Building Newton Applications with Newt

by Steve Weyer

Version 1.2, 4/24/95

The "Newt" discussed in this article is a shareware native programming environment on the Newton. Perhaps, you may use Newt as a generic nickname for your Newton MessagePad. So, this article will show how you can develop a simple application using Newt on your Newt — no cables, desktop computer or computer science degree required.

You will get the most out of this interactive article/tutorial if you have Newt (application) and NewtATut (book) installed on your Newton. NewtATut 1.2 requires Newt 3.0 (or later). See **Where to Find Newt and Further Information**. Improvements over earlier versions includes syntactic cleanup and creation of application packages directly on the Newton. A print version of this article appeared in *PIE Developers*, Vol. 2.4, July 1994; pp.14-18. (This article may also be available as an Acrobat PDF file (NewtATut.pdf), which requires Adobe Acrobat Reader (free) on your desktop system). NewtATut is freeware and may be freely distributed on line services as long as it is unmodified and includes the file NewtATut.1st. Copyright 1994, 1995, S. Weyer, All Rights Reserved Worldwide.

Current Newton applications you have used or heard about, including Newt, have been constructed using the Newton ToolKit (NTK) from APDA (Apple Programmer and Developer Association) on a Macintosh. Newton books, like this one, have been created using BookMaker, a pre-processor for NTK. NTK should be available on Windows real soon. Despite being the only game in town, NTK provides an excellent framework for any large, complex, industrial-strength application. However, NTK requires a heavy investment in terms of software, hardware and training — making it difficult and expensive for PC-based developers, students or developer-wanna-bes to find out more about, and actually do, Newton development.

Newt Genealogy

Like a chameleon, Newt has evolved to provide different functionality to different users. The first version of Newt in Oct. '93, inspired by the Inspector Gadget and Dot2Dot examples from Apple, allowed you to draw graphics using NewtonScript — the turtle, its amphibious cousin and name inspiration, used Logo. You, the learner, could explore mathematics via a turtle microworld, or add NewtonScript methods to emulate Logo commands and data structures. A separate book — NewtTurT — allows to experiment interactively with the turtle.

With enthusiastic feedback from early users, I shifted efforts and emphasis so that you could create objects based on Newton interface prototypes and save an application. Newt's icon (next) reflects both its original turtle personality (on left) as well as its later application personality (on right).

Using Newt from this book

Newt 3.0

If you have Newt installed, you can tap on its icon above to start it — if successful, Newt opens, then a few seconds later, the book reappears. You can access ("fall into") Newt later by closing this book. You can return to this book by selecting its title "Building Newton Applications with Newt" from Newt's overview list (bottom dot between two scroll arrows).

Later in this book, you can tap on underlined code to copy and evaluate it in Newt. This also saves the source text in the Notepad — and usually exits the book when there is usually a visible result. If you accidentally close Newt too soon, just start over. For directions on how to create applications directly in Newt, refer to the files accompanying Newt.

Consumer Alert!

Before delving into programming details, I offer a warning. Apple and Newton developers are still learning how to implement system and application software. In addition, while learning with Newt, you will take risks and make mistakes. For example, Newt allows you to execute arbitrary NewtonScript expressions. This can be a formula for enlightenment or disaster. So, remember that this is your Newton with your information and listen to your mother's advice about backing up your system.

Using common sense in following examples and suggestions, and limiting yourself to documented commands, you should reap many benefits and long hours of enjoyment from using Newt. At the same time, "a little learning [about programming] is a dangerous thing" (my apologies to Alexander Pope). I would caution against too much experimenting with random functions or methods. "Gee, I wonder what xxx does" might yield a simple error message, or it might zap a frame or soup in your system.

During the course of examples, this article will provide some glimpses, but not exhaustive explanations, of NewtonScript syntax, methods, frames, prototypes and views. Familiarity with programming concepts and syntax in general, and Lisp or Smalltalk in particular, would be helpful, though not necessary. For a more complete guide, I defer to the NTK documentation and introductory books on Newton programming, such as *Programming for the Newton* by Rhodes & McKeehan. There is a more information available in Newt's readme files, and, of course, you can register to receive a collection of program examples and a Newt manual, and to encourage me to write more interactive articles like this one.

Creating a "Hello World" Application

"Hello World" is the canonical test of any programming environment. In this tutorial, you will create an application that contains a button, an about box, some input and result fields and a checkbox. You will describe these *objects* in NewtonScript, and add them dynamically to a live application.

Remember: to return to this book, select its title from Newt's overview list. To start with a new, empty application, tap on the following (if Newt is present, the expression should highlight before exiting to Newt to show you the result):

```
MyApp
//:doObj('build,'MyApp)
{_proto:  protoApp,
//viewBounds -- defaults to full screen
title:  "Hello World",
_package: {shortTitle: "Hello",}
}
```

If you attempt to add an application or object that already exists, Newt replaces it. You should now have a completely functioning application complete with title and statusbar, with clock and closebox. If you are not satisfied with only that functionality, read on...

Adding A Text Button

To add a button to your application:

```
MyApp+button
{_proto:  protoTextButton,
viewBounds:  RelBounds(100,120,40,16),
text:  "About",
buttonClickScript:  func()
  if float exists
  then float:open()
  else PlaySound(@102), // ROM_funbeep
}
```

You define an object as a *template frame*, delimited by curly braces {}, with slot-value pairs separated by commas. The slot is a symbol followed by a colon. Templates typically have several slots that you supply or override:

- a `_proto` slot, whose value refers to a built-in system prototype or one that you have defined, here `protoTextButton`
- a `viewBounds` slot whose value is a frame that defines the location of this object on the screen. These coordinates are interpreted using its view justification and are relative to the parent. The `viewBounds` frame can be specified by calling the function `RelBounds` with the upper left corner and width and height, which yields the corresponding frame {left: 100, top: 120, right: 140, bottom: 136}.

• other slots that override system values or methods, or define application-specific ones. `text` is a typical slot, whose value is a string, here `"About"`. buttonClickScript defines a method that will be invoked when you tap the button. Tap it now (you should hear a sound since the about box (float) has not yet been defined).

Adding an About Box

In order to add an about box, you will add two new objects: a floating view and a text object inside that. First, add the floating view (no visible change):

```
MyApp+float
{ proto:  protoFloatNGo,
viewBounds:  RelBounds(20,140,150,100),
}
```

Now, add a text object to float:

```
MyApp.float+aboutText
{viewclass:  clParagraphView,
viewBounds:  RelBounds(5,5,140,90),
text:  "This demo created by"&&
      userConfiguration.name&
      ", with the help of Newt, the lizard wizard",
viewFlags:  3, //vReadOnly+vVisible,
}
```

If float is still open, close it. Now, when you tap on the About button, the about box appears.

Define a User Prototype

You will be adding several similar input fields to the application. To do this in an object-oriented style and avoid redundant code, add a user-defined prototype to the application (this expression will highlight, but not exit to Newt since no visible change occurs):

```
MyApp.myInputProto
{ proto:  protoInputLine,
text:  "",
value:  0,
viewFlags:  10753, //vVisible + vClickable + vGesturesAllowed + vNumbersAl-
lowed,
getTextValue:  func() // from text, set number value (used by total)
  begin
    self.value := StringToNumber(text);
    if not value then value := 0;
  end,
viewChangedScript:  func(slot,view)
  if slot='text
  then begin
    :getTextValue();
    if total exists then total:update();
    end,
viewSetupFormScript:  func()
```

```
      begin
      self.text := clone(text);
      :getTextValue();
      inherited:?viewSetupFormScript();
      end,
    }
```

Some information about this prototype before creating some instances. This user prototype is based on/inherits from a built-in prototype — protoInputLine. The viewFlags slot specifies recognition behavior — you override `viewFlags` to assure that your input field will recognize mainly numbers. The value of `viewFlags` is based on a set of bit switches. These switch names (indicated in the comment) are available in NTK at compile-time, and can be made available in Newt during development by linking constants via plug-in modules. It is easiest, for now, to provide the "magic numbers" yourself — a little crude but workable.

Next, if you want to provide a way of describing what should happen after text is entered into a field, you add a `viewChangedScript` method. Here, it converts the string value of its `text` slot to a real number, caches it in a `value` slot you added, and sets `value` to 0 if it is `nil` (`StringToNumber` returns this for empty strings). (It would have been less cumbersome to write `value := StringToNumber(text) or 0;` but NewtonScript's boolean operators, unfortunately, return only `true` or `nil`, although they will operate on any kind of value.) It then asks `total`, if it exists, to `update`. There are two new pieces of NewtonScript syntax: first, you can include multiple statements in a `begin…end` statement, separated by semi-colons; second, you can branch on logical tests using the `if…then…else` conditional statement (the `else` clause is optional).

Finally, to ensure that the field is correctly initialized, you specialize the `viewSetupFormScript` method. This makes a copy (clone) in the view of the prototype's `text` string and sets an initial numerical value. `viewSetupFormScript` conditionally sends the same message to the system variable `inherited` — always a good idea when you override a system method — so that `protoInputLine` can perform any additional initialization.

Adding Input Fields

Now, add two input fields that use this user prototype:

```
MyApp+num1
{ proto:  myInputProto,
viewBounds:  RelBounds(130,20,100,20),
}
```

You can write numbers into the input field, use the scrub gesture to erase, and double tap to popup a numeric keypad. This inherits slots, including behavior, from myInputProto, the user prototype you defined earlier. Now, add the second field:

```
MyApp+num2
{ proto:  myInputProto,
viewBounds:  RelBounds(130,45,100,20),
}
```

Adding Total

You might have noticed that `viewChangedScript` in myInputProto attempts to update the `total` field, but only if `total exists`. You will now create a `total` field:

```
MyApp+total
{ proto:  protoStaticText,
viewBounds:  RelBounds(130,80,100,16),
text:  "Total", // initial text
numVars:  ['num1, 'num2],
getValueText:  func() // return text from summing field values
  begin
  local tot := 0, field;
  foreach field in numVars // add up num1.value + num2.value etc.
  do tot := tot + GetVariable(self,field).value;
  if round exists and round.viewValue
  then tot := RIntToL(tot);
  NumberStr(tot); // return string
  end,
update:  func()
  SetValue(self,'text,:getValueText()),
viewSetupFormScript:  func()
  begin
  self.text := :getValueText();
  inherited:?viewSeutpFormScript();
  end,
}
```

When you change any of your input fields, the total should update automatically. Notice that the `numVars` slot is initialized to contain an array of symbols naming the fields to be totalled, in this case, `['num1, 'num2,]`. In the `update` method, you declare several local variables, and iterate over this array using the handy NewtonScript `foreach` construct. Since the field is a symbol name, `GetVariable` looks up the field name, such as `num1`, in the current context to obtain a reference to an input field frame. By using inheritance, `GetVariable` finds the name defined in `total`'s parent, in this case `myApp`, where `num1` and `num2` are defined. Next, it may round the value, using the non-mnemonically named built-in function `RIntToL`, depending on the state of a yet-to-be-added checkbox named `round`. Finally, it converts the number `tot` to a string using `NumberStr`, and sets its `text` field. Using the `SetValue` function ensures not only that the `text` slot is set to the new string, but also that the Newton view system will update the screen to reflect this change.

Adding a Checkbox

Finally, since you provided a little code in `update` to handle rounding of the result, you can now add a checkbox object named `round` as follows:

```
MyApp+round
{ proto:  protoCheckbox,
viewBounds:  RelBounds(20,78,50,16),
text:  "Round?",
valueChanged:  func()
  if total exists then total:update(),
}
```

Now, when you enter numbers with decimal points, most easily via the keypad, you can affect whether the total is shown as a decimal number or a rounded integer by toggling the checkbox.

Running and Saving (and Running) your Application

Of course, your application is already running within Newt. Also, you can reconstruct it quickly in a later session with Newt from your method sources saved in the NotePad — you can build from existing sources by evaluating `:doObj('build,'myApp)`. Perhaps you might like to edit the methods to change the name of the button, for example. Newt automatically compiles methods in the curent folder. You then would need to re-create the application and objects.

However, if you would like to save your finished application in a form so that you do not need Newt or so that you can give your application to someone without distributing your source code, you can save it as a package (using the NewtPack plug-in) or if this does not work, RUNewt can also be used to save, run, beam or email an application.

To save your application from Newt:

• the NewtPack or RUNewt package must be installed -- see packages.txt

• make sure your application is visible — select it from the overview list if necessary

• tap the Save button

If NewtPack was installed (and the save was successful), you can exit Newt and tap on the Hello icon in Extras. If you are using RUNewt,

• you can either select RUNewt from Newt's overview list, or from the Extras Drawer after exiting Newt

• select Hello World from the list of applications,

• tap the action button, select Run App

• after your application appears, you can close RUNewt or just drag it out of the way.

I hope that this whirlwind tour has provided a general introduction of how you can write NewtonScript, create objects and save this as an application on your very own Newton. Although this is a simple example, you can take the same basic ingredients, plus a few more, and concoct more interesting and complex recipes using Newt.

Some Final Information and Disclosures

Although Newt slices, dices, cures cancer and ensures happiness, Federation regulations require me to disclose some of Newt's potential limitations and possibly recalibrate user expectations.

How large an application can I build with Newt?

When you download a NTK-created package into your Newton, it is placed in a special area known as package memory. When you open your application, the package uses some dynamic memory — also known as frame heap — for run-time state; however, much of the application remains in package memory. Newt currently creates its application entirely in dynamic memory, except for references to built-in objects. Although this can be saved as a package so that it will occupy little heap at run-time, it does need to fit into heap during initial development. For RUNewt, the application is saved as a compressed frame in an application soup. When it is later executed by RUNewt, it again occupies frame heap. This means that as you develop Newt applications, you will eventually see the dreaded "Newton does not have enough memory to do what you want now. Do you want to Restart?" (or `Exception |evt.ex.outofmem|: (-48216) Ran out of Frames Heap memory`).

Handwriting consumes frame heap, especially in early systems. Open applications, and even some closed ones that rudely keep object references, consume additional heap. Often, restarting will help clean up unused objects that cannot otherwise be reclaimed through normal garbage collection. In addition to frame heap, other limits such as size of text notes and speed of access across a large collection of soup entries may also affect native application development.

The NTK version of a small application like my Pico Fermi Bagels game, which contains a roll browser and a handful of controls, occupies approximately 15K in package memory. Slurpee — an extension to the DTS Slurp example for soup entry transfer over a serial connection — consumes about 40K in its NTK version. Newt can construct both of these, though additional tricks such as virtual methods and incremental object creation are needed as applications become larger. Hopefully, newer revisions of the Newton system software will allocate more space for frame heap and manage it more effectively, and newer Newt versions will use less, allowing ever larger applications.

Can I access Newt-created applications via the Extras drawer?

If you save an application as a package, it behaves like any "normal" application. If you use RUNewt, you can run your application directly or you can install an icon for it so that is accessible via Extras. Although this Extras icon will survive system resets and card removals, it may not work properly with Extras/package utilities.

Can I create any kind of application using Newt?

Basically, yes, if you have enough documentation, frame heap, and perseverance. You may also need to structure your application and adapt NTK examples somewhat to fit Newt's style and to work around Newton system limitations. Current examples include:

• a more extensive version of "Hello World" — from the "kitchen sink school of interface design" — that demonstrates many of the system prototypes;

• an application that modifies rolodex entries in your Name soup;

• an application that adds panels to the Formulas application;

• a number guessing game (Pico Fermi Bagels)

• a serial communications example for transferring soups to the desktop (Slurpee)

• an application with a simple online help book

• calculators (scientific; intelligent assistant)

• versions of most of the Apple DTS (Developer Technical Support) examples

Which system prototypes are available?

Newt 3.0 currently includes and documents 59 common system prototypes and viewclasses (all those documented in NTK 1.0.1). User prototypes are an economical way to define your own version of a system prototype with your own default and additional slots and methods, and use it in several places in your application. You can also include objects that contain other views, for example, the float object would be called a "linked layout" in NTK. Finally, you can add named references to other prototype frames in the ROM or in other applications.

How does Newt differ from NTK?

Newt is a native rather than a desktop development environment. As a one person effort, it is also not as large, complete or well-documented as NTK. Newt can use Slurpee to transfer text sources to the Notepad, convert graphic and sound resources and provide a simple inspector for debugging. Newt 3.0 can save modest-sized applications as packages and provide limited support for constants. NTK is more robust and complete in all of these areas, plus it provides a layout environment for creating application objects graphically.

Who should be a Newt user?

Newt is appropriate if you want to learn about NewtonScript programming and Newton application development, if you would like to build and distribute small to moderate-sized applications, or if you want to do some portable prototyping or lack a Macintosh for development. Since Newt complements NTK, some Newt users are also using or considering NTK. Newt's turtle personality can provide a portable learning environment for children. Current world-wide Newt users include university students, professors, PC developers, financial traders, and my 13-year old daughter.

What's next for Newt?

Since Newt's evolving personalities and ambiguous name left more than a few early downloaders confused about its identity and utility, it's possible that Newt may be renamed and repackaged in the future. In functionality, Newt could evolve in many possible directions: more examples and system prototypes, more documentation, non-programmatic application interfaces, support for application-specific development like database forms, integration with other Newton applications, electronic articles like this one, etc. — as with most shareware, how Newt will evolve depends greatly on the feedback and level of support from users.

Where to Find Newt and Further Information

You should be able to obtain Newt 3.0 (or the latest versions) from the following online sources (usually as filenames similar to newt-devenv-30.sit/.hqx, newt30.sit/.zip):

• America Online(AOL): PDA:Software Libraries:Newton

• Compuserve: GO NEWTON (DL 8 or 9)

• eWorld: ShareWare:Newton

• Internet (anonymous ftp):

  -newton.uiowa.edu/pub/newton/software/dev or /app (or /submissions)
  -ftp.amug.org/pub/newton
  -sumex-aim.stanford.edu/info-mac/nwt/dev

• Usenet newsgroup: comp.binaries.newton

Registered Newt users receive a 70pp. manual that introduces NewtonScript and describes Newt commands and methods, a set of 160+ examples, notification and discussion of future releases, and relief from shareware procrastination and guilt.

The North Atlanta Newton User Group (NANUG) newsletter *_protoReality 1.3*, available on many networks, contains an interview with me and a turtle-oriented article by my daughter. Erica Sadun reviewed Newt as turtle in *PIE Developers*, Vol. 2.4, July 1994, pp. 6-7. Finally, I welcome comments and suggestions. You can contact me via one of several email addresses:

• weyer@netaxs.com

• AmericaOnline, eWorld, NewtonMail: SteveWeyer

• CompuServe: 74603,2051

• http://www.netaxs.com/~weyer (my home page with latest Newt info)

Bio

Over the past 20+ years, Steve has implemented and managed R&D projects involving object-oriented languages and prototyping environments, AI tools, hypertext systems and education. When not borrowing time from his family to work on Newton applications and generally recovering from the culture shock of transplanting from Silicon Valley to rural Pennsylvania, he consults for a pharmaceutical client on enabling technologies including pen-based systems.