



# Newton<sup>®</sup> Technology

J O U R N A L

Volume 1, Number 4

August 1995

## Inside This Issue

### New Technology

Taking Advantage of Newton Toolkit 1.51

### NewtonScript Techniques

Small Parts: A Faster Way to Develop Large Applications 1

### Communications Technology

Programming Wide-Area Communications Using the Marco 3

### Server Technology

The Wayfarer Enterprise Server NT 7

### Server Technology

BLACKSMITH: PDA-Mainframe Communications 9

### NewtonScript Techniques

Function Objects in NewtonScript 11

### Developer Group News

New Choices and Reduced Prices on Newton Developer Support Programs 14



Newton<sup>®</sup>

## New Technology

# Taking Advantage of Newton Toolkit 1.5

by Bob Ebert, Apple Computer, Inc.

### INTRODUCTION

This article is written for the developer who is already familiar with Newton Toolkit (NTK) and the Newton platform. It covers features added to the 1.5 release of Newton Toolkit for Macintosh.

The article is divided into three major sections. The first section describes productivity enhancements and general changes. The second covers the NewtonScript (NS) profiler and shows how to take advantage of it in your applications. The last covers the NS native compiler and describes how to make the best use of compiled code.

### ENHANCEMENTS

This section describes some of the many changes that are found in this release of NTK, along with a brief description of how these changes may affect your development practices.

### Project Improvements

The project window now allows you to control the build order for your files. You can sort by any column by clicking on the heading. Click on the Sequence column to see the order in which files will be processed. Option-up and -down arrows will move a file earlier or later in the build order.

You can include more than one text file in your project. You don't have to be frustrated

continued on page 16

## NewtonScript Techniques

# Small Parts: A Faster Way to Develop Large Applications

by Maurice Sharp, Apple Computer, Inc.

### INTRODUCTION

Developing small applications using Newton Toolkit (NTK) is easy and fast. Even developing large applications with lots of views and data is not too difficult. However, there are ways to make development of larger applications even faster.

Most large applications can be easily split into a number of smaller pieces. Sometimes applications can be broken up by functionality, sometimes by interface, and sometimes the pieces can be split between the interface and the data. If these parts are included in one project, it can take some time to compile and download, which reduces the advantage gained from the rapid prototyping provided by NTK. However, you usually don't have to recompile all the pieces of an application – and suffer a long compile/download cycle – when making a reasonably localized change.

### THE TRADITIONAL (SLOW) WAY

First, let's look at a sample application done the monolithic, single-package way. The example will be a map viewer. To simplify matters, we'll assume that the application will use static data structures instead of soups. A standard development cycle for the application would be

continued on page 15

Published by Apple Computer, Inc.

Lee DePalma Dorsey • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

David Glickman • *Coordinating Editor, Business Content*

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

Philip Ivanier • *Manager, Newton Developer Relations Technical Peer Review Board*

J. Christopher Bell, Bob Ebert, Jim Schram, Maurice Sharp, Bruce Thompson

#### Contributors

Bob Ebert, Maurice Sharp, Julie McKeeham, Neil Rhodes, Ed Colby, Russel D. Matichuk,

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Judith Chaplin • *Art Director*

© 1995 Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014, 408-996-1010. All rights reserved.

Apple, the Apple logo, APDA, AppleDesign, AppleLink, AppleShare, Apple SuperDrive, AppleTalk, HyperCard, LaserWriter, Light Bulb Logo, Mac, MacApp, Macintosh, Macintosh Quadra, MPW, Newton, Newton Toolkit, NewtonScript, Performa, QuickTime, StyleWriter and WorldScript are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AOCE, AppleScript, AppleSearch, ColorSync, develop, eWorld, Finder, OpenDoc, Power Macintosh, QuickDraw, SNA•ps, StarCore, and Sound Manager are trademarks, and ACOT is a service mark of Apple Computer, Inc. Motorola and Marco are registered trademarks of Motorola, Inc. NuBus is a trademark of Texas Instruments. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. Windows is a trademark of Microsoft Corporation and SoftWindows is a trademark used under license by Insignia from Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories, Inc. CompuServe, Pocket Quicken by Intuit, CIS Retriever by BlackLabs, PowerForms by Sestra, Inc., ACT! by Symantec, Berlitz, and all other trademarks are the property of their respective owners.

Mention of products in this publication is for informational purposes only and constitutes neither an endorsement nor a recommendation. All product specifications and descriptions were supplied by the respective vendor or supplier. Apple assumes no responsibility with regard to the selection, performance, or use of the products listed in this publication. All understandings, agreements, or warranties take place directly between the vendors and prospective users. Limitation of liability: Apple makes no warranties with respect to the contents of products listed in this publication or of the completeness or accuracy of this publication. Apple specifically disclaims all warranties, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

## Editor's Note

by Lee DePalma Dorsey, *Managing Editor*

### NTK 1.5 SIGNALS NEW OPENINGS ON NEWTON PLATFORM

Last month brought scores of developers to Apple Computer's annual Worldwide Developers Conference to hear the latest news and check out the newest technologies from Apple. Amidst the sessions on Macintosh, the Newton Systems Group packed its meeting rooms with developers eager to learn about Newton Platform directions and technologies. In a standing-room only setting, we delivered the latest news on the Newton Toolkit 1.5. The content of this session and the others on the Newton platform set the tone for Apple's future directions with respect to platform tools and Newton application development. In case you missed that session, let me highlight the key points and provide some more insight on where we're headed and how you fit in.

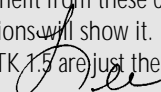
The newest version of the Newton Toolkit (NTK) is the product that you asked for and is indicative of the changes going on inside of the Newton Systems Group and the platform at large. NTK 1.5 is more open and flexible, as is the future of the platform. We've provided the compiler you asked for and delivered a profiler that allows you to test your code to determine whether compiling makes a difference in your code's speed and size. Unlike other compilers you may be used to, you should not always compile your code. In fact, you may even make your code larger and slower by compiling. Use of the profiler will let you determine which routines in your Newton application will benefit from being compiled. The choice of exactly what code to compile is up to you. Bob Ebert's article on NTK 1.5 in this issue of NTJ will highlight some key points in using the product and some areas where you should be cautious. This is critical reading for those looking forward to

working with this new version of NTK.

While we think you will laud the opening of the tools as another step in the right direction for you, it is clear that with the increased flexibility and choice comes increased risk and responsibility. When the Newton platform was introduced, the tools were closed and the developer was protected from mistakes. While that protection yielded easy application development, it reduced power, control, and creativity on the part of the developer. We look forward to seeing the results that the new choices and flexibility in NTK 1.5 allow from the applications you develop.

So the movement away from closed and protected to open and flexible means more choices, more power, and more responsibility for Newton developers. It also means greater possibilities. This is clearly the theme moving through all of the future directions for the platform and you will continue to experience more of this. Future tools will include Windows versions of the Toolkit, and C++ tools which will allow for even broader access. We'll be providing broader access to drivers for communications and PCMCIA development. Finally, we see the development of third-party tools playing a growing role in opening up the platform. All kinds of changes are afoot in the Newton Systems Group platform strategy that will benefit our business partners and developers.

What exactly do all of these changes signify to the Newton developer community? Is it a loss of arrogance from the Newton Systems Group? Is it simply a change in philosophy? Probably a little bit of both. But more importantly, it's a big change in the business model to that of a customer focused, customer driven model. We think you'll benefit from these changes and your applications will show it. The new features in NTK 1.5 are just the first



# Programming Wide-Area Communications Using the Marco

Excerpted from "Wireless for the Newton" by Julie McKeehan and Neil Rhodes. AP Professional.

Page references throughout the excerpt refer to sections of "Wireless for the Newton".

"Anytime, anywhere communications" is one of the mottoes of the Newton. Indeed, communications is one of the hottest areas of development within the computer industry, in general, and on the Newton, in particular.

The Motorola Newton, the "Marco," provides Newton users with an important feature previously unavailable in a Newton: built-in two-way wireless radio. This new feature gives the Newton user wireless access to a nationwide electronic network that includes a gateway to the Internet. This allows a wide range of new capabilities, including the ability to send and receive wireless mail to and from the Internet.

## Marco Specialized Transports for Supporting Wireless Mail

The Marco Newton comes with a dual protocol (MDC4800 at 4.8 Kbps, RDLAP at 19.2 Kbps) radio modem that connects to the ARDIS network. A user can have an ARDIS Personal Messaging (ARDIS PM) service agreement with ARDIS which enables sending messages to and from other ARDIS PM users or to hosts on the ARDIS network. A user can also establish a service agreement with Radiomail, which provides a mail gateway to the Internet.

To enable this feat of wireless magic, Motorola has provided two specialized transports which handle the interaction between the two services:

ARDIS Personal Messaging and users. This transport provides the ability to send and receive messages from other ARDIS PM users.

Radiomail. This transport provides the ability to send and receive messages from the Internet.

## Programming the Marco

Any application that has provided mail support (see "Mailing" on page 44) will automatically support wireless email. Once the user has chosen how to send mail (for example, ARDIS PM to another ARDIS PM user, Radiomail to send via the Internet, NewtonMail using a wired modem, or some custom transport), it will be automatically sent from any application via the specified transport.

## New Motorola APIs

Motorola has also offered the programmer other ways of making a Newton application interact specifically with the Marco. There are three APIs that a Newton application can use, depending on the type of application it is:

Mail-based (without receive mail user to use the should use the API.	Applications that send mail directly using the action menu) and directly (without requiring the In Box). These applications Universal Mail Services (UMS)
Transport-specific transport transport's API. are Radiomail and	Applications that require a specific and communicate with that Built-in transports on Marco ARDIS PM. Other transports could be provided in the future.
Dedicated host	These applications write directly to the Wireless Manager endpoint. They send packets to a specific host on the ARDIS network.

Use of any of these APIS's makes your application Marco-specific.

Of these three types of transports, we will be dealing with the first, mail-based applications. We will cover it in detail since the vast majority of Marco-specific applications will be of this type.

### Note:

*It is possible, using the Wireless Manager endpoint, to write your own transport (a service which is io box-based, and which is available to all applications). For example, you could provide a transport that accesses your enterprise-wide email system. By simply connecting to a host on the ARDIS network and writing transport software that would connect to the email system. Incoming mail would appear in the In Box. Users could use the action button to mail to a user on the enterprise email system. The mail would go to the Out Box, and then out via the ARDIS network to the host on that network.*

For information on the transport-specific APIs, on the Wireless Manager API, or on writing your own transport, you should contact Motorola at [MotoNewt.Dev@applelink.apple.com](mailto:MotoNewt.Dev@applelink.apple.com).

## Writing to the Mail-based API

There are two parts to mail-based applications: sending mail without going through the action button and mail slip, and receiving mail without requiring the user to go through the In Box. Sending mail without going through the action button has already been covered (see "Posting to the Out Box Programmatically" on page 57). The second part, receiving mail automatically, is done by registering your application.

## Registering an Application and Sending Mail

To register an application to receive mail, you use the `RegisterMailPackage` method (a method of the `motorola` global frame). This method takes one parameter, your application symbol.

When a piece of mail arrives, the mail system goes through the registered applications, sending each a message along with the piece of incoming mail. Each application checks to see whether the incoming mail belongs to it. If it does, it handles it and returns `nil`. Otherwise, it returns the mail, which is passed to the next registered application.

If no registered application handles the mail, it is placed in the In Box (as always). figure 2.2 shows a graphic representation of this process.

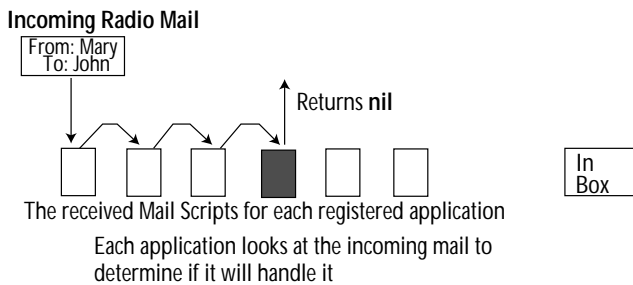


Figure 2.2 How the Marco handles incoming mail.

## Receiving Incoming Mail

The message sent when incoming mail arrives is `receivedMailScript`. The parameter this method takes is a frame containing at least the following slots:

<code>dateStr</code>	The date of the mail as a string.
<code>fromEmailAddress</code>	The email address of the sender.
<code>fromName</code>	The user-readable name of the sender.
<code>toEmailAddress</code>	The email address the item was sent to.
<code>name address</code>	The user-readable name at the mail the item was sent to.
<code>title</code>	The title of the mail message.
<code>text</code>	The body of the mail message.

## Unregistering an Application

To unregister your application, use the `UnregisterMailPackage` method. This is also a slot in the `motorola` global frame. You pass your application symbol as a parameter with code similar to the following:

```
UnregisterMailPackage(kApplicationSymbol);
```

The time at which you register and unregister your application will depend quite a bit on your particular application. Certainly, you'll want to register when you expect to receive mail. Different applications, however, might want to do this at different times. Here are some examples of when:

- When your application is opened.
- When your application is installed.
- When you send a message that requires a reply.
- At specific times. Perhaps your application receives mail at 10 am each day. In that case, you could register an alarm for 9:55 am that in turn registers your application for mail.

You'll deregister when you no longer expect to receive mail, or when your application can no longer receive it. The latter occurs when your application is removed. Therefore, you should minimally unregister in your `RemoveScript`. As to the former situation, any of the following answers might be appropriate for different applications:

- When your application is closed.
- When the mail you're expecting has finished arriving.
- At specific times. Perhaps your application listens for mail at 10 am, but if no mail arrives by 10:15 am, then it won't be arriving at all. In that case, you could unregister at 10:15 am.

In general, you should try to minimize the amount of time your application is registered. Otherwise, receiving mail can be quite time consuming for the user. Imagine the time involved if every application always looks at the incoming mail in turn and then passes it on to the next application.

## IMPLEMENTING A MARCO MAIL-BASED APPLICATION

Now that you have had an opportunity to get some sense of what is involved in creating a mail-based application for the Marco, let us look at a sample application that does this.

### A Stock Quote Application

The application we'll be creating obtains stock quotes from an Internet stock information service. The user selects a stock symbol and taps Lookup (see figure 2.3). This then sends a mail message to an Internet address which in turn sends back information about that stock symbol. This mail message is received by the application, which then displays it for the user in a view (see figure 2.4).

*Note:*

*The Internet service we are using for this application, QuoteCom, provides stock quotes and other information. They have many levels of service, one of which is free and allows up to 5 stock quotes per day. In order to receive free quotes, you must have registered for service. For further information, including information on registering for free quotes, send email to [info@quote.com](mailto:info@quote.com).*

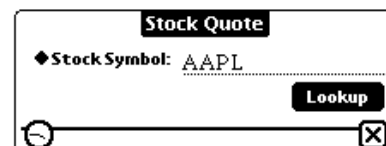


Figure 2.3 Looking up a stock quote.

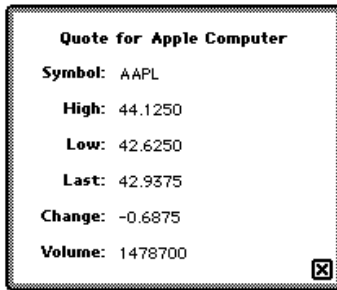


Figure 2.4 Receiving a stock quote.

## Sending a Stock Quote

The first thing that needs to be done is to send the request. To receive a quote, we must send mail to `services@quote.com` with the subject "Quote symbol". First we create the frame that we will send out:

```
local mailFrame := {
  appSymbol: kAppSymbol,
  connect:true,
  email: services@quote.com,
  hidden: true,
  text: "", // The text of the message is blank
  title: "Quote" && stockSymbol,
}
```

Now, we call the `Send` function specifying our newly created `mailFrame` and the transport we want to use:

```
call kSendFunc with ('mail', mailFrame);
```

### Note

*The Marco does allow sending frames in mail messages over its transports (just as NewtonMail does). Of course, the receiver of the mail must be a Newton in order to do anything with the frame.*

## Receiving a Stock Quote

Next, we must install our application so that it is notified when mail arrives. Although we could do this in our `InstallScript`, that would really be overkill. Rather, it is better to install only when we are expecting to receive mail. This way unasked-for mail is never sent to us; instead, mail comes in only in response to an outgoing item. Therefore, we will install our application to receive mail only after sending a piece of mail. Likewise, when our response comes we will want to deinstall. This is a little trickier than is sounds, however. Here are the situations we need to make sure we take into account:

- What if the user sends another stock lookup before the first has returned?
- What if the user removes the application while there is an outstanding request that hasn't been received yet?

Thus, we'll also need to keep a count of how many outstanding requests we have. When we send a stock lookup request, we'll increment that count. If the count just went from 0 to 1, we'll install

ourselves for mail notification:

```
numberOutstandingRequests:=
  numberOutstandingRequests + 1;
if numberOutstandingRequests = 1 then begin
  motorola:RegisterMailPackage(kAppSymbol);
end;
```

At this point, the mail has been sent, and our application has been registered to receive notification whenever mail arrives. If mail arrives, the `receivedMailScript` message is sent to our application's base view. Thus, we will create this method as a slot in our application base template, and within it check to see if the incoming mail belongs to our application or not. If the message does not belong to us, we also need to pass it along. Here is the `receivedMailScript`:

```
func(item)
begin
  if :IsQuoteResponse(item) then begin
    :RemoveRequestFromQueue();
    local error := :GetError(item);
    if error then
      :Notify(kNotifyAlert,
        EnsureInternal(kAppName),
        EnsureInternal(error))
    else begin
      local quote := :GetQuote(item);
      // EnsureInternal because floater may
      // be open when app is removed
      floater := BuildContext(EnsureInternal(
        pt_quoteResponseFloater));
      floater:Open();
      floater:SetResponse(quote);
    end;

    // let's delete the entry since we saw it..
    If IsSoupEntry(item) then
      EntryRemoveFromSoup(item);

    return nil;
  end else //it's not our mail
    return item;
end
```

In this code, we check to see whether the mail is a response to our quote request (with `IsQuoteResponse`). If it doesn't belong to us, we return the item, signifying that it's not ours. Otherwise, we need to take care of all of the following in this code:

- Reduce the count of outstanding requests (using `RemoveRequestFromQueue`).
- Deinstall our application from mail notification, if necessary, if the count is 0 (handled in `RemoveRequestFromQueue`).
- Read the information from the message. If the information is a valid quote, we need to display it in a floater (in `GetQuote`).
- If the information is not a valid quote, we put up a notify slip showing the error (handled in `GetError`).

## Checking for a Valid Quote

Note that a response to a quote request will have a return address of "services@quote.com" and a subject of "QuoteCom Email Response". We look for both of those in `IsQuoteResponse`:

```
func(item)
begin
  constant kQuoteFromEmailAddress :=
    "services@quote.com";
  constant kQuoteTitle :=
    "QuoteCom Email Response";
```

```

return StrEqual(item.fromEmailAddress,
  kQuoteFromEMailAddress) and
  StrPos(item.title, kQuoteTitle, 0) <> nil;
end

```

The text of a valid quote response will look like:

```

Tue Feb 7 14:24 EST Quotes may be delayed by exchanges
Symbol High Low Last Change
Volume
-----
BRK 24600.0000 24500.0000 24500.0000 +50.0000 60

Name Earn/Shr P/E Div/Shr 52-wk Price Range
-----
Berkshire Hathaway N/A 30.0 0.10 15150.00 - 25200.00

```

### Handling an Error

We'll assume that if the text starts with "Error:" or if the "---" can't be found that the response is some sort of an error (perhaps we aren't registered with Quote.Com or we've exceeded our number of quotes for the day). Here is the `GetError` method which returns an error message, or nil for no error:

```

GetError := func(item)
begin
  constant kCharactersBeforeQuote := "----\n";

  if BeginsWith(item.text, "Error:") then
    return item.text
  else if not StrPos(item.text,
    kCharactersBeforeQuote, 0) then
    return "Can't find the quote"
  else
    return nil;
  end
end

```

### Getting the Text for a Quote Frame

We need to extract the information from the quote text and put it into a frame containing that information. An example of a desired quote frame is:

```

{
  name:           "Berkshire Hathaway",
  symbol:         "BRK",
  high:           "24600.0000",
  low:            "24500.0000",
  last:           "24500.0000",
  change:         "+50.0000",
  volume:         "60"
}

```

Here is the `GetQuote` method, which searches through the text using brute force alone. The method then, in turn, creates a quote frame:

```

func(item)
begin
  constant kCharactersBeforeQuote := "--\n";
  local startOfQuote := StrPos(
    item.text, kCharactersBeforeQuote, 0);
  if not startOfQuote then
    return nil;
  end
end

```

```

local position := startOfQuote +
  StrLen(kCharactersBeforeQuote);
local stuff := {name: "", symbol: nil,
  high: nil, low: nil, last: nil,
  change: nil, volume: nil};

```

```

local GetString := func(s, sLength)
begin
  local endPos := position;
  // skip white spaces at the beginning
  while endPos < sLength and
    IsWhiteSpace(s[endPos]) do begin
    endPos := endPos + 1;
  end;
  position := min(endPos, sLength - 1);
  // now skip over
  while endPos < sLength and
    not IsWhiteSpace(s[endPos]) do begin
    endPos := endPos + 1;
  end;
  endPos := min(endPos, sLength - 1);
  local s := SubStr(s, position,
    endPos - position);
  position := endPos;
  return s;
end;

```

```

local length := StrLen(item.text);
stuff.symbol := call GetString with (
  item.text, length);
stuff.high := call GetString with (
  item.text, length);
stuff.low := call GetString with (
  item.text, length);
stuff.last := call GetString with (
  item.text, length);
stuff.change := call GetString with (
  item.text, length);
stuff.volume := call GetString with (
  item.text, length);

// name of stock follows after more "---"
position := StrPos(item.text,
  kCharactersBeforeQuote, position);
if position then begin
  stuff.name := Substr(item.text,
    position + StrLen(kCharactersBeforeQuote),
    kNameLength);
  TrimString(stuff.name);
end;

return stuff;
end

```

Once we've got a quote frame, it's a simple matter to set the various static texts in a floater. Here is the `SetResponse` method of the floater which does that:

```

func(quote)
begin
  SetValue(headline, 'text, headline.text &&
    quote.name);
  SetValue(symbol, 'text, quote.symbol);
  SetValue(high, 'text, quote.high);
  SetValue(low, 'text, quote.low);
  SetValue(last, 'text, quote.last);
  SetValue(change, 'text, quote.change);
  SetValue(volume, 'text, quote.volume);
end

```

### Handling the Unregistration

The application must be unregistered if there are no more outstanding requests. The `RemoveRequestFromQueue` method does that:

```

func()
begin

```

# The Wayfarer Enterprise Server NT

by Ed Colby, Wayfarer Communications, Inc.

Wayfarer provides developers a client/server framework on which to develop Newton applications and integrate them into enterprise networks and resources such as databases, terminal servers, e-mail systems and legacy systems. Wayfarer's framework, the Wayfarer Enterprise Server NT, hides the complexity of communications software, so developers can implement custom applications quickly and run them reliably over mobile networks. Wayfarer-based applications support Newton connectivity using TCP/IP protocols over most leading wire-line and wireless networks.

Programmers develop Wayfarer applications using standard languages such as Visual Basic™, PowerBuilder™, Visual C++™, and NewtonScript™. Developers use Wayfarer's APIs to create user interfaces on Windows™ PCs or Newton® MessagePad™ PDAs, and to work with application logic on the server. The server provides flexible access to multiple enterprise systems, such as databases, that can be incorporated into mobile applications. Real-time data access, transactions, messaging, work flow and other functions can all be integrated into individual custom applications for mobile work forces.

## DESIGNED FOR MOBILE NETWORKS

Wayfarer optimizes performance over low-bandwidth, high-latency mobile networks. The Server offloads most storage, computing and communications tasks from mobile computers and mobile networks onto a powerful server computer and high-bandwidth, fixed enterprise networks. Wayfarer supports most wire-line and wireless media in local- and wide-area networks, used in any combination.

## Access to Enterprise Systems

The Wayfarer Server provides tools to access and integrate data, programs and services from any standard or legacy system. As a distributed computing model, developers can access multiple systems simultaneously and can integrate them into custom mobile

applications. Developers can also use standard tools for standard systems, such as ODBC for SQL databases, or they can create their own tools.

## Communications and Work Flow

Wayfarer's system includes extensive messaging, communications and work flow support. The Wayfarer Server supports real-time messaging, alphanumeric paging and e-mail. Messages can be addressed or forwarded to individuals or groups of users. Using Wayfarer's APIs, programmers can easily integrate messaging into their custom applications. These communications functions can also be integrated into enterprise systems such as corporate e-mail. Wayfarer also supports e-mail over the Internet.

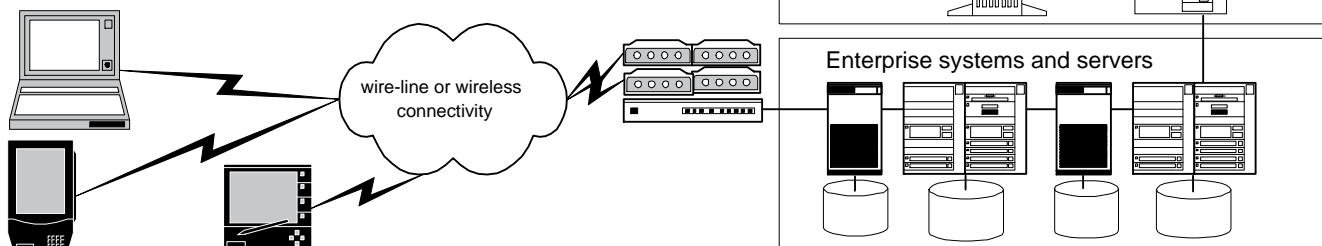
## Integration into the Enterprise

Wayfarer's system is built on the leading standards for enterprise computing, application development and internetworking. The Wayfarer Server, and Windows and Newton mobile clients all use TCP/IP networking protocols. Windows clients and the Server also support Named Pipes. The Wayfarer System Manager provides a graphical interface for system configuration, management and security. All system access and usage is configured and controlled centrally. The Wayfarer System Manager also communicates with mobile client users through the messaging functions built into Wayfarer's system.

Wayfarer's architecture optimizes performance on mobile devices and mobile networks by offloading the burden of computation, storage and communications with enterprise servers onto the Wayfarer Enterprise Server. These optimizations for mobile environments give rise to the server and services model. A client/server architecture enables many Newton MessagePads and Windows computers to be connected to a single Wayfarer Enterprise Server.

Wayfarer uses a client/server architecture to connect Newton

Framework to **develop** and **deploy**  
**custom applications** on  
**Windows** mobile computers *and*  
**Newton** MessagePads



MessagePads and Windows mobile **client** computers to the Wayfarer Enterprise Server. Wayfarer relies on TCP/IP protocols in Newton, included in Wayfarer's product, to provide network connectivity between mobile client devices and the Wayfarer Enterprise Server.

The **server** functions as a gateway between mobile client devices and enterprise servers, such as SQL databases and e-mail systems. It manages connections and communications with Newton MessagePads and Windows mobile clients, eliminating the need for programmers to manage these tasks.

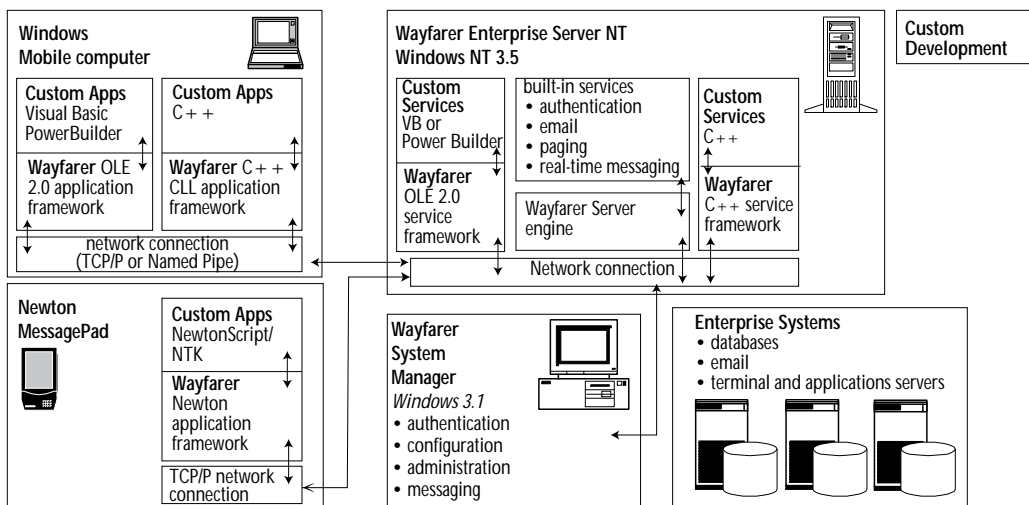
The Server provides **services** to Wayfarer mobile clients. Wayfarer's services include mail, paging, messaging, and authentication. Using enterprise information and communications resources, services provide specific functionality to Wayfarer client users. Programmers write programs on mobile client devices that use and integrate any or all of Wayfarer's services to meet specific applications requirements.

Programmers develop applications by calling Wayfarer's APIs available through NewtonScript and NTK. The APIs provide general functionality for mobile applications, as well as calls for all services available from the Wayfarer Server. Developers program Newton MessagePad applications exclusively in NewtonScript.

Wayfarer applications do not have to change to work with different connectivity media. By using TCP/IP as the network protocol, developers can deploy mobile applications with any standard form of wire-line or wireless connectivity for in-house, metropolitan or wide-area coverage. Applications can be developed with wire-line connectivity for convenience and cost, and can still be deployed with wireless connectivity media. Supported connectivity media includes land-line and cellular dial-up, CDPD, Metricom, Photonics, Dayna, Digital Ocean and RAM Mobile Data.

Wayfarer's system includes a Windows for Workgroups 3.11 program called the Wayfarer System Manager; this provides all administrative functionality for clients and servers on the Wayfarer system. Wayfarer System Manager requires no programming, but does require configuration in setting up a Wayfarer system.

Programmers can develop custom services using the Wayfarer C++/OLE Automation Custom Service Framework. This framework allows a developer to author custom services using Microsoft's Visual C++ , or Visual Basic, running under Windows NT 3.5. Using this framework, developers can easily create services that access enterprise-specific resources such as databases. Custom services are full-fledged services that can be accessed by client applications in the same manner as the built-in Wayfarer services.



NTJ



# BLACKSMITH: PDA-Mainframe Communications

by Russel D. Matichuk, CEL Corporation

Many large companies, universities, and government agencies have considerable resources invested in large, mainframe-based databases. Rather than replacing their existing systems to take advantage of advances in client/server technology, these organizations can now use a technology called middleware to enable PDAs, desktop computers, or World Wide Web browsers to create data entry and retrieval systems for applications that previously could be accessed only by terminal emulators. CEL Corporation's BLACKSMITH is a middleware development tool that allows Newton developers to add client/server communications capabilities to the Newton MessagePad.

## DESKTOP-DRIVEN CLIENT/SERVERS

Today most users communicate with mainframes by means of terminal emulation on PCs. But methods of interacting with mainframes have changed dramatically over the last ten years. With the cost of PC processing power at the lowest levels ever, it no longer makes sense to use PCs simply for terminal emulation. Instead, firms are looking to combine the flexibility, ease of use, and power of their personal computers – including PDAs – with the resources of their mainframes.

The interaction between the host computer and the desktop computer has evolved over the years, with the desktop machine taking on a greater share of the processing power. Originally, PCs were used only for terminal emulation, where the host-based system controlled all processing and used the PC only for displaying data or receiving user input. In front-ending, or simple automations, character streams from the host were mapped to objects in a graphical user interface or to keystroke commands for macros, making it easier for the user to interact with the host. In host-driven client/server applications, the desktop machine became a server capable of receiving messages from the host; some of the host's functionality was extended to the desktop machine, and desktop-based user interface capabilities such as menus or dialog boxes were possible. With desktop-driven client/server applications, logic is split between the client and the server; however, the desktop machine retains principle control of the application and determines what should be displayed and how user input should be handled.

In a desktop-driven client/server scenario, the desktop application submits queries or updates to a server on a host or another platform. There is a significant increase in system functionality because of the improved coordination between stand-alone desktop applications and the server. This approach allows servers to be optimized for data-management functions, such as high-volume transaction processing or resource-intensive database query processing, while allowing desktop computing power to be utilized more fully. An example of a desktop-driven client/server is a fourth-generation language written using extensions that send information to and retrieve information from

relational or nonrelational host systems.

In general, desktop-driven client/server solutions – also called *distributed applications* – refer to transaction programs that rely on specific client and server components designed to optimize performance and productivity. The server elements include the program engine and the management services governing security and administration. The client software contains the end-user interface and some level of intelligence that might include some local data storage error checking and other functions. The model allows the two components to be separated so that each may be independently tailored to specific users and functions.

This contrasts with traditional methodology that has a central department – such as management information systems (MIS) – generating more generic processes that accommodate many users, but are inherently less productive and are more difficult to adapt. Shifting the burden of data access – from a centralized system where MIS responds to departmental needs to a system where departments are self-sufficient – benefits the entire organization by allowing each group to satisfy its needs. Also, development time and costs decrease when similar development functions are conducted on the PC instead of the mainframe.

Another factor in this model is the distribution of processing power across the enterprise. MIS organizations face the rigorous assignment of allocating and managing mainframe resources. By integrating the intelligence of personal computers, many traditional mainframe tasks – such as error and syntax checking – can be handled by desktop systems. As the intelligence of front-end systems evolves, more of the information processing occurs at the desktop; response time increases and use of unnecessary network bandwidth diminishes. In addition, by sharing the processing burden across the enterprise, mainframes are better utilized for CPU-intensive applications.

In this view of corporate data, all users share the cost of the system. Departments can either allocate funds to develop custom tools and applications on the mainframe, or they can move development to the PC. Development tools for the PC such as Omnis 7, 4th Dimension, PowerBuilder, Visual Basic, Microsoft Access, Uniface, Visual AppBuilder, C++ – plus the introduction of PDAs and Web technology – have significantly reduced the investment in time and capital required for the development process. Increasingly, departments are taking responsibility for developing their own tools on the PC instead of investing heavily in similar tools on the mainframe.

## BLACKSMITH: A DESKTOP-DRIVEN MIDDLEWARE SOLUTION

The growth of client/server applications is due in part to the proliferation of a new generation of software development tools called

*middleware*. Because no single protocol adequately solves all data transmission issues, large companies tend to manage several protocols throughout their corporate network. These organizations are also using third- and fourth-generation languages (3GLs and 4GLs) extensively on microcomputers and are evaluating the use of PDAs and Web technology. This presents a number of challenges, especially when end users need to access information from various host systems employing differing network protocols. BLACKSMITH is designed to ease the process of writing distributed applications by abstracting the details of both the operating environments and the network protocols.

The key to improving user productivity is to break down the barriers between these distinct environments – mainframes and PCs – so end users can spend less time learning the intricacies of complicated mainframe applications and concentrate instead on accessing information. If users require data in different locations or on different systems, they shouldn't have to learn new methods for retrieving data, but should be able to use familiar front ends such as PCs, PDAs, or Web browsers. Middleware provides a migration path. Network developers and support staff are usually skilled in developing, maintaining, and operating host-based systems, leaving them reluctant to embark on a distributed-solution development. Thus, a middleware product is only viable for them if presented in an architectural framework that provides a slow migration path, enabling them to preserve their legacy investments, while gradually increasing the use of desktop and local server computing resources. Several factors – operational savings resulting from integrating desktop systems, the ability to create scalable systems, and a reduction in development and deployment time – should make distributed systems (desktop-driven client/server systems) increasingly attractive to network developers.

There are several ways of providing a communications link between a client and a server. The method that BLACKSMITH uses is called Terminal Data Stream (TDS) communications. The TDS is the physical connection that allows an operator to communicate with host computers; it includes data streams of all types (3270, 5250, VT100, etc.). Initially, only dumb terminals were connected through the TDS. With the advent of terminal emulators, microcomputers have since dominated the use of the TDS communications medium.

The key technology that spawned the terminal emulation industry –

and which is the critical ingredient in the recent explosion of development using client/server technology – is the development of the *terminal emulator API* (Application Programmers Interface). In an effort to expand their market, vendors of terminal emulation products wrote APIs that allowed third parties to create and sell software that depended on these products. However, these APIs are fraught with deficiencies. They were written to allow integration only by third-generation languages like C, are very difficult to use, and in most cases provide only very basic services. None of the APIs provides a reliable host-state management process. In addition, each vendor's API is quite different from those of other vendors.

BLACKSMITH is designed to eliminate the complexities of connecting microcomputer applications to host systems. With terminal emulator APIs or communications standards such as VT100, developers must have advanced communications programming skills; with BLACKSMITH, developers aren't required to know anything about the low-level communications issues. They can choose from a wide selection of third- and fourth-generation development tools to build applications that improve the data entry and retrieval processes used by their organization. In addition, developers can implement solutions that include PDAs or Web browsers communicating with a BLACKSMITH server. One set of BLACKSMITH commands can be used in any of the environments that are supported. This same set of commands will allow a developer to write applications that communicate with any host system, using any of the communications channels that are supported.

#### USING BLACKSMITH WITH NEWTONS AND THE WORLD WIDE WEB

BLACKSMITH 2.0 adds Newton and Web client access to mainframes and minicomputers. Organizations can now have their staff interacting with their host systems using graphical Macintosh, Windows, Newton, or Web applications, with no modification to the mainframe or minicomputer applications.

Using NewtonScript or HTML (Hypertext Markup Language) Web server code, BLACKSMITH developers can build functions that send a *Remote Procedure Call* (RPC) to a waiting BLACKSMITH server. The RPC is written using one of the Macintosh or Windows 4GLs or 3GLs that BLACKSMITH supports: PowerBuilder™, Visual Basic™, Omnis 7, 4th

NTJ



To request information on or an application for Apple's Newton developer programs,  
contact Apple's Developer Support Center at 408-974-4897  
or Applelink: DEVSUPPORT or Internet: [devsupport@applelink.apple.com](mailto:devsupport@applelink.apple.com).

# Function Objects in NewtonScript

by Julie McKeehan and Neil Rhodes, Calliope Enterprises, Inc.

In NewtonScript, function objects (sometimes called closures) are first-class objects which can be manipulated and stored just like other values. For example, you can store a function object in:

- a local variable
- an array
- a slot in a frame
- a soup entry

You can use function objects in a variety of ways as well: you can pass a function object as a parameter, Clone it, or DeepClone it. This consistency between function objects and other values makes NewtonScript very flexible.

These aspects of function objects are usually well understood by NewtonScript programmers. There is another aspect to a NewtonScript function object, however, that is less clear:

when a function object is created, by executing a `func` statement, it saves the environment that exists at that time.

By doing so, the function object can have access to local variables, parameters, and inherited variable lookup that existed at its creation time.

The term "function object," rather than just "function," is used to emphasize the fact that one `func` statement can give rise to many different function objects. These functions will differ based on the environment that exists at the time the `func` statement is executed.

## THE FUNCTION ENVIRONMENT

To understand how you can use this aspect of a function in your applications, let us first review the environment of most NewtonScript functions. Usually, you create function objects at compile time. They are slots in a template, edited using a slot browser, or they are functions created in a Project Data file. In either case, the environment that exists is the environment of NTK. Thus, there are no local variables, parameters, or inherited variables available when that function object is created.

Now, look at a different way to create a function object. In this case the environment in which it is created will be significant. This will be a run-time function object. Please note that by definition, these will be nested functions (ones created inside other functions).

Here is an example:

```
outerFunction := func(aParameter)
begin
  local aVariable := 3;

  local nestedFunction := func(nestedParameter)
```

```
begin
  local nestedlocal := 5;

  Print(aParameter);
  Print(aVariable);
  Print(nestedParameter);
  Print(nestedFunction);
  Print(nestedlocal);

end;
...
```

The function object, `nestedFunction`, is created as `outerFunction` is executing. At the run-time point when `nestedFunction` is actually created, the environment includes a local variable, `aVariable`, and a parameter, `aParameter`. Since `nestedFunction` has access to that environment, it can access both its own parameters and locals, as well as those of the function in which it is nested.

## HOW FUNCTION OBJECTS ARE CALLED

Before we actually talk about how you can use function objects in your programming, we need to address the manner in which you call a function object. There are four ways you can do this:

With a compile-time argument list (**Call**):

Call functionObject with (argumentList)

With a run-time argument list (**Apply**):

Apply(functionObject, argumentArray)

By sending a message with a compile-time argument list (**:** and **?:**):

frameExpression:message(argumentList)

By sending a message with a run-time argument list (**Perform**):

Perform(frameExpression, messageSymbol, argumentArray)

Here is some sample code that uses these four ways:

```
local Pow := func(num, iterations)
begin
  for i := 1 to iterations do
    total := total * num;
  return total;
end;

Call Pow with (2, 3) ⇨ 8

local argArray := [2, 3];
Apply(Pow, argArray) ⇨ 8

local account := {
  balance: 0,
  Deposit: func(amount)
    return balance := balance + amount,
```

```

}
account:Deposit(50) ⌘ 50
Perform(account, 'Deposit, [75]) ⌘ 125
Print(account) ⌘ {balance: 125,
  Deposit: <CodeBlock, 1 args #4419361>}

```

## HOW FUNCTION OBJECTS CAN BE USED

The sections that follow describe a several ways that you might use function objects. Two examples are:

- to implement abstract data types
- to support Date Find operations.

### Abstract Data Types

One use of function objects is to implement abstract data types. These are types that can only be modified procedurally; their actual data is hidden. Though it might appear so, frames with methods don't provide the same functionality. In a frame, the data values in the slots are visible and can be modified even when not using the appropriate methods.

Consider the following account generator:

```

MakeAccount := func()
begin
  local balance := 0;
  local Deposit := func(amount) begin
    return balance := balance + amount;
  end;
  return Deposit;
end;

```

Calling `MakeAccount` returns a function object:

```
myAccount := call MakeAccount with ()
```

This function object references the `balance` local variable from `MakeAccount`. Even though `MakeAccount` is no longer executing, since the nested function, `Deposit`, references `balance`, the `balance` variable continues to exist. Thus, calling `myAccount` modifies the hidden variable `balance`:

```

call myAccount with (50) ⌘ 50
call myAccount with (75) ⌘ 100

```

Notice also that one function object can return multiple function objects, each of which references shared data. For instance, suppose you want both `Deposit` and `Clear` capabilities in your account:

```

MakeAccount := func()
begin
  local balance := 0;
  local Deposit := func(amount) begin
    return balance := balance + amount;
  end;
  local Clear := func() begin
    balance := 0;
  end;
  return [Deposit, Clear];
end;

```

Because `MakeAccount` needs to return two values (two function objects), it returns them in an array:

```

myAccount := call MakeAccount with ();
myOtherAccount := call MakeAccount with ();
call myAccount[0] with (50) ⌘ 50

```

```

call myOtherAccount[0] with (40) ⌘ 40
call myAccount[0] with (75) ⌘ 125
call myAccount[1] with () ⌘ 0
call myOtherAccount[1] with () ⌘ 0

```

Using an array for the two function objects is somewhat inconvenient, however, since the numbers 0 and 1 don't describe the `Deposit` and `Clear` functions in a very useful manner. To fix this problem, we will rewrite `MakeAccount` to return the two function objects in a frame rather than in an array. This way, the function objects can be referenced by name, rather than by array location.

```

MakeAccount := func()
begin
  local balance := 0;
  local d := func(amount) begin
    return balance := balance + amount;
  end;
  local c := func() begin
    balance := 0;
  end;
  return {
    Deposit: d,
    Clear: c,
  }
end;

```

```
myAccount := call MakeAccount with ();
```

```

call myAccount.Deposit with (50) ⌘ 50
call myAccount.Deposit with (75) ⌘ 125
call myAccount.Clear with () ⌘ 0

```

Remember, however, that we are using the above frame only as a way to store two named values. We have not started using any object programming, however, as no messages are being sent.

Admittedly, this use of function objects to create Abstract Data Types is not common. There are cases, however, where function objects are necessary in your Newton programming. One of the most common examples occurs when an application is supporting Date Find.

### Using Function Objects to Support Date Find

Here is an excerpt of code from `DataFind`:

```

func(comparison, time, ...)
begin
  cursor := Query(..., {
    type: 'index,
    validTest: func(e)
    begin
      if comparison = 'dateBefore then
        return e.date < time
      else
        return e.date > time;
      end
    });
  return cursor to caller
end;

```

Even though the `validTest` function is embedded in the cursor, it is called from the Find results slip to display the found entries. Notice how `DateFind`'s comparison and time parameters are used by the nested function `validTest`. Keep in mind that the `validTest` is called after the `DateFind` function has finished executing.

Here is another example of how you might use function objects. Imagine that you want to count the number of entries that are in a particular query. In such a case, you might use a function object to count the number of entries in a cursor using `MapCursor`:

```
CountEntries := func(cursor)
begin
  local total := 0;
  MapCursor(cursor, func(e)
  begin
    total := total + 1;
    return nil;
  end);
  return total;
end
```

The function object passed to `MapCursor` increments a variable in `CountEntries`. Notice that the function object returns `nil`, and thus `MapCursor` will end up returning an empty array.

### STACK FRAMES/ACTIVATION RECORDS

In most programming languages, when a function is entered, an activation record (also called a stack frame) is pushed on the stack. This activation record contains the parameters to the function and local variables. When the function returns, the activation record is popped from the stack.

For NewtonScript, some allowance needs to be made for variables which are closed over; that is, variables which are accessed by nested functions. Since nested functions may need to access variables from outer functions even after the outer function has exited, a stack-based system which always pops outer references from the stack would fail.

One possible implementation could be to allocate activation records in dynamic memory (the heap).

Like all other allocated memory, when no more references are made to the memory, it can be garbage collected. Thus, the activation record is not freed when a function object exits if any nested functions still exist. Only when all nested functions are freed is the activation record available for garbage collection.

Another implementation might store part of an activation record on the stack, and part on the heap (only those variables referenced by nested functions need be on the heap).

### MESSAGE CONTEXT AS PART OF FUNCTION OBJECT

A function object has access to more than local variables and parameters from enclosing functions. It also has inheritance lookup based on the value of `self` at the time the function object was created. This means that a function object has access to all variables, including inherited slots, that are available to the code that created the closure.

This inheritance lookup is implemented by storing a message context as part of a function object. This message context contains the value of `self` at the time a closure is created. Calling a closure restores `self` to the value stored in its message context.

The major difference between calling a closure and sending a message is that sending a message sets the value of `self` to the frame where the message is sent. Thus, sending a message causes the message context of the closure to be ignored.

### Using Inheritance in a Function Object

There are times, however, when you need to use inheritance in a function that has not been executed in response to a message send. A common case of this in Newton programming is found in the implementation of filing. Filing is usually implemented by creating a

cursor that contains a `validTest`. The cursor is usually created when the application opens:

```
app.viewSetupFormScript := func()
begin
  ...
  self.theCursor := Query(..., {type: 'index,
  validTest: func(e)
  begin
    return labelsFilter = '_all or
    labelsFilter = e.labels;
  end,
  });
  ...
end
```

The cursor saves the `validTest` cursor and calls it every time the cursor is moved. When the `validTest` is called, the `labelsFilter` variable is looked up first as a local, and then using inheritance *based on the value of self at the time the validTest function object was created*. Since `self` was the application view when the `validTest` was created, `self` is set to the application base view when the `validTest` is called.

### Sending a Message Changes the Message Context

Thus, the major difference between sending a message and calling a function *has to do with the value of self*. When a message is sent, `self` is set to the frame that was sent the message. When a function is called directly, `self` is based on the message context of the function object.

### FUNCTION OBJECTS CREATED AT COMPILE TIME

Function objects created at compile time have no lexical environment or message context. When a top-level function object is created at compile time – either as a slot in a template editor, or in the top-level of the Project Data file – the lexical environment and message context are empty (Technically, the lexical environment and message context exist, but are nil'ed out after the function is created). This is important for you to remember when you are creating standalone closures (ones which have no references to your package). Examples of such closures are those copied to a soup, or the one used as a `postParse` routine for Intelligent Assistance. If you use a closure that has a non-empty message context here, the whole receiver will be copied into the soup (or into memory in the case of Intelligent Assistance). This is not a good idea, in most cases. Thus, for function objects which need to execute independently of your package, make sure they are created at compile-time, rather than runtime.

### SUMMARY

In NewtonScript, function objects are first-class objects which have access to the environment that exists at the time they are created. They have access to variables in enclosing functions, as well as to inherited slots based on the value of `self` at the time the function object was created. Because of these characteristics, you can do particular types of things with function objects that can not be done in most other languages.

# New Choices and Reduced Prices on Newton Developer Support Programs

by Lee Dorsey, Apple Computer, Inc.

Newton Developers now have access to a full suite of Newton developer support options, with just announced new programs, and enhancements with reduced prices to existing programs. Newton developers may choose from a range of support programs, from self-support to unlimited priority support, offered through the Apple Developer Group. Our goal is to help you succeed in your development efforts – so we have delivered programs that provide you with the level of support you need, at a cost you can afford. Read on for more details on new programs and reduced program pricing!

## THE NEWTON ASSOCIATES PROGRAM

Introduced in December of 1994, this program has been designed as the development support option for developers seeking low-cost, self-help development resources for Apple's Newton technology. Primary program features include access to on-line technical information, Q&As, updates to Newton development tools, the Newton Developer Mailing, the Newton Developer CD, discounts on Newton developer training classes, Apple hardware purchase discounts, and a full suite of developer assistance from the Apple Developer Support Center. A full features list follows.

Annual membership fees are \$250, reduced from \$400 in May of 1995.

### Associates Program Features:

- The Newton Orientation Kit
- Newton Developer Mailing
  - Newton Technology Journal
  - Newton Developer CD
  - Newton Development Tool Updates
  - Utilities
  - Q&A's
  - Sample Code
  - User Interface Guidelines
  - Apple Directions
- \$200 worth of Discounts on Newton development training classes
- Members-only access to the Newton area on Apple's on-line service
- Discounted rates for Apple's online Service
- Developer Handbook
- Developer Support Center resources
- Development hardware purchase privileges
- Worldwide Developer Conference invitation
- Newton Developer Conference invitation
- Third-Party Compatibility Test Lab

## NEW: THE NEWTON ASSOCIATES PLUS PROGRAM

This new program, announced in May, 1995, provides developers who need a limited amount of code-level support an option other than purchasing the unlimited support available in the Newton Partners Program. 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.

Annual fees for the Newton Associates Plus Program are \$500 and includes the following features:

### Newton Associates Plus Program Features:

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

## THE NEWTON PARTNERS PROGRAM

The enhanced Newton Partners Program boasts an even further reduced price and enhanced support features. The improved program includes all of the features of the Newton Associates program, plus unlimited expert-level programming support directly from Newton Systems Group engineers on the Newton platform via e-mail. Additional features include more hardware purchase privileges, and select participation in Apple-sponsored marketing activities, all available at a reduced price.

Annual fees are \$1500. Members of both the Macintosh Partners Program and the Newton Partners Program will now receive a discount of \$1000 off the combined membership fees, making dual platform support options even more affordable than ever before.

### Newton Partners Program Features:

- All Newton Associates Program Features
- Unlimited Newton programming-level support via e-mail
- Additional discounts on development-related Newtons
- Consideration as a test site for pre-release Newton products
- Select participation in Apple-sponsored marketing activities

While these programs are for US and Canadian Newton developers, many European developer groups will be following with similar programs and pricing structures. European developers should watch for more information and details on Newton programs. US and Canadian developers may obtain additional information and applications for the Newton Developer Programs by contacting the Apple Developer 

continued from page 1

## Small Parts: A Faster Way to Develop Large Applications

as follows:

1. Process the data for use on the Newton
2. Create the interface
3. Build and test the package
4. Refine the interface and data
5. Test the product
6. Build the final product

Step 1 can take quite a bit of time. A developer may often have a large amount of data that needs to be processed into data structures that can be used by a NewtonScript program, as well as the need to create indexes or other meta information that can be used to access the raw data. Whether this preprocessing step occurs on the Newton or as part of the NTK package build, it may well be quite a slow process. It's not uncommon for this step to take 10 minutes or more for a large amount of data, and adding 10 minutes to each compile/load cycle can drastically cut your productivity.

Steps 2, 3, and 4 are an ongoing process. Usually a developer would start by creating a simple interface for testing the data. Then the interface would be progressively refined. Note that each compile/download of the interface will have the overhead of recompiling and reprocessing the data; that means 10 or more minutes per iteration that could be better spent hacking instead of making coffee.

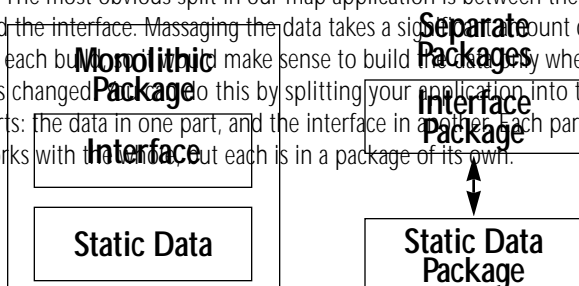
Once the product has been refined to a state that matches the original goals, the developer can proceed to step 5. Note that testing will probably require more changes, which means that each testing revision requires another 10-minute wait. You could easily waste several days worth of prime programming time waiting for those compile/download cycles.

Once the testing is completed and all those off-by-one errors have been fixed, you can build the final version of the product.

### A BETTER WAY

It's clear from the previous discussion that those long compile/download cycles are a time sink. A better approach would be to split the application up into logical parts so that you compile and download only what you need.

The most obvious split in our map application is between the data and the interface. Massaging the data takes a significant amount of time on each build, but it makes sense to build it once when it has changed. To do this by splitting your application into two parts: the data in one part, and the interface in another, each part works with the other, but each is in a package of its own.



A typical way to include a static data structure in an application is to define a constant and then reference that constant. In this case, you might see something like:

```
// In a project text file
DefConst('kMyStaticData, call BuildMyDataFunc with ());
...
// and somewhere in the application, we use myStaticData
val := ArrayPos(kMyStaticData, targetKey, 0, nil);
```

When you use this method, the static data is an integral part of the project; however, this means the data must be compiled and downloaded each time the project is built.

An alternative is to create a separate package for the data, and reference that data through a well known path. In the simplest implementation, you could build another form part and put the st **NTJ**

*continued from page 1*

## Taking Advantage of Newton Toolkit 1.5

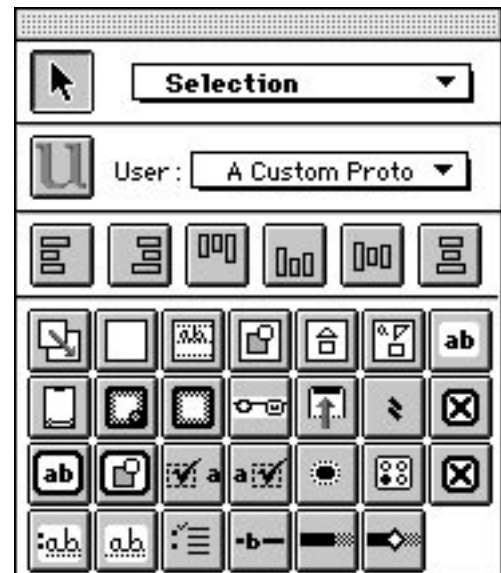
Seq_	Name	Type	Size	Mod. Date	Path Name
1	Included Package	Package	566	5/1/95-12:10 AM	Newton.Desktop.Folder.NTK
2	A Text File	Text	1201	5/1/95-12:10 AM	Newton.Desktop.Folder.NTK
3	A Custom Proto	Proto	1057	5/1/95-7:06 PM	Newton.Desktop.Folder.NTK
4	• A Layout	Layout	2346	4/30/95-10:25 PM	Newton.Desktop.Folder.NTK
5	A 2nd Text File	Text	1201	5/1/95-12:10 AM	Newton.Desktop.Folder.NTK

because your `InstallScript` needs to get at some object that isn't created until later in the build. You can still include a "Project Data" file early in the build order that includes all the project-wide constants, globals, or functions you need, but now you can add a text file named something like "Install & Remove Scripts" at the end of the build order to define these scripts.

It's much easier to create multi-part packages. If a package from another project is included in the project file, all the parts from the included package will be copied into the resulting package when the project is built. The build order is considered: if the included package is first, the parts it contains are output first; if the package is included at the end, the parts it contains are output after the part produced by the current project.

### Layout Improvements

The first thing you'll notice is that the floating palette that accompanies the Layout window is significantly smaller. Only the most frequently used protos are on the palette itself, which means there's a much better chance you'll be able to remember what all the icons are. All the protos are still available in the pop-up menu on the palette.



The new layout windows are much better about view justification, so the rectangles you see in the layout window now more closely match where the windows will appear on the Newton. (Tip: The rectangle shows where the outside of the view's border will be; the inner corner markers show the corners of the view's visible area.)



Finally, the two common screen sizes are readily available in the Layout preferences. (These are 240x336 for the Original Apple MessagePad, MessagePad 100, and Sharp ExpertPad, and 240x320 for the Apple MessagePad 110, Apple MessagePad 120, and Motorola Marco.) You can still enter your own sizes to see what your application would look like on a wristwatch- or whiteboard-sized Newton product.

### Browser Improvements

Developers spend most of their time working in the Browser window, so double-clicking a layout file in the project window now opens a browser instead of the graphical layout. `-B` still opens a browser as well, and layout windows are opened with `-L`. Once in a browser, it takes just a single click to open an editor on a slot. You'll find all of the browser-related menu items, including Browser Preferences, in the Browser menu.

The Text editor type is no longer required. Editing text slots is now done by default in a standard Evaluate editor, so you'll have to include double quotes for string literals. This change allows you to easily use constants or compile-time code for text slots, which aids in localization. The Text editor type is still available for compatibility. (Tip: Turn on "Show Slot Values" to see the type of each editor in the browser window.)

### Search

The List feature of NTK 1.0.1 has been extended into a useful project-wide search that lets you quickly locate views, slots, or text in slots. In NTK 1.5, searching is much faster and includes text files. The results are displayed in a list window; double-clicking on a line in that window will take you to a browser or editor with the result selected.

### Error Reporting

The same location features used in Search are also used when an error occurs while building. NTK displays a dialog box with the error message, adds the text of the error to the inspector window, and then highlights the line or lines where the error occurred, using an open browser if one is available. Compile-time warnings (such as using a global function that NTK doesn't recognize) are displayed in the inspector window only.

### Print Formats, User Protos, Layouts, and Constants

Print Formats as a special subclass of layouts are no longer necessary, so you won't find them in the New menu. A print format file was special in two ways: it was compiled first and it produced a variable so you could reference it in your application. Now you can put a layout file anywhere in the build order, and templates from all layouts are made available as described below, so print formats can be replaced with standard layouts.

Here "layout" means both proto templates and layouts. The only difference between a proto template and a layout in NTK 1.5 is that the proto template files in the project will be available in the layout palette as user protos.

A new function called `GetLayout` is provided to access the templates for layouts that have been compiled by NTK. `GetLayout` takes the name of the layout file as an argument, and evaluates to the template produced. `GetLayout` will not force the layout to be compiled; you'll have to change the order in the project window to fix

file dependencies, but `GetLayout` will generate an error if the layout hasn't been built yet.

As in NTK 1.0.1, every layout, print format, or user proto compiled also causes a constant to be created for the template produced. This constant, called `layout_thefilename`, is created as soon as the layout is compiled. However, `GetLayout` is the preferred method for getting at the template, because of the additional error checking it provides.

You must pass `GetLayout` a constant argument (a string literal or a constant containing a string). `GetLayout` is constant folded (see the following section), so you can safely use it in the body of a function such as `viewSetupChildrenScript`, as shown in this example:

```
viewSetupChildrenScript: func()
begin
  self.stepChildren := [GetLayout("My Special Proto")];
end;
```

### Constant Folding

Some expressions have only constant elements, so they can be evaluated as a function is compiled without changing the behavior of the program. This process is called constant folding, and NTK will now do this for some expressions. For example, in previous versions of NTK the expression `2 + 2` in a function body would produce code to add 2 and 2 at run-time. With NTK 1.5, this expression will be evaluated during the build and the constant 4 will be substituted. (More practical examples would be the folding of `vjParentCenterH + vjParentBottomV` for `viewJustify`, or `vClickable + vGesturesAllowed + vNumbersAllowed` for `viewFlags`.)

Frame and array access can be constant folded, if the frame or array being accessed is a constant. Consider the following constant frame:

```
constant kUninvitedGuests := `{
  thingOne: "A cat",
  thingTwo: "A hat",
}
```

If a function included the expression `kUninvitedGuests.thingOne`, the string "A cat" would be substituted by the compiler.

### More Constants

NTK 1.5 provides many more constants that allow your code to take advantage of the current compiler settings and other values stored with the project file.

The commonly used constants `kAppSymbol`, `kAppString`, `kAppName`, and `kPackageName` are now defined by NTK from the various settings dialogs. If your project already defines these constants, an error will occur during building when the constant is redefined. The old definition will be highlighted, and can easily be deleted. If for some reason you wish to keep the old constant definitions, the "NTK 1.0 build rules" switch in Project Settings will make NTK skip creation of these constants (among other things).

`kDebugOn` is set to `true` or `nil` depending on the state of the "Debug build" flag. `kProfileOn` will be `true` if compiling for profiling. `kIgnoreNativeKeyword` is set if the "Ignore native

keyword" value is checked; this will be discussed further later in this article. (The `debugOn` constant from NTK 1.0.1 is also set; it always matches `kDebugOn`.)

### Localization Support

Some features have been added to make it easier to localize applications for other languages. You define what's called a Localization Frame, which contains all the strings, bitmaps, or other data that needs to change based on the target language. To use localized information, you use `LocObj`, which will look up the appropriate information based on the currently selected language.

Since most developers will be starting with English projects, the English information is provided as a first argument to the `LocObj` function. For example, if you had a `title` slot in your application's base view with the string "A Demo Application", you might change the contents to this:

```
LocObj("A Demo Application", 'baseview.title)
```

When you have English selected as the language in the Project Settings dialog, `LocObj` simply evaluates to its first argument, so you'll still get the string "A Demo Application" in the `title` slot.

When some other language is selected, `LocObj` will use the second argument as a path expression to find the information in a localization frame. The localization frame contains a single slot for each language, and the path expression is applied to that slot to find the information. To specify the localization frame, you call the compile-time function `SetLocalizationFrame`. Here is a partial localization frame for French and German:

```
SetLocalizationFrame({
  french: {
    baseview: {
      title: "Un Logiciel de Démonstration", ...
    }, ...
  },
  german: {
    baseview: {
      title: "Ein Versuchsprogramm", ...
    }, ...
  });
```

When the language is set to French, `LocObj` will evaluate to the string "Un Logiciel de Démonstration"; for German, the string "Ein Versuchsprogramm" will be used.

Like `GetLayout`, `LocObj` must be passed constant arguments because it is constant folded. It's fine to use `LocObj` inside any function.

### Inspector Improvements

You'll notice a button bar along the top of the inspector window. The first button allows you to connect or disconnect the inspector from the Macintosh end without closing the window. The rest provide



easy access to debugging features like `PrintDepth`, `BreakOnThrows`, `StackTrace`, and `ExitBreakLoop`. There is also a button to set the global variable `trace` to `nil` (this button was at the bottom of the inspector in NTK 1.0.1).

The inspector output when an exception occurs at run-time is much prettier. The list of known exceptions has been extended, so you're less likely to need to look up error codes, and the exception symbol and all the exception data are displayed for those unusual cases. The output from `StackTrace` has also been made much more readable; only information that is relevant to debugging is displayed.

### SETTINGS AND PREFERENCES

Looking through the Project menu, you'll find three new dialog boxes that give you a lot more control over the way NTK builds your project and what it builds.

Project Settings gives you control over the aspects of the build-time environment for the project. You can choose the platform file from a list, and choose the language that applies to the `LocObj` function. Here you will also find several switches for the compiler, including controls for debug builds, profiling builds, the native compiler, and a compatibility option for old projects.

Package Settings gives you control over the package-wide parameters of the build, such as the user-visible name of the package, version number, and copyright message. Copy protection and compressions flags are here, as well as the flag that tells NTK to delete the old package when downloading a new one. One new flag is the "Auto remove package" setting. This is only useful with packages that contain auto parts, since it causes the package to be removed immediately after the `InstallScript` for the parts is run.

Output Settings lets you choose what the project produces. The default is still an Application, or form part. If you choose this part type, you'll be able to edit the name for the extras drawer, application symbol, and icon. It's now easier to create more than just applications, which is to say part types other than form parts. The Output Settings dialog has radio buttons for auto, book, and store parts, and is extensible.

### Building Books

NTK 1.0.1 produced book parts if you included a book file in the project. In NTK 1.5, book files (produced with the BookMaker application) are added like any other text file, and you must choose the Book part option. NTK 1.5 now lets you type in a name, symbol, and icon for the book parts. Normally this information comes from the book file, but if, for example, no icon is found in the book file, NTK will use the one in Output Settings instead.

### Auto Parts

Auto parts are now easier to create. These parts, which do not appear in the Extras drawer, can be selected in the Output Settings dialog. The name, symbol, and icon settings are not relevant to auto parts, and so are disabled. Auto-dispatch auto parts are created by choosing an auto part and selecting the "Auto remove package" option in the Package Settings dialog.

## Store Parts

NTK 1.5 now allows you to create package-based stores, called store parts or sometimes soup parts. Choosing "Store part" makes a compile-time global variable called `theStore` available. The functions available on this store are just like the ones for the internal or card stores at run-time. During build time you create soups on `theStore`, then add indexes and entries. It's important to give this store a name that's based on your registered signature – for example, `theStore:SetName("Demo:PIEDTS")`. When the part is downloaded, the package-based store – and the soups on it – becomes available through a pair of global functions.

`GetPackageStore` takes the store name and returns the store object contained in the part, and `GetPackageStores` returns an array of stores for all store parts installed.

You can't do much with package-based stores in the current Newton OS. They are read-only, and soups on them cannot participate in soup unions with soups on other stores. While you may consider using them for the convenient indexing that soups provide, you should keep in mind that soups are often not the most compact or efficiently accessed data structures for read-only data. Usually a collection of arrays or frames will be more efficient for read-only data. (See the article "Lost in Space" – available on the CD, AppleLink, and via ftp – for more details.)

## Custom Parts (Fonts, Dictionaries)

With the Custom Part feature and some tools which will become available around the same time as NTK 1.5, building custom fonts or recognition dictionaries is possible. If you choose Custom Part, you must specify the part type and the part contents yourself. The part type is entered in the small field next to the Custom Part radio button, and the part contents are entered in the field labeled Result. Commonly you will create a global frame or array in a text file, add data to it in other text or layout files, and then enter the global variable in the Result field. This will include all objects referenced by the variable in the custom part. (You could also type everything into the tiny Result field, or use the `Load` or `ReadStreamFile` functions to read in data, but these approaches don't take advantage of any of the nice features of NTK Projects.)

## Stream Files

You can take advantage of NTK's new stream file feature to speed up building some projects, especially those that include large data structures that don't change very often. For example, part number, price, or phone lists would be excellent candidates to place into a separate stream project.

## Making a Stream File

A stream file contains one NewtonScript object, but this can be a frame or array so in effect multiple objects can be included. Let's consider, for example, a database of employee names, phone numbers, and office locations. The data might be stored in three separate arrays, with each array sorted by the employee name. For example:

```
namesList := ["Anderson, Bob", "Walthrop, Royce", ...];
phonesList := ["315 555-4476", "419 555-3543", ...];
cubesList := ["PPR 51", "CL-217", ...];
```

(Tip: The NS array constructor is slow for very large arrays, so it's better to create them this way: `names := Array(1000); names[0] := "Anderson, Bob"; names[1] := "Walthrop, Royce", ...)`

Store the three arrays in a single object by putting references to them in a frame:

```
output := {
  kNames: namesList,
  kPhones: phonesList,
  kCubes: cubesList,
};
```

All this text is put in a text file (or several text files) and added to a project in NTK.

Specify that the stream file should contain this data by going to the Output Settings dialog, selecting the "Stream File" radio button, and typing `output` into the results window. (We could have simply typed the frame right there, but by using a single global variable we make it easier to add to the output later, as only the text file needs to be modified.)

Building this project will produce a file with a “.stream” extension, usually referred to as a stream file.

### Using a Stream File

Using the three arrays in another project requires that the stream file be read in to it during the build. This is done by using the `ReadStreamFile` compile-time function. You could add a new text file called “Streamed Data” to the project to control this, and put the following code in it:

```
call func() begin
  foreach symbol, value in
    ReadStreamFile(HOME & "MyData.stream") do
      DefConst(symbol, value);
  end with ();
```

Now the data from the streamed project is available in three constants for use in the rest of the project. Building a project using a stream file will be much faster than constructing the arrays each time the project is built. You just have to remember to build a new stream file when the data changes.

(Tip: During development it may be even faster to put the database in a separate package and make the three arrays available via a global variable. Doing this allows you to avoid the time needed to download the database each time the application changes. See the Application Design section of the Q&As for details on this approach.)

(Tip: It may be worthwhile to adopt a convention where the top-level object in the stream file defines an `Install` function, which would contain code like the above Streamed Data code. Then the line to read in the stream file and create the constants is simply `ReadStreamFile(HOME & "MyData.stream"):Install()`

### Combining Objects

It's possible to save space in most packages by combining objects that are identical. For example, if the string “New” appears in a text slot for two different buttons, both slots could reference the same string. The behavior of the program would be virtually unchanged.

In previous version of NTK, the only way to accomplish this kind of object sharing was to create a compile-time constant or global variable to hold the object, and explicitly use the “shared” reference wherever the object was needed.

NTK can now combine objects as a final step in the build process. Because this takes extra time, it will normally do this only when the “Compile for debugging” switch in Project Settings is off. Since combining objects is useful only for frame-based part types, NTK will not combine objects in store parts or stream files. In version 1.5, only frame maps and binary objects (including strings and bitmaps) will be combined.

Combining objects usually results in smaller packages; typically a 10% to 20% savings can be achieved for large projects. But combining objects can also change the behavior of the program in two ways. The first is that objects that would have been different may now be identical, so that the `=` operator will evaluate to `true` for these objects. Since the objects are always read-only, it would be unusual to compare them against each other, so it's difficult to imagine how this could affect most programs.

The other effect that combining objects can have is to change the

locality of the stored program. Here we need to take a short digression into how data is stored on the Newton.

When writing the code and data to a package, NTK generally ends up writing objects when references to them are encountered. Initializers for variables in a function tend to be stored near the function's instructions, and elements of an array or frame tend to be written close together. Because there is now only one copy of an object where before there may have been many, the object will be stored near the first place that the program references it. This may not be near other parts that also use the object.

This makes a difference because data stored in packages on the current Newton OS is paged in to system memory as it is accessed. (That's right, the Newton OS uses a virtual memory model for packages.) Because objects may not be near the code that references them, more segments of the project may need to be paged in to run the program. (The fact that there are fewer pages overall may reduce the cost for this paging.)

We don't currently know what performance impact combining objects has on applications, so NTK provides a way to skip the combination step. NTK 1.5 will check for a global variable called `consolidateObjectsAfterBuilding`. If the variable exists and has the value `nil`, consolidation will be skipped. (Setting the variable `true` will not force consolidation for debug builds.)

### Apple Events

Being MacApp based, NTK has always had support for AppleEvents, but in previous versions that was limited to the four required events (Open App, Open Doc, Print, and Quit). NTK now responds to two additional AppleEvents. One corresponds to the Build Package menu item, which will come in handy for those trying to automate building large projects. The other is a “do script” event, where the “script” is NewtonScript code. The script is compiled and executed in the NS environment on the Macintosh, and a text representation of the result is returned. (Currently this result is limited to 256 characters.)

### Projector Compatibility

NTK 1.5 is much friendlier to Projector. It will no longer delete ‘ckid’ resources in text files, layouts, protos, or projects, making it much easier to use Projector to manage your NTK projects. The output packages, stream files, and exported text files are still created from scratch each time a build or export takes place, so you may still have some difficulty keeping these produced files in Projector.

### PROFILER

A major new feature in NTK 1.5 is the NewtonScript profiler. The profiler collects and summarizes performance data about the system and your application.

### A System Update for MP100, MP110 Is Required

Profiling requires some support from the Newton OS. The supporting routines are built in to the Apple MessagePad 120 and Motorola Marco, and system updates are provided with NTK for the Apple MessagePad 100 and Apple MessagePad 110. (You cannot profile with an Original Apple MessagePad or Sharp ExpertPad.) Other than including the profiling support, these updates are identical to the

latest updates for those products. Because profiling support requires some memory and some changes to the way functions are called, installing the profiling updates may slightly reduce the products' performance. (That is, it's probably best *not* to install the profiling updates unless you actually plan to use the profiler.)

Profitable MP100s and MP110s can be recognized by a "p" at the end of the version number at the bottom of Prefs. When the NTK Toolkit App is installed on a profitable Newton product, an additional button will appear in its slip called "Profile Control." The same Toolkit App will function normally on non-profitable units.

### How to Use the Profiler

Profiling your code requires you to do two things. The first is to make your application profitable; the second is to add calls that tell the OS when to start and stop gathering data; and the third (three things) is to turn on the profiling features of the OS.

Making your application profitable is easy. Simply turn on the "Compile for profiling" check box in the Project Settings dialog. This causes NTK to keep information about your application when it builds. (Fine point: The information isn't saved with the project, so you'll need to make sure you build once before starting to profile after launching NTK.)

Because profiling native compiled code adds a higher overhead than profiling code that is interpreted, you have the option of skipping profiling for native functions. Check the "Profile native functions" checkbox in Project Settings to have the system gather data for native functions called from within other native functions. (If the box is unchecked, all the time spent in native functions will be attributed to the first native function called from an interpreted function.)

Telling the OS when to start and stop gathering data is accomplished by calling a new run-time global function called `EnableProfiling`. Pass `true` to begin gathering data, and `nil` to stop. The function returns the state the profiler was in before `EnableProfiling` was called. Typically you might turn off profiling during initial setup for a function call, or while interacting with the user, and turn it on when the heavy processing is taking place. This allows you to filter out the other work the system does as idle tasks or while waiting for input.

It may be a good idea to test the constant `kProfileOn` before calling `EnableProfiling`, to make it easy to turn off all profiling support for your application. For example:

```
if kProfileOn then EnableProfiling(true);
```

The profiling features of the OS itself are controlled through the Profile Control slip in the Toolkit App. When profiling is not active, a single button labeled "Begin Profiling Run" is visible. Tapping this button sets up the OS to gather data. When a run has started, two buttons are available, one to stop profiling and upload the results gathered, and another to abort the profile run. When data is actually being gathered (when `EnableProfiling(true)` has been called), the upload button changes to the text "Profiling Enabled." It is not possible to upload the data while it is being gathered.

In summary, there are four steps you need to take to profile your application. First, change the Project Settings. Second, insert calls to `EnableProfiling` before and after the code you're interested in.

Third, compile and download the application. Fourth, on the Newton end open the Profile Control slip, begin a profiling run, and run the application. Finally, (five steps,) upload the results back to NTK, where they will be displayed in the inspector window.

(Tip: The inspector does not need to be connected while profiling your application. Because the inspector connection takes some system memory, you'll get more accurate results by disconnecting it before profiling. If you tap the Upload Results button and the inspector is not connected, the Toolkit App will open so you can connect. Once connected, the upload will take place.)

### Profiling Options

There are two settings available in the Prefs for the Profile Control slip. The buffer size setting controls the amount of RAM allocated to storing profiling data. The default is 4K, which is enough to profile almost all applications. Since the memory is allocated from a shared area that is also used for running the OS, it's best to keep the allocation as small as possible. (If the profiler runs out of memory, statistics gathering is stopped and NTK will report this along with the partial results obtained. You should only increase the buffer size if this happens.)

The other setting in the preferences slip controls whether or not the profiler will gather data about calls to system functions. If "Detail System Calls" is checked, the time spent in each system function will be displayed individually. This can add a lot of data to your summary. If unchecked, the profiler will display just one line for all the system calls together. (I generally want to know which system functions are being called and how much time is spent in each, because I can change the way functions are implemented to minimize calls to expensive functions.)

### Interpreting the Results

The top section of the profiler results gives overview information about how much time was spent in your code and how much was spent in the system. Occasionally you will see an "Other" line; this shows time spent in functions in the NewtonScript heap that aren't from your application. Functions from other packages are included with the system functions in the summary. If native code is involved, you will see the number of calls to interpreted functions from native functions (this will be discussed in the next section). The summary will also show the number of times garbage was collected, and the time spent doing it.

Below that will be a detailed list of all the functions called while profiling. The profiler shows the number of times a given function was called, along with the total time spent in the function in milliseconds and the percentage of total profiled time taken up by that function. The list is sorted by percentage of time, with the worst offenders at the top of the list.

Interpreting the results depends on what the program is intended to do and how it is being profiled. Here are six things to look out for:

- Functions using most of the time. Can they be implemented more efficiently? Consider these functions for native compilation. (Be sure to read the next item first.)
- A function called twice where you only expect it once. This could

also be a function called twice as many times as neighboring functions. Usually these functions turn out to be "accessor" functions to get some data from some object. Performance can be boosted by keeping the result in a local variable instead of calling the function repeatedly.

- Several functions all called the same number of times. Because function calls do take significant time with the NewtonScript interpreter, it can be more efficient to write a single large function.
- A function called many times. Check the code that calls that function. If the function is called within a loop, check and see that the function is being called efficiently. Performance can usually be boosted by moving the code out of the function and into the loop, thus removing the function call overhead. (Doing this will often uncover other optimizations that can be made, given assumptions about how the function is called in this circumstance.)
- Many calls to GC or time spent in garbage collection. This means your application is generating a lot of intermediate structures. Cutting down on this can be hard, and often involves significant changes to code to re-use existing frames, arrays, or binary objects.
- (Five things)

The key to profiling effectively is to test, theorize, change, and test again. Change just one thing at a time. Some changes may help, while others may not, so if you make many changes at once you won't know which helped or how much. Sometimes a change that seems like a great idea (compile that function!) will slow things down because the application will be bigger or paged differently and cause thrashing.

### When You're Done

Don't forget to turn off the profiling switches and remove calls to `EnableProfiling` before making your application available to other people. While projects with profiling enabled will not damage data in units that don't support profiling, they will be slower than non-profiled code. The `EnableProfiling` call will not be available on units that don't have the profiling support, which may cause an "unknown global function" exception to be thrown.

Remember: don't optimize too soon. The right time to start working on performance is after everything else is the way you want it. Wait until you can identify the parts of the application that need performance help. Until then, you're better off writing clean and easy-to-read code. (Usually, most of your application won't need any help at all.)

### COMPILER

The native NewtonScript compiler for the ARM processor is the most eagerly awaited part of the NTK 1.5 release. Before you rush out and `nativ` – compile your entire application, there are some things you should know.

### Compiling Can Slow You Down

The native compiler is not a panacea. Typically, your application spends 90% of its time in system calls. Native-compiling a function that

just calls system functions does no good. It simply increases code size and slows down execution speed. Native compilation also increases the time it takes to build. It pays to be selective about which functions to native-compile.

Why would it slow things down? There are two reasons. Native functions are big, typically 8 to 10 times the size of the equivalent interpreted code. Bigger functions mean more paging.

Also, calling interpreted functions from within native functions is slow – much slower than calling interpreted functions from interpreted functions or native functions from native functions. The extra cost is attributed to the time needed to initialize the interpreter. Because many system functions are interpreted, you can't always remove all the interpreted calls from native code. The NS profiler will count the number of times an interpreted function is called from a native function; you should keep an eye on this number.

### Use the Profiler First

Before making functions native in your application, use the profiler to find good candidates for native compilation. Often, while doing this you'll find other coding improvements that will help more than simply compiling native. (No matter how much faster they are executed, efficient algorithms will be better than inefficient ones.)

### Using the Compiler

Enabling the native compiler is very easy. Simply insert the `native` keyword after the `func` keyword and before the argument list. `func(foo)` becomes `func native (foo)`. The `native` keyword is ignored for functions used at build time or functions typed directly into the Inspector.

### Taking Full Advantage of the Compiler

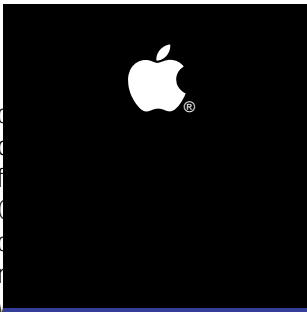
Simply adding the `native` keyword will get you native-compiled functions, but there are some additional things you should do to take full advantage of the compiler.

The most important thing (besides writing efficient code in the **NTJ** place) is to declare the types of your variables when possible. The compiler in NTK 1.5 recognizes integer and array types, and can generate much more efficient instructions if the variable type is



Newton®

If you have an idea  
for an article you'd like to write for Newton  
Technology Journal, send it via Internet to:  
[piesysop@applelink.apple.com](mailto:piesysop@applelink.apple.com)



# 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 unlimited expert-level development. All programs provide focused Newton development information and computer options development hardware, software, and tools—all of which can reduce your organization's development time and costs.

to take advantage of this, you should consider these drawbacks:  
integers will be shortened to 30 bits whenever they are passed to a

function or stored in an untyped variable; the interpreted bytecodes may be used instead of the native code on some platforms, and interpreted code will always be used if you ever set the "ignore native keyword" option.  
Development time and costs.  
code. For this reason, NTK offers an "Ignore native keyword" option in the Project Settings dialog. Turning on this option causes NTK to skip

## Newton Associates Program

Annual fees are \$250.

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*

## 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

### 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

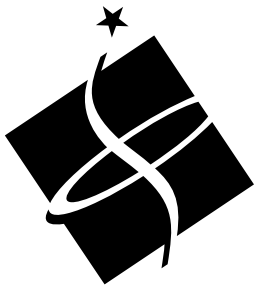
Cupertino, CA 95014-6299

AppleLink: DEVSUPPORT

Annual fees are \$1500.



Newton



**STARCORE™**

---

## *Dear Newton Developer,*

*We'd like to introduce you to StarCore, the software publishing and distribution arm of the Personal Interactive Electronics Division at Apple Computer, Inc. As a Newton developer, you are already involved in creating products for this exciting technology. There are many ways in which we can build relationships that will benefit you and the Newton platform.*

*At StarCore, we are actively recruiting titles for the Newton. StarCore can provide developers with a broad range of services and opportunities. The developer creates the software, StarCore provides the packaging, manuals, testing, user studies, marketing and end-user support.*

*We are anxious to talk with developers about products or concepts they would like to see published or distributed. We are particularly interested in business-oriented applications that would appeal to a mobile professional. We are also looking for products that have connectivity to Macintosh and Windows desktop applications.*

*Please contact us at:*

*StarCore  
Apple Computer, Inc.  
5 Infinite Loop, MS 305-3C  
Cupertino, CA 95014  
Attn: StarCore*