NEWTON Q&A: ASK THE LLAMA

My llama senses have been overwhelmed by a call for some information on performance. All the questions in this issue's column will relate to performance in some way. There are two important pc to remember: 1) None of these tips will work by themselves; you must *measure* your code. Use **Ti** use the **trace** global (see below), use **Print**. Find out where your code is slow, or where your application is bloated. 2) There is no silver bullet for a problem; you must *experiment* with different solutions. In the words of my wise programming master: "When is a llama not a llama? . . . When i guanacos." Or, "When you can snatch these coconuts from my hand, then it will be time for me to l

Q *I'm building an application that has a large set of static data. I search on a key term (a string) an all the data associated with that string. Mike Engber's "Lost In Space" article (in the May 1994 of PIE Developers magazine) says that I should include this data in my package and things will fast. But this doesn't seem to be the case. I have thousands of frames of data. Each frame conta one or more slots with strings that contain the key terms. I use FindStringInFrame to find all references to a key term but this takes a long time. Am I doing something wrong?*

A This may seem like a simple question, but it isn't. The root of the problem is that you've made assumption that functions provided in the ROM are fast, so they'll solve your problem. In this case, you assumed that FindStringInFrame would be fast. You're both right and wrong.

FindStringInFrame is fast, but it still has to linearly search every slot in every frame recursivel That means that if you have thousands of entries, it's checking thousands of frames. You can ta about how long something will take by calculating the worst case. FindStringInFrame has to se all your data frames (thousands of items), and for each frame it has to check each slot to see if i string. If so, it then has to check to see if the string you gave it matches the string it's looking f (step by step down the string). So if you had *n* strings (not just data items), and the average len of a string was *m* characters, that's *n* **m* checks. That makes *n* **m* time. In computer science terms, you would say that FindStringInFrame is an O(*n* **m*) operation (this is called Big-Oh notation and, in its simplest form, refers to the worst-case time).

This means you should think about other data structures and methods of accessing them. In you case, a simple change of data representation would result in a massive speedup. The idea is to m the expression in the Big-Oh notation have the smallest possible value. One way to do this is to reduce the search time for your key phrases. Since you have a fixed set of data, you can sort the and use a binary search algorithm. You can store the actual data in arrays and store indexes alon with the key items.

The nice thing about binary search is that you're always cutting your search space in half. On average, you only have to check log to the base 2 of the data. In Big-Oh notation, that's O(log *n*
Of co
you st

have to do the individual string comparisons, so you end up with O(*m* log *n*). So for 1000 item
FindStringInFrame takes 1,000,000 time units, but the modified method takes 3,000, a speedu
300 times! It's unlikely that a function implemented at a low level performs 300 times faster tha
custom NewtonScript code.

This excursion into computer science should make you think about your data structures and ho
you access them. Of course an academic exercise can take you only so far. You also have to ge
your feet wet and test the code. You can use Ticks to get rough estimates of time, and Stats (aft
GC) to get estimates of memory.

Q *The following is a viewClickScript from a pickList button in my application. Why does it take s
long to execute?*

```
viewClickScript.func(unit)
begin
   currentPickItems := [];
   for i := 0 to Length(defaultPickItems) - 1 do
      if i = currentSelectedItem then
         AddArraySlot(currentPickItems,
            {item: defaultPickItems[i], mark: kCheckMarkChar});
      else
         AddArraySlot(currentPickItems, defaultPickItems[i]);
   if :TrackHilite(unit) then
      DoPopUp(currentPickItems, :LocalBox().right+3,
               :LocalBox().top, self);
end
```

A There are several possible reasons why your code would execute slowly. Since they potentially
apply to lots of code out there, I'll go through each one separately. At the end is a rewritten
function that should execute considerably faster.

- Lookup costs. Assuming that currentPickItems, currentSelectedItem, and defaultPickItems
slots somewhere in your view hierarchy, at best they're slots in the pick button, at worst in
base application view. Remember that each access to a variable requires an inheritance look
check locals, then globals, then current context, then the _proto chain, then the _parent cha
This cost isn't high for single references but can be deadly in loops. Every cycle through yo
loop, you're doing three lookups; that's a lot of overhead. The solution is to use local varia
for faster access.
- Unnecessary object creation. The AddArraySlot call will grow, and potentially copy, the ar
the NewtonScript heap, resulting in a lot of unnecessary memory movement. Since you kno
length of the currentPickItems array in advance, you should preallocate the array and use th
accessor (that is, [n]) to add array elements. You can use the Array function call to allocate
ar

```
local pickItems := Array(Length(defaultPickItems), nil);
```

- Unnecessary execution. You need to create a new pick list only if the call to TrackHilite suc
  You should make the TrackHilite conditional the outer conditional:

```
if :TrackHilite(unit) then
   begin
      // construct pick list and DoPopUp
      ...
   end;
```

- Inefficient variable initialization. It's inefficient to use a loop for initializing currentPickItem defaultPickItems, because currentPickItems has only minor differences. It's better to use C for initialization. This way you get a new array whose elements are references back to the a items in defaultPickItems. All you need to do is replace the individual references in currentPickItems with their new or modified values. It's the difference between an O(*n*) ope (traversing all the array items in defaultPickItems) and an O(1) operation (accessing only th changed item). In other words, expect about an order of magnitude difference.
- Unnecessary slot. In this case you don't need to have a currentPickItems slot since its value recreated each time the viewClickScript is executed. You're better off using a local variable

The modified code is shown below. To illustrate the savings, I ran a brief test using a defaultPickItems array of 10 elements. Each function is called 100 times (note that TrackHilite always true) and found the following code to be over 6 times faster than the original code.

```
viewClickScript.func(unit)
begin
   if :TrackHilite(unit) then
   begin
      local pickItems := Clone(defaultPickItems);
      local selectedItem := currentSelectedItem;
      local l := :LocalBox();
      if selectedItem then
         pickItems[selectedItem] :=
            {item: pickItems[selectedItem],
            mark: kCheckMarkChar};
      DoPopUp(pickItems, l.right+3, l.top, self);
   end;
end
```

Q *I've written my own IsASCIIAlpha, IsASCIINumeric, etc. functions. They seem to be really s Why is that? Here's my IsASCIIAlpha:*

/
/

```
returns true if s is an alpha string
// i.e., if s is between a..z or A..Z
// s - string to check
IsASCIIAlpha.func(s)
begin
   local c := Upcase(Clone(s));
   local i;
   for i := 0 to StrLen(c) - 1 do
      if (StrCompare(SubStr(c, i, 1), "A") < 0) or
         (StrCompare(SubStr(c, i, 1), "Z") > 0) then
         return nil;
   true;
end;
```

A First comment. Newton is a Unicode based device. ASCII is a subset of Unicode (from 0x000 0x007F), but Unicode characters up to FFFD are documented. Make sure you realize that your routing is just checking some of the characters on page 0 (i.e., characters of the form 0x00*nn*), it must deal with all characters.

The main source of the slowness is that you are using string string functions when character functions would be faster. The distinction is subtle but important. In the code above, you loop through each length 1 substring of the target string to determine whether it's an alpha character. this takes time. The Upcase call is O(*n*), as are the SubStr and StrCompare. Of course, the StrCompare isn't really that slow, but it's still slower than you need.

The SubStr call is returning a single character at a time, but in the form of a string. That means there is a memory allocation for at least two characters (the content and the null terminator) for e call to SubStr. A better way is to compare each character of the string. In certain circumstances can access a character at a time with the array accessor (that is, []). An example of a function th does this is IsASCIIAlpha3 (see the code on this issue's CD). In general, when you need eithe single character from a string or character-by-character access, the array-like syntax is faster.

Note that the final fix to the code is that it doesn't do any preprocessing of the string; instead it a lookup in an pregenerated array of valid alphabetic ASCII characters. That gives it a significa speed advantage. Since timing in the Inspector is a useful technique, the code to do the timings print results is included on the CD. Also note that this function is specifically for ASCII charac so characters like é and ß would fail.

Q *I'm trying to use the* ***trace*** *global to get information on what methods are called. Unfortunately lots of output that doesn't start or end where I want. What can I do?*

A There are really two questions here: how to use **trace** effectively, and how to use the output. Usuall you w turn tracing on inside a method, then turn it off later on in the code. Unfortunately, you need to more than just set the value of **trace**; you also have to force the interpreter to notice that **trace** h changed. (The PIE Developer Technical Support NewtonScript Q&A on debugging tells you ho to do this.)

```
// to turn tracing on for functions
trace := 'functions;
// force interpreter to notice change in state of trace variable
Apply(func () nil, []);

// to turn tracing off
trace := nil;
Apply(func () nil, []);
```

Once you have the trace output, you should cut and paste it into a text processor. There are thre main bits of information you can get from a trace:

- You can look at how many messages are generated from an apparently simple call. You can **trace** in conjunction with function call timings made using Ticks to see why a particular cal so long. Using the find feature of your text processor, you can jump to the function call you looking at.
- You can look at the values passed in and returned by function calls.
- Perhaps most useful of all, you can use the text processor to strip away all the extraneous information (things like the lines specifying return values — that is, lines that contain the st "=>" as the first non-whitespace entry) so that you're left with the messages sent. Then you sort the messages and get a histogram of the results. This process is easier if you have a tex processor that supports **grep**-like text substitution (regular expressions) and sorts.

Q *I'm using the Newton Toolkit layout editor to organize my data object classes in my application have 20 classes with one layout per object type. To access the objects, I declare each class layou the main application. This gives me the benefits of parent inheritance. Unfortunately, even my t applications are memory hogs. I would expect a time penalty, but why is there such a large spac penalty?*

A The space penalty is much larger than it needs to be. You're using a layout editor to edit your classes so that you can graphically edit the classes' slots. But this has the disadvantage that you have to specify each class as some sort of view class or prototype, perhaps a simple clView. It' the cause of your space problem, because you also carry all the memory and runtime allocation goes with a view. Since your layouts are declared to your base application view, and since the default for a clView is visible, each of your classes is also a full runtime view. That can take a l amount of space on the NewtonScript heap. For a clView, the penalty is roughly 40 bytes, so th an extr 800 by of

NewtonScript heap that you can free.

A better solution is to avoid using the NewtonScript heap for your class (after all, that's one of advantages of prototype inheritance). You can do this in one of two ways:

- If you still want to use a layout editor to edit your class, you can use a user prototype insteа layout. At run time, you'll have access to the data class using the PT_<filename> syntax documented in the *Newton Toolkit User's Guide* (page 4-25). Remember that the user prot will be read-only.
- The other option is to textually define the class. You can do this in your Project Data file, o the Load command to read in a different text file. See the PIE Developer Technical Support NewtonScript Q&A document for more information.

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

Thanks
to our PIE Partners for the questions used in this column, and to jXopher, Bob Ebert, Mike Engber, Kent Sandvik, Jim Schram, and Maurice Sharp for the answers.

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

This code will go on the CD:

```
/* Some code to generate the table used in IsASCIIAlpha4
call func() begin
   s := Array(127, nil);   // ASCII is 7 bit, so only need 127
   for i := 0 to 255 do begin
      local c := chr(i);
      if (c >= $a and c <= $z) or (c >= $A and c <= $Z) then
         s[i] := TRUE;
   end;
   s
end with ();
*/

IsASCIIAlpha1 := func(s)
begin
   local c := Upcase(Clone(s)) ;
   local i ;

   for i := 0 to StrLen(c) - 1 do
      if (StrCompare(SubStr(c, i, 1), "A") < 0) OR
         (StrCompare(SubStr(c, i, 1), "Z") > 0) then
         return nil ;
   true ;
end;

IsASCIIAlpha2 := func(s)
begin
   local i ;
   local c ;

   for i := 0 to StrLen(s) - 1 do
   begin
      // the assignment of the current character to the local c
      // is inlined in the first compare, this is faster.
      if not(((c := s[i]) >= $A AND c <= $Z) OR
         (c >= $a AND c <= $z)) then
         return nil ;
   end;
   true ;
end;
```

I
s
A

```
SCIIAlpha3 := func(s)
begin
   local i ;
   local c := Upcase(Clone(s)) ;

   for i := 0 to StrLen(c) - 1 do
      if (c[i] < $A) OR (c[i] > $Z)  then
         return nil ;
   true ;
end;

// this table was generated using the code above, simple cut
// and paste :-)
constant kAlphaTable := '[NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL,
NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL,
NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL,
NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL,
NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, TRUE, TRUE, TRUE, TRUE,
TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,
TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, NIL, NIL, NIL, NIL, NIL,
NIL, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,
TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,
TRUE, NIL, NIL, NIL, NIL];


IsASCIIAlpha4 := func(s) begin
   try
      for i := 0 to StrLen(s)-1 do
         if not kAlphaTable[ord(s[i])] then
            return NIL;
      TRUE;

   // this code only handles first 127 characters, if the string
   // contains a unicode character, the index will be
   // out of the bounds of the array. Instead of checking
   // the bounds in each loop iteration, use an exception
   // handler. This ads no time to the loop, but a bit of setup
   // time for the exception
   OnException |evt.ex.fr;type.ref.frame| do
      if ord(s[i]) > 127 then
         nil;
      else
```

```
rethrow();
end;

call func()
begin
   local longPass := "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
   local longFail := longPass & " ";

   local timeFunc := func(targetFunc, s1, s2)
      begin
         local t := Ticks() ;
         for i := 1 to 100 do
         begin
            call targetFunc with (s1) ;
            call targetFunc with (s2) ;
         end;
         Ticks() - t ;
      end;

   Print("Long Strings ---------------") ;

   print("IsASCIIAlpha1: " & call timeFunc with (IsASCIIAlpha1, longPass,
longFail)) ;
   print("IsASCIIAlpha2: " & call timeFunc with (IsASCIIAlpha2, longPass,
longFail)) ;
   print("IsASCIIAlpha3: " & call timeFunc with (IsASCIIAlpha3, longPass,
longFail)) ;
   print("IsASCIIAlpha4: " & call timeFunc with (IsASCIIAlpha4, longPass,
longFail)) ;

   Print("\nShort Strings -------------") ;

   print("IsASCIIAlpha1: " &
      call timeFunc with (IsASCIIAlpha1, "a", "a ")) ;
   print("IsASCIIAlpha2: " &
      call timeFunc with (IsASCIIAlpha2, "a", "a ")) ;
   print("IsASCIIAlpha3: " &
      call timeFunc with (IsASCIIAlpha3, "a", "a ")) ;
   print("IsASCIIAlpha4: " &
      call timeFunc with (IsASCIIAlpha4, "a", "a ")) ;
end with () ;
```

"
L
o

```
ng Strings ---------------"
"IsASCIIAlpha1: 5078"
"IsASCIIAlpha2: 2389"
"IsASCIIAlpha3: 1981"
"IsASCIIAlpha4: 1176"
"
Short Strings --------------"
"IsASCIIAlpha1: 180"
"IsASCIIAlpha2: 111"
"IsASCIIAlpha3: 107"
"IsASCIIAlpha4: 79"
#2        NIL
```