Using Unit References to Speed Application Development

by Bob Ebert, Newton DTS, Apple Computer Inc.

The Problem

NTK is getting faster with each release. With NTK 1.6.x on a PowerMac, the compilation phase of even large projects is amazingly short. Unless you have to do it over and over and over again, all day long. What's worse, no matter how much faster NTK gets, it still takes time to download the package, and a PowerMac won't help with that. I find it annoying to have to wait a minute or two or twenty to see how a tiny change plays out in a large package.

Wouldn't it be great to be able to split an application into smaller projects? You could build each project separately, saving time both during compilation and during downloading, since only the smaller part you just edited would need to be built and downloaded. Separating an application into pieces could also allow a team of programmers to work on a project. Each engineer would produce one part, hack at it until it's working, then share it with the team.

Haven't We Solved That Problem?

Newton Technology Journal volume 1, number 4 , [199?], contained an article called "Small Parts: A Faster Way to Develop Large Applications" which addressed this problem. It showed how to split your application into separate modules that could be compiled and downloaded separately, but which would still work together. The small effort to split your application up was typically recovered in the first day of building and downloading smaller pieces. The DTS 1.x Q&As also go into detail on how to split up an application.

One of the drawbacks to using those approaches is that you have to edit your code to make the connections work properly, then edit it again when producing a final version. You also need to create global variables, and write code to hook these globals into your app. Even after mastering all that, getting data from another package into a template's _proto slot is very tricky.

This article builds on those techniques. Using the unit reference feature added in the 2.0 release of the Newton OS, you can split a large application into pieces which can be built and downloaded separately, or combined into a single part. What's more, the source files can now work either way without modification.

Background: Unit References

New in the 2.0 release of the Newton OS is the ability for one package to directly reference data in another. Actually, you've been able to do this all along using run-time references found in global variables, slots in the root view, or other places. What's new is that you can now have NTK compile in references to data in other packages—so you won't need to use any NewtonScript heap space to make the connections.

You may have heard of so-called *magic pointers*, which are references in your packages to objects in the ROM. What's "magic" about these pointers is that the objects themselves are in different locations in the various ROM releases. Yet your package still manages to find the right value, no matter which ROM it loads in. Magic!

It's not really magic, of course. Magic pointers work like handles, there's a double-dereference involved. Every ROM puts a lookup table for magic objects in a special place, and the OS looks up the real reference via the table when the magic pointer is used.

Unit References Work Like Magic

Unit references give the parts in your packages the ability to contain objects that can be referenced like magic pointers. Package A can create an array, frame, or binary object and export it. Package B can have a reference to that object, and the OS will make sure that when B looks for the object, it finds it, so long as A is installed. No matter where in memory A happens to be!

Unit References are more magical than magic pointers. There's no way of knowing in advance where in memory an exporting package will be located, so there's no easy way to locate the table of references for a given part. But it happens, and your importinpackage finds the correct object.

Now let's clean up the terminology. A *unit* is a group of zero or more objects, identified by a unique symbol as well as a major and minor version number. A *part* is what NTK typically produces, for example and application or auto part, and a part can export and import zero or more units. Parts go in *packages*, with zero or more parts per package. (But a package with zero parts isn't good for much besides debugging the OS.) Remember that it's at the part level that units are imported or exported, not the package level.

Unit references are great for all kinds of applications. Any time a bunch of applications need to share some read-only data, code, or

objects, you should consider using unit references. One alternative is putting copies of the shared data in each package, which wastes memory. Another alternative is using a soup, but that's typically complicated because you need to write code to make, find, and use the data at run-time. Large data objects, shared prototypes, widely used functions, or even modules from other programmers are all things that might be shared via unit references. This article focuses on using them during development to speed the build/download/test cycle.

For more details on the API for unit references, as well as documentation, supporting functions, and a cool example, check out the "Moo Unit" sample code by Mike Engber. "Moo Unit" is distributed with the DTS sample code.

Background: An Application

Each layout or user proto in NTK normally produces only a single object. That object is made available to the rest of the project through the build-time constant function `GetLayout("filename")`.

It is possible to create layouts in NTK that produce more than one value. BeforeScripts or afterScripts in the templates may create other constants, build-time global variables, or cause other side effects. While this can sometimes be handy, I think it's bad form for one layout to rely on "side effects" from compilation of some other layout. Layouts and protos are primarily declarative—they create an object—and relying on side effects of that object's construction can be confusing. It's easy to avoid programming this way by creating text files of common objects used by more than one layout.

Text files in NTK can be thought of as "nothing but side effects." They create global variables like the `InstallScript`, or constants like the localization frame that are used in other parts of your application. Even with text files it's usually a good idea to have each file produce a small, well-defined set of values. This set can be thought of as the "interface" between that file and the rest of the application.

User protos in NTK actually do a little bit more than simply create an object at build time. They also clue NTK into the fact that some new prototype object exists, which allows NTK to put an item in the "User Proto" popup in the palette. This currently doesn't buy you anything other than the ability to drag out a user proto in the layout view. We'll come back to this later.

Linked Layouts in NTK can be thought of as a special case of user protos. Unlike user protos, linked layouts constrain things so that a layout can only be placed in a project once. We'll come back to this, too.

Breaking a Project Into Smaller Parts

I'll assert that all the inter-file connections made while building your project are accomplished via build-time constants. This may not always be so, but it's a useful way of thinking about your projects, especially for the purpose of splitting it into pieces.

Any place where an object is shared only via a constant is a good place to split up an application. You do this by moving the protos, layouts, or text files out of the main project and into a new project of their own. This new project will create an auto part that exports the shared objects.

That's really all you need to know. That idea, along with the unit reference documentation, will let you break your big packages into smaller ones that can be downloaded individually, vastly reducing the time it takes to build, download, and test any one of them.

But keep reading. The rest of the article will describe one way to create the interface between the projects so that no existing code needs to change. It will also describe some things I do to help with debugging a project built this way.

Constant Agonizing

For constants provided via text files, either with `DefConst` or the `constant` keyword, both the code that defines the constant and the code that uses the value of the constant are using the same symbol. This seems obvious. It wouldn't work any other way!

Less obvious is that this is true for layouts and protos as well. Earlier I mentioned that NTK provides access to layouts and protos through the build time constant function `GetLayout`. The old way was to use a constant named `layout_`*`filename`*, and this is still supported. In fact, NTK 1.5 and 1.6 use the constant named `layout_`*`filename`* to implement the `GetLayout` function. The only thing a layout or proto really produces is a constant with the special name.

When sharing an object via unit references, you could name the reference anything you like. However, the symbol that's the name of the constant, e.g. `'layout_`*`filename`*, turns out to be an excellent choice. It's a good choice because all your code that uses the object is already written to use that symbol, and the name in the unit

reference declaration will be the only common between the exporting project and the importing project.

After the project is split, the code that defines the constant is in a different project than the code that uses it, so we'll actually create **two** constants, one in each project. By using the same symbol for the names of them in both projects, none of the existing code needs to be edited.

Build-time global variables don't fit into this scheme. That's OK, because build-time globals aren't the right tool for this kind of project design. Build-time constants fill the same need, and are handled better by the compiler. There are times when a build-time global variable is the right thing to use to solve a problem, but those cases don't require the global to be shared between projects, so they're irrelevant to this article.

Using Unit References

The core of the unit reference mechanism is implemented by three functions. `DeclareUnit` tells NTK that a unit is being used, the major and minor version of the unit, and the names of the objects within the unit. It must be called by both the importing and exporting projects. `DefineUnit` defines the objects that the unit will contain, and is called only by the exporting project. `UnitReference` gives you a "magic" reference to an imported object that will be hooked up at run time by the OS. `UnitReference` is needed in the importing application, though it can be used by the exporter as well.

`DeclareUnit` requires a declaration frame. This is a frame that declares what will be shared. The names of the slots provide the names for the objects in a unit, and the values of the slots are unique integers. The exporting project must provide a frame with unique sequential integers starting with 0. Importing projects are allowed to have gaps. This allows you to keep some objects "private" by removing their entries from the declaration frame when given to an importer. Read the unit reference documentation in the Newton 2.0 Q&As or the "Moo Unit" sample code for more detail on how unit references work.

The Interface File

Since we're using units for development only, we don't need to worry about which objects are public and which are private. The exporting project and the importing project will share the same declaration frame for a unit. It's convenient to put the declaration frame and the code that uses it into a text file, which I call an *interface file*.

The interface file we'll create will be used in both the exporting project and the importing project. Again, this isn't a requirement for using unit references with NTK, but it's a very convenient way to make sure the two parts stay in sync. Here's what an interface file will contain:

```
constant kPart1Sym := '|Part1:EBERT|;

constant kPart1Declaration := '{
   layout_protoFoo: 0,
   layout_AboutSlip: 1,
   kMungeAStringFunc: 2,
};

DeclareUnit(kPart1Sym, 1, 0, kPart1Declaration);
```

The symbol in `kPart1Sym` is the name of the unit, and must be unique in the Newton, so a registered signature is used. You should put this same symbol in the app symbol field of the auto part's project preferences. I'll tell you why in a moment.

I'm specifying three things in this unit: a user proto, a layout, and a constant that happens to contain a function. The unit has major version 1, and minor version 0. I never change these numbers during development.

Here's some additional code that I put in my interface files:

```
if kAppSymbol <> kPart1Sym then
   // building importing project, so create constants
   begin
      DefGlobalFn('ImpureUnitRef,
         constantFunctions.UnitReference);
      foreach slot, value in kPart1Declaration do
         DefConst(slot, ImpureUnitRef(kPart1Sym, slot));
   end
```

This code creates a constant for each slot in the unit reference declaration frame, but only for importing projects! The exporting project already has the constants in the text files, layouts, or user protos. That's why we made the unit symbol the same as the appSymbol—testing `kAppSymbol` against the unit symbol is an easy way of telling if the code is being compiled in the exporting or an importing project.

This code also does something unusual. `UnitReference` is a constant function, and so it can only be called with constant arguments. In order to make the loop work properly, we need to call the function with the `slot` variable from the loop. So we cheat and define a new "normal" global function called `ImpureUnitRef` that's the same as the

constant function `UnitReference`. This trick works in NTK 1.5 and 1.6, but it's not guaranteed to work in the future.

Note that exactly the same thing could be accomplished, in a supported way, like this:

```
DefConst('layout_protoFoo,
   UnitReference(kPart1Sym, 'layout_protoFoo));

DefConst('layout_AboutSlip,
   UnitReference(kPart1Sym, 'layout_AboutSlip)

DefConst('kMungeAStringFunc,
   UnitReference(kPart1Sym, 'kMungeAStringFunc)
```

This may seem simpler, and even shorter for this example. However, I don't do it this way because it requires me to edit more code every time I add, remove, or change the name of a shared object. If you use the loop, the only thing in the interface file that needs to be edited as the projects change is the `kPart1Declaration` frame. I think this is safe because during development I typically upgrade NTK much less frequently than I edit the contents of my units.

The Exporting Project

There's a little more code that needs to be written to make it all hang together. The exporting project needs to actually specify the objects to export. Create a new text file only for the exporting project with the following code:

```
DefConst('kPart1Objects, {
// <refSym>: <refValue>
   layout_protoFoo: layout_protoFoo,            // old
   layout_AboutSlip: GetLayout("AboutSlip"),    // new
   kMungeAStringFunc: kMungeAStringFunc,
});

DefineUnit(kPart1Sym, kPart1Objects);
```

That's it! You may ask "Why create the constant `kPart1Objects` at all? Why not just put the frame right in the call to `DefineUnit` and be done with it?" That would work fine, but once again I do a little bit more. Here's what else I put in the exporting project's definition file, strictly for debugging:

```
if kDebugOn then
   begin

      InstallScript := func(partFrame, removeFrame)
         begin
            DefGlobalVar(EnsureInternal(kPart1Sym),
               kPart1Objects);
            foreach slot, value in kPart1Objects do
```

```
                DefGlobalVar(EnsureInternal(slot), value);
            end;

        RemoveScript := func(removeFrame)
            begin
                foreach slot, value in GetGlobalVar(kPart1Sym) do
                    UndefGlobalVar(slot);
                UndefGlobalVar(kPart1Sym);
            end;

    end;
```

What this does is create a bunch of run-time global variables. One of the variables will have the same name as the unit, in this case `|Part1:EBERT|`, and will be a frame with all the exported objects. The `UnitReference` function doesn't work at run-time, so without sticking a reference to the exported objects in some easily accessible frame it could be hard to locate them later.

The rest of the globals each have the same name as the objects being exported. Note that I don't include my registered signature in the names of each of these constants. That means there's some chance one of my names will collide with some system object. I typically name things esoterically enough to prevent this. It's unlikely that a system object will be called `layout_protoFoo`, but you should keep the danger in mind if you do the same thing.

You're probably asking "Why even bother creating all those individual constants? Surely having the values available via the one global frame is sufficient?" I create the extra globals because I'm lazy. I like to prototype code in the inspector, to get it more or less working before I make it part of a project. Global variables and constants are accessed using identical syntax. Having the global variable available at run time for the inspector with the same name as the constant that's available at build time for the compiler means I can copy/paste code between the inspector and the project and not edit it. `call kMungeAStringFunc with ("Your Name")` works in either place.

If you haven't caught on yet, I like it a lot when the same code works in different environments or when put together different ways, especially when no editing is necessary. I'm a very lazy programmer.

About User Protos and Linked Layouts

User Protos do a wee bit more than just provide a constant. When a user proto is in a project, NTK knows to put its name in the palette so you can drag one out. After the split, a user proto may no longer be

in the same project as the code that uses it. So what do you drag out?

I drag out a protoFloater instead, then add an after script to fix up the contents of the _proto slot. Any predefined proto would work, but protoFloater just happens to be on the palette and not add any extra default slots. The afterScript looks like this:

```
thisView._proto := GetLayout("protoFoo");
```

The same can be done for linked layouts. After the split, the linked layout may no longer be in the same project as the layout it's linked to. protoFloater to the rescue again! Just drag out a protoFloater instead of a linked layout, and have the afterScript replace the _proto slot:

```
thisView._proto := GetLayout("AboutSlip");
```

This actually does not produce the same result as a real linked layout. We end up with an extra level of _proto inheritance that wouldn't be there with normal linked layouts. It is possible to completely simulate what NTK does when it links a layout. However, fully integrating a declared linked layout requires understanding of the Newton view system declare mechanism, which is beyond the scope of this article. Using a protoFloater is the simplest solution, since it lets you declare the views in NTK, just like you would with linked layouts.

Summary

It takes just a few simple steps to split a project up into separate compilation units that will exist at run time as separate packages and share objects via unit references. The steps are:

Remove the layouts, protos, or text files from the main project.

Put them in a new project that produces an auto part.

Create the interface file containing everything that's shared.

Place it at the beginning of the exporting project.

Create the definition file for the new project.

Put it at the end of the exporting project.

Build and download the exporting package.

Add the interface file at the beginning of the main project.

Build and download the main package.

Everything should end up working exactly as it did before. Notice that you didn't edit any of the "source" layouts, protos, or text files at

all! (Okay, except maybe to clean up the _proto slots for user protos or linked layouts.)

Next Steps

For interim or "beta" releases, you can make the main project a multi-part package that contains all the exporting parts as well as the main part, so your beta users only see one package. Remember that the unit reference mechanism works on a part-by-part basis, so there's no reason the exporting part and the importing part can't be in the same package.

When you're ready for your final build, you can just drop the source files right back into the main project. Put them right where the interface file was, and remove the interface file from the main project. This time you don't have to edit anything at all, even trivially.

There's actually no compelling reason to ever put the files back in a single part. The code will all work fine as a multi-part package. I have a suspicion that performance will improve if everything is in one part. I believe this because unit references must cost something, but I have not measured the costs. The locality of the package (in other words, which objects are next to which other objects) will also change when switching from separate packages to a multi-part package to an all-in-one package, and this can affect performance. I recommend trying the various configurations and choosing the one you like best.

A Bug to Watch Out For

There is a bug with unit references in the 2.0 OS. Sometimes when an importing package is installed before an exporting package, the unit references are not properly connected. When this happens, an exception will be thrown when the bad reference is followed by the importer. You can work around the problem by reinstalling the importing packages. By the time this is published, Apple Computer will probably have released a system update that fixes this bug, so that you can re-download an exporting package many times without touching the importer.

Conclusion

With the invention of unit references, it's now easier than ever to split a large application into separate compilation units. What's more, these units can be put in individual packages and downloaded separately. Over a full project development cycle, this could amount to days or even weeks of your time, much less than the time needed

to split up the application. The technique also encourages a good coding discipline, and can be used to allow teams of programmers to work together on large applications.