ViewFrame Single-Step Debugger

ßeta Documentation version 1 - 10/18/97

Note to testers: this document was originally written over two years ago, during the Newton 2.0 beta period, when the single-stepper first became operational - just before Apple made fundamental changes to how the NewtonScript interpreter works, flushing most of my work down the drain. I've updated it to cover the resurrected form of the stepper, but it's likely that there are some anachronisms still left. If any part of this document doesn't seem to describe the stepper as it currently works, please let me know.

DESCRIPTION:

The VF+Intercept Addition for ViewFrame has been expanded to include a Newton-based, single-step debugging feature for ordinary NewtonScript functions. This feature requires NS Debug Tools to be installed (tested under version 2.2, exact range of compatibility not known): it makes use of undocumented features of the Debug Tools package, so I cannot guarantee future compatibility. Debug Tools runs only on Newton 2.x devices, and therefore the debugger requires 2.x as well.

KNOWN BUGS:

• If you operate the debugger via pen, it will eventually stop accepting any pen input. This seems to be a problem with stroke units piling up in memory and never getting a chance to be cleared. Until this problem is fixed, I recommend using a keyboard for stepping, and using the pen only for display manipulation that can't be done any other way. Note that if this problem occurs, the Newton isn't actually hung. If you have a keyboard connected, you can use it to close the debugger and regain normal operation. Or, if you have an active Inspector connection, you can execute this line to get the same effect:

```
GetView('viewFrontKey):Close();
```

• The Call Trace display doesn't currently work.

• Editing local and stack values doesn't work: a "Not in break loop" exception is thrown. Note that you can still select local and stack values for display, and edit slots within them - that works just fine.

• Stepping into inherited: method calls currently doesn't work (the debugger can't find the method in order to determine what to do with it). However, stepping over such calls works just fine.

LIMITATIONS:

• The debugger only works with ordinary, bytecode-interpreted functions. Calls to native compiled functions, or built-in CFunctions, cannot be stepped into, although their parameters and return value can be observed and modified. TODO: allow stepping into native compiled functions (as long as bytecodes were not suppressed) by faking a call to its bytecode version.

• Problems may occur if you attempt to step into any global function or root view method that is used by the debugger, since this may cause the debugger to recursively trigger one of its own breakpoints. It doesn't look like this will be a serious concern: all of the system routines called by the debugger during the critical period of time are currently CFunctions, which can't be stepped into anyway. Attempting to step into the debugger itself, or to any of the NS Debug Tools routines, is of course completely out of the question.

• The debugger doesn't currently handle recursive functions very well. NS Debug Tools sets breakpoints based on the function object itself, rather than a particular call to the function, so the debugger can't tell which level of recursive call it's currently dealing with. TODO: some hints on

how best to handle a recursive function given these limitations.

• Each instance of the debugger takes a fair amount of memory, and stepping into functions creates additional debugger instances. Debugged functions can easily encounter out-of-memory errors that they wouldn't otherwise. Use the Close-In button whenever possible to avoid piling up debugger instances in memory.

• There may be problems trying to step through drawing scripts (`viewDrawScript`, `viewHiliteScript`, or any other method called via `:DoDrawing()`). If the drawing is being done into a clipped view, the same clipping gets applied to the debugger window, possibly making the debugger unusable (but the clipping doesn't affect pen taps, so you can always blindly close the window to regain control). I think this can be worked around, but there is a more serious potential problem: what happens when the debugger closes, and the views underneath get redrawn? It's easy to get stuck in an infinite loop where the debugger immediately reopens when it is closed. You may be able to close the view where the drawing takes place in the brief interval before the debugger reopens, but it's more likely that a reset will be needed. When installing an intercept on a drawing function, it would be best to give it a conditional expression that tests and sets a global variable, so that the intercept can trigger only once without manually resetting that variable. TODO: add a "once only" checkbox to the Interceptor to automate this process.

• There may be problems trying to debug repetitively called routines, such as viewIdleScripts or functions which add a deferred or delayed call to themselves. There can even be problems with individual deferred or delayed calls, depending on how the calling function is written. In all of these cases, the function may be called at a time that would not normally be possible, and the calling function may not be able to handle this. The debugger creates its own idle time by creative use of :ModalDialog(): a deferred action will typically execute before the next statement of the calling function, rather than waiting for all active functions to exit.

• If the function being stepped calls `GetView()` with `'viewFrontMost`, `'viewFrontKey`, or similar symbols, it will actually get a reference to the debugger itself. This may cause the function to behave differently that it normally would.

OVERVIEW OF USE:

There are three current ways to invoke the debugger, all of which are described in more detail in the existing VF+Intercept docs, or Change History 1b:

• Use the Install Intercept command to install an intercept of type "Single-step".
• Use the Step Into Next Button command (or its key equivalent provided by VF+Keys, which is Ctrl-Cmd-N) to step into a selected method of the next "button" pressed (actually, most anything that hilites when you tap on it).
• Use the `VF:MakeStepper()` method to invoke the debugger under program control.

THE DEBUGGER SCREEN:

The debugger always uses the full Newton screen: this avoids various reentrancy problems that might occur if any of the existing views could be seen or tapped on. The debugger's display consists largely of five scrolling lists. They all work similarly, so they will be described in general here, and in individual detail later.

DEBUGGER LISTS:

<illustration will go here>

Each of the five lists consists of some combination of these features:

• A title at the top left. In some cases this may be a popup allowing selection of different things

which can be shown in this list.
• A content area, consisting of either: a scrolling list of 1-line items (shown in the Simple/Geneva font), or a single word-wrapped text item (shown in the Fancy/New York font) which cannot be scrolled. If a list of items is shown, they can be tapped on (which generally shows more detail about the item), or the pen can be held down on them for one full second (which generally allows them to be edited).
• A description line, at the bottom left.
• Up/down scrolling tabs, at the bottom right. These are shown only if scrolling is possible in the corresponding direction. Scrolling will accellerate for as long as the button is pressed. The universal scroll arrows do not work in the debugger (one of the hazards of modal dialogs).
• A left-pointing arrow tab, at the top right. This is used to back up to a previous display in this list.

EDITING VIEWS:
<illustration will go here>
This is another common element of the debugger, used for editing NewtonScript objects. Generally it is invoked by holding down the pen for one second on an item in one of the lists.
• The title at the top lets you know what object is being edited.
• The entry line will accept an arbitrary NewtonScript expression. Note that its initial contents consist of a brief description of the object's original value: in some cases, it will NOT be a valid expression, and will have to be modified.
• Closing the view, or opening another editing view, aborts without changing the edited object.
• Tapping Change evaluates the entered expression and, if successful, applies the change to the edited object.
• Tapping on the keyboard icon at the lower right should bring up a keyboard, but this isn't working - apparently a limitation of modal dialogs. TODO: if this can't be worked around, put an embedded keyboard in the editing view.
• The NIL and TRUE buttons replace the contents of the entry line with the specified text. They could be used, for example, to change the value at the top of the stack so that a branch will be taken that otherwise wouldn't (or vice versa).
• The + and - buttons increment or decrement the value in the entry line if it is numeric, and have no effect otherwise. They might be handy to adjust the number of times a loop is going to execute, or the number of elements in an array that is about to be created.

THE LISTS:
The five lists that the debugger shows can be divided in two: the four minor lists, which are in a 2x2 grid at the top of the screen, and the main list at the bottom. Counterclockwise starting from the top left, the minor lists are:

• £iterals. Shows the contents of the function's literals array, which contains various objects (strings, symbols, large integer constants, etc.) that are used by the function but cannot be embedded in the function code itself. Throughout the debugger, these items are referred to as £*n*, where *n* is the array index. This use of the "£" character is intended to help distinguish literal references from other expressions that might look similar, and does not correspond in any way to normal NewtonScript syntax. Tapping on any item in this list will display it in greater detail in the Work Area, which is described later. Holding the pen down on any item will attempt to open an editing view for it, but this usually fails since the literals array is normally read-only.

• ÅrgFrame. Shows the current contents of the function's argFrame, or the equivalent area of the evaluation stack for new-style functions. The argFrame holds the function's parameters and declared local variables. Throughout the debugger, these items are referred to as Å*n*, where *n* is the item's position within the argFrame, followed by the actual name of the item if it can be

determined. Note that the displayed items start with Å3: Å0 thru Å2 correspond to internal variables (_nextArgFrame, _Parent, and _Implementor) that shouldn't be referred to by normal code. The first items shown are the function's parameters, if any, followed by its local variables. A dividing line in the list separates these two sections. The number of parameters is also indicated below this list. Tapping on any item will display it in the Work Area. Holding the pen down on any item will open an editing view for it, but saving the changes does not currently work.

• §tack. Shows the current contents of the evaluation stack. All NewtonScript bytecode operations work by pushing and/or popping items here. Throughout the debugger, these items are referred to as §*n*, where *n* is the position on the stack, with §0 being the current top of stack (which is actually displayed at the bottom of the list). This upside-down representation is conventional for stacks, and does have some display advantages: a sequence of pushed items will appear on the stack in the same order as the code steps which pushed them. Note that every time an item is pushed or popped, the position numbers of all other items will change. This is the nature of stacks: items are referred to relative to the current top of stack, rather than their absolute offset from the bottom of the stack. This means that any §*n* stack references you see in a code step are only meaningful when that step is about to execute: at any other time, the top of stack is likely to be at a different position. Tapping on any item will display it in the Work Area. Holding the pen down on any item will open an editing view for it, but saving changes currently doesn't work. This will give you great power over the execution of the function: you can modify the parameters to a system call after they've been pushed but before the call has been made, or change the result of an expression that controls whether or not a branch will be taken.

• Work Area. Gives detailed descriptions of objects that are tapped on anywhere in the debugger. If a frame or array is displayed here, its elements can be tapped on to display them in the Work Area as well. Think of this as a miniature version of ViewFrame. After an object is displayed here, the title shows the object or slot name currently being viewed, and can be tapped on to pop up a list of the objects in the path leading up to the current object. Selecting any object from this popup returns you to it. If there's more than one object in the path, a Back tab will appear that allows you to back up through the path. A brief description of the object appears below the list: it includes a "@*nnn*" reference if the object is a magic pointer, "RO" if the object is read-only, its PrimClass (and class if different), and its length (in slots, elements, or bytes, as appropriate). Holding the pen down on a frame or array element will open an editing view for it, assuming that the enclosing object isn't read-only.

THE MAIN LIST:

The wider list at the bottom of the screen shows one of three displays, which can be selected by tapping on its title:

• Trace history. This is the default display, and is automatically switched to whenever a step is taken. The underlined item is the step that is about to be taken: this is what will be executed when the Step button is tapped. Above this are the steps which have recently been taken: you can scroll back through up to 90 or so items. Of course, this section will be empty when the debugger first opens. Below the current step is a prediction of the next few steps that are likely to be taken. Unconditional branches are followed, and the prediction ends if a return statement is encountered. Conditional branches are currently not predicted very well: they are always assumed not to be taken, even if it is obvious that they will be. Tapping on any step, past, present, or future, will display that step as the sole contents of the main list: this is useful if the step is too wide to fit on one line. A Back tab will appear to return to the normal display. Holding down the pen on a step brings up a list of options concerning it, which currently contains only one item: "Run until step *nn*". This will cause execution to automatically step (faster than the debugger normally does, but much slower than

normal execution speed) until the selected step is reached. Running also stops if an exception handler is entered, or the function is about to return. The option popup may eventually include a "Skip to" option, which would allow you to bypass the normal flow of the routine (although at considerable risk of crashing).

• Function listing. This lists all of the steps in the current function, just like the Trace History display but in sequential order rather than in the order that they have or will be executed. Tapping or holding down the pen on an item has exactly the same effects as in the Trace History display. The "Run until" option is probably more useful from this display, since the exact step you want to run to may not appear in the history list.

• Call trace. NOT CURRENTLY WORKING, IGNORE THIS SECTION! This lists the sequence of function calls that led up to the current function, similar to the StackTrace() or QuickStackTrace() functions in the Inspector. Each listed function occupies 6 lines of the list, with a dividing line between each function. The topmost function is the one the debugger is currently viewing; the second is the function that called that one, and so on down the list to the function that originally responded to a system event. If the debugger was invoked via the Interceptor, the second function in the list will always be the installed intercept routine, and should generally be ignored. Note that both bytecode and native functions are listed here, although with slightly less detail in the latter case. Tapping on a line displays the data from that line in the Work Area. Holding down the pen on a line does nothing: none of the data here is meaningfully editable. The information shown for each function is:
1. LEVEL *n*. This is a header for the block of information, and may include a name for the function if one can be determined (there's lots of room for improvement here).
2. Receiver. The frame to which a message was sent: this is the value of 'self' in the corresponding function.
3. Implementor. The frame in which the function was actually found: this is typically a view template, while the receiver is typically the corresponding view. This is generally not a useful piece of data, but it's important for the system to maintain: if the function makes an inherited: method call, the implementor frame is the starting point for the _proto chain search for the inherited method.
4. ContextFrame. For old-style functions that have an actual frame as their argFrame, this is a reference to their working copy of the argFrame: tapping it will allow you to see the function's actual parameters and local variables. For new-style functions that have an array as their argFrame (or no argFrame at all), this is just an integer giving a stack offset to their parameters and local variables, and isn't really useful.
5. CodeBlock. A reference to the actual function object being executed at this level. You can tell what kind of function it is by the object description: "{CodeBlock:#5}" indicates an old-style bytecode function, "{:#5}" indicates a new-style function, and "{:#3}" indicates a built-in CFunction.
6. ProgramCounter. This is the offset within a bytecode function's instructions object that is currently being executed. For CFunctions, this is always -1.

DEBUGGER CONTROLS:

STEP:

Key equivalent: return or space.

Tapping this button executes the underlined step shown in the Trace History. If the step was a return statement, the current instance of the debugger is closed; otherwise, the ÅrgFrame, §tack, and Trace History lists are updated with the function's state after the step has been taken (the £iterals list isn't updated, since it cannot change; the Work Area isn't updated, although perhaps it should be). If repeated taps of the Step button hilite it but have no other effect, this indicates that the current function has been exited without the debugger being aware of it (probably because it triggered an exception that it didn't catch). When this happens, simply close the debugger. DO

NOT attempt to edit any objects from the ÅrgFrame or §tack lists under these circumstances: the lists may still appear to contain valid objects, but the underlying data structures died along with the function, and attempting to write to them will most likely crash the Newton.

STEP IN:
Key equivalent: letter 'I'.
This button is only available if the step about to be taken involves a function call of some sort (global function, user function, method call, or inherited:method call). It will attempt to create a new instance of the debugger for stepping through the called function: the current instance will remain open, and continue to track the current function after the called function returns to it. The operation may fail due to several reasons:
• Some specific functions may be disallowed from stepping, to avoid reentrancy problems. Currently, BreakLoop() is the only disallowed function.
• "No function to step into" indicates that the called function evaluated to NIL. This indicates either a bug in the function, or a conditional call (*obj*:?*method* or inherited:?*method*) to a nonexistant routine.
• "Not a bytecode function" indicates that the call was to a native routine that the debugger can't handle (or to a non-function object, indicating a bug in the function).

If a Step In operation failed, the function being debugged may actually be in the middle of a step. This is because exact determination of the function that will be called may require partial execution of the step that includes the function call. You may end up seeing items on the evaluation stack that aren't referenced in the trace history, such as the name of a method to be called. Just do a normal Step if this happens: it will pick up where the failed Step In left off, and the stack will be back in sync with the trace history.

TODO: global functions that specify another function to be called (Perform, Apply, AddDeferredAction, etc.) should eventually support Step In, opening a debugger instance for the specified function rather than for the global function itself.

CLOSE IN:
Key equivalent: 'C'.
This button is available in the same circumstances as the Step In button, and performs much the same actions. The difference is that this button closes the current debugger instance before stepping in. When the called function returns, it will continue the current function at full speed, and the Newton will return to normal operation (unless there was an earlier function from which a Step In was done). The reason for having this button is to prevent build-up of debugger instances in memory when deeply stepping into functions. You should use it instead of Step In if:
• there is no more meaningful code to execute in the current function, or
• you know that the code of interest to you is contained entirely in the called function.

CLOSE BOX:
Key equivalent: Cmd-W or Cmd-period.
Closing the debugger window will allow the current function to finish executing at full speed. If a previous instance of the debugger is still open, it will pick up where it left off as soon as the current function returns to its caller. If all debugger instances have been closed, the Newton should be back to its original state, with no lingering traces of the debugger (although I'm not prepared to place much of a bet on that). Closing the debugger may also be necessary in cases where the current function was aborted without the debugger being aware of it.

LIST SIZE ADJUSTMENTS:
There are two buttons at the top left corner of the screen for adjusting the relative sizes of the

debugger's lists. One button adds one line to each of the four minor lists, at the expense of two lines of the main list; the other button does just the opposite. The minor lists can be shrunk down to a single line each, but the main list must always be at least 4 lines high.

INSPECTOR OUTPUT:
Tapping the title of the main list produces a popup that includes, in addition to the three display options mentioned before, a "Listing >Inspector" item. This dumps to the Inspector the entire list of steps comprising the current function, as shown in the Function Listing display. Also, individual steps are sent to the Inspector as the function is stepped through.

UNDERSTANDING DEBUGGER OUTPUT:

In order to effectively use the single-step debugger, you'll need to have at least a basic understanding of how the Newton's bytecode interpreter works. The bytecode language bears little resemblance to NewtonScript code: it isn't merely a tokenized form of NewtonScript. If you use NS Debug Tools directly, you step through the function's bytecode operations one by one: individually, the operations don't do very much, and it's hard to see the big picture when you're dealing with such low-level elements. My goal was to produce a debugger that collected multiple individual operations into logical steps, that could be more easily understood. In fact, the debugger could collect an entire NewtonScript statement into a single step, but this would be a bad idea in many cases. Consider this statement:

```
A(B(C));
```

If this was executed as a single step, you wouldn't have any idea what value B() returned. If something was going wrong in this statement, you'd have no way of telling whether B() returned the wrong value, or if A() incorrectly handled that value. You might also wish to examine or modify the value of C before it is passed to B(). Therefore, the debugger would break this statement into three separate steps: retrieving the value of C, passing that value to B(), and passing its result to A(). The debugger's steps may be anything from a single bytecode operation, to an entire NewtonScript statement. The general rule is that a single step will either produce values, or use values: no step will produce an interesting value and then use it before you have a chance to see it. "Interesting" is roughly defined as being non-obvious: for example, the value of a local variable isn't considered interesting, since you can always see it in one of the debugger's lists.

In NewtonScript, as is the case with most programming languages, there are two basic forms of expressions:
• Infix expressions, such as the basic arithmetic operations, in which the operation appears between the values it is to operate on.
• Prefix expressions, such as function calls, in which the operation appears ahead of the values it is to operate on.
These expression forms make sense to human beings who are used to common mathematical notation. However, to a computer attempting to evaluate expressions, they're absolutely lousy: what's the point of specifying the operation before the values it is to operate on are known? Therefore, the bytecode language uses a postfix representation, in which operations don't appear until after all their needed values are specified. In this respect, the bytecode language is much more closely related to postfix programming languages (such as FORTH and PostScript) than to NewtonScript.

The bytecode interpreter maintains a stack for holding values that have not yet been used. All sources of values (constants, local variables, etc). push their value onto this stack. All operations pull the values they need from this stack, then push their result (if any). In fact, there are only two bytecode instructions that don't affect the evaluation stack: unconditional branches, which are used to construct various kinds of loops, and the pop-handlers instruction which ends a try block.

Example: the expression 2+3*4 would be represented as the following operations:

```
push 2
push 3
push 4
multiply
add
```

Some simple constants can be represented directly in the bytecode language. Three other sources of values can be referenced:

• The function's literals array, which contains constants that can't be directly represented, and symbols used for various purposes.
• The argFrame, which holds the parameters and local variables specific to a particular instance of the function.
• Values previously pushed on the evaluation stack.

The debugger displays references to these three sources with the special symbols £, Å, and §, followed by an integer giving a position within the source (argFrame references may also give the actual name of the parameter or variable if it can be determined). These symbols have no particular meaning: I'm using them simply to make these references compact and unambiguous.

Operations that take values from the stack may incorporate the operations that pushed those values on the stack, to the extent that those intermediate values were "uninteresting" (such as simple constants, or argFrame values). This produces debugger steps that correspond to multiple bytecode instructions. "Interesting" values are referred to with a §*n* reference, and the operations that pushed them are left as separate steps. Examples:

Example 1:

```
NewtonScript: 2+3
Bytecodes: push 2 , push 3 , add
Debugger step: (2 + 3)
```

Example 2:

```
NewtonScript: A()+3
Bytecodes: call A() , push 3 , add
Debugger steps: A() , (§0 + 3)
```

Example 3:

```
NewtonScript: 2+B()
Bytecodes: push 2 , call B() , add
Debugger steps: 2 , B() , (§1 + §0)
```

In example 3, notice that the left side of the addition was a simple constant, but it couldn't be combined into the addition operation due to the intervening function call to B(). Only contiguous bytecode instructions can be combined into a single step.

ELEMENTS OF DEBUGGER STEPS

Each debugger step is labelled with a number, which is the offset within the function's instructions object of the step's first instruction. Step numbers are generally not contiguous: each bytecode instruction may consist of 1 or 3 bytes, and there may be any number of instructions combined into a single step. In the function listing, step numbers will always appear in increasing order, but in the actual trace history they may skip around due to the effects of loops and conditional expressions.

After the step number, there is a ">" or "»". Presence of a "»" indicates that the step is the destination

of a branch instruction elsewhere: this is a visual aid to help you find where branches go.

Next is the actual operation performed by the step. To the extent possible, this appears in standard NewtonScript syntax. However, there may be £*n* and Å*n* references to clarify exactly where values are coming from, and §*n* references to tie together the pieces of an expression that couldn't be shown as a single step. Also, operations related to the flow of control are rather different in the bytecode language than in NewtonScript, and appear enclosed in angle brackets. If the step is too wide to fit in the trace listing, remember that you can tap on it to display the whole step.

Following is a list of all of the operations that can appear in debugger steps, along with their NewtonScript equivalents, rules for combining with other operations, etc. Each will have an "Apple name:" note giving the official name of the operation, as you'd see it displayed when using the NS Debug Tools directly.

constant
Simple constants that can be represented directly in the bytecode language appear in plain NewtonScript form: these include NIL, TRUE, integers up to 8191, many character constants, and magic pointers (@*nnn*). The constant value is simply pushed on the stack. These are always considered as uninteresting values, and may be combined into the operation that uses the value from the stack. Apple name: push-constant.

(£*n*)*constant*
Constants outside of the range that the previous operation can represent are stored in the function's literals array, and referred to by their array index. This also includes string constants, symbols, frame maps, and many other constant values required by the function. Frames and arrays are shown in an abbreviated form: for example, "{#4}" would indicate a frame with four slots. To see exactly what those slots are, tap on the object in the £iterals list to display it in the Work Area. Apple name: push.

'*symbol*
Symbols are just a special case of the previous operation, but are handled differently because they are so commonly used. Unlike literals in general, symbols are considered uninteresting and may be combined into the operation that uses them.

'*sym1*.*sym2*...
Path expressions consisting entirely of symbols are also a specially handled case of literal constants. Expressions like W.X.Y.Z are handled by looking up a single path expression ('X.Y.Z) in the object W, rather than doing separate slot lookups for X, then Y, then Z.

value1, *value2*, ...
If multiple uninteresting values are pushed in a row, they may be treated as a single step. After all, there isn't anything useful you could do between the individual pushes that you couldn't do later after all of the values were pushed. This will only be done for a series of values that are parameters to a single function call.

;
All NewtonScript statements are expressions, that return a value. However, statements other than the last one in the function don't do anything with their return value, so it is simply popped off the stack. This corresponds almost exactly with the required placement of semicolons in NewtonScript, so I use a semicolon to represent the pop operation. The semicolon may be stuck onto the front of the following operation, thus avoiding having a step that does nothing but a pop. The few

operations which don't push a result on the stack also end with semicolons, for consistency. Apple name: pop.

<dup??>
This operation duplicates the value at the top of the stack. It does not appear to be used at all: if you encounter one of these, please let me know where you found it. Apple name: dup.

self
Pushes the value of 'self' (the receiver of the current message) onto the stack. This value may be combined into the operation that uses it. If 'self' is used as the target of a message send, then it is omitted completely: this produces the common abbreviated form of ":*method*()" rather than "self:*method*()". Apple name: push-self.

get £*n*: *symbol*
Finds a slot of the given name and pushes its contents onto the stack. The slot is searched for in the function's context using full inheritance. This is generated when you write a variable name that isn't one of the declared parameters or locals of the function: if you see this operation unexpectedly, you probably forgot a local declaration. This operation's value is always considered interesting, since the debugger can't show you what the value is until the operation actually executes: therefore, it will never be combined with any other operations. Apple name: find-var.

(Å*n*)*symbol*
Å*n*
The contents of the indicated argFrame element (parameter or local variable) is pushed onto the stack. The second form is used if the name of the item cannot be determined: this is commonly true of built-in functions, and functions in packages that weren't built in debug mode. This operation is considered uninteresting (since the values of argFrame items are always visible in the debugger), and may be combined with other operations to form more complex expressions. Apple name: get-var.

{*slot1*: *value1*, *slot2*: *value2*, ...}
A frame is constructed and pushed on the stack. The slot names come from a frame map object from the function's literals array. The values are pushed onto the stack prior to this operation. To the extent possible, the operations that generated the values are combined into this operation, rather than being left as separate steps. This means that some of the values, starting from the right end, may appear as actual constants or expressions in this step. The rest of the values will appear as §*n* stack references. Apple name: make-frame.

[*value1*, *value2*, ...]
[*symbol*: *value1*, *value2*, ...]
An array is constructed and pushed on the stack. The second form is used if the array is given a class other than the default of 'Array. The values are pushed onto the stack prior to this operation: to the extent possible, the operations that generated them are combined into this operation, rather than being left as separate steps. Therefore, some of the values (starting at the right end of the array) may appear as actual constants or expressions. The rest of the values are shown as stack references. If there are more than two such values, they are shown as a range (such as §3...§0) rather than a list of individual references. Apple name: make-array.

[<§0 elements>]
This is a special case of the array constructor operation, apparently used only for allocating the result array for a foreach..collect loop. An array with a specified number of NIL elements is created,

rather than specifying the initial values of each element.

frame.*slot*
frame.(*expr*)
§0.*slot*
§0.(*expr*)
§1.(§0)
A slot is looked up in a specified frame, using _proto inheritance only. The different formats depend on the extent to which the operations that pushed the slot and frame can be combined into the current operation. If the slot can be combined in, the format also depends on whether the slot is a simple symbol or path expression, or something more complicated. In the first two formats, the resulting operation is eligible to be combined into other operations. Apple name: get-path.

frame.*slot* := *value*
frame.(*expr*) := *value*
§0.*slot* := *value*
§0.(*expr*) := *value*
§1.(§0) := *value*
§2.(§1) := §0
A value is stored into a slot of a specified frame. As with the frame accessor operation above, the format used depends on the extent to which other operations can be combined into this one, and on the type of expression used for the slot name. There are two variations to this operation: one (formats are as shown above) leaves a copy of the value on the stack, the other (formats have a semicolon added at the end) leaves nothing on the stack. Apple name: set-path.

(Å*n*)*symbol* := *value*;
Å*n* := *value*;
A value is stored into one of the argFrame elements (parameters or local variables). The second form is used if the name of the element cannot be determined. The operation that generated the value may be combined into this operation, otherwise the value will be shown as §0. This operation never leaves a value on the stack, which is indicated by the semicolon at the end. Apple name: set-var.

set £*n*: *symbol* := *value*
A value is stored into a slot somewhere in the function's context. This is generated when you assign a value to a name that isn't one of the function's declared parameters or local variables. The operation that generated the value may be combined into this operation, otherwise the value will be shown as §0. Apple name: set-find-var.

- FUNCTION CALLS -
Function calls are never combined with the operations that pushed their parameters, nor with the operation that uses their return value (but they may still be composed of more than one operation, such as the one that pushes the name of the function to be called). This gives you a chance to examine and modify all aspects of the call. Since the parameters are always pushed by separate steps, the parameter list in the step that actually does the call will simply consist of a list of stack references. Some possible forms are:

()	*no parameters*
(§0)	*one parameter*
(§1, §0)	*two parameters*
(§3...§0)	*four parameters*

name(*params*)
§0(*params*)
Call to global function. The second form should never be seen: it would indicate that the name of the function wasn't a literal. Apple name: call.

call *value* with (*params*)
call §0 with (*params*)
Call to user function. The second form is used if the operation that pushed the function object couldn't be combined with the current step. Apple name: invoke.

func(<*value*>)
Prepares a local nested function for use with the call..with operation above, by linking its lexical environment to that of the calling function. This is only done for functions actually nested inside the calling function: references to functions elsewhere are not processed by this operation, and therefore can't access the calling function's context. Apple name: set-lex-scope.

receiver:*name*(*params*)
:*name*(*params*)
§0:*name*(*params*)
§1:§0(*params*)
Message send. The various forms are used depending on the extent to which the calling information about the send can be combined into the current step. The second form is used if the receiver is 'self'. The fourth form should never occur. Apple name: send.

receiver:?*name*(*params*)
:?*name*(*params*)
§0:?*name*(*params*)
§1:?§0(*params*)
Conditional message send. The different forms have the same meanings as described for the ordinary message send, above. Apple name: send-if-defined.

inherited:*name*(*params*)
inherited:§0(*params*)
Inherited message send. The second form should never occur. Apple name: resend.

inherited:?*name*(*params*)
inherited:?§0(*params*)
Conditional inherited message send. Apple name: resend-if-defined.

- FREQUENT FUNCTIONS -
There are 25 commonly used functions which have corresponding bytecode instructions, thus avoiding the overhead of a global function call. Unlike ordinary function calls, the operations that pushed their parameters may be combined into these operations. Also, if all parameters were combined into the operation (in other words, no §*n* stack references are used), the operation itself is eligible to be combined into other operations. All infix operations are enclosed in parentheses, to avoid any possible ambiguity about the order in which operations are performed. Apple name: freq-func; the number and name of each individual instruction is shown after the descriptions below.

(*op1* + *op2*)	//0, +
(*op1* - *op2*)	//1, -
op1[*op2*]	//2, aref

(*op1*[*op2*] := *op3*) //3, setAref
(*op1* = *op2*) //4, =
not(*op1*) //5, not
(*op1* <> *op2*) //6, <>
(*op1* * *op2*) //7, *
(*op1* / *op2*) //8, /
(*op1* div *op2*) //9, div
(*op1* < *op2*) //10, <
(*op1* > *op2*) //11, >
(*op1* >= *op2*) //12, >=
(*op1* <= *op2*) //13, <=
BAnd(*op1*, *op2*) //14, band
BOr(*op1*, *op2*) //15, bor
BNot(*op1*, *op2*) //16, bnot
Note: BNot actually takes only one parameter, therefore this freq-func is never used.
-special- //17, newiterator
This freq-func is handled specially: see <foreach in...> in the Looping Constructs section.
length(*op1*) //18, length
clone(*op1*) //19, clone
SetClass(*op1*, *op2*) //20, setClass
AddArraySlot(*op1*, *op2*) //21, addArraySlot
stringer(*op1*) //22, stringer
HasPath(*op1*, *op2*) //23, hasPath
Note: "HasPath(X, 'Y)" is how the expression "X.Y exists" is implemented.
ClassOf(*op1*) //24, ClassOf

- PROGRAM FLOW -
<return §0>
Returns from the function, using the value on the top of the stack as the return value. A return with no value specified pushes NIL first, since a value is always returned. This operation is never combined with the operation that pushed the value onto the stack: this guarantees you a chance to modify the value on the stack before returning. Stepping over this operation automatically closes the debugger. Apple name: return.

<goto *step*>
Unconditionally branches to another step. Backward branches are used to implement the NewtonScript 'loop' keyword. Forward branches are used to skip over an 'else' clause that isn't executed. Apple name: branch.

<if §0 goto *step*>
Branches to another step if the value on the top of the stack has a non-NIL value. The value is popped from the stack. Backward branches are found in 'while' loops. Apple name: branch-t.

<if not §0 goto *step*>
Branches to another step if the value on the top of the stack is NIL. The value is popped from the stack. Backward branches are found in repeat..until loops. Forward branches are used in 'if' statements to skip to the 'else' clause (or to the end of the statement) if the condition is false. Apple name: branch-f.

- LOOPING CONSTRUCTS -
Loop, while, and repeat..until constructs are built out of the branch instructions above: no special

bytecode instructions are needed to support them.

For loops make use of two compiler-generated local variables, and a couple of special instructions. If variable names are present in the function, the two special variables can be recognized as being the name of the loop variable with a suffix added. *name*|limit holds the ending value of the loop, and *name*|incr holds the loop's increment value, or the constant 1 if the loop didn't have a 'by' clause.

<loop increment (Å*n*)*name*>
<loop increment Å*n*>
The loop control variable is incremented by the value of the *name*|incr variable pushed in the previous step. The second form is used if the variable names in the current function cannot be determined. The increment amount is left on the stack, and the new value of the loop variable is pushed onto the stack, for use in testing whether the loop is done. Apple name: incr-var.

<if loop not done goto *step*>
When this operation executes, the top three items on the stack will be: §2 - the loop increment; §1 - the current value of the loop variable; and §0 - the loop limit value. These are popped, and examined to see if the for loop has reached its limit. If not, execution branches back to the specified step. Apple name: branch-if-loop-not-done.

Foreach loops use one or two special variables and a few operations. The variable names are a concatenation of the loop's slot name variable (if present), its slot value variable, and a suffix. *SlotNameSlotValue*|iter references an iterator object that keeps track of the progress of the loop. It is currently an array of 7 elements. Element 0 is always the value of the slot name variable, and element 1 is the slot value variable. Other elements hold the object being iterated over, a flag indicating whether the 'deeply' option is being used, and other internal values. In foreach..collect loops, there is also a *SlotNameSlotValue*|result variable which holds the array of items being collected.

<foreach in *value*>
<foreach deeply in *value*>
An iterator object is created for the specified frame or array. Which of the two formats is shown depends on the value of a second parameter to this operation: NIL for a normal loop, TRUE for the 'deeply' option. The operation that generated *value* will be combined into this operation if possible, otherwise §0 will be shown. This operation will in turn be combined into the next operation, which should be an assignment to the *name*|iter variable. The next step will be an unconditional branch down to the bottom of the loop, where its exit condition is tested: this allows the loop to exit immediately if an empty frame or array is given. Apple name: freq-func 17 (newiterator).

<advance foreach (Å*n*)*name*|iter>
<advance foreach Å*n*>
The specified iterator object is advanced to point to the next element of its frame or array. Apple name: iter-next.

<is foreach (Å*n*)*name*|iter done?>
<is foreach Å*n* done?>
TRUE or NIL is pushed on the stack, depending on whether the specified iterator has reached the end of its frame or array. This will be followed by a <if not §0 goto...> that branches back to the top of the loop. Apple name: iter-done.

- EXCEPTION HANDLING -

<begin try block, handlers at *step1*, *step2*, ...>

When a try block is entered, this operation transfers information from the evaluation stack to the exception handler stack. For each exception handler associated with this try block, two items will be pushed onto the stack: the exception symbol, and the instruction offset of the start of the handler. Normally, all of these pushes are combined into a single step. The location of each handler is shown in this step, and the corresponding exception symbols are attached to the handlers themselves as a comment. Apple name: new-handlers.

<end try block>

When a try block is exited, its handler information must be popped off of the exception handler stack, and that's what this operation does. Note that there are two ways to exit a try block: successful completion of the block itself, or leaving via one of the exception handlers, and you'll find this operation in both places. Apple name: pop-handlers.

SUMMARY:

To gain familiarity with the single-step debugger's display of functions, try stepping through known, simple functions. ViewFrame Editor is ideal for this since it can now directly call the debugger. Start with straight sequential code, then try conditionals, AND/OR expressions, and simple loops. Work your way up to 'for' loops, and finally 'foreach'. When you fully understand how the Newton executes a foreach..collect loop (the most complex construct in NewtonScript), you can truly consider yourself to be a Master of Bytecodes.

Jason Harper