Bit Parts

DRAFT 6

Michael S. Engber
Apple Computer - Newton ToolBox Group

This article was (will be) published in the May 1994 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

Introduction

Many people are unfamiliar or unclear about the terms package and part. Yet these are very basic concepts in Newton programming. This is because prior to NTK 1.0.1 final you didn't have to explicitly decide what kinds of packages and parts you produced. This article will try to explain these terms and discuss the possibilities introduce with NTK 1.0.1. I assume the reader has some experience using NTK to write Newton applications.

Packages and Parts

On the Newton, a package is the basic unit of downloadable software. Newton Connection and Newton Package Installer each allow you to download packages. The Remove Software button in Prefs and the card dialog allows the user to remove a package.

Packages contain parts. There are a variety of different kinds of parts currently supported by Newton. Each has a four character type. Some of these types are shown in Table 1.

part type	part code	description
form	`form`	general purpose applications created by NTK
book	`book`	books created by Newton Book Maker
auto	`auto`	faceless background apps (or inits) of the Newton world
font	`font`	additional fonts
dictionary	`dict`	custom dictionaries
store	`soup`	store containing read only soups

Table 1 - Part Types

Prior to NTK 1.0.1 you could only create packages containing a single part a form part or a book part. Using NTK 1.0.1 you can create auto parts and multi-part packages.

Auto Parts

How Form Parts Work

Form parts are what you're already used to creating with NTK. I only mention them here to contrast them with the discussion of auto parts in the next section. When a form part is installed the following steps take place:

1. Its `partFrame` is TotalCloned (except for `theForm` slot) and then the clone's `InstallScript` is called with one parameter, the `partFrame`
2. A view is made using `partFrame`.theForm as the template. This view is put in a slot of the root view using the app symbol as the slot name.
3. An icon is added to the extras drawer for that part.

When a form part is removed its `RemoveScript` is called. It is passed one parameter, the same `partFrame` that was passed to the `InstallScript`.

How Auto Parts Work

Auto parts work very differently than form parts. When an auto part is installed the following steps take place:

0. Its `partFrame` is **not** TotalCloned.
1. Its `InstallScript` is called with two parameters, the `partFrame` and the `removeFrame`. The `InstallScript` has access to other data in the package through `partFrame.partData`. There is no `partFrame.theForm` slot.
2. A view is **not** made.
3. An icon is **not** added to the extras drawer.

When an auto part is removed its `RemoveScript` is called. It is passed one parameter, the `removeFrame` that was passed to the `InstallScript`.

Dispatch-Only Auto Parts

There is a slight variation on an auto part called a dispatch-only auto part. It installs like a regular auto part, but then it's automatically removed. Its removal works the same as a regular auto part except for that it is triggered automatically. Dispatch-only auto parts load, their `InstallScript` runs, their `RemoveScript` runs, then poof, they're gone.

How To Create Auto Parts

NTK 1.0.1 doesn't "officially" support auto parts. You won't find information on how to create them in the manual. Expect the user interface for creating auto parts to change in future versions of NTK.

To create an auto part project in NTK all you need to do is specify "`auto`" for your application name and symbol in the projects settings dialog ("`auto!`" if you want a dispatch-only auto part). You will need to create a dummy main layout so your project will compile, but this layout won't actually be put in your package. The code for your auto part all goes in `"Project Data"`. You create your `InstallScript` and `RemoveScript` in a similar fashion to form parts. You'll use code that something like the code below.

```
InstallScript := func(partFrame,removeFrame)
begin
…
end;

RemoveScript := func(removeFrame)
begin
…
end;
```

In addition, if you want your auto part to have some data you can define a `partData` global variable in the `"Project Data"` file. For example:

```
partData := {s1: "moe", s2: "larry", s3: "curly"};
```

The contents of the `partData` variable will be tucked away in your part frame in a slot by the same name. You should make `partData` a frame whose slots contain the various pieces of data used by your `InstallScript`. The `partData` slot is mainly a convenience to help you organize your code. You're free to embed data in your package as literals or constants in your `InstallScript`. One situation where `partData` is especially useful is accessing templates you layed out graphically in NTK. This is described in the next section.

How To Use Layouts in Auto Parts

Of course, you can always explicitly define your templates as NewtonScript frames. (In this section I use the term template to generically mean templates, user-protos, or print formats.) However, using NTK to define your templates is easier and less error prone. Naturally, you will want to take advantage of NTK to define the templates you use in your auto parts.

As was mentioned before, the layouts in your project window will not be automatically included in your package. Auto part `partFrames` do not have a `theForm slot`.

To use templates you lay out in NTK with auto parts you should:

create your templates in NTK as you normally would for a form part

add them to your project

give each template's top-level view an `AfterScript` which tucks away the template in `partData`.

An example `AfterScript` is shown below.

```
begin
  partData.fooTemplate := thisView;
end
```

You will be able to access this template from your `InstallScript` using `partFrame.partData.fooTemplate`.

When To Use Auto Parts

The main thing that distinguishes auto parts from form parts is they don't have an icon in the extras drawer. For example if your package consists of a series new panels for the Formulas roll, then there is no reason to create an icon in the extras drawer. Similarly, you might use an auto part to provide a panel in the Prefs roll.

Another use for auto parts is to provide data for another application. For example, you might have a game with various levels or screens. Rather than build all this data into one gigantic package you might break this data up into separate auto parts. This allows your users to install only the levels they need.

Store parts might seem like a more natural part type for providing data. Apart from the fact that NTK doesn't currently support creating store parts, soups have various disadvantage and advantages as a means of storing data. This topic is discussed further in the "Lost In Space" article. (this issue of PIE Developers Magazine)

It's not so clear what dispatch-only auto parts are useful for. A dispatch-only auto part gets one shot at doing something. The changes a dispatch-only auto part makes are lost when the Newton is reset (except for changes made to soups). Unlike non dispatch-only auto parts, it won't stick around so its `InstallScript` isn't run every time the Newton is reset. One possible use for dispatch-only auto parts might be to create a soup. Once the soup is created there is no need for the auto part to remain installed.

Accessing Data in Auto Parts

Auto parts have no base view, no application name, and no app symbol. This means you can't get to their data as you would with form parts `GetRoot().(kAppSymbol)`.

If you want the data in your auto part to be accessible then its `InstallScript` will have to have code to make it available in some way. Consider the scenario of a form part that is a game and an auto part that provides data for the game. The `InstallScript` of the auto part can put the data into the base view of the main application as show below.

```
InstallScript := func(partFrame,removeFrame)
begin
  local gameView := GetRoot().(kGameAppSymbol);
  if gameView  then
    gameView.(EnsureInternal('Level42Data)) := partFrame.partData;
end;

RemoveScript := func(removeFrame)
begin
  local gameView  := GetRoot().(kGameAppSymbol);
  if gameView  then
    RemoveSlot(gameView ,'Level42Data);
end;
```

This assumes that the application using the data is already installed. This may or may not be a reasonable assumption. Another approach is to use a global variable. You want to keep the number of global variables to a minimum. I suggest you keep it to one a frame. Below is some representative code.

```
InstallScript := func(partFrame,removeFrame)
begin
  local gData := GetGlobals().|Game:PIEDTS|;
  if not gData then
    begin
      gData := EnsureInternal({});
      GetGlobals().(EnsureInternal('|Game:PIEDTS|) := gData;
    end;

  gData.(EnsureInternal('Level42Data)) := partFrame.partData;
end;

RemoveScript := func(removeFrame)
begin
  local gData := GetGlobals().|Game:PIEDTS|;
  RemoveSlot(gData,'Level42Data);
  if Length(gData) = 0 then
    RemoveSlot(GetGlobals(),'|Game:PIEDTS|);
end;
```

In this example there is a single global variable (named using our signature to ensure its uniqueness) that holds the data for the levels that the various auto parts provide. The `InstallScript` creates this global variable if necessary. The `RemoveScript` eliminates it if we remove the last datum.

Auto Part Warnings and Gotchas

Remember, the `partFrame` of an auto part is not copied into internal RAM before being executed. That means the code in the `InstallScript` as well as the data in `partData` reside in your package. Remember this when you're mucking with the `removeFrame`. For example, if your `InstallScript` creates a slot in the `removeFrame` for your `RemoveScript` to use, then it had better be sure to use `EnsureInternal` on the slot name and possibly the contents of the slot as well.

```
removeFrame.(EnsureInternal('foo)) := EnsureInternal(…);
```

In some circumstances, a simple `Clone` may suffice for the contents of the slot it depends on what your `RemoveScript` will be doing with the object.

On the other hand, the `RemoveScript` and the `removeFrame` **are** copied into internal RAM. They have to be because when the `RemoveScript` executes your package will no longer be available. This means you normally do not need to use `EnsureInternal` in your `RemoveScript`.

The topic of exactly what `EnsureInternal` does and when to use it is complex. It is explained in detail in the "Newton Still Needs the Card You Removed" article. (February 1994 issue of Double-Tap Magazine, pp. 12-18)

Other Part Types

Book parts are thoroughly documented in the Newton Book Maker manual so I will not discuss them here. The remaining part types (font, dictionary, and store) are much more straightforward to describe since they only provide data. They do not normally contain any NewtonScript code (only form parts and auto parts have install and remove scripts)

NTK 1.0.1 does not currently support the creation of any of the other part types so I will only describe them briefly.

Font Parts

Font parts install new fonts on Newton when they are loaded. See the latest "Monaco Test" PIE DTS sample code for an example of a font part and how to use. You may find the font part provided with this sample especially useful since it's a mono-spaced font. Currently, none of the built-in Newton fonts are mono-spaced.

Dictionary Parts

Dictionary parts provide a way for you to provide custom dictionaries. Currently dictionaries of your own creation must reside in the NewtonScript heap. Dictionary parts will allow the dictionary data to remain in your package.

Store Parts

Store parts allow your package to contain a read-only store. This read-only store contains soups with the data of interest. Unfortunately, the four character code for store parts is soup. This, plus the fact that store parts are only useful in that they provide soups, will undoubtedly lead to them being erroneously called "soup parts."

Multi-Part Packages

What Are Multi-Part Pacakges

As their name implies, multi-part packages are packages containing more than one part. If a multi-part package contains two form parts, then when it's installed the two icons for the form parts will show up in the extras drawer. The net effect will be the same as if you had created the two form parts in separate packages and then installed both packages.

Creating a multi-part package only serves to bundle parts together for purposes of installation and removal. It does not link any code together. It does not give the parts any special way to access each others data.

How To Create Multi-Part Packages

NTK 1.0.1 creates multi-part packages by merging packages together. You specify the packages to merge by putting them in the same folder with your project and renaming them `"include.pkg"`, `"include.pkg1"`,`"include.pkg2"`, ...

NTK will create a single package containing the part from the current project followed by the parts in the various included packages in order by their file names. The ordering of parts in packages is not normally of interest. However, part ordering does determine the order in which the parts are installed on the Newton. For form and auto parts this determines the order in which the install and remove scripts are called.

Package-Wide Settings

Certain options in the NTK Settings dialog (package-name, compression, copyright, version, copy-protection) apply to a package as a whole, not to its individual parts. Therefore, when building a mulit-part package, the values for these options are taken from the the project being built ignoring the corresponding options used when the various `"include.pkgx"` files were built.

When To Create Multi-Part Packages

I don't think we have enough experience developing Newton applications yet to give a definitive answer as to when multi-part packages are appropriate. My feeling is they'll turn out to be less frequently useful than you might initially think.

For example, one use that probably comes to mind immediately is to bundle together your application with its documentation, a book. This has the advantage of giving your users one thing to install and one thing to remove. The disadvantage is that it takes away the flexibility of allowing experienced users to save space by installing your application without the book.

Another case where it makes sense to use multi-part packages is when your application is broken up into modules for development purposes. In order to speed up your build cycles, some of you with very large projects may have started breaking up your program into separate modules (packages) and compiling/downloading the individual modules separately to save time. The advent of auto parts make this strategy even more appealing because each of your modules doesn't have to have an icon in the extras drawer. During the development cycle you probably want to keep all the modules as separate packages. However, now that you can create multi-part packages you can opt to bundle all the parts together for your final build as opposed to merging the packages together by hand.

In general, I advise carefully thinking about your decision to use multi-part packages. If you are shipping your product on a floppy you might even consider providing it in various configurations and letting your users decide what is most convenient.

Unique Name Generation and Multi-Part Packages

Multi-part packages throws a spin on the original DTS guidelines for generating unique names. For example, if your package contains two form parts you can't have them both use your package name as their app symbol. This is pretty obvious stuff. In practice, I think such conflicts will be a rare occurrence. The original guidelines should still work pretty well.