Newton Technology JOURNAL

Volume II, Number 5 October 1996

Inside This Issue

NewtonScript Techniques

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 , August 1995, 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

continued on page 3

Newton Communications

Newton 2.0 Messaging Enabler - Get Your Messages Movin'

by Jason Rukman, Apple Computer, Inc.

Connectivity! Do you carry a cell phone, pager or possibly some other device to stay in touch? If you do, then you would probably like to transfer some of this information onto your Newton PDA. When I have my Newton with me I'd like the information on my pager to go directly to my Newton.

The Messaging Enabler is a 2.0 transport that solves this problem by providing a framework specifically designed for 1-way and 2-way wireless messaging. With the Messaging Enabler, developers are able to get a messaging device working quickly, easily and consistently with the Newton. The messaging device will be integrated with the rest of the Newton system and Newton applications that support routing.

For the Messaging Enabler to be useful, it requires a plug-in driver, called a message module, for a particular hardware device. A message module must implement a set of APIs for communicating with the particular hardware device and the Messaging Enabler looks after the rest. (This API is distributed with the Messaging Enabler development kit.)

Your specialty may be developing communications software. The idea behind the Messaging Enabler is to remove the user

continued on page 6

Newton Technology Journal

Volume II, Number 5 *October 1996*

Published by Apple Computer, Inc.

Jennifer Dunvan • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

Gabriel Acosta-Lopez • *Coordinating Editor, DTS and Training Content*

Technical Peer Review Board
J. Christopher Bell, Bob Ebert, David Fedor, Ryan Robertson, Jim Schram, Maurice Sharp, Bruce Thompson

Contributors
Bob Ebert, Cami Hsu, Julie McKeehan, Neil Rhodes, Jason Rukman, Bill Worzel

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

Will Iverson • *Technical Production Review*

InfoGraphix • *Design/Production*

Editor's Note

Letter From the Editor

by Jennifer Dunvan

The Excitement is Building

I just returned from Macworld Expo/ Boston, where I met a good number of enthusiastic Newton developers at Developer Central™. Granted, I took some heat for demoing Newton Toolkit for Windows on a Pentium (at a Macintosh Expo no less), but as the implications of making Newton development available to the Windows community began to dawn on folks, their eyes lit up. “This is a very good thing” they said. I have to agree – the possibilities are endless, and the excitement is growing.

Speaking of excitement, I want to invite each of you to the upcoming Fall Newton Developers Conference on November 4, 5, and 6th in San Francisco, California. Bring a hat to hold onto, as there will be plenty of extremely interesting news, technical presentations, hardware demos, and labs. Be sure to bring your existing code, for reasons you'll hear about later. Be prepared to be **blown away.** (I'm really not very good at keeping secrets, can you tell?) For conference registration information, please visit the Newton Development Home page at http://devworld.apple.com/dev/newtondev.shtml

Speaking of secrets, you've probably heard by now that Apple has wisely formed the Information Appliances Division, of which Newton is a part. Jim Groff was named Senior Vice President of the division by Dr. Gil Amelio in early June. Mr. Groff has a strong and vocal commitment to Newton and a rich history with Apple which spans a wide range of product and market areas, from Internet servers to Education. Sandy Benett was formally named as Vice President of the Newton Systems Group (he was acting in the position for quite some time), and another new face is that of Barbara Groth, who now heads up the Newton Solutions Marketing group (formerly known as Developer Relations). You'll be hearing more detail on the structure of this key group in months ahead.

In this issue of the Newton Technology Journal, we bring you coverage of Newton Communications in the form of Messaging Enabler and FDILs articles. We heighten your programming finesse with articles on User Interface tips and how to shorten your development cycle using unit references. In the debugging arena, you'll find advanced techniques using mock entries, and an excerpted article on debugging techniques from the newly updated book “Programming for the Newton, 2nd Edition” by Julie McKeehan and Neil Rhodes. And we bring you important information on the long-awaited Newton Connections Utility for Newton 2.0.

Keep your eyes open for December's issue, which no Newton developer will want to miss. To all of you who spoke with me at Macworld Expo, thank you for your product demos, and to all Newton developers, thank you for your enthusiasm, your belief in your products, and your ongoing commitment to Newton. I am confident that all of your hard work is about to pay off. The future is very bright on the Newton front.

See you at the developer conference!

Jen Dunvan
dunvan@newton.apple.com

continued from page 1

Using Unit References to Speed Application Development

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 importing package 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 an 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 commonality 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.

NTJ

continued from page 1

Newton 2.0 Messaging Enabler - Get your Messages Movin'

interface requirement from the Original Equipment Manufacturer (OEM). This allows a consistent user interface for all wireless messaging products that use the Messaging Enabler.

Is it for you?

Of course, the Messaging Enabler won't be suitable for every messaging system. It has been designed to enable as many of the wireless messaging systems as possible. If you want to enable a piece of hardware for the Newton platform, then it may be worth using the Messaging Enabler if the hardware can receive, and possibly send, messages wirelessly (for instance pagers, wireless PC cards, etc.).

A good understanding of routing is recommended for developers using the Messaging Enabler or writing a message module. Although an understanding of transports may be helpful when writing a message module, it is not required. You can find information on routing and transports in the "*Newton Programmers Guide: Communications.*"

The Messaging Enabler does most of the work for you, so that you can have your hardware up and running as soon as possible. The exciting features of the Messaging Enabler are covered below.

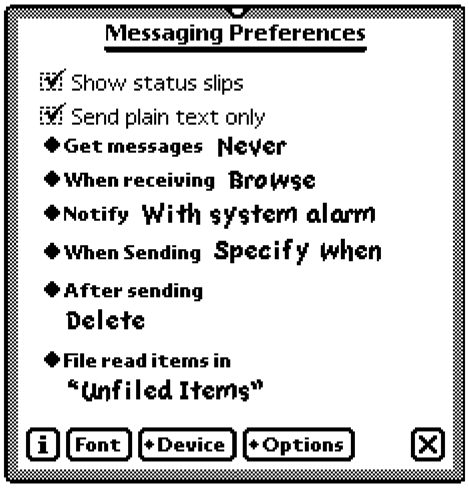
Message modules have many options that may be customized so you can support features specific to the messaging device being used.

Most Newton applications that support routing can use the messaging device via the Messaging Enabler with no changes. This is due to the NewtonScript routing mechanism. Applications can also be designed to control the Messaging Enabler directly which is ideally suited for applications targeted to a vertical market.

Features

The following list of features should give you a better idea of some of the functions provided by the Messaging Enabler:

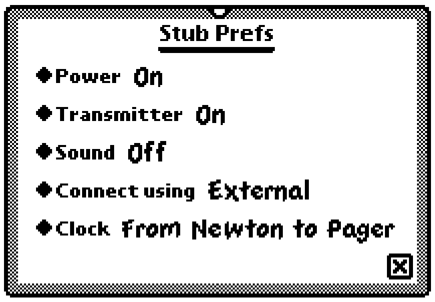
Standardized preferences for wireless messaging.


Messaging Preferences

☑ Show status slips
☑ Send plain text only
◆Get messages Never
◆When receiving Browse
◆Notify With system alarm
◆When Sending Specify when
◆After sending Delete
◆File read items in "Unfiled Items"

Figure 1: Messaging Enabler Preferences

Device specific preferences. Note that not all of these preferences will necessarily be displayed for each device.


Stub Prefs

◆Power On
◆Transmitter On
◆Sound Off
◆Connect using External
◆Clock From Newton to Pager

Figure 2: Device Specific Preferences

Send routing slip(s).

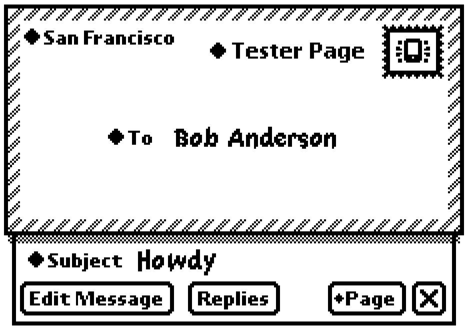


Figure 3: Paging Routing Slip

Paging address data definitions. This extends the in-built address data definitions.

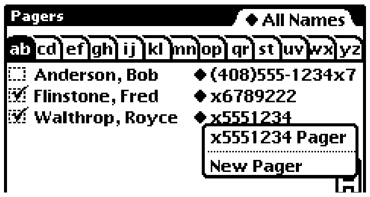

Pagers
◆All Names
ab cd ef gh ij kl mn op qr st uv wx yz
Anderson, Bob ◆(408)555-1234x7
Flinstone, Fred ◆x6789222
Walthrop, Royce ◆x5551234
x5551234 Pager
New Pager

Figure 4: Pager Address Listpicker

Message replying. Some 2-way messaging devices can reply to received messages, for instance the Motorola Tango™.

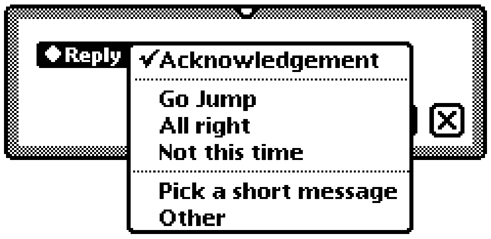
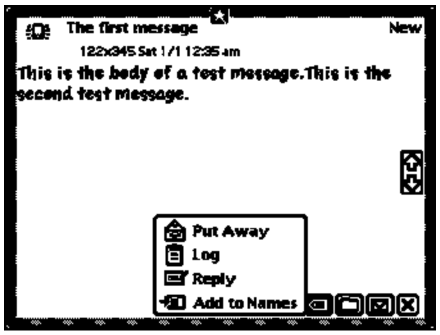


Figure 5: Message Reply Picker

Automatic message retrieval. Automatic control and combination of multi-part messages.

Text message viewing/editing/and put away.

The following message shows an item in the In Box made up of two combined messages.



This is the body of a test message.This is the second test message.

Figure 6: Displayed In Box Message

Manage and control the I/O Box.

User status feedback.

How it ticks

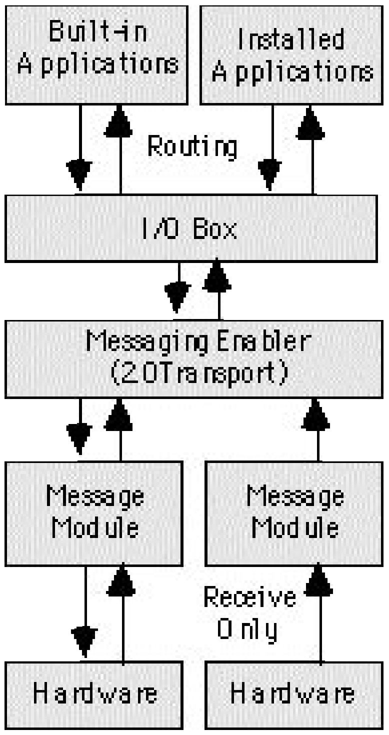


Figure 7: Messaging Enabler Hierarchy

As this diagram shows the Messaging Enabler fits in at the same location in the Newton communications layering as a 2.0 Transport.

Installing a message module

To add a message module to the system, you create an auto part. This means that when a message module is installed, it adds services to the system but does not add an application to the extras drawer; however, an icon is added to the "Extensions" folder. You define a message module based on the prototype, `protoMsgModule`. To add the defined message module to the system, you call the `RegMsgModule` platform file function from your auto part's `InstallScript` function with the template you have defined.

```
call kRegMsgModuleFunc with (
  kAppSymbol,
  partFrame.partData.TestEnabler
);
```

To unregister the message module you need to supply two part frame functions: DeletionScript and RemoveScript. The DeletionScript function will call the DeleteMsgModule platform file function to remove any preferences for the message module, and also ensures your message module RemoveScript is called.

```
SetPartFrameSlot(
  'DeletionScript,
  func() begin
    call kDeleteMsgModuleFunc with (
      kAppSymbol
    );
  end
);
```

In your `RemoveScript` of the auto part you should deregister the message module by calling the `UnRegMsgModule` platform file function:

```
call kUnRegMsgModuleFunc with ( kAppSymbol );
```

Working with callbacks and events

Most of the methods defined in `protoMsgModule` take a callback as one of the parameters. The Messaging Enabler will call methods that you override in `protoMsgModule` when the Messaging Enabler needs to perform a particular operation. For example: When the Messaging Enabler needs a message from the message module, it may send the `GetNextMessage` message, which could be implemented as follows:

```
GetNextMessage := func( callBack ) begin
....
.... // go get the next message
....
:doEvent(
  kEV_PROGRESS,
  { type: 'vBarber,
    statusText: "Almost done..."
  }
);
....
:doCallBack(
  callBack ,
  kRES_SUCCESS,
  message  // your retrieved message
);

end; // GetNextMessage
```

Note that the calls to the internally defined methods of `protoMsgModule`, `doEvent`, and `doCallBack`. The method `doEvent` was used to change the status display. The Messaging Enabler provides a default status display but by sending events to the Messaging Enabler this can be customized. Sending the `doCallBack` message is required to inform the Messaging Enabler when the operation for `GetNextMessage` has been completed. This also returns the result from the requested operation.

You may send other events to the Messaging Enabler to let it know when certain things happen. For example, if a new message has been received, you would send a `kEV_MESSAGE` event to alert the Messaging Enabler to read the message. You would do this by sending the `doEvent` message.

Need to send messages?

To support sending you need a `SendOptions` frame and a `SendMessage` method. The `SendOptions` frame defines options for sending messages. The Messaging Enabler will call the `SendMessage` method when a new item needs to be transmitted. The main slot required for the `SendOptions` frame is `routeSlipType`. This defines the addressing type to use when sending. The Messaging Enabler adds a paging data definition to the system. For more information about data definitions, please see the "Stationery" chapter in the *Newton Programmers Guide: System Software.*

A new item will be added to the action picker based on the `SendOptions` frame contents. A typical `SendOptions` frame might be similar to the following:

```
{ routeSlipType: '|nameRef.people.pager|,
  replyTypes: [ 'ack, 'user, 'canned ],
  dataTypes: [ 'text, 'frame ],
  group: 'page,
  groupIcon: ROM_RoutePageIcon,
  groupTitle: "Page"
}
```

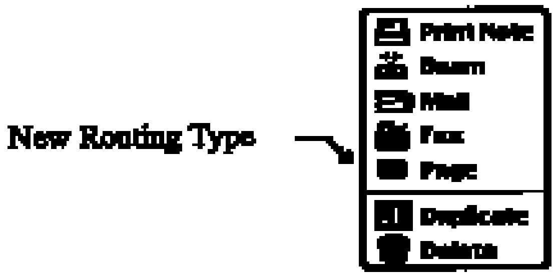


Figure 8: Notepad Routing Picker

This means that any Newton application that supports routing for either `'text` or `'frame` datatypes will now be able to send this data as a page. See the "Routing Interface" chapter in the *Newton Programmers Guide: Communications.*

What's your preference?

The Messaging Enabler provides several different mechanisms for controlling user preferences. The main preference slip as seen in Figure 1 contains several items that will only be visible if your message module overrides certain prototype slots. For example, the first option, "When receiving," will only be visible if your message module sets the `dirSupport` slot to `true`. (Note, however, that this labelpicker may still be visible if another installed message module has this set.)

A separate view displays the hardware preferences for each messaging device. The Messaging Enabler provides five generic preferences that you may use as seen in Figure 2. These preferences are very easy to set up. All that is needed is an array of strings that become the options for each preference (such as the `labelCommands` for the `labelPicker`). For example, you could set `soundStrings` to the following array:

```
[ "Off", "Tunes", "Annoying", "Loud" ]
```

to correspond with the hardware options for the particular messaging device. Note that the first array item will be the default for each of the preferences, so it is important to make the most reasonable preference setting the first item in the array. The Messaging Enabler determines when these preferences need to be set and will call the `SetConfig` method of the message module at the appropriate time.

A third way to provide user preferences gives more customization control, but also requires more work. Provide your own preference view template. You might need to do this if there is some special setting that is not covered by any other preference controls. You supply this view template in the `prefsTemplate` slot of your message module.

As you can see, there are several levels of control for the user preferences. In most cases, it is important to remember that less is often better. Most users work better with devices that function in an expected manner, rather than having to set a bunch of preferences to get them to work a particular way.

Controlling the Messaging Enabler.

The Messaging Enabler may also be controlled by an installed Newton application. This feature is intended primarily for vertical applications (such as a health-care dispatch application) that would need to set the preferences explicitly.

To change the Messaging Enabler preferences an installed Newton application would call the `TransportNotify` global function.

For example:

```
TransportNotify(
  'MsgEnabler,
  'ChangeConfig,
  [ callBack,
    { disable: true,
      autoStatus: nil,
      hideItems: nil,
    },
    { deviceSym: 'MM_msgModule,
      powerIndex: 1,
      portIndex: 2
    }
  ]
);
```

This does the following:

- Disables the user access to the Messaging Enabler preferences so they cannot be changed.
- The status dialog will not be automatically opened. (The user can still see the status if it is selected from the Notify Icon at the top of the screen.)
- The Messaging Enabler items will not be displayed in the I/O Box.
- The installed message module `MM_msgModule` will have its preferences set to the second item in the `powerStrings` array and the third item in the `portStrings` array.

As you can see, this gives an application the necessary control over the Messaging Enabler. There are many other preferences of the Messaging Enabler that can also be set in this way.

Note that this function is designed to be integrated with a single Newton application and is ideally suited for vertical market applications. If two separate Newton applications were to attempt this operation, the Messaging Enabler preferences would be set to a combination of these two applications and the results would be unpredictable.

Give me those In Box items!

So how does an installed Newton application get the items from the Messaging Enabler once they are in the In Box? Because the Messaging Enabler is a transport, any installed application can receive items from the messaging enable using the standard Newton routing APIs.

Please refer to the *"Newton Programmers Guide"* for a description of the different mechanisms available for routing items from the In Box, specifically `RegInBoxApps`, `RegAppClasses`, `PutAway` and `AutoPutAway`.

* And remember that the names of the innocent have been changed to protect the guilty.

Editing support; R. Robertson, J.C. Bell & A. Weiss.

NTJ

Advanced Techniques

Mock Entries for Debugging

by Bob Ebert, Newton Developer Technical Support.

The Problem

The NSDebugTools package, part of NTK 1.6, allows you to set break points in NewtonScript code. This is very powerful and will help you find many of your bugs which don't raise exceptions. However, once in a while what you need to know is when some object is accessed, not when some line of code is executed. One way to do this is to set the global variable `trace` to `TRUE`, but then you typically have to wait a long time for the trace output to stop scrolling by, then wade through reams of data to find the accesses you're interested in.

Mock Entries to the Rescue

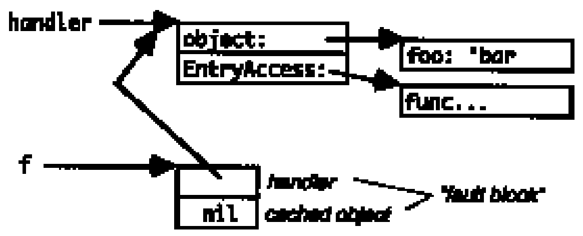
A little-known and even less frequently used feature that was added to the Newton OS in the 2.0 release is the ability to create entry-like objects, called *Mock Entries*. These objects are composed of two parts. One part is a simple NewtonScript frame, often called the *cached entry*. The other part is a *handler* which knows how to create and save the cached entry. The parts together are sometimes called a *fault block*; when something needs the cached entry and it doesn't exist, a *fault* occurs and a message is sent to the handler, which then creates or *faults in* the entry.

The Newton OS in 2.0 allows you to create these fault blocks for your own objects, which means you get to write the code that executes when the cached object needs to be faulted in. The code that creates the cached entry can also do other things, for example it could enter a breakloop, which would give you a chance to look at the stack and see what's accessing the frame.

Here's a simple example of using a fault block for debugging. We'll create a simple frame that prints an exclamation point in the inspector and beeps the speaker every time it's accessed.

```
handler := {
  object: {foo: 'bar},
  EntryAccess: func(mockEntry)
    begin
      write("!");
      GetRoot():Sysbeep();
      object;
    end,
  };
f := NewMockEntry(handler, nil);
```

`f` is now a Mock Entry. Here's how it looks to the OS:



When any slot in `f` needs to be accessed, the OS checks to see if there is a cached object. Because we passed `NIL` as the 2nd argument to `NewMockEntry`, and `EntrySetCachedObject` hasn't been called, there will be no cached object for `f`. When there is no cached object, the OS calls the handler's `EntryAccess` method, which is expected to create the cached object, tell the OS about it using `EntrySetCachedObject`, and return it.

We cheat and don't call `EntrySetCachedObject`. The OS uses the return value of `EntryAccess` as the object for this access, and since we return `handler.object` everything works, but next time something touches `f`, `EntryAccess` will be called again. There's our hook—every time some part of the OS reads or writes any slot in `f`, `EntryAccess` is called.

To almost all of the Newton OS, `f` looks like a regular frame. `f.foo` evaluates to `'bar`. `ClassOf(f)` is `'frame`. `f.baz := 42` will add a slot to the frame, which is also referenced as `handler.object` in our example, so the next time the frame is accessed, the modified object will be returned. The illusion is complete, only the test function `IsMockEntry`() can tell that `f` is a mock entry and not a normal NewtonScript frame.

An Improvement

If you try this, you'll see that the frame is accessed a lot more often than you might think. Getting the value of `f.foo` calls `EntryAccess` twice. Setting a slot also calls `EntryAccess` twice. Creating a new slot calls it 11 times. The `print` function must do a lot, because printing `f` in the inspector calls the `EntryAccess` method a whole bunch of times.

In our application debugging example above, we really only want to know when the object is being used for the first time in a while. Recall that if the OS finds that a cached object exists, it won't call `EntryAccess` but will simply use the object. So to prevent excessive calls, our `EntryAccess` method will now create the cached object. The trick then becomes clearing the cached object. I've found that clearing the object at a deferred time works well—typically it's cleared the next time control returns to the top level, which is soon enough to catch most bugs. Here's how to do that: (the bold text is new)

```
handler := {
  object: {foo: 'bar},
  EntryAccess: func(mockEntry)
    begin
      write("!");
      GetRoot():Sysbeep();
      EntrySetCachedObject(mockEntry, object);
      AddDeferredCall(
        GetGlobalFn('EntrySetCachedObject),
        [mockEntry, nil]);
      object;
    end,
  };
f := NewMockEntry(handler, nil);
```

You might want to experiment with clearing the cached object at other times.

Limitations

The `EntryAccess` method of the handler object is called only when a slot in the frame is accessed. That is, a statement like `g := f` won't cause the `EntryAccess` method to be called, since no slot in `f` is accessed. The result of that statement will be that you now have two references to the mock entry "fault block", and either `g.foo` or `f.foo` will cause the `EntryAccess` method to be called.

The Newton 2.0 OS only supports creating mock objects that are backed up by frames. While it's not guaranteed, you might experiment and see what happens if the object is an array or a binary object. (But back up your data first!) Try some special case binary objects like strings or real numbers. Depending on your situation, you may be able to use this debugging technique with arrays or binary objects as well as with frames.

I've found that some parts of the OS work normally in this case—the mock object is treated just like a string, bitmap, array, or whatever. Other parts of the OS "notice" that the object is a fault block and not the appropriate object type, which typically causes a throw. The error messages in this case can be interesting. For example, putting the string `"foo"` in the `object` slot of the `handler` will create an object that appears to be a string. Printing works, but functions like `StrLen(f)` or the accessor `f[0]` "notice" that the object isn't a string, and throw with the seemingly contradictory error message: "`Expected a string, got "foo"`." This happens because the exception printer *doesn't* notice that the mock object isn't a string, and so it calls `EntryAccess`, gets the string, and prints it.

Advanced Techniques

You can put any NewtonScript code in the handler, specifically in the `EntryAccess` method, so you can use this trick to do other things. For example, you could add a counter and find out how many times a frame is accessed during some operation. You could add in a test to see if some slot in the frame has changed, and stop when it gets a certain value.

Unfortunately, when the `EntryAccess` method is called you don't have any information about what's happening. You can't tell if a slot is being read or set. You can't tell which slot (or element, or byte) is being accessed. If you write code that watches for changes you end up finding out after the change takes place.

But this can still be useful. Consider if your handler set `trace` to `TRUE` and then set it to `NIL` again in a deferred call. This would do a great job of limiting the trace output to only code that actually used the object. The `EntryAccess` method might also watch for some change and, upon detecting the change, set `trace` to `NIL` and enter a breakloop. That way you'd know without doubt that the last section of trace output was the one you needed to look at.

You might even consider using mock objects to implement a kind of sentinel that lets you know when other applications access your objects.

Conclusion

Being able to have your code execute when a frame is accessed is powerful, and has lots of good uses. However, keep in mind that any observations you or I make about how the OS deals with mock objects should be used only for debugging. That is, it would be a mistake to write production code that relies on `EntryAccess`

NTJ

Debugging Tutorial: Finding & Fixing Bugs in an Application

Julie McKeehan and Neil Rhodes, Academic Press. 1996

Excerpted from Programming for the Newton, 2nd edition

Beeping Button Brouhaha

We've written a very simple application in which the user can write a number. Pressing the "Beep" button makes the Newton beep that many times. Figure 1 shows the application. Figure 2 shows the application templates in NTK.

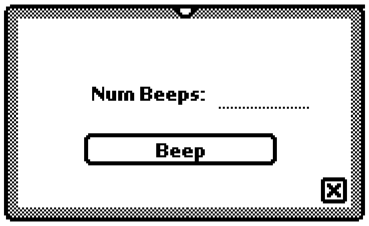
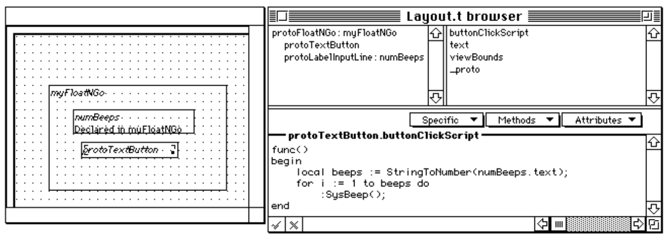


Figure 1: Beeping Button application.



```
protoFloatNGo : myFloatNGo
   protoTextButton
   protoLabelInputLine : numBeeps
```

```
buttonClickScript
text
viewBounds
_proto
```

protoTextButton.buttonClickScript

```
func()
begin
    local beeps := StringToNumber(numBeeps.text);
    for i := 1 to beeps do
        :SysBeep();
end
```

Figure 2: Structure of the Beeping Button project.

First Problem

The first thing that happens when we write in a number and then press the Beep button is a notification on the Newton: "Sorry, a problem has occurred (-48807)." Let's turn on `breakOnThrows` (clicking the icon in the Inspector Window) and press the button again. Now, the Inspector prints out

```
  Undefined variable: numBeeps
  evt.ex.fr.intrp;type.ref.frame
  -48807
(#6008D1D1).buttonClickScript(), 3: Push 'text
Entering break loop: level 1
```

If we look at the code we see that we're trying to access the `numBeeps` variable from our `buttonClickScript`. That variable should refer to our `protolabelInputLine` view. What could be wrong? If we've correctly declared `numBeeps` to the `protoFloatNGo`, the `protoFloatNGo` view should have a slot named `numBeeps`. Let's look in that view for `numBeeps`. (That view is the parent of our current self.).First, let's get `self`:

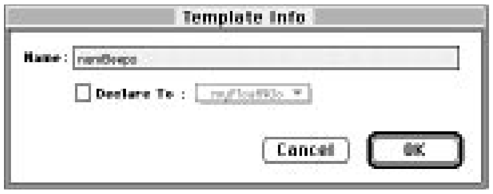
```
// get current self
beepingButton := GetCurrentReceiver(0);
```

```
#440F63D {_parent: {_parent: {#440D221},
       _proto: {#6008D0C1},
       viewCObject: 0x110926D,
       viewclipper: 17863292,
       base: <1>,
       viewFlags: 577,
       viewBounds: {#440F5C1}},
       _proto: {buttonClickScript:<function, 0 arg(s)#6008D239>,
       text: "Beep",
       viewBounds: {#6008D539},
       _proto: {@226}},
       viewCObject: 0x110A530,
       viewFlags: 515}
```

Now, let's get the parent slot and we will have the right view:

```
floatNGo := beepingButton._parent
#440F5DD  {_parent: {minute: 178,
                      downButton: {#440B2E9},
                      calculator: {#4406159},
                      mailEditor: {#44064C1},
                      extrasDrawer: {#4409671},
                      defaultTransport:Newton: {#4405DD9},
                      OutOfMemoryAlert: {#4405D95},
                      notification: {#4405D35},
                      remindSlip: {#44060C5},
                      namesButton: {#44063D9},
                      folderEdit: {#4405DF1},
                      phoneKeyboard: {#4405ECD},
                      ovButton: {#440643D},
                      upButton: {#4406461},
                      thegang: {#44065F9},
                      printerSerialPicker: {#4405D05},
                      ...},
          _proto: {viewBounds: {#6008D199},
                     stepChildren: [#6008D1B9],
                     _proto: {@180},
                     debug: "myFloatNGo",
                     appSymbol: |Demo:NTK.Demo|},
          viewCObject: 0x110926D,
          viewclipper: 17863292,
          base: <1>,
          viewFlags: 577,
          viewBounds:{left:-25, top:173, right:139, bottom:265}}
```

The `numBeeps` slot doesn't seem to be in the `floatNGo`. The view otherwise appears to be correct. It sounds like a problem in declaring. Let's check the Template Info dialog for that template (see Figure 3). Well, well, well. Turns out it actually wasn't declared. We'll checkmark the "Declare To:" checkbox and rebuild.



Name: numBeeps

☐ Declare To : myFloatNGo

Figure 3: Template Info dialog showing undeclared numBeeps.

Second Problem (a Hard One)

We rebuild, download, and rerun. We write in a number and tap the "Beep" checkbox. We get the following error in the Inspector:

```
  Undefined variable: numBeeps
  evt.ex.fr.intrp;type.ref.frame
  -48807
(#6008D229).buttonClickScript(), 3: Push 'text
Entering break loop: level 1
```

This is exactly the same error at the same place we had it before. Let's check the `floatNGo` view again:

```
beepingButton := GetCurrentReceiver(0);
floatNGo := beepingButton._parent
and here is what we find when the Inspector returns our
result:
#4412B0D  {_parent: {minute: 181,
                     downButton: {#440B129},
                     calculator: {#4406159},
                     mailEditor: {#44064A1},
                     extrasDrawer: {#4409651},
                     defaultTransport:Newton: {#4405DD9},
                     OutOfMemoryAlert: {#4405D95},
                     notification: {#4405D35},
                     remindSlip: {#44060C5},
                     namesButton: {#44063D9},
                     folderEdit: {#4405DF1},
                     phoneKeyboard: {#4405ECD},
                     ovButton: {#440643D},
                     upButton: {#440EF4D},
                     thegang: {#44065D9},
                     printerSerialPicker: {#4405D05},
                     ...},
           _proto: {viewBounds: {#6008D1F1},
                    stepChildren: [#6008D211],
                    _proto: {@180},
                    debug: "myFloatNGo",
                    numBeeps : NIL,
                    stepAllocateContext: [#6008D731],
                    appSymbol: |Demo:NTK.Demo|},
           viewCObject: 0x1108C2C,
           numBeeps : {_parent: <2>,
                       _proto: {#6008D5D1},
                       viewCObject: 0x1109F0C,
                       entryLine: {#4419229},
                       labelLine: {#4418E49},
                       width: 73,
                       indent: 75,
                       height: 13},
           viewclipper: 17863746,
           base: <1>,
           viewFlags: 577}
```

The `numBeeps` slot seems to be there and seems to point to what looks like it could be our view. Let's try to access it from the Inspector:

```
floatNGo.numBeeps
#2         NIL
```

That doesn't make sense. We can see that it is there. Let's try another way to get to that view using the `Debug` function:

```
Debug("numBeeps")
#2         NIL
```

Curiouser and curiouser. However, look very closely at the way the `numBeeps` slot prints out versus any other slot:

```
_proto: {viewBounds: {#6008D1F1},
                    stepChildren: [#6008D211],
                    _proto: {@180},
                    debug: "myFloatNGo",
                    numBeeps : NIL,
                    stepAllocateContext: [#6008D731],
                    appSymbol: |Demo:NTK.Demo|},
           viewCObject: 0x1108C2C,
           numBeeps : {_parent: <2>,
                       _proto: {#6008D5D1},
```

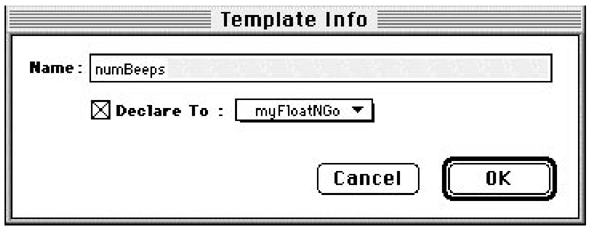
Other slots have no space before the colon (":"), while the `numBeeps` slot has one space there. Could this have anything to do with our problem? What if that space were significant? Let's try calling `Debug` with an extra space after `numBeeps`:

```
Debug("numBeeps ")
```

and here is the Inspector return result that we get:

```
#4418AF5  {_parent: {_parent: {#4412B25},
                     _proto: {#6008D0C1},
                     viewCObject: 0x1108C2C,
                     numBeeps : <2>,
                     viewclipper: 17863746,
                     base: <1>,
                     viewFlags: 577},
           _proto: {viewBounds: {#6008D681},
                    label: "Num Beeps:",
                    entryFlags: 10753,
                    _proto: {@189},
                    debug: "numBeeps ",
                    preAllocatedContext: |numBeeps |},
           viewCObject: 0x1109F0C,
           entryLine: {_parent: <2>,
                       _proto: {#356429},
                       viewCObject: 0x110A83B,
                       viewFlags: 10753,
                       viewBounds: {#4418F4D},
                       text: "2"},
           labelLine: {_parent: <2>,
                       _proto: {#356569},
                       viewCObject: 0x110A871,
                       text: "?Num Beeps:",
                       viewFont: {@100},
                       viewBounds: {#4418E2D}},
           width: 73,
           indent: 75,
           height: 13}
```

So if it acts as though the name had an extra space—maybe it does. Let's check the Template Info dialog for that template more carefully (see Figure 4). Indeed, there is a trailing space after `numBeeps`. We'll delete it and rebuild.



Name : numBeeps
☒ Declare To : myFloatNGo

Figure 4: Template Info dialog for numBeeps with an extra space at the end.

Third Problem

We rebuild, download, and rerun. We write the number "2" and tap "Beep." We get the following error in the Inspector:

```
Expected an integer, got nil
evt.ex.fr.type;type.ref.frame
-48406
(#6008D229).buttonClickScript(), 27: PushConstant NIL
Entering break loop: level 1
```

We're still in our `buttonClickScript`, about to execute the code at program counter 27. Here's the NewtonScript code for the `buttonClickScript`:

```
func()
begin
  local beeps := StringToNumber(numBeeps.text);
  for i := 1 to beeps do
    :SysBeep();
end
```

Let's look at the disassembled code for the `buttonClickScript`:

```
Disasm(GetCurrentFunction(0))
    0: FindVar              numBeeps
    1: Push                 'text
    2: GetPath              1
    3: Push                 'StringToNumber
    4: Call                 1
    5: SetVar               beeps
    6: PushConstant         1
    7: SetVar               i
    8: GetVar               beeps
    9: SetVar               i|limit
   10: PushConstant         1
   11: SetVar               i|incr
   12: GetVar               i|incr
   13: GetVar               i
   14: Branch               23
   17: PushSelf
   18: Push                 'SysBeep
   19: Send                 0
   20: Pop
   21: GetVar               i|incr
   22: IncrVar              i
   23: GetVar               i|limit
   24: BranchIfLoopNotDone  17
   27: PushConstant         NIL
   28: Return
#2        NIL
```

We're about to execute the code at offset 27—it looks as though we're at the end of the loop. Let's look at the values of our variables to see if they are reasonable:

```
GetAllNamedVars(0);
#4419641  {beeps: NIL,
           i: 1,
           i|limit: NIL,
           i|incr: 1,
           numBeeps: {_parent: {#4418345},
                      _proto: {#6008D5D9},
                      viewCObject: 0x110A57D,
                      entryLine: {#44186F5},
                      labelLine: {#44181FD},
                      width: 73,
                      indent: 75,
                      height: 13},
           text: "Beep",
           SysBeep: <function, 0 arg(s) #350E8D>}
```

It doesn't seem right that `beeps` is nil. In fact, that would explain the error we got. The for loop expected an integer, but got nil. But why is `beeps` nil? It was obtained from calling `StringToNumber` on `numBeeps.text`. Let's start by looking at `numBeeps` more closely:

```
numBeeps := GetNamedVar(0, 'numBeeps);
#4417EA9  {_parent: {_parent: {#4411D11},
                     _proto: {#6008D0C1},
                     viewCObject: 0x110A483,
                     numBeeps: <2>,
                     viewclipper: 17863765,
                     base: <1>,
                     viewFlags: 577},
           _proto: {viewBounds: {#6008D689},
                    label: "Num Beeps:",
                    entryFlags: 10753,
                    _proto: {@189},
                    debug: "numBeeps",
                    preAllocatedContext: numBeeps},
           viewCObject: 0x110A57D,
           entryLine: {_parent: <2>,
                       _proto: {#356429},
                       viewCObject: 0x110A58C,
                       viewFlags: 10753,
                       viewBounds: {#4418301},
                       text: "2"},
           labelLine: {_parent: <2>,
                       _proto: {#356569},
                       viewCObject: 0x110A5C2,
                       text: "?Num Beeps:",
                       viewFont: {@100},
                       viewBounds: {#44181E1}},
           width: 73,
           indent: 75,
           height: 13}
```

Now let's look at the value of the `text` slot in `numBeeps`. Expecting to find the value of 2, we get a surprise instead:

```
numBeeps.text
#4632AD   ""
```

There isn't a text slot in the `numBeeps` view or template. So the value of `numBeeps.text` must be coming from the `protoLabelInputLine` **itself**. Notice, however, that there does seem to be a `text` slot with the value "2" in `numBeeps.entryLine`. But of course! For a `protoLabelInputLine`, the input text isn't found in the text slot of the `labelInputLine` view, but in the text slot of the child view where the input is actually done. With this bit of fresh information we can now modify the code in the `buttonClickScript`. Our code should actually be:

```
func()
begin
  local beeps :=
    StringToNumber(numBeeps.entryLine.text);
  for i := 1 to beeps do
    :SysBeep();
end
```

Fourth Problem

We rebuild, download, and rerun. We write in "2" in the input line and tap the Beep button. We get the following error in the Inspector:

```
Expected an integer, got <a real number>
evt.ex.fr.type;type.ref.frame
-48406
(#6008D229).buttonClickScript(), 27: PushConstant NIL
Entering break loop: level 1
```

Hmm. This is the same location where we broke before. Last time the error was "`Expected an integer, got nil.`" This time it got a real number. Let's take a look at `beeps`:

```
GetNamedVar(0, 'beeps);
#4418E79  2.00000
```

Well, that's the problem. The `beeps` variable is a real number, not an integer. Using our brilliant deductive capabilities we realize that `StringToNumber` must return a real number. Let's check within the Inspector:

continued on page 19

Avoiding Common UI Mistakes

This article is an excerpt from *Newton User Interface Guidelines*, published by Addison-Wesley and summarizes what you should do to avoid the top 20 user interface mistakes. The book provides detailed discussions of the correct approach you should use for each of the situations described here.

Info Button

Use the Info button—with the "i" icon—and its picker for information options such as Help, About, and Prefs. Always place the Info button at the far left end of the status bar unless your application includes an Analog Clock, which is optional.

New and Show Buttons

If users can create new items or display different views of information in your application, include a New button and a Show button like the ones in the built-in applications. Put the New button near the left end of the status bar next to the Info button (if present), and put the Show button to the right of the New button.

Screen Size

Design your application to handle any screen size and aspect ratio. If your application can't scale its views small enough or can't rearrange view contents to fit the aspect ratio, notify the user before closing your application.

Tapping v. Writing

Tapping is faster than writing, so for data input favor pickers, scrolling lists and tables, radio buttons, sliders, and so forth over written input.

Picker Placement and Alignment

Align the top of a picker with the top of its button or label. Make exceptions for overview pickers, for other very wide or very tall pickers, or for small screens.

Display a picker so its button or label is at least partially visible, and keep the button or label highlighted while the picker is open. (An overview picker can cover the label or button that makes it appear.)

Field Alignment

Be consistent in how you align field values with field labels (including picker labels). Generally you should line up a field's label with the field's displayed value, not with the dotted line (if present) on which a user edits the field value. In a view that has several fields in a column, line up the labels at their left edges to insure a neat, orderly appearance for your application.

Close Box Size

Use a regular (small) Close box in a view where there are no adjacent buttons. Use a large Close box only where there are adjacent text buttons or standard-height picture buttons.

Button Location

Put buttons that affect an entire view at the bottom of the view, and put buttons that only affect part of the view elsewhere. Group buttons that affect content and appearance at the bottom left of a view, and put buttons that control or initiate action at the bottom right.

Button Spacing

Space adjacent buttons three pixels apart, and leave four pixels between buttons and the border of the view they're in.

Button Size

Make every text button 13 pixels high and center the button's name vertically. Make the button just wide enough that with the button's name horizontally centered there are three or four pixels between the name and the button's left and right borders.

Capitalization

Capitalize the following items like sentences: check boxes, field labels, and picker items. Capitalize the following items like book titles: view titles, text button names, and radio buttons. In some contexts it makes sense to capitalize differently, but your should be consistent within an application.

Picker Icons

Think twice before including icons in pickers. They're hard to design and have limited benefit.

Dismissing a Slip

If dismissing a slip does not cause an action to take place (other than accepting changes made to data in the slip), use a Close box for putting away the slip. In this context the Close box means "close" or "put away." Use a take-action button and a Close box if users have a choice when dismissing the slip of initiating an action or canceling. In this context the Close box means "cancel."

Take-Action Button

Name a slip's take-action button with a specific verb such as Print, Fax, or File. Only use vaguely affirmative names such as OK and Yes where you want to force users to scan other parts of the slip to verify what action the button initiates.

Fonts

Use fonts carefully. For the voice of the system and application use the bold style of the System font in 9- or 10-point sizes. For values a user can change use Casual 10- and 12-point. (Those are the fonts that are preset by the system protos.)

Keyboard Button

If your application includes a Keyboard button on the status bar or at the bottom of a slip, use the larger-size button (as in the Notepad) unless space on the status bar is constrained (as in the Date Book).

Punctuation to Avoid

Don’t use ellipses (...) in button names, picker labels, or list-picker items.

Do put an ellipsis at the end of the title or the message text in a status slip, but use three periods rather than an ellipsis character. Also use an ellipsis to accommodate an item whose text is too long to fit on a line in the space available for it (for example, in overviews).

Don’t use a colon at the end of a title, a heading, or a field label.

Extras Drawer Icons

To avoid overlapping icons in the Extras Drawer, make yours no more than 29 pixels tall and wide. Leaving a little space helps separate icons.

Limit the length of an Extras Drawer icon’s name to between 9 and 11 characters per line. Put a blank space in the name where you want it to break and wrap onto another line.

Make a Newton icon more distinctive and easier to identify by giving it a distinctive silhouette rather than a boxy shape.

Storage

Allow users to move your application’s data between storage locations with the Filing button in the Extras Drawer’s status bar. This is the method used by the built-in applications.

Date and Time Input

To input dates and times use the specially designed Newton pickers.

NTJ

Newton Directions

Get Connected!

Newton Connection Utilities 1.0 Beta this Summer, Shipping this Fall.

by Cami Hsu, Apple Computer, Inc.

Newton Connection Utilities 1.0 (NCU) is enabling technology that will integrate your Newton device with your existing Windows PC or Macintosh OS based computer. NCU will enable you to get connected with your favorite desktop computer applications.

The features include Backup, Synchronize, Install, Import/Export and Keyboard. The Backup function creates a backup file of your Newton PDA and restores information from that file to your Newton device whenever necessary. The Synchronize function enables direct exchange of information between popular desktop applications and your Newton PDA. The Install Package function enables the installation of Newton software onto your Newton PDA. The Import function enables the sending of information from PIMs, word processors, or text files into your Newton PDA. The Export function enables the sending of information from your Newton PDA to your personal computer in a variety of formats, including word processing and PIM formats. The Keyboard function enables usage of your PC keyboard to enter information directly into your Newton PDA.

Integrate the Newton with the following applications using NCU:

Mac OS based computer:

- Now Up To Date 3.0/3.5
- Now Contact 3.0/3.5
- Now DateBook 4.01
- Now TouchBase 4.01
- Claris Organizer 1.0
- RTF
- Delimited ASCII

Windows PC:

- Schedule+ 7.0
- Ecco 3.03
- Lotus Organizer 2.1
- SideKick95
- Sidekick for Windows 1.0
- Sidekick for Windows 2.0
- Delimited ASCII
- Word 2.0/6.0/7.0

NTJ

The High Level Frame Desktop Integration Library (HLFDIL)

by Bill Worzel, Newton Developer Training

The High Level Frame Desktop Integration Library (HLFDIL) is used to move Newton frames and arrays to and from a desktop machine, running either the Macintosh OS or Windows. (In common usage, HLFDIL is usually shortened to Frame Desktop Integration Library, or FDIL, and the two terms are used interchangeably throughout this article.) Before the HLFDIL can be opened a connection between the Newton and the desktop machine must be established using the Communications Desktop Integration Library (CDIL).

The FDIL is used to map the dynamic structure of Newton frames to the static structures used on desktop machines. As with the CDIL, it is implemented in C++ but has a C language API. The basic assumption of the FDIL is that the format of the Newton frames being moved is already known. A C language structure having a one-to-one correspondence with the Newton frame can be defined, and the desktop programmer can define the mapping of slots to fields. However, since there are cases when the programmer may not know the structure of the Newton frame in advance or when additional slots have been added to the frame, there must be a way to handle these unexpected or unknown slots. This is handled by the FDIL by uploading these slots as unbound data, that is, data for which there is not a previously defined memory location of the appropriate size and type. The unbound data is therefore put into memory on the desktop machine in a tree structure which defines where the data is in the NewtonScript frame in relation to other slots in the frame.

Throughout this article, the use of the FDIL to transfer Newton frame data will be discussed. However, it should be remembered that it is also used to transfer Newton array data. It is worth noting that the FDIL cannot be used to transfer simple data such as integers or strings unless they are part of a frame or array. It is recommended that such data be transferred using the CDIL mechanism, or put into a simple NewtonScript frame, such as `fooFrame:={myInteger:3}`.

Opening the FDIL and Creating Objects

After the CDIL connection, or pipe, has been opened, the FDIL may be initialized and used. The FDIL routine FDInitFDIL is called as follows:

```
fErr = FDInitFDIL();
```

Any error in initializing the library will be returned. FDIL errors fall in the -28000 range and a complete list may be found at the end of the FDIL section of the document *Newton Desktop Integration Libraries* which can be found on the DIL web page at http://dev.info.apple.com/newton/tools/dils.html.

Next, one or more FDIL objects must be created using the routine FDILCreateObject which is defined as:

```
DILObj *FDCreateObject(short objType,
char *objClass);
```

The first argument describes whether the object being created is an array or a frame, and the second argument is an optional class name which NewtonScript arrays may carry. Each frame or array which is transferred must have a separate object created for it. This includes sub-frames and sub-arrays within a parent frame or array.

For example, for the following NewtonScript frame, three separate FDIL objects must be created before the frame can be defined on the desktop:

```
aFrame:={
name:"foo",
phone:["333-444-5555","101-202-3333"],
address:{street:"123 Main", town:"Fooburg",
state:'GA, postalCode:12345}
}
```

There must be an FDIL object for the main frame, `aFrame`, one for the `phone` sub-array and one for the `address` sub-frame.

The calls to create these FDIL objects would look like this:

```
DILObj *aFrameObj, foneObj, addrObj;

aFrameObj = FDCreateObject(kDILFrame, NULL);
foneObj = FDCreateObject(kDILArray,"homePhone");
addrObj = FDCreateObject(kDILFrame, NULL);
```

In this example `FDCreateObject` is called twice with the predefined constant `kDILFrame` to create an object for defining a frame and once to create an array object using the `kDILArray` constant for the first argument. In the case of the array, we also supply the class name `"homePhone"` as it is a named (classed) array. Named arrays are described on page 2-9 of the *NewtonScript Reference*. (An electronic version of this book is available from the Newton documentation web page noted above.)

As of this writing a known bug exists which will throw an error on the Newton when a string is downloaded, if an array includes an unnamed string (that is, a classless string) as one of its elements. The work around is explicitly to give unclassed strings the default class of "string" when they are array members. See the routine `DearchiveFrame` in the FDIL Archive Lab solution code for an example of this work around. This bug should be fixed in future releases of the HLFDIL.

Defining Objects

Once you have created an FDIL object, you may begin to bind memory locations to the object. This process describes to the FDIL where data being transferred to or from the Newton will come from or go to. In other words, by binding a memory location on the desktop to a slot in a Newton frame, the FDIL knows where to put data or where to go to get data. The memory locations used in this binding may be a variable on the stack, a global memory location or a dynamically allocated heap location, depending on the use and duration of the data being transferred.

The routine `FDbindSlot` is used to connect the memory location with the Newton slot. Its formal definition is:

```
objErr FDbindSlot(DILObj *theObj,char* slotName,
void *bindVar, short varType, long
maxLen,long curLen, char *objClass);
```

The first argument (`theObj`) is the object created to define the frame being transferred, and the second (`slotName`) is the name of the slot that is being bound.

The third (`bindVar`) is a pointer to the memory location to which the slot is being connected. When data is to be downloaded to the Newton this will be the location of the data being sent. In the case of an upload from the Newton this is the place where the received data will be put. In the latter case it is up to the programmer to make sure there is enough room to hold the data being received. The `maxLen` argument gives the total size of the object's buffer, and the `curLen` argument tells how much of that buffer has been filled by your desktop application (that is, `maxLen` is used for receiving data from the Newton device, and `curLen` is used for sending it). In the case of an array or frame, `maxLen` and `curLen` should be set to -1.

The `varType` argument describes what type of data object the slot being bound is expected to be. The following table shows the existing types:

```
kDILPlainArray        // as opposed to a classed array
kDILArray             // array with class name associated
kDILBoolean           // true or false
kDILUnicodeCharacter  // 16-bit character
kDILCharacter         // 8-bit ASCII
kDILFrame             // object is a frame
kDILSmallRect         // packs a Newton rect into a long
kDILImmediate         // Newton immediate value (not int)
kDILInteger           // 30-bit integer
kDILNIL               // Newton nil value (NULL)
kDILBinaryObject      // double or other binary sequence
kDILString            // char *
kDILSymbol            // handled as char *
```

Of these types several are worth special mention.

`kDILArray` versus `kDILPlainArray` - many arrays on the Newton are simply sequences of data but a few have a class associated with them. `kDILPlainArray` has no class associated with it while `kDILArray` does.

`kDILBoolean` and `kDILNIL` - both have platform-specific definitions for the values true, false and nil.

`kDILSmallRect` is automatically used when a frame on the Newton which is used to store a rectangle is sent and the rectangle values (top, left, bottom, right) all fit in a single byte value. In this case the values are packed into a single long value.

`kDILBinaryObject` is used for any unspecified binary object (such as sounds, pictures, and so forth) as well as for the Newton Real type. Note, however, that unlike integers or immediates, platform-specific byte ordering must be accounted for.

Make sure you compile using 8-byte doubles if you are going to transfer Real numbers. When transferring Newton Real numbers, you must set your development environment to use 8-byte double values or it will interpret the reals received as a different value than expected. This is usually an option specific to your system. For example, Metrowerks defaults to 4-byte doubles and the 8-byte double option must be specified in the compiler options in the Preferences menu item. MPW defaults to 8-byte doubles and so no special action must be taken.

`kDILString` includes both classed and unclassed strings.

`kDILSymbol` is a Newton symbol (such as 'mySymbol) transferred to and from the desktop as a C string.

The last argument in `FDbindSlot (objClass)` is the class, if any, of the object. While many objects on the Newton have no class associated with them, many may have a class assigned by the programmer or by the system. For example, while most strings have no class associated with them, they may have one assigned explicitly by the Newton programmer. Conversely, Newton Real numbers always are transferred as objects of type `kDILBinaryObject` with a class of "Real." If an object has no special class associated with it, this argument should be a NULL to use the default class symbol ("string", "array", "etc…). If it has a class it is transferred with a C string, such as for the class name.

An example taken from the SoupDrink sample code involves downloading a name frame to the Newton Name soup. The following pseudo-NewtonScript describes a card frame and its associated FDIL type:

```
card := {

  cardType:kDILInteger,
  Name: kDILFrame,
  Address: kDILString,
  City: kDILString,
  Region: kDILString,
  Postal_Code: kDILString,
  phones: kDILArray,
  sorton: kDILString
}
```

This is only a code fragment from the SoupDrink example. The variables shown (such as the name strings) are assigned values in earlier code. To see the full code, see the SoupDrink code walk through.

In the same way the `Name` frame and the `phones` array may be defined as:

```
Name := {

  class: kDILSymbol,
  first: kDILString,
  last: kDILString
}

phones := [kDILString, kDILString, ...]
```

While not shown here, the elements of the `phones` array each have a separate class associated with them such as "HomePhone," "WorkPhone," and so forth.

The code to create the FDIL object for this structure is:

```
name = FDCreateObject(kDILFrame, NULL);
FDbindSlot(name, "Class", (void *)&pClass, kDILSymbol,
    strlen(pClass), -1, NULL);
FDbindSlot(name, "first", (void *)&fName, kDILString,
    strlen(fName), -1, NULL);
FDbindSlot(name, "last", (void *)&lName, kDILString,
    strlen(lName), -1, NULL);

phones = FDCreateObject(kDILArray, NULL);
FDbindSlot(phones, NULL, (void *)&phoneNo, kDILString,
    strlen(phoneNo), -1, "HomePhone"));

entry = FDCreateObject(kDILFrame, NULL);
FDbindSlot(entry, "cardType", (void *)&cardType,
kDILInteger, sizeof(int), -1, NULL) ;
FDbindSlot(entry, "Name", (void *)name, kDILFrame, 0, -1,
    NULL);
FDbindSlot(entry, "Address", (void *)&addr, kDILString,
    strlen(addr), -1, NULL) ;
FDbindSlot(entry, "City", (void *)&town, kDILString,
    strlen(town), -1, NULL) ;
FDbindSlot(entry, "Region", (void *)&state, kDILString,
    strlen(state), -1, NULL) ;
FDbindSlot(entry, "Postal_Code", (void *)&zip,
kDILString, strlen(zip), -1, NULL) ;
```

```
FDbindSlot(entry, “phones”, (void *)phones, kDILArray, 0,
     -1, NULL) ;
FDbindSlot(entry, “sorton”, (void *)&sName, kDILString,
     strlen(sName), 0, “Name”);
```

Here the FDIL object's `name` and `phones` are bound to slots in the entry object. Note also that the string in the `phones` array has a class associated with it, `HomePhone`. Once the `entry` object is defined it is ready to be used to download it to the Newton.

Transferring Data

Once an FDIL object is created and the slots are defined by binding them to desktop memory, it may be used to transfer data to or from the Newton. This is done by using the routines FDget to upload and FDput to download the object and its associated data. These routines are defined as:

```
objErr FDput (DILObj *entry, short type, CDILPipe *pipe);

objErr FDget (DILObj *entry, short type, CDILPipe *pipe,
long timeOut, CDILPipeCompletionProcPtr
callBack,long refCon);
```

Notice that the CDIL pipe is used here since this is the first time that data will actually be transferred. Note also that the FDget routine may be called asynchronously by passing a procedure pointer in for the fifth argument.

In both routines the first argument (`entry`) is the master object being transferred. It may have sub-frames and arrays objects bound to it but this is the top level object. The next argument (`type`) is the FDIL type (`kDILFrame` or `kDILArray`) of this master object. The third argument (`pipe`) is the CDIL pipe being used to transfer the data.

For FDget we have these additional arguments:

The fourth argument (`timeOut`) defines how long the FDIL should attempt to transfer the object before giving up and returning an error. The timeout value is measured in milliseconds on Windows machines and ticks on Macintosh OS machines.

The next argument (`callBack`) is a pointer to a callback routine if the FDget call is made asynchronously. If you are making the call synchronously, simply pass NULL for this argument.

The last argument (`refCon`) is an arbitrary value which can be passed to your completion routine.

SoupDrink downloads the previously defined entry FDIL object to the Newton using this call to `FDput`:

```
  FDput(entry, kDILFrame, ourPipe);
```

The Newton must be in a state to receive a frame or array before it is sent from the desktop machine. See the SoupDrink code for the Newton for an example of code which imports and exports frames.

Unbound Data

Unbound data is data which arrives at the desktop but is not bound to any memory locations by calls to `FDbindSlot()`. This typically occurs in two situations: when the desktop programmer does not know the structure of the Newton data beforehand and when there are previously unknown slots which have been added to a frame.

An example of the first case is a program which allows the programmer to select what will be transferred. SoupDrink does this when it allows the programmer to name a Newton soup which is to be uploaded. In this case SoupDrink provides a universal way to handle all soups by uploading the information as unbound data which it then writes to file.

The second situation occurs either when there is a system or program update or when applications which “tag along” on system soups are added. For example, if a new version of the Names application added new slots to the soup frames stored for a Name card, SoupDrink would get the data from the old slots if it used the entry definition shown above but would not have a place allocated to put new data slots. Similarly, if a new contact application was loaded into the Newton, it might choose to build off of the existing Names application but add additional information to the soup entries. In this case SoupDrink would get the standard data but would not be prepared for the data added to the Names soup entries.

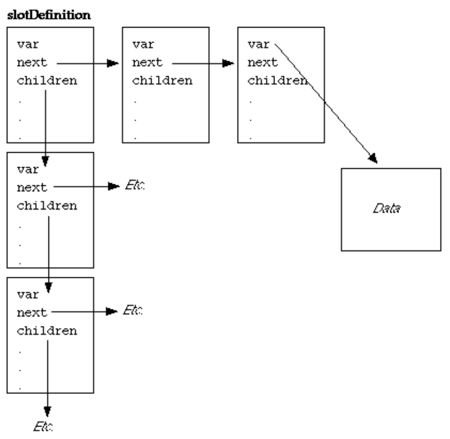
In either case, unbound data provides a way to accept unknown data slots and then to parse them for further disposition. Following is a description of unbound data and how to extract information about the data once it has arrived at the desktop.

Unbound data is linked together in a chain of data structures of type `slotDefinition`. The `slotDefinition` data type is defined as follows:

```
typedef struct slotDefinition
{
short varType;      // Data type of this variable
void *var;          // Actual pointer to data
ulong length;       // Length of data (strings)
ulong maxLength;    // Maximum Length of data
ulong streamLen;    // internal
ulong bufIndex;     // internal
long namePrec;      // internal
long classPrec;     // internal
char *slotName;     // Name of slot for this var
char *oClass;       // class of this object
short slotType;     // Data type of this slot
ulong truncSize;    // Current size of truncated object
long childCnt;      // Number of child nodes
long peerCnt;       // Number of peer nodes
short dataFilled;   // TRUE if data added in this op
long internalFlags; // Internal state flags
short boundData;    // internal
struct slotDefinition *children;
struct slotDefinition *next;
} slotDefinition ;
```

The fields which are significant to understanding the structure of unbound data are the `childCnt`, `peerCnt`, `children` and `next` fields. Unbound data is stored as a series of linked `slotDefinition` structures which may have siblings (peers) or children. In this scheme, `peerCnt` is the number of peers an item has, `childCnt` is the number of children, `next` is a pointer to the `slotDefinition` for the next peer in the list and `children` is a pointer to the start of the chain of `slotDefinitions` for the children of the item.

This is shown in the following diagram:

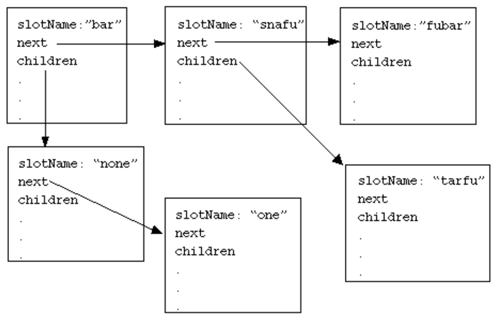


Here the `slotDefinition` structures are linked laterally by the next field while the children field points to the start of the list of all of the item's children. They each have their own list of children which do not intersect.

A more concrete example of this structure is shown below for the following simple frame:

```
foo:= {
  bar:{none:nil, one:1},
  snafu:{tarfu:2},
  fubar: 3
}
```

Unbound data structure for foo frame



Not shown in the second diagram is the var field which points to the actual data associated with the slot. With the addition of this slot we now have enough information to write the following pseudo-code for walking through the unbound data list printing the data:

```
parselist(list)
 begin
   item:=list[0];
   while (item != nil)
   begin
      if (item !=kDILFrame & item !=kDILArray)
         output(item->var)
      else
         parselist(item->children)
      item:=item->next
   end
 end
```

The only thing remaining is to find the start of the unbound data list and dispose of the memory allocated by the FDIL for the unbound data. The following routines do what is necessary:

```
SlotDefinition *FDGetUnboundList(DILObj *theFDILObj);
objErr FDFreeUnboundList(DILObj *theFDILObj,
SlotDefinition *list);
```

`FDGetUnboundList` returns a pointer to the list of items in the top level (master frame) of the unbound data while `FDFreeUnboundList` frees all the memory allocated for unbound data when `FDget` is called and there is data uploaded which has not been defined using `FDbindSlot`.

The use of unbound data may be summarized as follows:

Unbound data is the data received from the Newton after a call to `FDget` returns slots which do not have a specified location in memory connected to them. This may occur because the desktop programmer did not know what the structure of the data would be or because additional data was added to the frame being uploaded. Once transferred to the desktop machine the data is put into a linked list of slotDefinition structures which contain pointers to the next unbound item as well as items which are children of the current item. The unbound list may be parsed and extensive information about the slot associated with the item (including its type, name, class, and the actual data associated with the item) may be extracted from its slotDefinition structure.

The list of items of unbound data are kept in an array and the children field in a `slotDefinition` is an array of child items. `SoupDrink` uses this to parse the unbound data by looping through each of these arrays. Both algorithms work since *children is equivalent to children[0] in C.

Destroying Objects and Closing the FDIL

When you are completely finished with an FDIL object, call `FDDisposeObject` to destroy the object and all associated memory. Note that calling `FDDisposeObject` does not deallocate memory explicitly allocated by the desktop application. If memory is allocated off of the heap and then bound to a slot in an object, `FDDisposeObject` will not deallocate the heap memory. It must be explicitly deallocated.

Finally, the FDIL should be closed by calling `FDDisposeFDIL`. This should be done before closing the CDIL.

The definition of these routines as follows:

```
objErr FDDisposeObject ( DILObj *theObject );
objErr FDiposeFDIL();
```

NTJ

continued from page 13

Debugging Tutorial: Finding & Fixing Bugs in an Application

```
StringToNumber("2");
#4419331  2.00000
StringToNumber("2.5");
#44193ED  2.50000
```

We'll use the `Floor` function, which rounds down a real number to an integer, to fix our problem:

```
Floor(StringToNumber("2"));
#8         2

Floor(StringToNumber("2.5"));
#8         2
```

Let's rewrite the `buttonClickScript`. We'd better keep in mind that `StringToNumber` might return nil if we pass in a non-numeric string (wonder what would happen if we called `Floor` with `nil`?):

```
func()
begin
  local beeps :=
    StringToNumber(numBeeps.entryLine.text);
  if beeps then
    for i := 1 to Floor(beeps) do
      :SysBeep();
end
```

We rebuild, download, and rerun. We write in "2" in the input line and tap the Beep button. Lo and behold, the Newton beeps (although it's hard to tell there are two beeps because the second beep starts before the first beep finishes). Just to be thorough, we write in "two" in the input line and tap the Beep button. Nothing happens, just as we desire.

NTJ

Apple Developer Group

Newton Developer Programs

Apple offers three programs for Newton developers – the Newton Associates Program, the Newton Associates Plus Program and the Newton Partners Program. The Newton Associates Program is a low cost, self-help development program. The Newton Associates Plus Program provides for developers who need a limited amount of code-level support and options. The Newton Partners Program is designed for developers who need ujnlimited expert-level development. All programs provide focused Newton development information and discounts on development hardware, software, and tools – all of which can reduce your organization's development time and costs.

Newton Associates Program

This program is specially designed to provide low-cost, self-help development resources to Newton developers. Participants gain access to online technical information and receive monthly mailings of essential Newton development information. With the discounts that participants receive on everything from development hardware to training, many find that their annual fee is recouped in the first few months of membership.

Self-Help Technical Support
- Online technical information and developer forums
- Access to Apple's technical Q&A reference library
- Use of Apple's Third-Party Compatibility Test Lab

Newton Developer Mailing
- *Newton Technology Journal – six issues per year*
- *Newton Developer CD – four releases per year* which may include:
  – Newton Sample Code
  – Newton Q & A's
  – Newton System Software updates
  – Marketing and business information
- *Apple Directions – The Developer Business Report*
- *Newton Platform News & Information*

Savings on Hardware, Tools, and Training
- Discounts on development-related Apple hardware
- Apple Newton development tool updates
- Discounted rates on Apple's online service
- US $100 Newton development training discount

Other
- Developer Support Center Services
- Developer conference invitations
- *Apple Developer University Catalog*
- *APDA Tools Catalog*

Annual fees are $250.

Newton Partners Program

This expert-level development support program helps developers create products and services compatible with Newton products. Newton Partners receive all Newton Associates Program features, as well as unlimited programming-level development support via electronic mail, discounts on five additional Newton development units, and participation in select marketing opportunities.

With this program's focused approach to the delivery of Newton-specific information, the Newton Partners Program, more than ever, can help keep your projects on the fast track and reduce development costs.

Unlimited Expert Newton Programming-level Support
- One-to-one technical support via e-mail

Apple Newton Hardware
- Discounts on five additional Newton development units

Pre-release Hardware and Software
- Consideration as a test site for pre-release Newton products

Marketing Activities
- Participation in select Apple-sponsored marketing and PR activities

All Newton Associates Program Features:
- Developer Support Center Services
- Self-help technical support
- Newton Developer mailing
- Savings on hardware, tools, and training

Annual fees are $1500.

New: Newton Associates Plus Program

This new program now offers a new option to developers who need more than self-help information, but less than unlimited technical support. Developers receive all of the same self-help features of the Newton Associates Program, plus the option of submitting up to 10 development code-level questions to the Newton Systems Group DTS team via e-mail.

Newton Associates Plus Program Features:
- All of the features of the Newton Associates Program
- Up to 10 code-level questions via e-mail

Annual fees are $500.

For Information on All Apple Developer Programs
Call the Developer Support Center for information or an application. Developers outside the United States and Canada should contact their local Apple office for information about local programs.

Developer Support Center at (408) 974-4897
Apple Computer, Inc.
1 Infinite Loop, M/S 303-1P
Cupertino, CA 95014-6299

AppleLink: DEVSUPPORT

Internet: devsupport@applelink.apple.com

Newton