NEWTONSCRIPT PROGRAMMING LANGUAGE

Introduction

This document addresses NewtonScript Programming Language issues that are not available in the currently printed documentation . Please note that this information is subject to change as the Newton technology and development environment evolve.

TABLE OF CONTENTS:

NewtonScript Q&As

- Only 7-Bit ASCII In NewtonScript Symbols (1/26/94) (Obsoleted by NS Guide 1.0 Final Documentation)
- Nested IF Statements, Constant Condition Problems (6/9/94)
- NEW: How to Avoid _parent Problems (6/28/94)

NEW: NewtonScript Object Sizes (6/30/94)

Generic

The Newtonscript Objects are the objects that either reside in the read-write memory, pseudo ROM memory or inside the package or ROM. These objects are aligned to 8-byte boundaries. This alignment causes a very small amount of fragmentation (usually under 2%) so this issue could be ignored.

Immediates

The Newton Object System has four built-in primitive classes that describe an object's basic type: Immediates, binaries, arrays and frames. In the case of an Immediates (integers, character, boolean and so on) we are dealing with a 30+2 bit object.

In the case of binaries, arrays and frames we are also dealing with objects containing a so called Object Header.

Object Header

Every object has a 12-byte header that contains information concerning size, flags, class, lock count and so on. This information is implementation specific.

Strings

A string object contains a 12 byte header plus the Unicode strings plus a null termination character. Note that Unicode characters are two-byte values. Here's an example:

```
"Hello World!"
```

This string contains 12 characters, in other words it has 24 bytes. In addition we have a null termination character (24 + 2 bytes) and an object header (24 + 2 + 12 bytes), all in all the object is 38 bytes big. Note that we have not taken into account any possible savings if the string was compressed (using the NTK compression flags).

Binary Objects

A binary object contains a 12 byte header plus the size of the actual data.

Array Objects

Array objects have an object header (12 bytes) and additional four bytes per element. In

addition you need to calculate the amount of data stored in the arrays as well (references) if you want to calculate the total amount.

Here's an example:

```
["Hello World!", "foo"]
```

We have a header (12 bytes) plus four bytes per element (12 + (2 * 4) bytes). In addition we have string objects that we refer to (12 + (2*4) + 38 + 20 bytes), all in all 78 bytes. Again we have not taken into account savings concerning compression. Note that the string objects could be referred by other arrays and frames as well, so if we want to take this into account we should make sure the string is counted only once per package.

Frame Objects

We have two kinds of frames, frames that don't have a shared map object, and frames who do have a shared map object. We take the simple case first (no shared map object).

An array object map is an array of symbol pointers and one additional slot. The actual frame is two arrays, one contains the slot names, and the other contains the actual slot entry.

The map is 16 bytes, if we add the object header to this the basic header size of a frame is 28 bytes. If we want to add the size taken by slot entries we multiply N slots with 8 (two array entries). Here's an example:

```
{Slot1: 42, Slot2: "hello"}
```

We have a header of 28 bytes, in addition we have two slots (28 + (2 * 8)), all in all 48 bytes. Once again we didn't calculate the actual slot entry objects.

In the case of a frame with a shared map multiple similar frames (look the same) could share the one and only map. This will save space, reducing the marginal overhead per frame to the same as an array with the same amount of slots. In addition we need to take into account the amortized map size that multiple frames share. In other words the magic formula this time is 12 bytes for the header plus 4 times the amount of slots plus the amortized map size.

Here's an example of a frame with a shared map:

```
{Slot1: 42, Slot2: "hello"}
```

We have a header of 12 bytes, in addition we have two slots (2 * 4), and additional 16 bytes for the size of a map with no slots, all in all 36 bytes. We should also take into account the shared map, in the worst case there's only one of these frame objects, in other words we have additional 16 bytes.

When do we create objects with a shared map?

1. When we clone the system will make sure that the cloned object uses the same map. A trick to make use of this is to create a common template frame, and when we need to create additional frames we clone this template frame over and over.

2. The system will make sure that the frame uses a shared map if the frame is created by a a

frame constructor expression in most cases a function that returns a frame. This is the reason we use RelBounds when we create the viewBounds frame, in other words there's just one single viewBounds map in the system, and it's stored inside the ROM.

Note again that these figures are for objects in their runtime state, ready for fast access. Objects in transit or in storage (packages) are compressed into smaller stream formats. These figures are neither true for flattened objects that are sent over a comms endpoint, neither true for objects stored in soups.

Symbols

Asymbol is a frame with two slots, one pointing to a string containing the name, the other pointing to the next symbol in an internal hash table. Symbols share one map, so each symbol occupies 12+2*4 = 20 (round to 24) bytes for the frame and 12+length (rounded) bytes for the name, for a total of 36+length bytes.

A symbol is a binary object that contains a four-byte hash value and the name is a null terminated ASCII character . Each symbol is 12 (header) + 4 (hash value_) + length of object + 1 bytes (null termination ASCII char).

PLAY CATCH - NEWTONSCRIPT EXCEPTIONS

(9/15/93) (Obsoleted by NS Guide 1.0 final docs)

Introduction

The NewtonScript exception system provides a structured way for a module to report a failure to another module. It communicates the type of the failure, and allows (optionally) transmission of data that may explain the reason for the failure. The NewtonScript implemention uses the exception system to report errors (for example, errors in an application, failures due to lack of resources, or internal errors). Developers can use the exception system to communicate errors between different modules in an application.

Basic Syntax

The Exception handling is based on a `try block`, and if an exception is thrown from somewhere, the code should catch this with a special `onException` block.

Try Statements

The general format of the try statement is:

```
try
      <statement list>
onexception <exception symbol> do
      <statement>
onexception <exception symbol> do
      <statement>
...
```

An example of this might be:

```
try
begin
      // create entries and store them to soups
end
onexception |ext.ex.store.err| do
begin
      // problems with the store, most likely not enough memory
      // do something
end
```

The <exception symbol>s must be single part exception symbols. Details about exception symbols — that is, what multiple parts mean — are discussed below.

One or more onexception clauses are allowed. The try statement executes <statement list>. If no exceptions are thrown in the process, then the value of the try statement is the value of the last statement in <statement list> and the onexception clauses are never executed.

If during execution of <statement list> an exception is thrown, then the execution of <statement list> stops and control is transferred to one of the onexception clauses. When an exception, X, is raised, the onexception clauses of the try statement are examined in order. The first clause whose <exception symbol> is a prefix of any of X's parts is executed and its <statement> value becomes the value of the try statement.

If no onexception clause matches the exception, then the exception is passed to the next enclosing try statement for processing. In a Newton application, the exception will ultimately be handled by the view system (putting up an error dialog box) if your application doesn't catch it.

NOTE: It is important that the search for a matching onexception clause uses dynamic scoping, not lexical scoping.

Since all exceptions have an evt.ex part, an exception clause with evt.ex as its symbol will catch any exception. (See the following section for a description of "parts".) If your try statement has an evt.ex clause, it should be last, since onexception clauses occuring after it will never be executed. In general, you should order your onexception clauses from most specific to least specific.

Exception Symbols

Exceptions have names which are symbols. These symbols have a particular format which must be adhered to. An exception name consists of one or more parts separated by semi-colons. Each part is a structured name beginning with evt.ex.

A few facts about exception symbols:

- They can have as many as 127 characters.
- They can contain periods, so the symbols must be enclosed in vertical bars (|'s)
- They can have multiple parts, separated by semi-colons.
- They must have a part starting with evt.ex.

The simplest possible exception symbol is |evt.ex|. An example of an exception symbol with two parts would be '|evt.ex;type.ref|. Some more examples: '|evt.ex.div0|, '|evt.ex.fr.intrp;type.ref.frame|.

Exception Frames

Associated with every exception is an exception frame. When handling an exception you can get this frame using the global function, CurrentException(). An exception frame always has a name slot which contains the exception symbol. It will contain one other slot whose name and contents depend on the type of exception as follows: (this info is summarized on the NS Quick Ref Card)

• if type.ref is a prefix in the exception symbol, then the other slot will be called data and can contain anything.

```
|evt.ex;type.ref|: {name: <string>, data: <frame>}
```

```
Ex: {name: "the llama exception", data : {type: 'inka, size: 42, weight:
177}}
```

• if evt.ex.msg is a prefix in the exception symbol, then the other slot will be called message and contain a string

```
|evt.ex.msg|: {name: <string>, message: <string>}
```

```
Ex: {name: "Ho ho exception", message: "You have a serious problem,
mate"}
```

• otherwise, the other slot will be called error and will contain an integer (error code)

```
|evt.ex|: {name: <string>, error: <integer>}
```

```
Ex: {:name: "Hi Ho exception", error: -48666}
```

Here are examples of exception frames from real life:
• Example A, division by zero:

```
{name: |evt.ex.div0|, error: 1764744}
```

If you want to catch division by zero errors, here's the trick:

```
try
      5/0
onexception |evt.ex| do
      CurrentException();
```

• Example B, undefined variable:

```
{name: |evt.ex.fr.interp; type.ref.frame|, data: {error:-48807, symbol:
foo}}
```

How to Raise Gentle Exceptions

You can throw an exception using the global function, Throw(<exception symbol>, <exception data>). The value you pass for <exception data> is put into the "other" slot of the exception frame. Make sure it is the correct type (as per the above rules) or your call to Throw will raise another exception.

If the code block has more than one try statement in effect at one time, you can pass control to the next enclosing try statement using the Rethrow function.

Here are examples of the three different ways to create and throw exceptions. Note that you need to send an exception symbol ('):

```
Throw('|evt.ex.foo|, 99);

Throw('|evt.ex.msg|, "string");

Throw('|evt.ex;type.ref.something|, ["a", "b", "c"])
```

Default Exception Handling

Unfortunately, there is currently no general way for your application to specify a default exception handler, for example, viewExceptionScript. This means you need to use try statements wherever you want to catch exceptions.

Examples

Here's some code to test out the exception system. This tries various cases and is designed to print no output, assuming the exception system works as advertised. However, the NTK Inspector prints out every exception that is thrown, whether it's handled or not. Expect to see output; however, none of thePrint("*error?*) statements should execute.

```
//simple cases & a nested block
try
   begin

      try
         Throw('|evt.ex;type.ref;foo|,"dsafsda");
         Print("*error*1");
      onexception |foo| do
         begin
            //Exception handled quietly
         end;

      try
         Throw('|evt.ex;type.ref;foo|,"dsafsda");
         Print("*error*2");
      onexception |type.ref| do
         begin
            //Exception handled quietly
         end;

     //outer block should catch this one
      Throw('|evt.ex.foobar|,42);

     Print("*error*3");

   end
onexception |evt.ex| do
   if CurrentException().error <> 42 then
      begin
```

```
         Print("*error*4");
         Print(CurrentException())
      end;



//try out rethrow()
try
   begin

      try
         Throw('|evt.ex|,42);
         Print("*error*5");
      onexception |evt.ex| do
         begin
        //outer block should catch this one
            Rethrow();
         end;

   Print("*error*6");

   end
onexception |evt.ex| do
   if CurrentException().error <> 42 then
      begin
         Print("*error*7");
         Print(CurrentException())
      end;


//try it out with fn calls
try
   begin

      call func()
         begin
            try
               call func(esym) Throw(esym,42) with ('|evt.ex|);
               Print("*error*8");
            onexception |type.ref| do
               begin
              //outer block should catch this one
                  Rethrow();
               end;
         end
      with ();

      Print("*error*9");

   end
onexception |evt.ex| do
   if CurrentException().error <> 42 then
      begin
         Print("*error*10");
         Print(CurrentException())
      end;
```

```
Print("no news, is good news");
```

NewtonScript Q&As

Garbage Collection (10/15/93) (Obsoleted by NS Guide 1.0 Final Documentation)

In NewtonScript, the run-time module, not the programmer, is responsible for allocating storage for objects and for reclaiming the storage of objects that are no longer used. Garbage collection is carried out by the basic object system, so it's not technically NewtonScript that does it. The storage taken up by an object will be freed sometime after the last reference to that object goes away.

One of the many benefits of garbage collection is that the programmer has to think about freeing objects only in a small number of important places, as opposed to dealing with the constant background worry that objects must be freed. Wth automatic garbage collection you only need to consider disposing objects in rare cases.

One place where you *should* think about it is when you close an application; here you want to free as much storage as possible. You do this by either removing slots or by setting slots to `nil` in the application's context frame. Setting the value of all slots and variables referring to a particular object to `nil` allows the Garbage Collector to destroy the object.

Within the Newton operating system, automatic garbage collection is triggered every time the system runs out of memory. There's not really any reason to invoke garbage collection manually. However, if you must do so, you can call the global NewtonScript function `GC`. Consult the *Newton Programmer's Guide* to find out more about the `GC` function.

Order of Slots in Frames (9/15/93)

Q: The description of the foreach function in the manual might be taken as a hint that slots in frames are ordered. Does slot or array order imply anything about layout of data in memory (as with C structs?)

A: Slots are not ordered, nor have we seen any performance gains from ordering slots for particular sequences or search criteria. Don't count on their being in any particular order, nor on data structures in memory being ordered with any regard to the placement of slots (as in the ANSI Common LISP object extension CLOS).

Q: Where is a newly-defined slot placed in a frame? at the end?

A: Because slots are unordered, the location of a new slot cannot be predetermined.

Structured Literals (9/15/93)

Q: Is it really dangerous to assign a string literal to a variable, as in the following example?

```
s := "abc";
```

A: That depends on what you're going to do with s. If you're going to modify it, then yes, it's dangerous. In that case, you should use

```
s := Clone("abc");
```

In general you should treat string literals in NewtonScript as read-only (just as you should in ANSI C). String literals might reside in read-only memory or share storage with other literals.

The same warning applies to quoted array and frame literals, as in the following example:

```
'[1,2,3] or '{one: 1, two: 2, three: 3}
```

Here are some rules regarding string literals and quoted array/frame literals:

- Treat them as read only. Use clones when necessary.

- Realize that they always represent the same object. In the example below,

```
func foo() begin "abc"
end;
```

- always returns exactly the same string - not different, equivalent strings.

- In your code, an unquoted array or frame literal will generate a new array or frame each time it is evaluated. So it's acceptable for a function to return [1,2,3] or {one: 1,two: 2,three: 3} as long as they're not quoted.

Passing Arguments (9/15/93)

Q: Is NewtonScript call-by-reference or call-by-value?

A: A seemingly simple question. The short answer is “call-by-value,” the litmus test being that

```
func foo(x) x := 1;
```

does not affect the argument passed to it. However, this is not meant to imply that NewtonScript functions can never make changes to their arguments. If the argument has structure—for example, if it’s an array, frame, or string—then changes to the internal structure are persistent. Consider the following code fragments:

```
func foo(x) x[0] := 1;
// changes the 1st element of array arguments

func foo(x) x.slot1 := 1;
// changes the slot1 slot of frame arguments
```

Depending on what you’re used to, this example might be confusing because the concepts of call-by-reference, call-by-value are mixed, and copying arguments are intertwined (and often misunderstood.) The following discussion of this terminology with respect to other languages may help clarify the differences.

Pascal - Supports both call-by-reference (VAR parameters) or call-by-value (the default). When array or record arguments use call-by-value, copies of the arrays/records are made and passed in. To Pascal programmers, NewtonScript may seem to pass arrays and frames as VAR parameters.

C - Only supports call-by-value. Struct (not struct*) arguments are passed as values; that is, a copy of the struct is passed. However, array arguments pass in the address of the first element of the array. Thus, C programmers will find the array arguments to be treated normally.

LISP - Only supports call-by-value. Copies are not made of structured arguments (lists, structures, arrays, objects,...) Argument passing in Newton Script works just like it does in LISP.

Slot (Variable) Lookup (9/15/93)

Q: What differences are there among the various ways to access a slot?

A: The means by which you access a slot specifies the search path and inheritance mechanisms used to locate it. Each of the five ways to access a slot is described below.

•Specifying the slot name only

If you specify only the name of the slot, as in the example

```
theSlot
```

the search begins in the current function frame with lexically-visible variables, followed by slot names. In other words, if the currently-executing function has a local variable named theSlot, it is found before a local variable named theSlot in the enclosing function, and so on. Similarly, any of those variables are found (in scope order) before an actual slot named theSlot is found. If a variable named theSlot is not found, global variables are searched next. If a global variable is not found, the current receiver is searched for a slot named theSlot.

If the slot is not found in the receiver of the message, the remaining frames are searched in order of prototype and parent inheritance. An exception is thrown if you try to access a slot that can't be found. If you do an assignment operation using an undeclared slot, the slot is created if you place the self. prefix in front of the slot name; if you do the operation without the self. prefix you will get a local variable.

•Using the dot (.) operator

If you use the dot operator to specify the frame and the slot, as in the example

```
myframe.myslot
```

the search begins in the specified frame. If the slot is not found in the specified frame, the remaining frames are searched in order of prototype inheritance. Parent inheritance paths are not searched. If the slot does not exist, the system returns NIL.

•Using the GetSlot function

If you call

```
GetSlot(frame, slot)
```

only the specified frame is searched for the specified slot. Inheritance paths are not searched.

•Using the GetVariable function

If you call

```
GetVariable(frame, slot)
```

the search begins in the specified frame. If the slot is not found in the current frame, the remaining frames are searched in order of prototype and parent inheritance.

•GetVar will be removed from the language

You can usually simulate GetVar using GetVariable(self,slot). The difference will be that local variables will not be found - only slots.

If you use GetVar you may find that is not available in future products, in which case your code will throw an exception; or GetVar may work differently not finding local variables anymore, in which case your code may subtly break. DTS advises against using GetVar.

Testing the Existence of a Slot (Variable) (9/15/93) (Obsoleted by NS 1.0 Final Documentation).

Q: What differences are there among the various ways to determine whether a slot exists in a frame?

A: The means by which you test for a slot's existence specifies the search path and kinds of inheritance used to discover it. The following examples describe three ways to discover the existence of a slot in a frame.

- Using the exists operator

The exists operator follows the same rules as the expression to which it is applied. For example, if you use this operator to test a slot access expression, as in the example

```
myframe.myslot exists
```

the search begins in the specified frame. If the slot is not found in the current frame, the remaining frames are searched in order of prototype inheritance only. Parent inheritance paths are not searched.

However, when using the exists operator to check whether an identifier exists, as in the example

```
myvar exists
```

both prototype and parent inheritance paths are searched.

- Using the HasSlot function

If you call

```
HasSlot(frame, slot)
```

only the specified frame is searched for the specified slot. Inheritance paths are not searched.

- Using the HasVariable function

If you call

```
HasVariable(frame, slot)
```

the search begins in the current frame. If the slot is not found in the current frame, the remaining frames are searched in order of prototype and parent inheritance.

Calling Methods Out Of Slot Context (9/15/93) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: I'm trying to pull a method out of a slot and call it. Unfortunately, the functions apply and call both seem to set self to {}. Is there another way?

A: The global function Perform(frame, message, paramaterArray) does what you want, except that the method needs to be in a slot in the current frame if you really want to preserve the value of self.

Inherited (10/13/93) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: What is the function of "inherited:" and "inherited:?"

A: In NewtonScript, the inherited:X() message send specifies a NewtonScript function call on the function X, where X is found UP the inheritance chain starting from the caller's _proto. In other words, lookup for that method ONLY starts up the caller's _proto chain, NOT in "self" (the currently executing frame). The conditional message send (`:?`), with `inherited` specified, operates almost exactly the same except that no error results if the given method (function) is NOT found. The conditional message send (`:?`), with `inherited` specified, is helpful in complex heirarchies because you can use it without worrying about whether any such method was ever implemented higher up in the inheritance chain.

Deeply (foreach deeply in...) (10/13/93)(Obsoleted by NS Guide 1.0 Final Documentation)

Q: What is the difference between "foreach" and "foreach deeply in"?

A: 'Deeply' specifies that only the slots/elements of the current frame/array are listed, except that if passed a frame, slots of ancestors up the _proto chain are considered elements of the current frame.

Here is some Inspector code to illustrate the difference between the two:

```
//oo DEFINITIONS
normallist := func (param)
begin
      local tempItem;
      foreach tempItem in param
            collect tempItem;
end;
```

```
deeplylist := func (param)
begin
      local tempItem;
      foreach tempItem deeply in param
            collect tempItem;
end;



//oo SAMPLE DATA
x := {one: 1, two: 2, three: 3};

y:= {four: 4, five: 5, combo: x};

z:= {six: 6, _proto: y};

complex := {a: "first string", b: ["array1", "array2",
      {_proto: x}], c: 3.1415926};



//oo TESTS
:normallist(x)          // same
#4413441  [1, 2, 3]
:deeplylist(x)
#44137D9  [1, 2, 3]


:normallist(y)          // same
#4413A11  [4,
     5,
     {One: 1,
      two: 2,
      three: 3}]
:deeplylist(y)
#4413C49  [4,
     5,
     {One: 1,
      two: 2,
      three: 3}]


                  // For z, the answer is different.
                  // the deeply version travels proto chains!
:normallist(z)
#4416E29  [6,
     {four: 4,
      five: 5,
      combo: {#4415D79}}]


:deeplylist(z)
#4416FE1  [6,
     4,
     5,
     {One: 1,
```

```
        two: 2,
        three: 3}]


             // for this complex frame, the answers are the same
             // because there is no _proto slot in the *MAIN* frame
             // hence there is nowhere to go deeply into.
:normallist(complex)
#4418741  ["first string",
     ["array1", "array2", {#44183B9}],
     3.14159]

:deeplylist(complex)
#4418AC1  ["first string",
     ["array1", "array2", {#44183B9}],
     3.14159]
```

Compile Function (10/9/93) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: I need the ability to download the validtest and endtest functions to the Newton for searches. I do not see how I can convert the incoming textual representation of the function to code to execute. How could I do this?

A: You could use the special compile function in NewtonScript. Here's an example of how to use compile:

```
   begin
       local x;

       // Set x to be a function that returns the new function.
       x := Compile("Print(\"Hello\")");

      // Now call the actual compiled function.
      call x with ();

      // At this point, "Hello" will have been printed to the
inspector...
   end;
```

Note that compile returns a function taking no arguments whose body is the string passed to compile. If you need to create a function taking arguments, you can create a function with Compile whose return value is a function that takes arguments, like this:

```
f := call Compile("func(x) x*x") with ();
call f with (10);  // prints 100.
```

Function Definitions (10/9/93)

Q: Is it "safer" to use a return statement at the end of a function or is it just a style issue?

A: All functions return the last value in the function body. This means that you have a choice of explicitly writing a return statement, or assuming that the code reader knows that the last statement is automatically returned.

One might argue that a return statement should always be used. However there are cases where the code looks prettier with no return statement, and the return statement does generate code. Compare the following two examples:

```
begin
   ...
   if expr then "" else text
end


begin
   if expr then
      return ""
   else
      return text
end
```

Additionally, in the current NewtonScript compiler, an extra byte or two is used for the return statement, but in most cases it will make no apparent difference in speed.

Closures and Perform (10/9/93) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: The NewtonScript manual indicates that the receiver during a call is the receiver that made the call. For example, the code

```
a := {
id: "a",
foo: func() Print(id),
};

b := {
id: "b",
bar: func() call a.foo with (),
};

b:bar();
```

should print "b".

When I run this in an Inspector, however, it appears that the receiver during the execution of foo() is the top level frame in which a (and b) is defined. How is the receiver actually determined? Is it always the top level of the Inspector?

A: What you really create is a closure, which is the executable code bound up with the dynamic and lexical environments at the time the function was created. That is, whatever self evaluated to when the function was compiled is what it will evaluate to when it's called.

The Perform call, or message-passing does change the closure's dynamic environment (self) to the frame that was specified by the message pass or Perform call.

To do what you seem to be attempting, you could try this:

```
a := {
id: "a",
foo: func() Print(id)};

b := {
id: "b",
bar: func() begin self.temp := a.foo; :temp() end};

b:bar();
```

Instead of :temp() above, you could have used Perform(self, 'temp, []); Perform is handy when you don't know at compile time which message you need to pass. For example, you can write Perform(self, jumpTable[i], []); where jumpTable contains an array of symbols representing methods. Perform takes a frame, a message and an array of message parameters.

Symbol Hacking (11/11/93) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: I would like to be able to build frames dynamically and have my application create the name of the slot in the frame dynamically as well. For instance, something like this:

```
MyFrame:= {}; tSlotName := "Slot_1";
```

At this point is there a way to then create this ?:

```
MyFrame.Slot_1
```

A: There is a function called Intern, that takes a string and makes a symbol. There is also a mechanism called path expressions (see the NewtonScript manual), that allows you to specify an expression or variable to evaluate, in order to get the slot name. You can use these things to access the slots you want:

```
MyFrame := {x: 4} ;           // MyFrame -->{x: 4}

theXSlotString := "x" ;
MyFrame.(Intern(theXSlotString)) := 6
                              // MyFrame --> {x: 6}

tSlotName := "Slot_1" ;       // the following code creats a
```

```
                                    // slot called Slot_1 in MyFrame and
                                    // assigns it the number 7.
MyFrame.(Intern(tSlotName)) := 7 ;
                                    // MyFrame --? {x: 6, Slot_1: 7}
```

Q: Is there a way to look for a slot programmatically starting with a slot name string?

A: Here's the code which can do this:

```
if MyFrame.(Intern(slotToFind)) exists then
      x := MyFrame.(Intern(slotToFind)) ;  // use the slot
```

Literals and run time (11/11/93)

Q: Is the literals slot really needed during run time? I noticed that I could figure out quite a lot about how my functions work from the literals used (examining the literals slot). I would rather not make it easy for others to reverse engineer certain routines.

A: The literals slot is indeed needed during run time (in the current implementation of NewtonScript) The interpreter is using it at run time. Here's an example:

```
f := func() "abc";
#4409CB9  <CodeBlock, 0 args #4409CB9>
f.literals
#4409B61  [literals: "abc"]
f.literals[0] := {foo: 'bar}
#440A1D1  {foo: bar}
call f with ()
#440A1D1  {foo: bar}
```

Don't assume anything about the implementation of the literals slot in future Newton system software.

Missing Semicolons cause Weird Errors (12/11/93) (Obsoleted by NS Guide 1.0 Final Documentation)

As with Pascal, semicolons are statement separators, not terminators. If you forget to add a semicolon at the end of the Newtonscript statement, the interpreter will try to interpret a larger statement than intended, causing strange error reports.

The following is an example of such a case.

You are getting an "expected array, frame, or binary but got 8" exception because you wrote the following code:

```
:Emit(dest, 'subprim, "Send", [t2, t1, Length(node.r3)])
 :PopTemp('ref);
```

Note the missing semicolon on the first line. This is parsed as

```
(:Emit(blah blah)):PopTemp('ref);
```

The :Emit call resulted in an 8, which was used as the receiver for the PopTemp message. This was NOT what the programmer intended.

There are two habits that would have prevented this error. You might want to consider them as part of your personal coding style, though neither of them is extremely important since this error is pretty rare.

- Always use "self:foo()" instead of ":foo()".
- Always put a semicolon at the end of a statement, even when you don't need to.

Classical OOP programming withNewtonScript (12/17/93) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: I would like to see the ability to use more of a C++ approach, where I could inherit code — not necessarily user-interface (view) oriented code — with functionality like templates.

A: You can easily simulate most aspects of traditional class-instance OO programming (as in Smalltalk or C++) using NewtonScript. There aren't any specific features in NTK 1.0 to support this, but you can do it without too much trouble, using the Project Data window.

The basic idea is to use _parent inheritance to provide the link between "instances" and "classes". You define a frame in the Project Data to hold the methods (this is the "class"), and you construct "instances" whose slots are the "instance variables" (or "members" if you prefer C++). Quotes will continue to be used around these words to emphasize that there is no built-in concept of a class in NewtonScript; we are using the prototype-based inheritance system to simulate class-based inheritance.

Suppose we want to create a "class" to represent a stack. We will represent the stack as an array. Each "instance" needs to store the array and the index of the top element of the stack, so an "instance" will look like this:

```
{ _parent: ...,                 // pointer to the "class"
  items: [...],                 // array of items
  topIndex: 2 }                 // index of top item
```

The "class" contains the methods, like so:

```
Stack :=
      { Push:     func (item) begin
                              topIndex := topIndex + 1;
                              items[topIndex] := item;
                              self
                          end,

                Pop:      func () begin
                              if :IsEmpty() then
                                  NIL
```

```
                        else begin
                            local item := items[topIndex];
                            items[topIndex] := NIL;
                            topIndex := topIndex - 1;
                            item
                        end
                    end,

        IsEmpty:    func ()
                        topIndex = -1,

        Clone:      func () begin
                        local new := Clone(self);
                        new.items := Clone(new.items);
                        new
                    end,

        New:        func ()
                        {_parent: self,
                         items: Array(10, NIL),
                         topIndex: -1}
    }
```

Obviously, this ignores such niceties as stretching the array and signalling error conditions, for the sake of clarity.

To get a new Stack "instance", you send the New message to the "class":

```
s := Stack:New();
```

Now you can send messages to s to get work done:

```
s:Push('a)
x := s:Pop()
if s:IsEmpty() then ...
s2 := s:Clone()
// etc.
```

You can put "class variables" and "class methods" in the "class" if you want; they're all the same thing in this model, just slots of the "class". The New method is an example. Note that you can get a new instance from an old instance (i.e., s:New()) because of this unity.

Those who have been paying attention to the Listener output may already have noticed this method of programming, because it is used by soups, stores, and cursors. (But you should assume that anything undocumented you notice in the Inspector may be changed in the next ROM.)

To use this technique in NTK, you can put the "class" in the Project Data window. To access it from your code, you'll need to put a slot that refers to it in your main view (or lower down if you don't need the class throughout your application). You can just add a slot called "Stack" whose value is "Stack", or if you prefer, you can use distinct names (for example, call it "StackClassFrame" in Project Data).

Here's an example, in NTK text-export form. This does not include the project data section, because it looks just like the "Stack :=" section above. This is just an application with a button that prints "30" in a complicated way using a stack object.

```
main :=
   {title: "xxx",
    viewBounds: {left: -1, top: 0, right: 236, bottom: 333},
    Stack: Stack,
    _proto: protoapp,
   };

_view000 := /* child of main */
   {text: "Button",
    buttonClickScript:
      func()
      begin
         local s := Stack:New();
         s:Push(10);
         s:Push(20);
         Print(s:Pop() + s:Pop())
      end,
    viewBounds: {left: 74, top: 74, right: 154, bottom: 106},
    _proto: prototextbutton
   };
```

You can use an extended form of this technique to put an object-oriented interface on soup entries. "Wrap" the soup entry in an "instance" frame like the one above by pointing a slot of the "instance" to the soup entry. You can have "class methods" to do queries and return these "instances". You can also have "instance methods" to change, validate, undo, flush, and so on. You can also choose _proto as the slot that points to the soup entry, so you inherit the entry's slots--but remember that slot assignments will go into the "instance", not the entry.

If you're intrigued by all this, see the various papers on the language "Self", by Ungar, Chambers, and others at Stanford (now at SunSoft). Many aspects of NewtonScript were inspired by Self. The postscript files are available via ftp on Internet from self.stanford.edu.

Only 7-Bit ASCII In NewtonScript Symbols (1/26/94) (Obsoleted by NS Guide 1.0 Final Documentation)

Q: My frame has some slot symbols that contain accent and special letters. For example:

```
f := {|à|:"help me please", |é|:"Thanks in advance", ...,...}
```

My program received a letter from ProtoInputLine and turned it into a symbol. I have tried many different ways to do this but I could not convert it into a symbol.

A: The basic problem is you can only use 7-bit ascii in symbols.

Nested IF Statements, Constant Condition Problems (6/9/94)

Q: I have found a mysterious problem with Scripts containing nested IFs. In some cases like the following example they are executed in an abnormal way.

The true conditions in the real code are constants defined in the project data file with values true or nil. These are used for conditional compiling.

```
protoApp.viewSetupDoneScript : FUNC ()
begin
if TRUE then
begin
if TRUE then
      begin
      if TRUE then
      begin
            if TRUE then
            begin
                  Print ("Idlescript1");
            end
      end
  end
end;
Print ("Idlescript2");
end
```

There should be only two prints. In fact there are five prints in the following manner:

```
"Idlescript1"
"Idlescript1"
"Idlescript1"
"Idlescript1"
"Idlescript2"
```

If I define Boolean slots with value truein the base view, and if I use those slots as conditions, the code works like it should. Why can nested IFs with constant conditions work like a loop?

A: This is a bug in the compiler's handling of IF statements with constant conditions. The bug occurs only if the result of the IF statement is not used. For example,

```
func () begin
    if TRUE then
        if TRUE then
            Print(1);
    Print(2);
end
```

will trigger the bug, but

```
func () begin
    local x := if TRUE then
                   if TRUE then
                       Print(1);
    Print(2);
end
```

will not.

The workaround is to use the result of the `if` statement. You don't really want to keep the result around, so the above example is not recommended . The simplest workaround is probably the following:

```
func () begin
    (
        if TRUE then
            if TRUE then
                Print(1);
    ) and TRUE;
    Print(2);
end
```

This works because the NTK 1.0.1 compiler does not throw away the "and true" part, but the result of the "if" is dropped immediately.

NEW: How to Avoid _parent Problems (6/28/94)

Q: I read somewhere that developers should never access a view _parent slot directly? Are there any cases when it is safe to access the _parent slot?

A: In most cases, you should use the :Parent() view method to determine a parent view. However, there are some cases when you want to access the _parent slot of a view directly. The situation which is bad is to use _parent as a simple variable because results will vary depending on implementation of the current function. Some guidelines:

`_parent`	is bad
`view:Parent()`	is OK
`self._parent`	is OK (and should be the same as :Parent())
`view._parent`	is OK (and should be the same as view:Parent())
