Mock Entries for Debugging

by Bob Ebert, Newton Developer Technical Support

The Problem

The NSDebugTools package, part of NTK 1.6, allows you to set breakpoints in NewtonScript code. This is very powerful and will help you find many of your bugs which don't raise exceptions. However, once in a while what you need to know is when some object is accessed, not when some line of code is executed. One way to do this is to set the global variable trace to `TRUE`, but then you typically have to wait a long time for the trace output to stop scrolling by, then wade through reams of data to find the accesses you're interested in.

Mock Entries to the Rescue

A little-known and even less frequently used feature that was added to the Newton OS in the 2.0 release is the ability to create entry-like objects, called *Mock Entries.* These objects are composed of two parts. One part is a simple NewtonScript frame, often called the *cached entry*. The other part is a *handler* which knows how to create and save the cached entry. The parts together are sometimes called a *fault block*; when something needs the cached entry and it doesn't exist, a *fault* occurs and a message is sent to the handler, which then creates or *faults in* the entry.

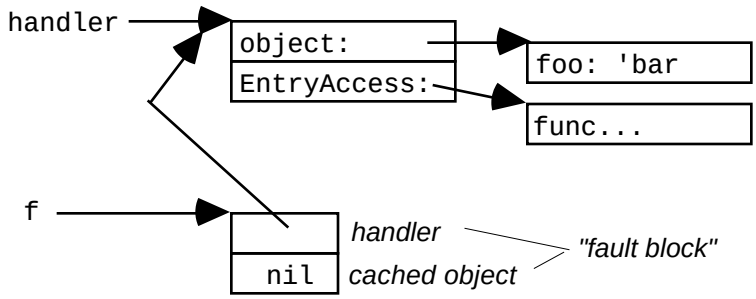
The Newton OS in 2.0 allows you to create these fault blocks for your own objects, which means you get to write the code that executes when the cached object needs to be faulted in. The code that creates the cached entry can also do other things, for example it could enter a breakloop, which would give you a chance to look at the stack and see what's accessing the frame.

Here's a simple example of using a fault block for debugging. We'll create a simple frame that prints an exclamation point in the inspector and beeps the speaker every time it's accessed.

```
handler := {
   object: {foo: 'bar},
   EntryAccess: func(mockEntry)
      begin
         write("!");
         GetRoot():Sysbeep();
         object;
      end,
   };

f := NewMockEntry(handler, nil);
```

f is now a Mock Entry. Here's how it looks to the OS:



When any slot in f needs to be accessed, the OS checks to see if there is a cached object. Because we passed NIL as the 2nd argument to NewMockEntry, and EntrySetCachedObject hasn't been called, there will be no cached object for f. When there is no cached object, the OS calls the handler's EntryAccess method, which is expected to create the cached object, tell the OS about it using EntrySetCachedObject, and return it.

We cheat and don't call EntrySetCachedObject. The OS uses the return value of EntryAccess as the object for this access, and since we return handler.object everything works, but next time something touches f, EntryAccess will be called again. There's our hook—every time some part of the OS reads or writes any slot in f, EntryAccess is called.

To almost all of the Newton OS, f looks like a regular frame. f.foo evaluates to 'bar. ClassOf(f) is 'frame. f.baz := 42 will add a slot to the frame, which is also referenced as handler.object in our example, so the next time the frame is accessed, the modified object will be returned. The illusion is complete, only the test function IsMockEntry() can tell that f is a mock entry and not a normal NewtonScript frame.

An Improvement

If you try this, you'll see that the frame is accessed a lot more often than you might think. Getting the value of f.foo calls EntryAccess twice. Setting a slot also calls EntryAccess twice. Creating a new slot calls it 11 times. The print function must do a lot, because printing f in the inspector calls the EntryAccess method a whole bunch of times.

In our application debugging example above, we really only want to know when the object is being used for the first time in a while. Recall that if the OS finds that a cached object exists, it won't call EntryAccess but will simply use the object. So to prevent excessive calls, our EntryAccess method will now create the cached object. The trick then becomes clearing the cached object. I've found that clearing the object at a deferred time works well—typically it's

cleared the next time control returns to the top level, which is soon enough to catch most bugs. Here's how to do that: (the bold text is new)

```
handler := {
   object: {foo: 'bar},
   EntryAccess: func(mockEntry)
      begin
         write("!");
         GetRoot():Sysbeep();
         EntrySetCachedObject(mockEntry, object);
         AddDeferredCall(
            GetGlobalFn('EntrySetCachedObject),
            [mockEntry, nil]);
         object;
      end,
   };
f := NewMockEntry(handler, nil);
```

You might want to experiment with clearing the cached object at other times.

Limitations

The `EntryAccess` method of the handler object is called only when a slot in the frame is accessed. That is, a statement like `g := f` won't cause the `EntryAccess` method to be called, since no slot in `f` is accessed. The result of that statement will be that you now have two references to the mock entry "fault block", and either `g.foo` or `f.foo` will cause the `EntryAccess` method to be called.

The Newton 2.0 OS only supports creating mock objects that are backed up by frames. While it's not guaranteed, you might experiment and see what happens if the object is an array or a binary object. (But back up your data first!) Try some special case binary objects like strings or real numbers. Depending on your situation, you may be able to use this debugging technique with arrays or binary objects as well as with frames.

I've found that some parts of the OS work normally in this case—the mock object is treated just like a string, bitmap, array, or whatever. Other parts of the OS "notice" that the object is a fault block and not the appropriate object type, which typically causes a throw. The error messages in this case can be interesting. For example, putting the string `"foo"` in the `object` slot of the `handler` will create an object that appears to be a string. Printing works, but functions like `StrLen(f)` or the accessor `f[0]` "notice" that the object isn't a string, and throw with the seemingly contradictory error message: `"Expected a string, got "foo"."` This happens because the exception printer

doesn't notice that the mock object isn't a string, and so it calls `EntryAccess`, gets the string, and prints it.

Advanced Techniques

You can put any NewtonScript code in the handler, specifically in the `EntryAccess` method, so you can use this trick to do other things. For example, you could add a counter and find out how many times a frame is accessed during some operation. You could add in a test to see if some slot in the frame has changed, and stop when it gets a certain value.

Unfortunately, when the `EntryAccess` method is called you don't have any information about what's happening. You can't tell if a slot is being read or set. You can't tell which slot (or element, or byte) is being accessed. If you write code that watches for changes you end up finding out after the change takes place.

But this can still be useful. Consider if your handler set `trace` to `TRUE` and then set it to `NIL` again in a deferred call. This would do a great job of limiting the trace output to only code that actually used the object. The `EntryAccess` method might also watch for some change and, upon detecting the change, set `trace` to `NIL` and enter a breakloop. That way you'd know without doubt that the last section of trace output was the one you needed to look at.

You might even consider using mock objects to implement a kind of sentinel that lets you know when other applications access your objects.

Conclusion

Being able to have your code execute when a frame is accessed is powerful, and has lots of good uses. However, keep in mind that any observations you or I make about how the OS deals with mock objects should be used only for debugging. That is, it would be a mistake to write production code that relies on `EntryAccess` being called twice when reading the value of a slot. See the Newton Programmer's Guide for Newton OS 2.0 for more information on using Mock Entries, including what's supported and how to do mundane things like simulating real soup entries.