Surviving the Grip

Note: This is a follow-up article to *"Newton still needs the card you removed"* by Mike Engber, first published in February 1994 in Double-Tap magazine.

Introduction and Terminology

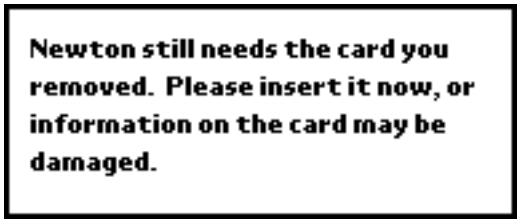

Figure 1 (you blew it)

If you see this slip, you have encountered the **Grip of Death** (aka **G.O.D.** or **the grip**). This happened because something has **touched** the card while it was being **unmounted**, causing the unit to require the card back to resolve the reference. If the reference isn't caught when the card is unmounted, it becomes a **bad pkg ref** which can cause throws at a later time.

Briefly, the grip happens when the OS follows a reference to an array, frame, binary object, symbol, frame map, VBO, or soup data on a card while the card is being unmounted.

> *Grips happen when you touch data on a card while the card is being unmounted.*

That isn't enough of a description for you to understand the problem or why it occurs, so read the *"Newton still needs the card you removed"* article. (You may want to read it two or three times.) This article assumes the reader is familiar with the common causes and workarounds for the grip. The original article is currently available on-line at

`ftp://ftp.apple.com/pub/engber/newt/articles/NewtonStillNeedsTheCard.rtf`

and on the Newton Developer CDs in

`Technical Information:Newton 1.x-related ARCHIVES:ARCHIVE - 1.x Articles:NewtonStillNeeds article`.

Since that article was written, we've undergone one major and several minor revisions to the Newton operating system. With the 2.0 (and later) OS, many of the things that used to be common causes of the grip are now easily avoided. For example, most of the OS

registries now ensure that the necessary elements of what's being registered are stored internally so that the corresponding unregistration can be done without touching the card or leaving bad references.

There have been many other changes and enhancements added to making working with PC cards easier. Smarter registries, soup definition frames, better package handling, atomic store operations, better cleanup, and debugging hooks make tracking down and avoiding the grip easy and almost convenient. This article discusses those changes and enhancements.

The most recent devices from Apple now have multiple PC card slots. The support has been in the OS since 2.0, but never before has hardware been available. This enhancement does not complicate the card removal process as far as your applications are concerned. The only thing to note is that removing a card can cause the OS to touch the remaining card, which isn't a problem unless you happen to eject both cards at the same time, in which case the OS itself will give you the grip on one of the cards.

Better Notification

Figure 2 (I blew it)

In the older versions of the OS, only the slip from Figure 1 would appear, and this slip still appears when there is no information available about the package that caused the problem. The slip in Figure 2 is displayed on 2.0 and later units when the OS can determine which package caused the problem. It's intended to provide a clue to the user about which package must be removed to fix the card. Don't let it be yours!

Removable/Remountable Cards

In 1.x, if you got the grip you were stuck in a classic "lather, rinse, repeat" loop. The Newton device would not proceed until you reinserted the card. On reinsertion, the OS would finish the unmounting process, then immediately remount the card.

Remounting the card reintroduced the grip. The only way out was to reset the unit. In 2.0, the OS stops after unmounting the offending card and gives you a chance to remove it safely, a significant improvement.

Figure 3 (Newton saved you)

Bad Package References

After a package is deactivated, the 2.0 OS replaces any references to data in the package with a special object, called a bad package reference. This is done by searching the NewtonScript heap, and doesn't take very long. (It's roughly the same as garbage collection.)

This is an important change. In the 1.x OS the invalid reference would remain unchanged, and could be used later. If the user were lucky, they'd get a -10401 "Bad package" error when this occurred. If they were unlucky, some other data would happen to be at that location in memory and the result would be unpredictable.

On 2.0 and later, using a bad pkg ref typically causes a throw of an `evt.ex.fr` exception with the error code -48221: "Reference to deactivated package." Depending on what operation is being done with the bad pkg ref, other errors are possible. For example, trying to find the length of a bad pkg ref will generate an `evt.ex.fr.type;type.ref.frame` exception with error code -48418: "unexpected immediate." The string `<bad pkg ref>` will be displayed in the `'value` slot of the exception frame, which provides the needed clue that it's a card problem.

The change makes it much more likely that you'll notice immediately when using a bad pkg ref, so it's now much easier to find out what your application has done to cause the problem. Typically simply setting `breakOnThrows` to `TRUE` and doing a stack trace when the error occurs will tell you enough to find and fix the problem.

-48221 errors or any error involving a <bad pkg ref> means you have a grip related problem where the OS simply didn't notice the bad reference until later.

Suppressing Package Activation

Occasionally during development you'll create a package so horrible it resets or locks up the unit during installation. With 1.x, you'd either have to have had the foresight to put the package on a card during debugging and then erase the card by having the prefs application open when it was mounted, or you'd have to do a hard reset, erasing all data on the internal store.

With 2.0 there's a way out. Hold down the pen along the left edge of the mostly-black splash screen during a reset and you'll get a chance to prevent packages on the internal store from being installed, or hold the pen there while inserting a card to mount the card without activating any packages on that card. Once this is done, you can simply delete the problem package and preserve the rest of your data.

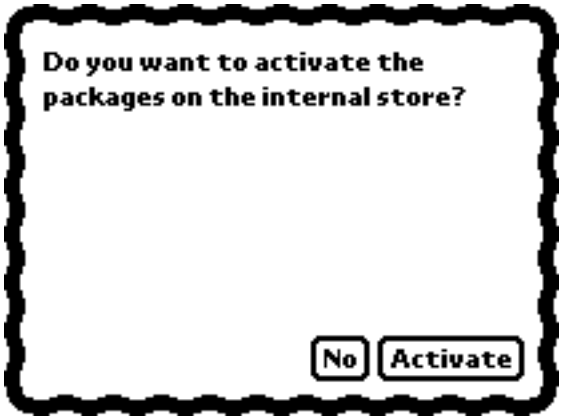

Figure 4 (phew)

By "along the left edge" we don't mean all the way against the edge of tablet, but rather over the display area, within about 1/4 inch from the edge. Holding down the pen along the top edge of the screen not only allows you to avoid mounting packages on the internal store, but also resets the backdrop application to Notes.

Hold down the pen along the left edge of the screen during reset or card mounting to prevent activation of packages.

Open prefs then insert a card to erase the card without touching any data on it.

Hold down the power key while resetting to erase the internal store.

Locking a Package

When an application on a card is open or in use, ejecting the card is guaranteed to cause the grip. During card unmounting, packages on the card are deactivated. Deactivating a package executes the `RemoveScript` for each part in the package, and also closes the base view for any form parts in the package. If nothing else, the system searches your base view's `_proto` chain for the `Close` method, and this will cause the grip.

This is fairly safe, and users quickly learn to close applications before ejecting a card. It's not practical to work around this, so don't try. Any workaround would require trying to `EnsureInternal` enough of your application so that it could be closed without causing the grip, and this would be a total waste of precious NewtonScript heap space.

However, this process does mean your application may be closed in unusual circumstances, even when it's the backdrop or doesn't provide a close box. For certain operations this may be a problem; live connections may be have to be aborted or gathered data may need to be discarded. To help avoid this, the 2.0 release of the OS has support for reserving a package, and the store on which the package exists.

Reserving a package is done by calling `MarkPackageBusy`, giving a reference to the package along with some text information to be shown to the user if necessary. `MarkPackageNotBusy` cancels the reservation. When a package is reserved, the user gets an extra warning if they attempt to delete it, freeze it, or move it to another store. These functions are documented in the Newton Programmer's Reference.

Reserving a package effectively reserves the card that contains the package. Attempting to eject such a card does something new. The "Newton still needs the card you removed" slip still appears, but with the reserving application's name as provided to `MarkPackageBusy`. Unlike during normal unmounting, this slip appears immediately when the card is unlocked, and processing is suspended until the card is reinserted. On reinsertion, processing picks up right where it left off. The card is not unmounted, the applications are not closed, and as far as most operations in NewtonScript are concerned, nothing happened. The card is effectively "locked" into the unit.

The OS provides a way to lock a store briefly so that it may be used without fear of being ejected. The store method `BusyAction` will execute a call-back function with the store reserved until the call-back completes. Ejecting the card while the call-back is executing produces the same "locked" result.

If you are designing robust applications, you might also check out the store method `AtomicAction`. This executes all the store-related operations in a call-back function as an atomic unit, so that if any failure (such as a store full error) occurs during the call-back the store is left in the same state it was in before the call-back started.

You may wonder why all apps don't simply reserve their packages when they open and unreserve them when they close. When a user ejects a card, it's generally their goal to get the card out of the unit. If the card is locked by your application, the user would reinsert the card, close the offending application, and then eject the card again.

Not reserving the packages is cleaner. Consider what happens if two applications are open. Only one produces a grip message when the card is unlocked. The user reinserts the card, closes that app, ejects the card again, and then the second application produces a grip message. Repeat as necessary, and it quickly gets annoying. Not locking the packages means the user is notified only once. After reinserting the card, package deactivation continues grip-free for all remaining packages.

A rule of thumb is to mark packages busy for as brief a time as possible, to minimize the inconvenience to the user. Only lock a package if the inconvenience caused by removing the package or closing the application is greater than that caused by having to remount the card and manually abort some operation.

> *Mark packages or stores busy only when necessary to avoid user inconvenience.*

Tracking Down the Grip

When the grip is encountered while unmounting a store, the 2.0 OS tucks away a reference to the offending data. With the help of a debugging package written by Mike Engber, you can get this reference, and find out exactly what it was that caused the problem.

This debugging package is called FindGOD, and the package is available on-line at

`ftp://ftp.apple.com/pub/engber/newt/FindGOD.sit`

and on the Newton Developers CD (#12) in

`Tools:The UN Files:FindGOD`

Using it is fairly simple: install the FindGOD package on the internal store, yank the card to reproduce the grip, reinsert and then remount the card, then call `FindGOD()` from the NTK inspector. The object that caused the grip will be displayed, which typically will tell you exactly what you need to fix to avoid the problem. The README that comes with FindGOD gives more details, so read it.

Use FindGOD to quickly track down the cause of the grip.

Conclusion

The 2.0 OS vastly improves the situation with respect to the grip of death conditions. You still need to be careful when writing your applications, but at least now if you encounter the problem, tracking it down is easier.

Read the original "Newton still needs the card you removed" article.