# "Newton still needs the card you removed."

## DRAFT 5

Michael S. Engber
Apple Computer - PIE Developer Technical Support

This article was (will be) published in the February 1994 issue of Double-Tap magazine. For information about Double-Tap , contact the Software Frameworks Association at FRAMEWORKS@APPLELINK.APPLE.COM or (408)253-2765.

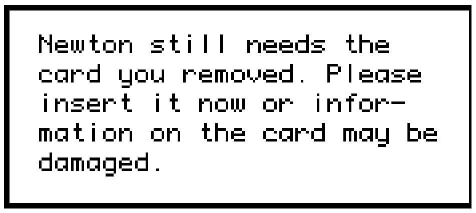

Newton still needs the card you removed. Please insert it now or information on the card may be damaged.

Figure 1 – Card Reinsertion Message (you blew it)

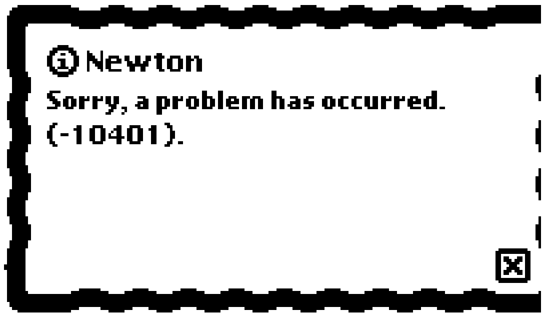

Newton
Sorry, a problem has occurred. (-10401).

Figure 2 – -10401 error (you blew it)

This article will explain how to avoid programming errors that result in the messages shown Figures 1 and 2. Well-written Newton applications should be able to reside on a PCMCIA card and not trigger the reinsertion message or -10401 error when the card is removed. All you have to do is make sure you don't leave any references to your card around after it has been removed.

This sounds simple enough, but first you need to understand fully why the reinsertion message occurs, when it occurs, and what techniques are available to avoid it. This, in turn, requires an understanding of a variety of other topics. While I may seem to take a long time to get to the point, be advised this is an area where "a little bit of knowledge can be dangerous." I suggest taking the time to read and digest the whole article rather than just sprinkling calls to EnsureInternal throughout your code. I assume the reader is familiar with NewtonScript, has experience using NTK to write Newton applications, and has some idea about what InstallScripts and RemoveScripts are.

An additional disclaimer: Some of the details discussed in this article may change in future Newton products or with future versions of NTK. I've tried to point these out whenever possible.

## Where Objects Reside – NewtonScript Heap Vs Packages

NewtonScript data types can be divided into two categories; immediates and references. Immediate types are integers, booleans, and characters. Immediate data is stored completely in 32 bits, not as a pointer to the actual data. Everything else is a reference. That is, a pointer to the actual data. If you want to check out a particular object, use `PrimClassOf(foo) = 'Immediate`.

Non-immediate NewtonScript objects reside in one of two places: the NewtonScript heap or a package. The NewtonScript heap is NewtonScript's working memory. When your program allocates an object dynamically (e.g., `foo := [x, y, z]`), the object resides in the NewtonScript heap.

The other place objects can reside is in packages. For example, if a string literal appears in your code, that string resides in your package. That string is a read-only object and you will get an error if you try to change a character in it (e.g. `s[0] := $Z`) or change its length (e.g. `SetLength(s,4)`). A good analogy is string literals in ANSI C. In C, you need to treat string literals as read-only because the compiler reserves the right to place them in protected memory or have different literals share the same memory. In many C implementations you can often get away with being sloppy, but on Newton your package is always in protected memory.

Here is a list of some common ways you can get references to objects in your package in your NewtonScript code:

- string literals – `"abc"`
- real literals – `3.14`
- quoted symbols – `'foo`
- quoted arrays - `'[1,2,3]`
- quoted frames – `'{slot1: 42, slot2: nil}`

Note that [É] or {É} appearing in your code without the quote means that a new array or frame will be created in the NewtonScript heap when the code executes. However, it's still possible for the resulting structure to have references to your package. Consider the following code:

```
{slot1: "abc", slot2: '[1,2,3]}
```

A new frame is allocated in the NewtonScript heap, but its slots contain references to objects in the package. The problem can be even more subtle. Consider the following code:

```
{slot1: nil, slot2: 42}
```

Even though the slots don't contain references to the package, the symbols used for the slot names reside in the package. This is a subtle point that I will address several times in this article.

I should also point out that all the template and proto layouts you create using NTK reside in your package. These are actually frames that are created at build-time. Since they are created at build-time, their slot initializers are evaluated at build-time. This leads to another way you can run into package references: slot initializers. For example, if you use `[1,2,3]` to initialize some slot in a template, that slot's initial value will be an array that resides in your package. It doesn't matter if you quote the array or not. It doesn't even matter if you use `Clone([1,2,3])` instead. The array will reside in your package. Remember, the slot's initial value is computed at build-time.

A related point is that objects you create in your "Project Data" file are also created at build-time. For example, when you create a build-time global in "Project Data" using `gFoo := {a: 1, b: 2}` and then use `gFoo` to initialize some slot, you end up with a reference to your package. Things work the same way for BeforeScripts and AfterScripts. Only code that runs after your package is downloaded to the Newton can create objects in the NewtonScript heap.

You may already be familiar with the distinction between objects in the NewtonScript heap and objects in packages from experiences with error -48214, "object is read-only." Objects in packages are read-only. You must use copies in the NewtonScript heap if you need to modify them. This is when most people discover the functions Clone and DeepClone. These functions are sufficient to solve the read-only object problem, but they are not always sufficient to solve the invalid reference problem.

## Invalid References

The Newton uses PCMCIA cards more like an extended RAM space than like an external file system. This combined with the fact that the user can remove a PCMCIA card at any time, causes an interesting problem. To draw an analogy to traditional programming, when the user removes a PCMCIA card it's more like yanking out some of a running computer's RAM than like removing a floppy disk.

This means it is possible to have a valid reference suddenly become a reference to memory that is no longer available – that is, a reference to an object on the PCMCIA card the user is now holding in their hand. I will use the term "invalid reference," to describe such references. Drawing on your own programming experience, you might think of these as dangling pointers (pointers to objects that have since been deallocated or moved), but they are actually worse. Invalid references refer to memory that no longer even exist.

There is another way users can make your package unavailable besides removing the card it's on. They can use Remove Software – via Prefs for internally stored packages or Card for packages on cards. The issues just discussed still apply. However, in this case you really do create dangling pointers as opposed to references to non-existent memory. For simplicity, this article discusses the problem in terms of card removal, but keep in mind the issue is package removal in general, not just card removal.

## Using an Invalid Reference

Newton handles attempts to use an invalid reference in one of two ways. If the invalid reference is to a card that's in the process of being removed, Newton puts up the reinsertion message, freezes (probably in the middle of executing your RemoveScript) and waits until the user reinserts the card so it can continue where it left off. If the invalid reference is encountered after the card removal process completes, the Newton throws a -10401 exception and aborts the code that was executing.

When users get the reinsertion message, they really have only two choices: reinsert the card or reboot. Reinserting the card is safest because it lets the code finish executing. This doesn't necessarily mean they will subsequently may be able to remove the card without incident. The most troublesome applications continue causing the reinsertion message and the user must either remove the application from the card (with Remove Software) or reboot the Newton.

In the -10401 case, users have no choice but to dismiss the error dialog. At this point they may be able to continue using Newton or it's possible that the invalid reference remains and will continue to cause errors until the Newton is rebooted.

In either case, rebooting will get rid of any invalid references. Recall that these references exist in the NewtonScript heap and, when you reboot, the NewtonScript heap is reinitialized. This is not to say rebooting is a panacea. Rebooting from the reinsertion dialog means aborting some code that was executing – probably your RemoveScript. Similarly, the -10401 error means some code was aborted (by an unhandled exception). In practice, usually this isn't a problem, but, if the aborted code was part way through some operations involving soups, it is possible things could be left in an inconsistent state.

## PCMCIA Card Removal

The Newton is designed so that as soon as the user unlocks a card it's no longer available. It doesn't matter when they physically eject the card. From your program's point of view it's gone as soon as the unlock switch is thrown. This might be a useful fact to note when you're stress testing your application for handling card ejection. You can save time and reduce wear on the eject button and PCMCIA connectors by simply unlocking and locking the card.

Your program doesn't get a chance to do something before the card goes away, but you do get a chance to do something immediately afterward, in your RemoveScript. That is often where programs get into trouble, usually because the RemoveScript tries to access an object residing on the card.

This access might be as blatant as dereferencing a slot in your base view's template or as seemingly innocuous as trying to use RemoveSlot to clean up a printing format you tucked away in the root view. There are a wide variety of ways to access your card inadvertently. These errors and how to avoid them are discussed in a later section.

The sequence of calls when a user removes a card, hopefully, will consist of the user unlocking the card followed by your RemoveScript executing – end of story. However, if your RemoveScript uses an invalid reference the sequence will be:

- app is closed - ViewHideScript  and ViewQuitScript called
- sometime later the user unlocks the card
- RemoveScript called
- invalid reference encountered – RemoveScript suspended
- user reinserts card
- RemoveScript continues
- InstallScript runs (due to card reinsertion)

If your application happens to be open when the user removes the card, the sequence of calls will be slightly different:

- user unlocks the card with your app open
- RemoveScript called
- invalid reference encountered – RemoveScript halts
- user reinserts card
- RemoveScript continues
- app is closed – ViewHideScript  and ViewQuitScript called
- InstallScript runs (due to card reinsertion)

Even if you fix your RemoveScript so it doesn't use any invalid references there's not too much you can do about the occurrence of the reinsertion message during this second sequence. After your RemoveScript executes, your application is sent a Close message. Searching for the Close method will cause access to an invalid reference, the _proto slot of your base view. Your base view's _proto slot references its template which is in your package.

The purpose of presenting these detailed sequences is not so you'll write your code to rely on them. You definitely should not. These details may change in future products.

The two points you should note are:

- Your RemoveScript and ViewHideScript/ViewQuitScript can run in any order. Don't assume that the RemoveScript runs last.

- The one case where it's ok to trigger the card reinsertion message is when the user removes the card while your application is open. This one is presently beyond your control.

## TotalClone() and EnsureInternal()

The weapons at your disposal to fend off the reinsertion dialog are the global functions TotalClone and EnsureInternal. They are cousins of the global functions Clone and DeepClone.

Most people grasp the difference between Clone and DeepClone. Clone copies only the "top level" of an object, while DeepClone copies all the objects it references, and all the objects those objects reference, and so on. So the question that arises is: "If DeepClone copies everything, what does TotalClone do?"

The answer is that DeepClone doesn't actually copy everything. It copies everything necessary to ensure that no change you make to a DeepClone of an object will affect the original. However, a DeepClone of an object may still share certain parts with the original – parts that aren't normally destructively changed like symbols. Currently, there are no NewtonScript functions that destructively change a symbol, so symbols aren't copied by DeepClone. A common example would be the symbols used as slot names by frames. There is no reason for DeepClone to copy them and not copying them saves memory.

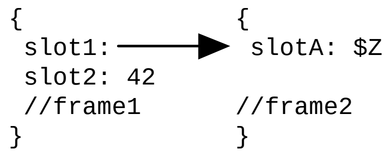
This is not normally a problem, unless the original object happens to reside in your package. Your package can be removed (card removed or Remove Software) leaving your DeepClone with invalid references. To deal with this problem, the function TotalClone is provided. It makes a "deep" copy of an object, but more importantly, it guarantees the object it returns resides entirely in internal RAM or ROM.

There is a related function, EnsureInternal, which simply guarantees the object it returns is entirely in the internal RAM or ROM. It may or may not return a new object. It only copies as necessary to ensure the object is in internal RAM.

Strictly speaking, packages in Newton's internal store are already in internal RAM. So to be precise I should say TotalClone and EnsureInternal guarantee the objects they return are entirely in the NewtonScript heap or the system ROM. The terminology gets a little sloppy here. Generally, when someone talks about to an object being internal, they mean it's in the NewtonScript heap or system ROM. If you want to nit-pick, you can argue that EnsureInternal should have been called EnsureInNSHeapOrROM.

Normally, the function you want to use is EnsureInternal. This is because usually the only thing of importance is that the result reside internally, not that you get an entirely new object. I've noticed a lot of people using TotalClone when EnsureInternal would have been sufficient. EnsureInternal may use less memory if parts of the object are already in internal RAM. In practice, you may find that EnsureInternal often ends up making a "TotalClone" of the object. I still recommend using EnsureInternal because it saves memory in some cases and it clarifies that your code relies only on the result being internal and not on it being a completely new object.

Here are some examples contrasting EnsureInternal and TotalClone. Figure 3 shows a simple data structure consisting of `frame1`, which contains a reference to `frame2`. Table 1 shows how TotalClone and EnsureInternal behave differently (with respect to what is copied) depending on whether or not `frame1` or `frame2` are entirely internal.



```
{               {
 slot1: ──────▶  slotA: $Z
 slot2: 42
 //frame1       //frame2
}               }
```

Figure 3 - Data Structure for Table 1

| frame1 | frame2 | TotalClone | EnsureInternal |
|---|---|---|---|
| internal | internal | clone frame1 & frame2 | no cloning |
| internal | external | clone frame1 & frame2 | clone frame1 & frame2 |
| external | internal | clone frame1 & frame2 | clone frame1 only |
| external | external | clone frame1 & frame2 | clone frame1 & frame2 |

Table 1 - TotalClone Vs EnsureInternal

## Fifty Ways to Leave Your Reference

### Leaving a Direct Reference Around

This is the most obvious case. You tuck a reference somewhere and then forget to clean it up later. For example, in your ViewSetupFormScript you do something like:

```
AddArraySlot(SoupNotify,kMySoupName);
AddArraySlot(SoupNotify,kAppSymbol);
```

The two elements you're adding to the SoupNotify array are references to your package – a reference to a string literal and a reference to a symbol. If you entirely forget to remove these two elements, you'll corrupt SoupNotify. You may be thinking, "That's pretty obvious. I would never do that," but there's still a way you can get caught.

You also have to think carefully about when you plan to clean up. If you wait until your RemoveScript runs it may be too late. At that point your package is gone and the references are invalid. If you knew the indexes of your two elements of SoupNotify you could remove them with ArrayRemoveCount without actually having to access them. Unfortunately, in the case of SoupNotify you can't know those indexes with certainty. They might have changed from their original positions as other applications removed their elements from SoupNotify. To find your element you need to search SoupNotify, but even a simple pointer equality check will cause the bad references to be accessed and trigger the reinsertion message.

One possible solution to this dilemma is to remove the elements in your ViewQuitScript. You can be sure the references are valid when ViewQuitScript executes. Another solution would be to call EnsureInternal on the elements before adding them. This will allow you to safely remove them during your RemoveScript.

## Corrupting a Frame with an Invalid Slot

This is probably the most common and most subtle of the errors. When you add a slot to a frame, you have to consider the symbol used as the slot name and possibly the actual contents of the slot. I commonly see someone creating a slot, being careful to call EnsureInternal on its contents, and then later using RemoveSlot to clean up. Two common examples are installing formatting frames in the root view and installing routing frames in the Routing global variable. Here is some typical code:

```
//install routing frame (typical error)
Routing.(kAppSymbol) := EnsureInternal(myRoutingFrame);
---
//clean up - (in RemoveScript)
RemoveSlot(Routing,kAppSymbol);
```

The above installation code does exactly the opposite of what it should have done. It uses EnsureInternal on the contents of the slot it's creating and not on the slot symbol. When the remove script runs, the reinsertion message will appear because the symbol used as the slot name in the global routing frame has become an invalid reference. The content of the slot doesn't matter, it isn't touched. It's the slot name that's the problem.

Once a frame is corrupted in this way, accessing any slot can be a problem. This is because accessing a slot requires searching the frame for that slot. If the search comes across an invalid reference, boom! You might notice situations in which some slots are safe to access and some aren't. You shouldn't take advantage of this observation. The details of slot access may change in future implementations of NewtonScript. If any slot symbol is corrupt, the whole frame should be considered corrupt.

The above installation code can easily be corrected as follows:

```
//install routing frame (works)
Routing.(EnsureInternal(kAppSymbol)) := myRoutingFrame;
```

This may look a bit strange, but it solves the problem. Notice that there is no need to call EnsureInternal on `myRoutingFrame` because RemoveSlot doesn't access the contents of the slot being removed.

A subtle variation on this problem is changing the contents of an existing slot. You might run into a situation where you're just changing an existing slot. You might automatically conclude that you don't have to call EnsureInternal on the slot symbol because you're not adding a new slot. Consider the following example:

```
//f is a frame: {_proto: {slotA: 42}};
f.slotA := nil;
//f is now: {slotA: nil, _proto: {slotA: 42}};
```

Recall that the first time you change a slot inherited from the proto chain, a new slot is created. So you may actually be creating a new slot. Consequently, you need to consider using EnsureInternal on the slot name. In the above example you could use:

```
f.(EnsureInternal('slotA)) := nil;
```

You can run into this problem changing any frame with a _proto slot. Some practical examples would be the root view and the UserConfiguration global variable (which inherits its default values from its proto chain).

**AddDelayedAction and AddDeferredAction**

I once ran into a case of a developer who would get a -10401 error only if he removed the card while his application was running. It turns out that he was using AddDeferredAction in his ViewQuitScript. When you remove the card with the application open, the deferred action is run after the card is gone. I suppose if I closed his application and then immediately unlocked his card I might be able to achieve the same error if I was quick enough.

The problem was that the code for the deferred action was inside his package. He got a -10401 error instead of a reinsertion message because the deferred action ran after the card removal was complete. In general, you have to be concerned about any deferred actions you set up as part of your ViewHideScript/ViewQuitScript code and any you set up from your RemoveScript.

The solution is to EnsureInternal the closure you pass to AddDeferredAction, but you don't want to be passing any old closure to EnsureInternal. Closures are more than function pointers. At the time they're created they close over the current lexical environment and the current receiver (self). This means when you EnsureInternal a closure you may be cloning a lot more than just the compiled code.

For example, if in your ViewQuitScipt you tried something as simple as EnsureInternal(func() 42) you'd get an out of memory error. Why? Because self (your base view) has a _parent slot that points to the root view. So you'd end up trying to make an internal copy of the root view.

You probably only want to EnsureInternal closures with empty lexical environments and receivers. Closures created at the build-time "top level" fit the bill. Examples of these would be evaluate or script slots that are initialized to func(É)É. . So a solution to the above developer's problem was to define his deferred action in a slot in his base view and then EnsureInternal the contents of this slot and pass it to AddDeferredAction.

Adding deferred actions in your ViewQuitScript is a pretty strange thing to do anyway. I can't think of any reason it why would be necessary. The real solution to this developer's problem was to eliminate that code entirely. However, delayed actions are a different story.

Since a delayed action executes after a specified time delay, anything could happen between when you call AddDelayedAction and when the closure you passed in gets executed. This means that to be safe, you have to call EnsureInternal on any closure you pass to AddDelayedAction. Of course, the above notes on using EnsureInternal on closures still apply.

In case you were wondering, normally you never have to worry about EnsureInternal when using ViewIdleScripts. The system takes care of shutting down your idle scripts when your view is closed. They won't be called after your package has been removed.

**Touching theForm or Your Base View in the RemoveScript**

Your InstallScript and RemoveScript are passed a single argument, a partFrame. One of the slots in partFrame is theForm. It contains a reference to your base view's template. When your RemoveScript is called, theForm will be an invalid reference. Trying to access it will trigger the reinsertion message.

As a way around this, some people have discovered that when their RemoveScript is called their base view is still in the root view and can be accessed using:

```
GetRoot().(kAppSymbol)
```

This is a bad idea. Under some circumstances you can get away with references to slots in your base view, but it's a risky thing to do. One problem is that the _proto slot of your base view is an invalid reference to its template (theForm). So if your code causes any inheritance to be used, the _proto slot will be accessed and the reinsertion message appears. Even if you know the slot you're after is in the base view (no proto chain search necessary) you're still likely to run into slots whose symbols aren't internal as described above.

My advice is: do not access your base view in your RemoveScript. However, it is safe to use the following code in your RemoveScript. Notice it doesn't access any slots in your base view or send messages to your base view.

```
  RemovePowerOffHandler(GetRoot().(kAppSymbol));
```

**Soup Related Problems**

Soup entries are one of the easier cases to handle with respect to invalid references. The system takes care of soup entries for you. You don't have to call EnsureInternal on a frame before passing it to AddToDefaultStore. However, you may want to Clone it for other reasons. Remember, when you pass a frame to AddToDefaultStore it's destructively changed.

The entry frames you deal with in your code reside in the NewtonScript heap. They are caches of the actual information in the soup which resides in a user store. That it means it's possible for you to modify one of these frames and taint it with an external reference. However, as soon as you call EntryChange or EntryUndoChanges, the system ensures the entry is internal.

Soup entries are not a problem, but there are some other things you need to watch out for when using soups.

In general, you want to set the slots you created in your base view to nil in your ViewQuitScript. This will allow these to objects to be garbage collected provided there are no references to them elsewhere. If you forget, it can be especially troublesome to leave references to soups or cursors. Leaving a cursor reference is a common source of -10401 errors.

Another way you can get a -10401 error is with SetInfo or SetAllInfo (both the store and the soup methods). Both the slot symbol and the value you pass to SetInfo need to be internal, as does the frame you pass to SetAllInfo. Use EnsureInternal on these arguments. If not, the info frame can become corrupted. This is not to say the information in the soup will become corrupted, rather, only the frame which serves as a cache for the actual soup data will be corrupted. Rebooting will solve the problem.

### Notify Related Problems

You need to make sure the parameters you pass to Notify are internal. You might have noticed that Notify retains references to the last four messages it displayed and allows the user to scroll through them using the universal scroll arrows. Because of this, you need to ensure that the strings you pass as the second and third parameters to Notify won't become invalid references if your application is removed. The first parameter to Notify is an immediate, one of four pre-defined integer constants, so it's not a problem.

If you mess up, the next time Notify is called it will access an invalid reference and attempt to display a -10401 error using Notify. The result is that the Newton beeps endlessly as Notify keeps trying, and failing, to display -10401 errors. You can see the -10401 exceptions in the NTK Inspector if you have it connected.

## Being Too Paranoid

I don't want you to go sprinkling calls to EnsureInternal throughout your code as a result of reading this article. You need to make sure you understand when and where you need it and use it only when necessary. Otherwise you'll increase your code size and probably waste a lot of RAM. In this section I make some points about when you need and don't need EnsureInternal.

### Slot Access

When people learn that they can corrupt the root view using code like: `GetRoot().foo := myFormat` they start writing code like: `y := GetRoot().(EnsureInternal('myFormat))`. When you're simply accessing a slot there is no need to use EnsureInternal. It's only when you're creating a slot that you have to worry.

### InstallScripts and RemoveScripts Execute From Internal RAM

Your InstallScript and your RemoveScript are copied into internal RAM before they are executed. (This is always true for the applications you make with the current version of NTK. The rules will vary when NTK supports other types of code.) The implication is that you don't have to worry about using EnsureInternal in your InstallScript. (This goes without saying for your RemoveScript because at that point it's too late to use EnsureInternal.) Revisiting my previous SoupNotify example:

```
AddArraySlot(SoupNotify,kMySoupName);
AddArraySlot(SoupNotify,kAppSymbol);
```

If this code appears in your InstallScript, there is no need to use EnsureInternal. However, it's common practice for InstallScripts to utilize code that's actually stored in your package. Two common ways of doing this are with code like:

```
partFrame.theForm:Install()
 or
AddDeferredAction( func() GetRoot().(kAppSymbol):Install(),'[]);
```

These examples assume there is an Install method defined in your base view's template to handle your application's installation needs. Whether you call it directly in the template or in a deferred action to your base view depends on exactly what you need to do. The second method is probably more straightforward to get working. The advantage of keeping the bulk of your installation code in your package is that it lets you keep your InstallScript small. This is important since your InstallScript is copied into memory. Unfortunately, this same technique won't work for your RemoveScript. If you have to ask why, you need to re-read this article.

But I digress. The point is: if you keep your installation code in your package, it won't be copied into internal RAM and you will have to use EnsureInternal as necessary.

A slight variation on the above is that sometimes an InstallScript will need to grab data from theForm. For example, a printing format may be in a slot of theForm. Remember, theForm has not been copied into internal RAM. Any time you get data from theForm you need to consider whether or not you need to use EnsureInternal. In most cases you probably don't, but you should think about it.

There is yet another variation on the above theme. Your application has no way to access the partFrame that's passed to your InstallScript and RemoveScript. Sometimes people pass the partFrame to an installation method in their base view. Sometimes they even tuck a reference to the partFrame in a slot somewhere. It's sometimes useful to be able to access the partFrame so you can communicate with your RemoveScript – for example, provide it with information it needs to clean up. If your InstallScript puts something in a partFrame slot, there's no problem since the InstallScript is in internal RAM. However, if you've got code in your package operating on the partFrame you need to be careful not to corrupt it. Specifically, make sure to call EnsureInternal on the symbols for any slots you create in the partFrame otherwise you will corrupt it. If you expect the RemoveScript to be able to access an object referenced by one of those slots, the object needs to be internal too.

**ViewSetupFormScript, ViewQuitScript, et al**

Your ViewSetupFormScript, ViewSetupDoneScript, ViewHideScript, ViewQuitScript, etc., aren't copied into internal RAM as your InstallScript and RemoveScript are, in general, no other code is treated specially like the InstallScript and RemoveScript. That doesn't mean you automatically have to use EnsureInternal everywhere. Generally, you don't have to worry about calling EnsureInternal on references that you will clean up in your ViewQuitScript (or ViewHideScript). This is because your ViewQuitScript is guaranteed to run when your references to your package are valid.

**InstallScript Vs ViewSetupFormScript**

There's a nice symmetry we can observe. If you "do it" in your InstallScript and "undo it" in your RemoveScript there's no need to worry about EnsureInternal. Nor is there any worry if you "do it" in your ViewSetupFormScript and "undo it" in your ViewQuitScript. It's when you break the symmetry that you have to start worrying.

Unfortunately, you may not get to take advantage of this symmetry. Installing a full-blown application will actually involve a fair amount of code and data. Consider all the IA templates, meta-data specifications, routing frames, formatting frames, etc. You probably will need to keep most of it in your package, and this breaks the symmetry.

As a general guideline, do as much as possible in your ViewSetupFormScript and ViewQuitScript and as little as possible in your InstallScript and RemoveScript. Keeping your InstallScript small conserves memory (recall your InstallScript and RemoveScript are copied into internal RAM). As was noted in the previous section, you can also keep the size of your InstallScript down by using code that's stored in your package. However, it's still important to minimize the work you do in your InstallScript so as not to slow down the card insertion process.

There are some services it makes sense to register with as soon as the card with your application is inserted, such as IA or global Find – the user shouldn't have to have your application open in order to use Find or IA. There are other services that you only need if your application is actually open, such as RegisterCardSoup – there is no reason you need to be creating soups on every card you see unless your application is open. There are other cases that might be deceiving. Initially, it might seem you only need to be registered with SoupNotify when your application is open. However, the SoupNotify array is also used to notify applications when the user has edited the filing categories. If your application supports filling, you probably want to register with SoupNotify right away.

Basically you should put off registering with system services as long as possible and unregister as soon as possible. You need to think carefully about when you will be utilizing system services in order to decide between registering in your InstallScript or in your ViewSetupFormScript.

**Minimizing the Depth of Copies**

Sometimes you can get away with using Clone instead of EnsureInternal. Remember, EnsureInternal often ends up making a "deep" copy. For example, maybe you need to make sure a reference to an array doesn't become invalid, but you don't care about the elements. Use Clone to make a shallow copy of the array in internal RAM. Watch out. Clone won't copy the class symbol. This means the new array will share its class symbol with the original. Hence, its class can become an invalid reference. Below is a one-liner to make a shallow-internal copy of an array – including its class (note that SetClass returns its first argument as its result).

```
SetClass(Clone(origArray),EnsureInternal(ClassOf(origArray)));
```

Frames are a little trickier. Recall that Clone won't copy the slot symbols. So we can't start by simply using Clone to make a shallow copy. Instead, we can create a new slotless frame and then copy over the slots from the original one at a time – making sure the slots we create are internal. Below is some code that makes a shallow-internal copy of a frame.

```
local newFrame := {};
local slot,val;
foreach slot,val in origFrame do
 newFrame.(EnsureInternal(slot)) := val;
```

Building up a frame one slot at a time isn't the most efficient way to do things, but in this case we can't do better unless know ahead of time about the structure of `origFrame`.

**Never Clone Needlessly**

In general, applications should only do enough copying to make sure they can do their cleanup without accessing an invalid reference. For example, if you're creating a slot in the base view for a printing format, you'll remove the slot in your RemoveScript, but you won't access the slot's contents. So you only need to make sure the symbol you use for the slot name is internal, not the slot's contents.

Another example is adding an element to a global array such as FormulaList. FormulaList is an array of templates used to generate the panels in the built-in Formulas application. Your application can add a panel of its own by adding a template to FormulaList. Later, when it's time to remove the template you added, your RemoveScript must search FormulaList to find the template. Therefore, it's essential that the template doesn't become an invalid reference (yes, simple equality testing, with =, causes references to be accessed). However, there's no reason to make a deep copy of your template to add to FormulaList. You can create a shallow internal copy using the techniques previously discussed. In fact, your RemoveScript will not access any of the template's slots anyway, so a simple Clone should be sufficient (and use less RAM).

If you think carefully about what your code is doing, you should be able to decide when you need to call EnsureInternal, when Clone is sufficient, and when you don't need to use either of them. Of course, you should test your hypotheses thoroughly. It's very easy to overlook something, but don't let this discourage you from trying to reduce the amount of copying your application does. Your users will thank you for minimizing your application's RAM requirements.

## Conclusion

The reinsertion message and the -10401 error are two of the most common problems applications have. They can be especially annoying because they're usually discovered when you think you've almost finished. It works "perfectly," except when you try to remove the card. This, combined with the somewhat esoteric nature of the problem, makes for some world-class frustration.

Unfortunately, there is no simple recipe I can give you. This article has touched on most of the relevant issues. Perhaps some of it even makes sense. My advice is to test your application constantly by unlocking and locking the card it's on. Start this testing early in the development cycle and make changes to your code in small increments. Too often, people don't notice these problems until they're near the end of the project. There's nothing worse than a huge amount of code that doesn't work for some totally mysterious reason. Unfortunately, I frequently find myself in this situation. Typically I'm on the phone trying to talk some developer "off the ledge," usually without the luxury of being able to look at the source with my own eyes. Most of the time I can quickly track down the offending code using the following general guidelines:

• Insert the card and remove it without running your application. If you get the reinsertion message then it's probably something your InstallScript did. Most likely your InstallScript executed installation code in your package. Another possibility is that your RemoveScript is trying to use an invalid reference (e.g., `partFrame.theForm` or `GetRoot().(kAppSymbol).foo`)

• Insert the card, open your application, close your application, and remove the card. If you get the reinsertion message then you can be pretty sure it's something in your ViewSetupFormScript or ViewSetupDoneScript. It's possible that it's your ViewQuitScript. Judiciously commenting out code should let you narrow it down from here.

• If you reinsert the card in response to the reinsertion message, can you subsequently remove the card? If you can't, it indicates that the problem resides in the InstallScript or RemoveScript. If you can, it indicates it has to do with actually opening the application.

• If it's a -10401 error, make sure your ViewQuitScript sets to nil any base view slots that you have created. Use the Inspector to take a look at your base view. Be especially watchful for references to soups and cursors. Also, check out any deferred or delayed actions and calls to Notify. Remember, if the -10401 occurs when downloading your package, it may indicate a corrupted package. Are you in 32 bit mode? Try building on another machine.