



Newton[®] Technology

Volume 1, Number 1

February 1995

Inside This Issue

Business Opportunities	
Newton Delivers for Vertical Markets	1
Communications	
Newton Communications	1
Developer Group News	
New Developer Support Programs For Newton Developers	3
NewtonScript Techniques	
Exception Handling in NewtonScript	4
Understanding NewtonScript	
NewtonScript Functions	7
Newton Communications	
Beam Me Up, Newt!	10



Business Opportunities

Newton Delivers for Vertical Markets

By Jane Curley, Apple Computer, Inc.

Apple's Newton platform delivers powerful new technologies that can help create entirely new markets for developers. The combination of power, ease-of-use and personal portability makes it a great vehicle for attacking and solving problems whose solution can not only open up new business opportunities, but also make a real difference in how people can do their work. Recently, we came across a project in Boston that provides an excellent example of this process.

Health care professionals have a unique need for access to the resources of massive amounts of information at their fingertips, often immediately. Much of the information they reference daily in life or death situations is not only clinical data, but also educational resources, decision aids, and other professional and practical information. Many times, health care professionals do not consult reference materials when making clinical decisions. This is due, in part, to the fact that there is a lack of availability of resource information in areas where it is needed most. But Apple's Newton MessagePad in the health care environment is changing some of that.

Boston's Brigham and Women's Hospital will begin a joint pilot program with the Massachusetts General Hospital using Apple's

continued on page 14

Communications

Newton Communications

by Bill Worzel, Arroyo Software
ArroyoSeco@eworld.com

Newton communications programming is often viewed with a mixture of awe and fear. Somehow the notion of programming a PDA to connect to everything from wireless devices to infrared to networks seems slightly unreal.

Once you begin to explore the communications chapter in the Newton Programmer's Guide (NPG), astonishment may give way to puzzlement. The situation often becomes worse when a programmer begins to write and debug code.

This article is intended to make things easier for the programmer by giving more details, code examples and inside looks at the architecture of the communications side of the Newton. This article first discusses the architecture and fundamentals of creating, connecting and using Newton endpoints. Next, it talks about serial and modem connections. Lastly, a related article (Beam Me Up Newt!) discusses use of the infrared link for connection both to other Newtons and to remote home appliances such as televisions, VCRs, CD players, and so on.

This article assumes you are familiar with the Newton, NewtonScript and the Newton Toolkit (NTK). Because unusual programming techniques and constructs will be discussed, it also helps if you have some experience in doing communications programming. This article is designed to cover much the same ground (though not in as much detail) as is

continued on page 15

Published by Apple Computer, Inc.

Lee DePalma Dorsey • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

Tony Espinoza • *Coordinating Editor, Technical Tools Marketing*

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, Steve Strong, Bruce Thompson

Contributors

Jane Curley, Steven E. Labkoff, Julie McKeenan, Neil Rhodes, Bill Worzel

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

Scott T Boyd • *Editor*

John Kawakami • *Editorial Assistant*

Judith Chaplin • *Art Director*

© 1994 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, Light Bulb Logo, Mac, MacApp, Macintosh, Macintosh Quadra, MPW, Newton, Newton Toolkit, NewtonScript, Performa, QuickTime, 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, and Sound Manager are trademarks, and ACOT is a service mark of Apple Computer, 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. 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*

Welcome to the premier issue of Newton® Technology Journal! The Personal Interactive Electronics (PIE) division at Apple Computer, Inc. is pleased to be able to bring you this new, bi-monthly publication. From its first conception, this journal has been designed and tailored specifically for you, the Newton platform developer. Our intent is to make Newton Technology Journal your primary resource for the latest information on Newton platform technology, Newton development tools, and PDA business and market news from Apple Computer and our licensees. While our focus will be primarily technical, we also want to provide our developers with a well-rounded picture of the opportunities that the Newton platform is building for the developer community.

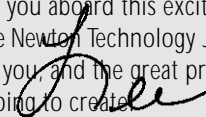
As Managing Editor of Apple's Newton Technology Journal, I know I speak for the entire publishing team when I say I'm thrilled to be able to provide such a support resource to Newton developers. With our recent introduction of a full range of Newton Developer Programs in December, 1994, we're really filling out our range of services to assist developers in creating and publishing winning Newton applications. I hope Newton Technology Journal will go a long way in making those support services indispensable to you.

In this premier issue, you'll find articles on Newton communications, excep-

tion handling, and function objects, written by expert Newton programmers Bill Worzel, Neil Rhodes, and Julie McKeenan. You'll also get a recap of the recently introduced Newton Partners and Newton Associates Developer Programs. On the business side, we'll give you a look at a vertical market success story in the medical arena.

We'll look forward to bringing you the latest in technical and business news on the Newton platform in future editions. We also invite you to send your comments, suggestions, and article ideas to us via Internet at piesysop@applelink.apple.com. The Newton Technology Journal is the perfect forum to share your tips, tricks, and programming expertise. The Newton platform is growing fast, and we hope to grow our community of authors and contributors equally fast with your help. Do you have an article you'd like to write and have published in Newton Technology Journal? Just let us know.

For those of you currently developing for Newton, congratulations on your accomplishments and successes thus far. For readers just beginning their Newton development experience, I'm happy to welcome you aboard this exciting platform. The Newton Technology Journal is for all of you and the great products you're going to create.



Apple Computer, Inc.
would like to thank Xplain Corporation (the publishers of MacTech™ Magazine)
for lending their expertise in producing the Newton Technology Journal.

New Developer Support Programs For Newton Developers

Apple Computer, Inc.'s Personal Interactive Electronics Division (PIE) and the Apple Developer Group are pleased to announce the introduction of the Newton Associates Program for Newton MessagePad and Newton platform developers, as well as several enhancements to the Newton Partners Program (formerly the PIE Partners Program). Effective December 1, 1994, Newton developers now have a range of support services from which to choose to meet all of their development needs.

"Answering the developers' call for affordable, quality support, the establishment of the Newton Associates Program further reinforces Apple's ongoing commitment to the Newton platform and its developers," said Shane Robison, Vice President and General Manager of Apple's Personal Interactive Electronics Division. "Developers are crucial to the platform's success and viability and we are very pleased to be able to expand support to a broader developer community."

Newton Associates Program

The Newton Associates Program is a low cost, high quality, self-help development support program. It is available for a US\$400 annual fee and includes the following features:

- Support services from Apple's Developer Support Center
- Discounted rates for online technical information
- Access to a technical Q&A reference library
- Discounts on Newton and Macintosh hardware
- Receipt of regular Newton developer mailings which include the Newton Developer CD and the Newton Technology Journal

- Use of Apple's third party compatibility lab
- Discounts on Newton development classes
- Automatic invitation to the Newton and Worldwide Developer Conferences
- Eligibility to participate in StarCore's Affiliate Label Program

Newton Partners Program

The Newton Partners Program (formerly called the PIE Partners Program) includes all the features of the Newton Associates Program plus services such as expert-level programming support via e-mail, free updates to Newton development tools and participation in select Apple marketing and PR opportunities. The price has been reduced from US\$2850 to US\$2500 annually. In addition, the Newton Partners Program has been enhanced with new features such as a Newton-focused monthly mailing, a Newton orientation kit, regular delivery of the Newton Developer CD, and the Newton Technology Journal, a new technical publication for Newton developers.

Both the Newton Partners Program and Newton Associates Program are only available to developers in the US and Canada at present. Developers outside of the US and Canada should contact their local Apple office for details on current or planned Newton support options in their region.

For more information on joining the Newton Associates or Newton Partners Program, you may contact the Apple Developer Support Center at (408)974-4897, Internet: devsupport@applelink.apple.com or AppleLink: DEVSUPPORT. N



To request information or an application on Apple's Newton developer programs,
contact Apple's Developer Support Center
at 408-974-4897

or Applelink: DEVSUPPORT
or Internet: devsupport@applelink.apple.com.

Exception Handling in NewtonScript

AN OVERVIEW OF EXCEPTIONS

Exceptions are a way of dealing with errors. An exception is a report of an abnormal condition. It terminates the execution of the current function and of the function that called it, and so on up the call chain until an exception handler is found to respond to the error. The exception handler in turn is usually responsible for cleanup and for reporting the error/exception to the user. Exceptions are an alternative to returning error codes.

Figure 1 shows how this whole mechanism operates. Here we have function A calling B calling C calling D calling E. Function B has the exception handler and function E throws the exception. At this point, Function E terminates followed likewise by D and C. Function B's normal flow is stopped and execution continues within B's exception handler.

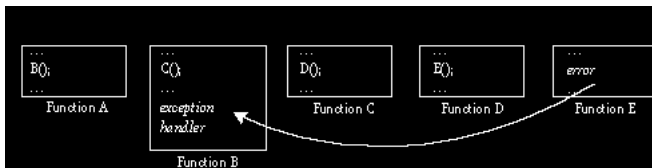


Figure 1.

THE ADVANTAGES OF USING EXCEPTIONS

In your programming, there are a couple of clear advantages to using exceptions as opposed to returning error codes. The first advantage is that functions that do not generate exceptions (like C and D), but which call functions that do, don't need to do anything special. Without any extra work, the exception will be propagated to calling functions (like B). On the other hand, if you relied only on error codes, C and D would need to return error codes as well, since they call the function with the error in it.

The second advantage is the simplification of functions. Since the execution of a function terminates as soon as an exception occurs, you can be sure that the statements prior to the exception executed successfully. Conversely, functions which use error codes commonly look like this:

```
X := func()
begin
  error := A();
  if error == noErr then begin
    error := B();
    if error == noErr then begin
      error := C();
    end;
  end;
  return error;
end;
```

It's clear that the logic of the functions can get lost in the error-checking. With exceptions, however, X could be neatly written as:

```
X := func()
begin
  A();
  B();
  C();
end;
```

WORKING WITH EXCEPTIONS

Having a fairly good idea why programming with exceptions makes sense, note the type of code that would profit from it use. Code that typically follows this pattern will be of interest:

```
set some state
code which could cause an exception
restore the state
```

Of the types of states that benefit from exception handling, it is worth reviewing those states you might be familiar with from other types of machines, and then looking at the situation with the Newton.

EXCEPTION HANDLING IN OTHER ENVIRONMENTS

On desktop machines using traditional languages, exception handlers are common. Here are some examples of common states that require exception handlers:

- Allocate some memory
Do something with the memory that could cause an exception
Deallocate the memory
- Lock some memory
Do something with the memory that could cause an exception
Unlock the memory
- Open a file
Write to the file
Close the file
- Create a temporary file
Use the temporary file
Delete the file

Each of these requires an exception handler. When a resource has been acquired, it must be given up in the event that an exception occurs.

EXCEPTION HANDLING IN THE NEWTON ENVIRONMENT

Because of NewtonScript's superior handling of memory, the need for exception handlers is minimized. Since NewtonScript has garbage collection, explicitly deallocating memory is not necessary. Likewise, the Newton has no memory locking, making unlocking unnecessary as well. Another nice point is the Newton's lack of files; if you don't have a file open, you don't have to close it.

As you can see then, none of the typical states on other platforms are a problem on the Newton. So what is? In applications, two common places you will need exception handling are when calling the `EntryChange` method and the `PutAway` method used for beaming and mailing. A rarer situation involves the creation of a temporary soup on the Newton; in this case you would use exception handling to ensure its deletion.[†]

The Structure of Exceptions

This section describes how you would write an exception handler using proper NewtonScript syntax. Next, it looks at the hierarchical relationship of exceptions.

Exception Handling Syntax

The actual syntax for exception handlers in NewtonScript is:

```
try
  code which could generate an exception
onException exceptionSymbol do
  code to handle this type of exception
onException exceptionSymbol2 do
  code to handle this type of exception
```

To propagate an exception from within an exception handler, use the `Rethrow` function:

```
Rethrow();
```

The Hierarchy of Exceptions

There is a hierarchy of exceptions (Figure 2 shows a small subset of the whole tribe). This hierarchy exists so that you can write exception handlers for a specific exception, a group, or for all of them.



Figure 2.

Every exception symbol contains an `evt.ex` part (since the symbols contain embedded periods, they must be surrounded by `||`). Portions after the `.` part signify lower levels in the hierarchy. For example, the exception `|evt.ex.fr.intrp|` is a kind of `|evt.ex.fr|`

exception. You use semicolons (as shown in Figure 2) to separate multiple exception types.

So, if you wish to catch all exceptions use:

```
onException |evt.ex|
```

You can also use more specific exception types. For instance, to catch only `type.ref` exceptions, use:

```
onException |type.ref|
```

It is also possible to cascade exception handlers. In this case, the first one that matches the current exception is the one that gets called. For example:

```
try
  code
onException |evt.ex.div0|
  deal with divide-by-zero error
onException |type.ref|
  deal with a type error
onException |evt.ex|
  deal with all other errors
```

Using Exceptions in Newton Applications

There are two typical problems you might encounter on the Newton that could benefit from the use of exceptions. The first involves using `EntryChange`, and the second involves the `PutAway` method.

Exception Handling and EntryChange

The `EntryChange` method is the mechanism used on the Newton to allow an application to save a modified entry back into its soup. Although the view system will provide a reasonable notification to the user (see Figure 3), you will see that more is needed.

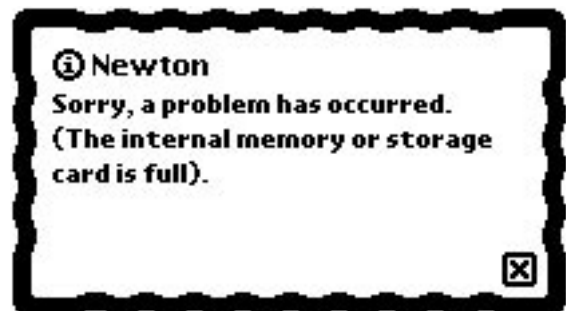


Figure 3.

Here is why. These code snippets handle saving a modified entry when the user scrolls up:

```
viewScrollUpScript: func()
begin
  ...
  :SaveModifiedEntry(currentEntry);
  :DisplayPreviousEntry();
end;

SaveModifiedEntry: func()
begin
  ...
```

```
EntryChange(currentEntry);
...
end;
```

If `EntryChange` throws an exception, the notification shown in Figure 3 will be presented by the view system. A problem remains, however. The user is stuck viewing the current entry and any effort to save it (like closing the application), causes `SaveModifiedEntry` to throw an exception (because of `EntryChange`). Thus, no progress can be made.

The solution is to change the behavior of `SaveModifiedEntry`. Rather than having it swing between successfully saving or throwing an exception, modify its behavior to successfully save or to notify the user of the error. You can also rename the function to describe more accurately its new responsibilities:

```
viewScrollUpScript: func()
begin
...
:HandleModifiedEntry(currentEntry);
:DisplayPreviousEntry();
end;
```

```
HandleModifiedEntry: func()
begin
...
try
EntryChange(currentEntry)
onException |evt.ex.fr.store| do
begin
:Notify(kNotifyAlert, EnsureInternal(kAppName),
EnsureInternal("There is not enough space to
save the modified item"));
EntryUndoChanges(currentEntry);
end;
...
end;
```

The call to `EntryUndoChanges` is necessary; that is how you wipe out modifications made to the entry. Otherwise, the soup holds the unmodified entry, while the entry cache contains a modified entry. The right course of action is to have the view reflect the fact that you were unable to save the entry. Thus, you must revert to the saved entry and then display that version.

WARNING

The union soup method `AddToDefaultStore` does not throw

an exception when a store is full, although the documentation says that it does. In reality, `AddToDefaultStore` shows that it fails by returning `nil` and by calling `Notify` to alert the user. This failure to follow documented behavior can create problems for applications that do not check to ensure that `AddToDefaultStore` succeeds.

You might want to consider writing a wrapper around the `AddToDefaultStore` method that will throw an exception in the event that the `AddToDefaultStore` method returns `nil`:

```
DefConst('kMyAddToDefaultStore, func(unionSoup, frame)
begin
result := unionSoup:AddToDefaultStore(frame);
if not result then
Throw('|evt.ex.fr.store|, '{error: 10617}');
return result;
end);
```

Exception Handling and PutAway

The other place where exception handling is commonly needed is in the `PutAway` method, used for beaming and mailing. Remember that `PutAway` calls `kRegisterCardSoup`, adds an entry to a soup, and then calls `kUnregisterCardSoup`. Since adding an entry may throw an exception, an exception handler is needed to call `kUnregisterCardSoup`. Here is a stripped-down `PutAway` method that shows the problem:

```
PutAway := func(item)
begin
local soup;

soup := call kRegisterCardSoupFunc with (kSoupName,
kSoupIndexes, kAppSymbol, kAppObject);
soup:AddToDefaultStore(item.body);
call kUnregisterCardSoupFunc with (kSoupName);
end
```

The Sample Code with Exception Handling

Now, look at how to fix the problem with exception handling:

```
PutAway := func(item)
begin
local soup;

soup := call kRegisterCardSoupFunc with (kSoupName,
kSoupIndexes, kAppSymbol, kAppObject);
try
```

N



To send comments or to make requests for articles in Newton Technology Journal,
send mail via internet to: piesysop@applelink.apple.com

NewtonScript Functions

FUNCTION OBJECTS IN NEWTONSCRIPT

In NewtonScript, function objects (sometimes called closures) are first-class objects that 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, make a `Clone`, or `DeepClone` of 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.

Function Environment

To understand how you can use this aspect of a function in your applications, it's good to review the environment of most NewtonScript functions. Usually, you create a function object at compile time. These could be slots in a template, edited using a slot browser, or they could be functions created in a Project Data file. In either case, the environment that exists is the top-level 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 matter. This will be a run-time function object. Note that by definition, these will be nested functions (ones created inside other functions).

For 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 local variables, as well as those of the function in which it is nested.

How Function Objects Are Used – Call/Perform/Apply

Before talking about how you can use function objects in your programming, it's necessary to address the manner in which you call a function object. There are four ways to 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>
```

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. Frames with methods don't provide the same functionality, though it might appear they do. 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`. Though `MakeAccount` is no longer executing, the nested function `Deposit` references `balance`, and so 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, you 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 the frame above is used only as a way to store two named values. No object programming has started, 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 `DateFind`:

```
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 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;
  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.

In NewtonScript, some allowance needs to be made for variables that are closed over; that is, variables that 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 could not help but 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

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 function object.

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 function object is created. Calling a function object uses the value of `self` stored in its message context.

The major difference between calling a function object 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 function object to be ignored.

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` function object 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

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.

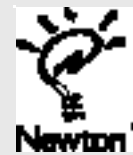
Function Objects Created at Compile Time Have No Lexical Environment or Message Context

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 to remember when you are creating standalone function objects (those that have no references to your package). Examples of such function objects are those that will be copied to a soup, or the one used as a `postParse` routine for Intelligent Assistance. If you use a function object that has a non-empty message context here, the whole message context will be copied into the soup (or into memory in the case of Intelligent Assistance). 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 run time.

N

SUMMARY

In NewtonScript, function objects are first-class objects which have



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 or AppleLink: PIESYSOP

Beam me up, Newt!

INTRODUCTION

The infrared (IR) beaming device that is embedded in the head of the Newton can be used to send messages between Newtons. You may have already used this facility to "Beam" a note or card from your Newton to another. This article will show how to send messages and data directly from your application. Another use of the IR beam is to control remote devices such as a VCR or TV. You will learn how to set up a control pattern that will control a CD player.

IR Beaming Protocol

Beaming between Newtons requires a half-duplex, asynchronous framed protocol. This means that only one Newton can send messages at a time. Delivery of the message to the receiver cannot be not guaranteed. If the Newtons are not pointed correctly, the data can't be passed between them. However, if the data does arrive, the message is guaranteed to be complete and correct. The sending Newton will try for about two minutes to complete a transmission before giving up and throwing an exception.

The IR Endpoint

As is the case with the other Newton communication tools, the focal point is the endpoint through which NewtonScript links to the outside service. The IR Beaming endpoint service is defined as follows:

```
myEndPoint:= { _proto: protoEndpoint,
               configOptions: [
                 {label: kCMSSlowIR,
                  type: 'service',
                  opCode: opSetRequired }
               ]
             }
```

Although two Newtons passing IR information define the endpoint in the same way, they connect the endpoint differently. The passive receiver must set the endpoint to wait for a connection using:

```
myEndPoint:Listen(nil);
```

while the active sender must connect the endpoint using:

```
myEndPoint:Connect(nil,nil)
```

If the two Newtons are unable to make a successful connection within 2 minutes, an exception `error -38001` is created. To avoid this error condition, and to allow a user to abort the connection in a more reasonable time frame, like 10 seconds, you may want to connect using `AddDeferredAction` and `AddDelayedAction`. The `AddDeferredAction` method happens almost immediately, but runs in a separate Newton task thread, allowing you to continue with

your own code at the same time that the connection attempt process is being handled. For the active sending Newton, the complete endpoint creation and connection functions are as follows:

```
TryToConnect: func() begin
  err := myEndPoint:Instantiate(myEndPoint,nil);
  if err then begin
    :HandleError();
    return;
  end;
  AddDeferredAction(func(ep) ep:Connect(nil,nil),[myEndPoint]);
  AddDelayedAction( func(ep) begin
    if ep:State() < 5 then begin // Not Connected
      ep:Abort(); // Kill the connection attempt
      AddDelayedAction( func(ep) ep:Dispose(), [ep], 500 ;
        // Remove the endpoint
      GetRoot():Confirm("", "Try Again?", base,
        'ConnectAgain);
    end;
  end,
  [myEndPoint], 10000);
end,
ConnectAgain: func(OK) begin
  if OK then
    TryToConnect();
end,
```

The `Confirm` method, which is accessed through the root, throws up a dialog with a message (TryAgain?) in which the user can tap on OK or Cancel. It then sends a message (`ConnectAgain`) to your base view with an argument of `true` or `nil`.

Message Sending Between Newtons

Because of the half-duplex nature of the IR beam, a Newton must alternate between being a sender and a receiver. This must be part of the high-level protocol established between the two Newtons by your program.

To change from being a receiver to a sender, you must kill the current `inputSpec` and set the next `inputSpec` to `nil`:

```
myEndPoint.nextInputSpec := nil;
myEndPoint:SetInputSpec(nil);
```

Only then can you begin to output data. IR output must specify the start and end of the data using system defined flags:

the `kFrame` flag indicates a low-level data frame (**not** a Newton frame)

the `kMore` flag indicates more data to come

These flags are used as arguments to the `Output` and `OutputFrame` messages passed to the endpoint.

```
myEndPoint:Output("the beginning", kFrame+kMore);
myEndPoint:Output(" and almost end", kFrame + kMore);
```

```
myEndPoint:Output(unicodeCR, kFrame);
```

The receiver must match the framing flag by setting the `recvFlags` slot in the endpoint to `kFrame`:

```
myEndPoint:= { _proto: protoEndpoint,
  configOptions: [
    {label: kCMSSlowIR,
     type: 'service',
     opCode: opSetRequired }
  ]
  recvFlags: kFrame,
}
```

Since Newtons know about frames that can hold a lot of information, it makes good sense to build a transaction of data as a frame, such as:

```
trade := { action: "Buy",
  security: "appl",
  quantity: 11000,
  price: 85.25,
  trader: "SJ",
  allocation:[ {client:"XYZ Co", quantity:1000},
    {client:"ABC INC", quantity:5000},
    {client:"NewtDTS", quantity:5000}
  ]
}
```

and then transmit the entire transaction as a frame using `OutputFrame`:

```
myEndPoint:OutputFrame(trade, kFrame);
myEndPoint:FlushOutput(); // good idea with frames to flush
```

Using frames, the state machine can be quite complex, but easy to follow.

The complexity lies within the content of the frame, not in the external state machine. A single `inputSpec` can handle all incoming messages and decide what to do, based on the value of a single key slot such as 'action'.

It should be noted that for the time being, you must use `Output` at least once before using `OutputFrame`. Failure to do so causes an error.

TRACKING THE ELUSIVE NEWTON

Earlier, it was noted that the IR endpoint cannot guarantee delivery of a message if the Newtons lose IR contact with one another. If this loss of contact occurs during a transmission, the receiving Newton will wait happily forever, while the sending Newton will either freeze, or time out with an error after about two minutes.

To avoid this error condition, the sending Newton needs to use an `AddDeferredAction` method and an `AddDelayedAction` method similar to the ones described for connecting. The difference will lie in determining whether the message did or did not get through, and what resulting action to take. Ideally, the action should be to allow the two Newtons to search for each other, and when contact is reestablished, to resend the original message.

The following code example is a description of how to conduct such a search for that elusive Newton. The sender tries to output a repeating message at regular intervals, in this case, every second. The message contains the delay in 1/60 of a second since the last time the message was sent. If the last message got through successfully, this would

be a value of 60.

```
viewIdleScript: func() begin
  if quitSearch then return nil;
  :SendTest();
  1000;
end,

SendTest: func() begin
  local thisTick := Ticks(); // a clock time in 1/60 second
  if NOT lastTick then // lastTick is initialized to nil
    local delay := 60;
  else
    local delay := thisTick - lastTick;
  :SetNextValue(thisTick); // set new value for lastTick
  myEndPoint:Output(UnicodeSTX,kFrame + kMore);
  myEndPoint:Output(NumberStr(delay),kFrame + kMore);
  myEndPoint:Output(UnicodeETX,kFrame);
  myEndPoint:FlushOutput();
end,
```

The receiving Newton decodes the sender's delay between messages and compares this with the delay at the receiving end. The `InputScript` method computes a value for a gauge meter, based on the difference between the two delays.

```
waitForTest: {
  inputForm: 'string,
  endCharacter: unicodeETX,
  discardAfter: 10,

  InputScript: func(endpoint, s) begin
    local thisTick := Ticks();
    local meterVal := 0;
    local stx := "\u0002\u";
    local startStx := StrPos(s,stx,0);
    if startStx then // did we get whole message?
      begin
        // extract sender's delay between messages
        local txt:=SubStr(s,startStx+1,StrLen(s)-
          startStx-2);
        local val:=StringToNumber(txt);
        local expected := endpoint:GetLastTick();
        if expected then begin // not the first?
          expected := thisTick - expected; // receiver's delay
          // compare sender delay & receiver delay
          meterVal := Max(10,100-(val-expected));
          end;
        else
          meterVal := 70; // first test message
          end;
        else
          meterVal:=50; // only part of message

          endpoint:UpdateGauge(meterVal);
          // update gauge meter
          endpoint:SetNextValue(thisTick); // set new
          // value for lastTick
        end,
      }
}
```

The receiver uses a gauge view to display the degree of contact between the two Newtons. The gauge is pushed up as the difference between the sender's delay and the receiver's approaches zero, and is pushed down a small amount at intervals of half a second.

```
SearchGauge := {
  viewValue: 0,
  minvalue: 0,
  maxvalue: 100,
  ...,
  viewIdleScript: func() begin
    local newValue := Max(0, self.viewValue - 5);
    SetValue(SearchGauge, 'viewValue, newValue);
    500;
```

```

end,
}

```

When the gauge stabilizes at a high value, the receiver gets a message that contact has now been reliably established.

```

UpdateGauge: func(newValue) begin
  if (newValue >= 80) AND
  (SearchGauge.viewValue >= 80) then
    :PutDataInStatusArea(
      "Locked on target\nYou can Stop Search ");
  else
    :PutDataInStatusArea("Searching...");

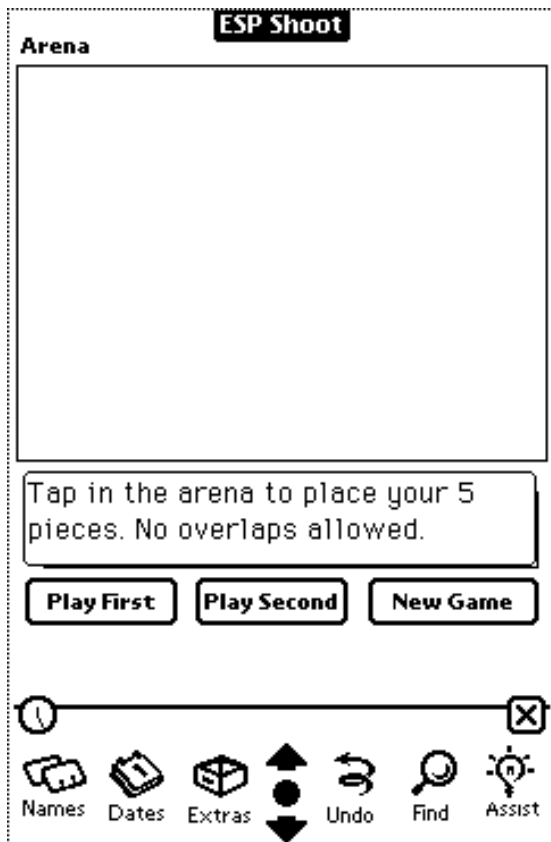
  SetValue(SearchGauge, 'viewValue, newValue);
end,

```

At this point, the sender and receiver would cancel the search and return to normal operation. If this type of code is included in an application, the user will feel much more in control and a good deal less frustrated. Forcing the user to press the reset button is an option of last resort and much to be avoided.

AN EXAMPLE OF IR USING A BOARD GAME

An excellent way to develop your IR skills is to build a simple game in which moves and results are passed back and forth as frames of information. This game contains: a board view, in which the game is played; a status view, to tell the player what is expected next; a glance view, to provide transitory messages on the success of the last move; and some buttons to set the game in motion.



The 'Arena' is the Board view. The space below the buttons is a protoGlance view for transient information.

The game is very simple:

- Each player places five pieces (spots) on their board
- The players decide who goes first, and tap either Play First or Play Second
- Each player takes a turn to tap once on the board, to try to locate the other player's pieces
- The game continue until one player has located all five pieces on the other player's board

The state machine breaks down into four parts:

1. Setup

- Get taps on Board and create playing pieces
- Connect players and handshake for proper connection
- Change to My Turn if Play First,
- Change to Their Turn if Play Second,

2. My Turn

- If all pieces hit, send win message and change to end of game
- Display history of my shots: misses as O and hits as X
- Get tap on board and send shot
- Wait for a hit or miss message and display the result
- Change to Their Turn

3. Their Turn

- Display my pieces and history of opponent's shots
- Wait for next message (shot or win)
 - If a win message, change to end of game
 - Else Display new shot,
 - Determine if a hit,
 - Send hit or miss message
- Change to My Turn

3. End of Game

- Disconnect
- Prepare for new game setup

This scenario describes the fundamentals of almost all board games, including Chess, Checkers and Go.

REMOTE IR

Remote IR is using the IR beam to control a VCR, TV, CD Player or other device.

It is a one-way broadcast of an IR signal. The Newton filters incoming remote signals at the hardware level, so there is no way to receive input from a remote device.

There is also no particular listener specified and no guarantee of good delivery, even if the sender points the beam in the right direction. As with other remote controllers, this means that if the device doesn't respond the first time, a user must press the button again. Given the number of times this happens in daily life when all that is being transmitting is about 2 bytes of command, you can appreciate why a three-

to-four-foot limit is required for Newton-to-Newton IR communications, in which you are transmitting hundreds or thousands of bytes, with guaranteed integrity.

Setting up, Using and Disposing the Connection

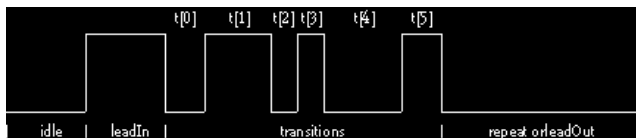
There is no endpoint defined as yet for remote IR. The methods to control remote IR are not built in to the Newton system, but are available through an upgraded version of the Message Pad file.

```
cookie := :OpenRemoteControl(); //call once if error,
// return nil.
:CloseRemoteControl(cookie); //after all sending
// finished. always returns nil
:SendRemoteControlCode(cookie, command, count); //the 'command' is
// sent count times. returns nil after commands have been sent
```

The `cookie` is a reference object, but not a Newton object to which messages can be sent.

Command Codes

Each command wave form has the following structure:



The code structure to produce this command wave form is currently implemented as a Macintosh resource:

```
struct IRCCodeWord {
  unsigned long name; // identifier for ref only
  unsigned long timeBase; // the clock cycle in microseconds
  unsigned long leadIn; // duration in timeBase units of the lead bit cell
  unsigned long repeat; // duration in timeBase units of the
  // last bit cell for loop commands
  unsigned long leadOut; // duration in timeBase units of the last bit cell for non-loop
  unsigned long count; // count of transitions following (origin 1)
  unsigned long transitions[]; // array of transition durations in timeBase units
}
```

Note that the repeat time is used only when the code is sent multiple times.

The transitions array carries the wave form for a particular command to a particular device, such as "CD Play". Life would be too simple if all manufacturers used the same standard to define the wave form. Instead, there are several standards, each with their own values for the `timeBase`, `leadIn` and other variables.

Further, each manufacturer has different codes for devices and commands, and finally, they have different ways of encoding their commands into the wave transitions to be sent by the IR beam.

If this all begins to sound a little too complicated, things can be simplified by sticking with one manufacturer for an example. Sony encodes each command as an AGC burst (2400 μ s) followed by a 7-bit command field and a 5-bit device field, transmitted `leastSignificantBit` (lsb) to `mostSignificantBit` (msb).

Example:

Using device 17 (CD player) and command 50 (Play).

The "CD play" instruction is therefore:

MMM 10001 0110010

AGC CD Play
but the device/command field is sent lsb to msb, so really the bit stream to be sent looks like:

MMM 0100110 10001

To convert this into a wave form, you need the following rules
The timebase is 600 μ seconds.

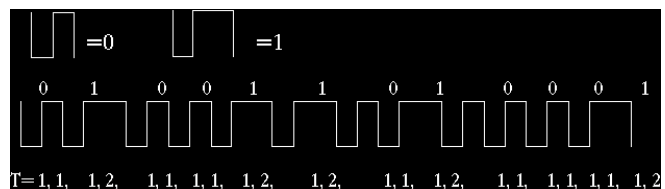
A "M" bit is the mark condition:

This gives us a LeadIn of 4 units

A "0" bit is encoded as 600 μ s idle followed by a 600 μ s pulse.

A "1" bit is encoded as 600 μ s idle followed by a 1200 μ s pulse.

This is interpreted as follows to give the transition waveform.



The transition waveform for the command "CD Play"

Returning to our Macintosh resource definition, we can now define a Sony CD Play resource as:

```
{
  'cdpl', // reference name
  600, // timeBase, in microseconds
  4, // leadIn, in timeBase units
  74, // repeat, in timeBase units (~44 ms)
  833, // leadOut, in timeBase units (~500 ms)
  24, // count or number of transitions
  [ // array of transitions
    1, 1, 1, 2, 1, 1, 1, 1,
    1, 2, 1, 2, 1, 1, 1, 2,
    1, 1, 1, 1, 1, 1, 1, 2 ] }
```

DEVELOPING A REMOTE IR APPLICATION

The following is some sample code which would allow you to build your own remote controller.

```
//-----Project Data File
// constants
kSonyCD := 0;
kSonyTV := 1;

// Get commands from resource file
rf:= OpenResFileX(Home & "Sony.rsrc");
sonyCDPlay:=GetNamedResource("IRCD", "SonyCDPlay",
  'resource');
sonyCDStop := GetNamedResource("IRCD", "SonyCDStop",
  'resource');
...
CloseResFileX(rf);

//-----Main.t file
baseView:= {
  // constants defined in platform file
  OpenRemoteControl: kOpenRemoteControlFunc,
  SendRemoteControlCode: kSendRemoteControlCodeFunc,
  irCodes: {
    sonyCD: [sonyCDPlay, sonyCDStop, ... ],
    sonyTV: [sonyTVOn, sonyTVOff, ... ],
  }
}
SendCode: func(device, code, count) begin
  if device = kSonyCD then
```

continued from page 1

Newton Delivers for Vertical Markets

Newton® MessagePad™ to put evidence-based, background and portable content into the hands of its residents for the purpose of facilitating patient evaluation and management in a project called Constellation.

The goal is to provide residents with immediate access to general background information on disease processes. A collection of specially developed tools assist professionals in navigating this content. The Constellation project will put information tools at the touch of a resident's fingertips, giving him or her immediate access to general disease and management descriptions, references to the recent and relevant medical literature, and access to a selected set of medical guidelines — all in order to apply this content to their patients. Traditionally, to access this information, the resident relies on a library, textbooks, computers or fellow residents and must take the time to go and research the needed information. Now with the Newton, physicians can tap into the equivalent of over 1,500 pages of medical and drug information in the hospital corridor or at a patient's bedside. In less time than it would take to do any kind of research, a doctor can flip open his or her Newton and with a few stylus taps on the screen, learn about the ramifications of prescribing a particular drug, research a rare congenital disease, or browse through a series of medical journal articles.

Dr. Steven Labkoff, an internist and head of the Decisions Systems Group at Brigham, is also head of the Constellation Project and is responsible for some of the custom programming of the units. He credits Sandeep Shah, President of K2 Consultant, Inc. with devising the data compression and search strategy programs.

Shah and Labkoff chose the Newton MessagePad for the project at the hospitals for a number of reasons. Among them, the Newton form factor provided the right size with the appropriate amount of computing power to provide for the physicians needs. While there were smaller form factors available in other products, none were as flexible.

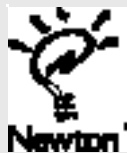
Newton's touch input technology appealed to a physician's keyboard shyness. Additionally, the ease of programming with Newton Book Maker was very appealing. Shah needed to reference information from a variety of sources in a number of different formats. The fact that he could deliver everything from outlines to full chapters on specific subjects, with full search and find capabilities, and with fonts embedded in data, allowed delivery of a fast, user friendly interface that doctors are comfortable using.

Currently, the Newtons contain an electronic version of the American College of Physicians' (ACP) Medical Knowledge Self Assessment Program, the American College of Physicians' Journal Club, the Electronic Monthly Prescribing Guide, handbooks and residency phone books for both Brigham and Women's and Massachusetts General hospitals, and the Intensive Care Unit/Cardiac Care Unit Drug Reference Book. All of these publications are being referenced on a regular basis by residents in the Constellation program.

Shah commented that "Newton has the technology that we need today in order to deliver on the product we want, without relying on the promise of future technologies". He views the Newton as providing great opportunities for developers interested in bringing technology into a number of vertical markets.

The Constellation project has delivered a wide variety of information into the fingertips of doctors at Brigham and Women's Hospital and at Massachusetts General Hospital. It has been a success at demonstrating how Newton technology can be used to solve previously unsolvable problems while creating new markets for developers to explore and own in the process.

Developers interested in creating and conquering new markets should rethink the way problems are solved today. The first step is to think about what would a "perfect world" solution be if you didn't have to keep any technical or computer related considerations in mind. The



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

continued from page 1

Newton Communications

covered in Apple PIE's class: *Newton Programming: Communications*.

HARDWARE

The first and most commonly used piece of communications hardware in the Newton is an SCC (Serial Controller Chip), identical to the one used on most Macintoshes. This is what is used for serial connections, modem connections and AppleTalk. While the SCC chip can be programmed to run at a variety of speeds without specialized hardware or cabling, it cannot normally be used at speeds of over 19,200 to 38,400 bps (bits per second).

The Newton also has a built-in infrared transceiver. This can be used to communicate between Newtons and can also be programmed to interact with remote IR devices such as TVs, VCRs, and so on.

The last hardware interface is the PCMCIA slot, which can be used to add such things as fax modems, cellular communications cards, and so on. These cards must have specialized drivers written for them; to date there have been no documents or tools related to this available from Apple. However, as this article is being written, the Driver Developer's Kit (DDK) is being put into Alpha testing by Apple for release sometime down the road. Drivers will be written in C++ using the DDK.

ARCHITECTURE

Figure 1 shows how the Newton is organized. The top layer shows system software written in NewtonScript. This mostly consists of prototypes and other NewtonScript interfaces to the various services. The next layer down shows the OS software, which is primarily written in C++. The last layer is the Newton hardware including the SCC and the IRDCU (infrared controller).

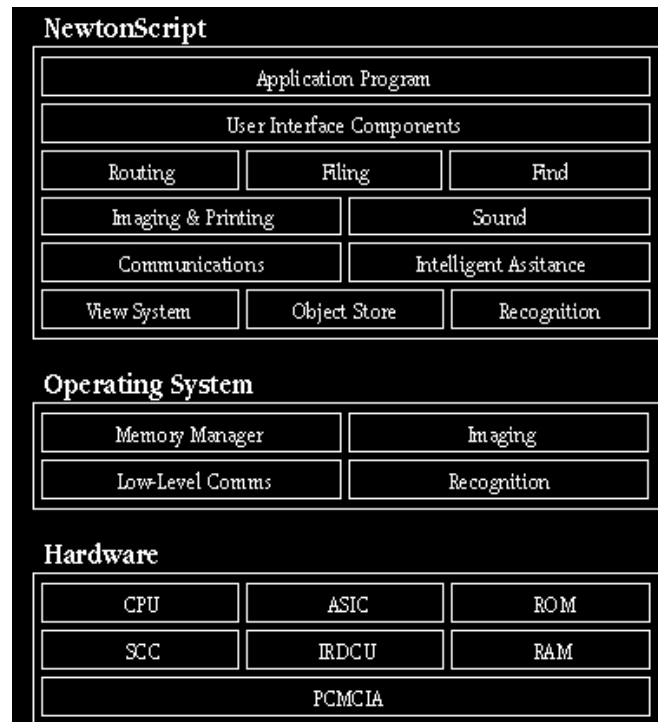


Figure 1

For communications, the important thing is that there is a high-level communications interface written in NewtonScript – the endpoints – and a low-level communications interface – communications tasks, tools and drivers – written in C++, which talks directly to the hardware.

The NewtonScript interface provides a common interface to all types of communications hardware via a virtual connection called an *endpoint*. To create and use an endpoint, appropriate connection parameters must be established to describe the endpoint. For example, for a serial connection you must specify baud rate, stop bits, parity, and so on, before you can connect to the service. Once a connection is established, an endpoint acts as a pipeline down which bytes are sent or received.

The steps necessary to create and use an endpoint are as follows:

1. Define the endpoint frame, including the type of communications service, the particular connections options (baud rate, error correction, and so on), the methods for receiving data through the endpoint, and any other slots and methods you may want to access from the endpoint.
2. Instantiate the endpoint. This creates a NewtonScript object and notifies the low-level communications code of the existence of

the endpoint.

3. Open a connection to the hardware through which communications can take place.
4. Send and receive data through the endpoint. Sending is done by calling the `Output` method. Receiving is done by establishing a frame describing expected input format and containing an `inputScript` method or other input methods.
5. Disconnect the endpoint. When you are done, disconnect after first aborting any pending incoming transfers.
6. Destroy the endpoint. Having disconnected successfully, the endpoint may now be destroyed. This destroys the NewtonScript object and the connection to the low-level communications code.

Details and code examples for each of these steps will be discussed in depth in this article.

The Newton has a true multi-tasking micro-kernel-based architecture. Figure 2 shows several of the tasks which commonly run under direction of the micro-kernel task scheduler.

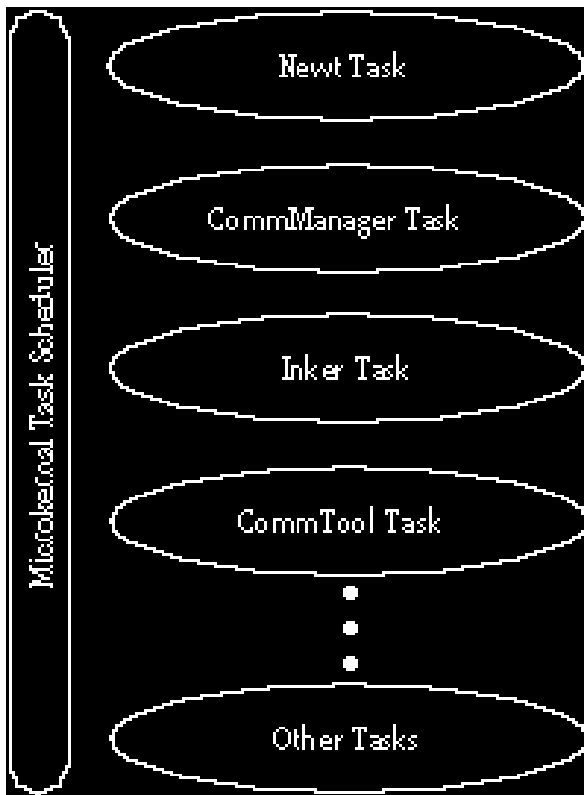


Figure 2

There are several things here that have an impact on communications programming. The first and most obvious point is that there

are separate tasks for the low-level Communications Manager and for the Communications Tool that is currently active. An equally important though less obvious factor is that there is only one NewtonScript task. This means that all NewtonScript code runs in the same task space, though not necessarily within the same application context.

For example, in doing communications programming you may provide an exception handler that deals with communications problems that occur after an endpoint is connected. This exception handler will be called from the root view and so may not have access to your application's slots, frames and methods. Because of this, it may be necessary to find your application from the root view. For example,

```
GetRoot().|MyApp:MyCompany|.AMethod
```

Figure 3a shows the relationship between the NewtonScript task and the low-level communications tasks when an endpoint is instantiated. A message is sent from the NewtonScript task to the Communications Manager task to open a communications connection.

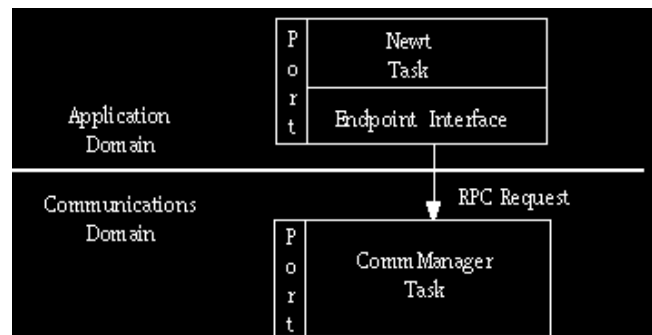


Figure 3a - During Instantiation of Endpoint

Figure 3b shows the response of the Communications Manager to this request. It has launched the appropriate communications tool for the type of connection specified in the endpoint frame, which in turn has established the appropriate connection with the hardware.

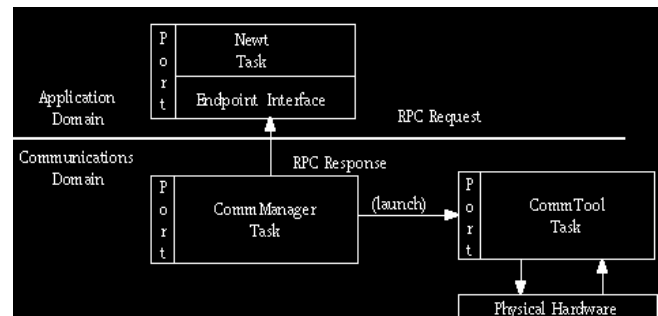


Figure 3b: Response From Instantiation

Figure 3c shows the next step of connecting an endpoint to the device. Here the Newton endpoint sends a connection request to

the Comm Tool which was launched when the endpoint was instantiated. In response, the Comm Tool sends a response to the Newton task describing the success of the request.

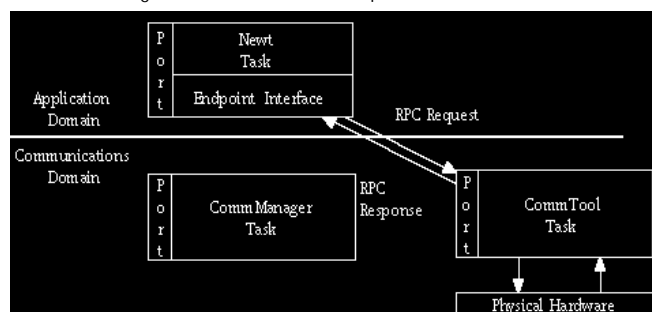


Figure 3c: Connection

The significance of this sequence is primarily to point out that communications with hardware is done through a separate task from the NewtonScript task. This means that all communication to the device is truly asynchronous to your code. You cannot be certain when a communications action occurs or of the status of the connection represented by the endpoint, without explicitly checking the status of the endpoint. It is for this reason that in certain code you must check the status after you have sent a message to the endpoint but before you do something drastic to it.

An example would be to abort ongoing data transfers prior to disconnecting and destroying the endpoint. This also means that in debugging the endpoint you must take care not to make assumptions about when an error might occur, as it is possible that the last command sent to the endpoint has not yet executed.

Another less obvious implication of the sequence of instantiation described in the preceding figures is that at any time only one endpoint can be active. Otherwise the separate NewtonScript endpoint methods collide. This means that when an endpoint is instantiated there can be no other instantiated endpoints.

For example, you should not have an IR endpoint and a serial endpoint active at the same time. From a programming point of view this is particularly important, as it means that you cannot use NTK's Inspector to debug endpoint code, because it uses a serial endpoint to talk to the Newton.

ENDPOINTS

As mentioned before, endpoints are virtual connections created in NewtonScript. What this means is that in theory, once an endpoint is created in NewtonScript and a service is connected, the same code could be used in all cases to send and receive data. In practice, this is *almost* true.

Creating an Endpoint

To create an endpoint you must first define a NewtonScript frame which describes the endpoint you wish to create. You do this by creating a frame which has a `_proto` slot with a value of `ProtoEndpoint`, and a series of other slots that define the char-

acteristics of the endpoint. This is typically done in the `viewSetupFormScript` method of the base view, so that the endpoint is defined the entire time an application is running.

For example, below is an endpoint definition you might use for a serial endpoint:

```
mySerialEndpoint := {
  _proto:protoEndpoint,
  configOptions: [
    {
      label: kCMSASyncSerial,
      type: 'service,
      opCode: opSetRequired },
    {
      label: kCMOSerialIOParms,
      type: 'option,
      opCode: opSetNegotiate,
      data: {
        bps: k9600bps,
        databits: k8DataBits,
        stopBits: k1StopBits,
        parity: kNoParity } },
    {
      label: kCMOInputFlowControlParms,
      type: 'option,
      opCode: opSetNegotiate,
      data: {
        xonChar: unicodeDC1,
        xoffChar: unicodeDC3,
        useSoftFlowControl: true,
        useHardFlowControl: nil } },
    {
      label: kCMOOutputFlowControlParms,
      type: 'option,
      opCode: opSetNegotiate,
      data: {
        xonChar: unicodeDC1,
        xoffChar: unicodeDC3,
        useSoftFlowControl: true,
        useHardFlowControl: nil } },
  ]
}
```

Note that `configOptions` is an array of frames, each of which defines some aspect of the endpoint. The `configOptions` frames contain different data depending on what they are setting, but they all have a frame with a `type` slot whose value is the symbol `'service`. This identifies that this option describes the kind of endpoint being established (for example, SlowIR, Serial, AppleTalk, and so on). The `label` slot of this option frame names the service. In the example above, this is the value `kCMSASyncSerial`, a system constant defining a serial connection.

Each `configOptions` array also has an `opCode` slot which describes whether the particular option must have the value set (`opCode:opSetRequired`) or open to negotiation between the Newton and the hardware (`opCode:opSetNegotiate`). A required option must be exactly what is specified, whereas a negotiable option means you can accept something less than what is asked for. An example of a negotiated option is the baud rate setting shown in the previous example. You are *asking* for a connection of 9600 bps, but since this is a negotiable option you can accept a data rate of less than 9600.

As you can also see from the example, the third component of a `configOptions` array is the `data` slot. This slot is usually another frame which provides the necessary information for the option being set. You must read the documentation in the Newton Programmer's Guide, Chapter 14, to find out what data (if any) is

needed for each option.

While you typically set the options once when defining an endpoint, they may change later as you instantiate and connect the endpoint (or even later after sending or receiving data, though this is not recommended). However, it is far more common to set the values once when defining the endpoint and leave them thereafter.

You should also beware that not all options *can* be successfully changed after the endpoint is instantiated. Which ones cannot be changed? This has not yet been documented so if you wish to try this, beware, as you might crash unexpectedly. Just to make it interesting, the success or failure of changing an option is dependent on the communication tool in use ; what works with one endpoint may not work with another.

Note that you may add other slots to your endpoint. As you will see later, there are at least a couple more slots you will typically want to add in order to receive data and handle endpoint specific exceptions.

As discussed in the previous section, before connecting an endpoint to a service you must first instantiate it, in order to create a NewtonScript object and to open the correct communication tool. You do this by sending the `Instantiate` message to the endpoint. For example:

```
mySerialEndpoint:Instantiate(mySerialEndpoint, nil);
```

The second argument is an optional set of `configOptions` similar to those just discussed. By specifying `nil` for this argument, you are simply using the ones defined in `mySerialEndpoint`.

Having instantiated the endpoint, you can now connect it by sending the `Connect` message:

```
mySerialEndpoint:Connect(nil, nil);
```

Here the first argument is a frame which describes the address of the thing you wish to connect to; in the case of a serial endpoint, you need no address as you are connecting to the device on the other end of the cable. However for a modem endpoint, this would probably be the phone number you wanted to dial; an AppleTalk connection would have the NBP of the entity you wished to connect to. An example of an address frame might be:

```
anAddress:= {
  type:'address,
  label:kCMARouteLabel,
  opCode:opSetRequired,
  data:{addressType:kNamedAppleTalkAddress,
        addressData:"LlamaFarm:LlamaServer@*"}
}
```

The second argument is again an opportunity to change the endpoint options and as with `Instantiate`, you typically pass a `nil` value.

In most cases this is all that is necessary to connect an endpoint, and if successful, you would now be open for business. However, as detailed in the related article in this issue, since infrared beaming is half-duplex (cannot send and receive at the same time), the receiving device must send a `Listen` message to its endpoint before a connection can be established.

Sending Data

There are two endpoint methods which may be used to send data out from the Newton: `Output` and `OutputFrame`. The `OutputFrame` method is used to send flattened frames; that is, NewtonScript frames in which all of the references have been resolved into a stream of bytes. While this is useful, it relies on both sender and receiver knowing that a frame is coming. Since the exact format of a flattened frame is not documented, this method is generally only useful when sending between two Newtons.

For this reason `Output` is usually used in all cases except for Newton-to-Newton communications. The exact format of the `Output` and `OutputFrame` methods is as follows:

```
Output(data, flags)
OutputFrame (data, flags)
```

where the `data` argument is the data being sent and the `flags` argument is used to control output in the infrared case. Other than in infrared communications then, the `flag` argument will be `nil`.

The data which can be sent using `Output` includes Newton strings, integers, and binary arrays. In the case of strings all outgoing data will go through a Unicode-to-ASCII translation, as all Newton strings are kept internally in Unicode (16-bit character values) format, but it is assumed that any external device will be expecting ASCII. If you wish to send the raw values without this translation you can send an array of character values. Since the array is not a string, it will be sent as raw Unicode values.

Currently this Unicode-to-ASCII translation is the only translation available and is the default, but in the future you may be able to specify your own translation tables.

In NewtonScript, integers are represented as 30-bit values. However, when integers are sent from the Newton, only the least significant byte is actually sent. This means that to send a 30-bit value you must send 4 values, one for each byte in the integer. This is shown below:

```
ep:Output(len >> 24, nil) // output 1st (MSB) byte
ep:Output(len >> 16, nil) // output 2nd byte
ep:Output(len >> 8, nil) // output 3rd byte
ep:Output(len, nil); // output 4th (LSB) byte
```

To use `OutputFrame`, you simply pass a reference to the frame you wish to send as the data and again use a `nil` for the flags argument unless you are sending the frame over an infrared link. A brief example of this is shown here:

```
aFrame={ slot1:4, slot2:"abcd" };
mySerialEndpoint:OutputFrame(aFrame,nil);
```

When outputting data, you should flush the output channel in order to avoid losing data in an internal buffer if something should go wrong. This is done by sending the message `FlushOutput` to the endpoint.

Receiving Data

To receive data through an endpoint you must set up a special

frame called an *inputSpec* that has in it a method that will be called when a certain input condition is reached. The input conditions are described in this frame as well.

You then use the method `SetInputSpec (inputSpec)` to notify the endpoint that this is the *inputSpec* you want for future input. Note that you may (and probably will) change the *inputSpec* that is current as your application runs. Complex state machines created by chaining *inputSpecs* will be discussed in a future article.

An example *inputSpec* is shown in here:

```
receiveNameSpec:= {
  inputForm: 'string,
  endCharacter: unicodeCR,
  InputScript: func (theEndpoint, inputStr)
  begin
    baseView.nameSlot:=inputStr;
  end
}
```

This *inputSpec* will now be discussed in detail.

There are a number of conditions that can be used as triggers for the input method. The commonest of these is to use a single character specified as by the `endCharacter` slot of the *inputSpec* frame. This essentially keys the `InputScript` method to be called when the character specified is received. In this case the character is a carriage return.

The `inputForm` slot defines what the expected input will be. In this case the `'string` symbol specifies that the input is expected to be characters and, since the Newton uses the Unicode character set for strings, the input will be automatically converted to Unicode from ASCII.

The `InputScript` slot is the method that is called when the trigger condition (in this case an incoming carriage return character) occurs. It is common for an *inputSpec*'s `InputScript` to activate a different *inputSpec*.

There are several other triggers that may be used — the most common is setting an input length of characters received. This is done by setting a slot in the *inputSpec* frame called `byteCount` with an integer value that specifies the number of input characters to be received before calling the `InputScript`.

There are other ways to handle input which will not be covered here. However, an important point in all input schemes is that the sender and the receiver must know what to expect. In other words, they must have a higher-level protocol to coordinate the format and meaning of successive data received.

You set the *inputSpec* as the one currently in use with the following code:

```
ep:SetInputSpec ( receiveNameSpec );
```

NEWTON SERIAL & MODEM COMMUNICATIONS

As described earlier in this article, all Newton communications software is directed through an endpoint. Connection, input and output are all channeled through this virtual software device. An endpoint can handle many forms, including infrared and AppleTalk,

but this article is devoted to its form as a serial and modem service. On the hardware front, the Newton is equipped with a serial plug (mini DIN-8) which allows it to link via a cable to either a desktop computer serial port or to a modem.

Serial communication may be used to move data to and from a desktop computer or a data-gathering device such as a scientific probe or an industrial sensor.

Modem communications can be used to hook into the myriad services now being provided worldwide, including the Internet, and for sending data to an office LAN, or placing an order with the local grocery store.

Serial Communications

The endpoint in NewtonScript is the key to linking your application to the outside world. As has been shown earlier in this article, the endpoint is an interchangeable object that allows your code to talk to remote devices, without requiring you to make changes in your application code.

The minimum definition of a serial communications endpoint includes a statement that it is an asynchronous service, and an option describing the speed of the communications link, its parity and number of data and stop bits.

```
mySerialEndpoint := {
  _proto:protoEndPoint,
  configOptions: [
    { label: kCMSASyncSerial,
      type: 'service,
      opCode: opSetRequired },
    { label: kCMOSerialIOParms,
      type: 'option,
      opCode: opSetNegotiate,
      data: { bps: k9600bps,
             dataBits: k8DataBits,
             stopBits: k1StopBits,
             parity: kNoParity },
    ]
}
```

Note that all names starting with `k`, such as `kCMSASyncSerial` are defined system constants. Also note that if the value of `opCode` is `opSetNegotiate` (another system defined constant), it means that when the requested values such as 9600 baud, are not available, a reasonable substitute will be accepted. The Newton supports asynchronous speeds from 300 baud up to 57,600 baud.¹

In the `bps` slot you can use any of the following:

CONSTANT	VALUE
<code>k300bps</code>	300
<code>k600bps</code>	600
<code>k1200bps</code>	1200
<code>k2400bps</code>	2400
<code>k4800bps</code>	4800
<code>k7200bps</code>	7200
<code>k9600bps</code>	9600
<code>k12000bps</code>	12000
<code>k14400bps</code>	14400
<code>k19200bps</code>	19200
<code>k38400bps</code>	38400
<code>k57600bps</code>	57600

In the `dataBits` slot, you can use the following constants:

CONSTANT (# of Data Bits)	VALUE
<code>k5DataBits</code>	5
<code>k6DataBits</code>	6
<code>k7DataBits</code>	7
<code>k8DataBits</code>	8

In the `stopBits` slot, you can use the following constants:

CONSTANT (# of Stop Bits)	VALUE
<code>k1StopBits</code>	0
<code>k1pt5StopBits</code>	1
<code>k2StopBits</code>	2

In the `parity` slot, you can use the following constants:

CONSTANT	VALUE
<code>kNoParity</code>	0
<code>kOddParity</code>	1
<code>kEvenParity</code>	2

Flow Control

Flow control for the serial endpoint can be handled by either software or hardware. For example, to use XON/XOFF software control for your endpoint, add the following options to the `configOptions` array:

```
{ label: kCMOInputFlowControlParms,
  type: 'option',
  opCode: opSetNegotiate,
  data: { xonChar: unicodeDC1,
         xoffChar: unicodeDC3,
         useSoftFlowControl: true,
         useHardFlowControl: nil } },
{ label: kCMOOutputFlowControlParms,
  type: 'option',
  opCode: opSetNegotiate,
  data: { xonChar: unicodeDC1,
         xoffChar: unicodeDC3,
         useSoftFlowControl: true,
         useHardFlowControl: nil } },
```

You may well recognize UnicodeDC1 as control-Q or ASCII 13hex, and UnicodeDC3 as control-S or ASCII 11hex. Note that software handshaking can drop characters if a modem is overrun, (that is, if a modem is overrun before the XOFF can be sent) so for high transfer rates, hardware handshaking may be necessary and is supported in the Newton.

The following constants are useful for specifying Unicode characters:

CONSTANT	VALUE
<code>unicodeNUL</code>	<code>\$(u0000)</code>
<code>unicodeSOH</code>	<code>\$(u0001)</code>
<code>unicodeSTX</code>	<code>\$(u0002)</code>
<code>unicodeETX</code>	<code>\$(u0003)</code>
<code>unicodeEOT</code>	<code>\$(u0004)</code>
<code>unicodeENQ</code>	<code>\$(u0005)</code>
<code>unicodeACK</code>	<code>\$(u0006)</code>
<code>unicodeBEL</code>	<code>\$(u0007)</code>

<code>unicodeBS</code>	<code>\$(u0008)</code>
<code>unicodeHT</code>	<code>\$(u0009)</code>
<code>unicodeLF</code>	<code>\$(u000A)</code>
<code>unicodeVT</code>	<code>\$(u000B)</code>
<code>unicodeFF</code>	<code>\$(u000C)</code>
<code>unicodeCR</code>	<code>\$(u000D)</code>
<code>unicodeSO</code>	<code>\$(u000E)</code>
<code>unicodeSI</code>	<code>\$(u000F)</code>
<code>unicodeDLE</code>	<code>\$(u0010)</code>
<code>unicodeDC1</code>	<code>\$(u0011)</code>
<code>unicodeDC2</code>	<code>\$(u0012)</code>
<code>unicodeDC3</code>	<code>\$(u0013)</code>
<code>unicodeDC4</code>	<code>\$(u0014)</code>
<code>unicodeNAK</code>	<code>\$(u0015)</code>
<code>unicodeSYN</code>	<code>\$(u0016)</code>
<code>unicodeETB</code>	<code>\$(u0017)</code>
<code>unicodeCAN</code>	<code>\$(u0018)</code>
<code>unicodeEM</code>	<code>\$(u0019)</code>
<code>unicodeSUB</code>	<code>\$(u001A)</code>
<code>unicodeESC</code>	<code>\$(u001B)</code>
<code>unicodeFS</code>	<code>\$(u001C)</code>
<code>unicodeGS</code>	<code>\$(u001D)</code>
<code>unicodeRS</code>	<code>\$(u001E)</code>
<code>unicodeUS</code>	<code>\$(u001F)</code>

Compression

To improve the performance of serial communications, a number of compression features have been added under the general heading of MNP. To create a serial endpoint with MNP compression, you must specify a different service — "`kCMSMNPID`" — and add three options: one to allocate a buffer for MNP compression; one to specify the type of MNP compression you want; and one to set the rate for data transfers.

```
mySerialMNPEndpoint := {
  _proto:protoEndpoint,
  configOptions: [
    { label: kCMSMNPID,
      type: 'service',
      opCode: opSetRequired },
    { label: kCMOSerialIOParms,
      type: 'option',
      opCode: opSetNegotiate,
      data: { lps: k9600bps,
            dataBits: k8DataBits,
            stopBits: k1StopBits,
            parity: kNoParity } },
    { label: kCMOMNPAllocate,
      type: option,
      opCode: opSetRequired,
      data: kMNPDoAllocate },
    { label: kCMOMNPCompression,
      type: 'option',
      opCode: opSetRequired,
      data: kMNPCompressionV42bis },
    { label: kCMOMNPDataRate,
      type: 'option',
      opCode: opSetRequired,
      data: k9600bps },
  ]
}
```

The `kCMOMNPAllocate` option specifies whether you want a buffer allocated for MNP compression; possible data slot values are as follows:

CONSTANT	MEANING
<code>kMNPDoAllocate</code>	allocate buffer

`kMNPdontAllocate` do not allocate buffer

The `kCMOMNPCompression` option specifies which MNP compression to use; possible data slot values are as follows:

CONSTANT

`kMNPCompressionNone` connect with no compression
`kMNPCompressionMNP5` connect with MNP 5 compression
`kMNPCompressionV42bis` connect with V42bis compression

MEANING

connect with no compression
 connect with MNP 5 compression
 connect with V42bis compression

If you specified `kMNPdontAllocate` for the `kCMOMNPAllocate` option, you must specify `kMNPCompressionNone` for the compression type. If you specified `kMNPdoAllocate`, you need to specify either `kMNPCompressionMNP5` or `kMNPCompressionV42bis`.

There is a "fall-back" mechanism whereby if you request MNP V42bis compression and it isn't available, the modem tool will try MNP5 compression. If MNP5 compression is unavailable, the endpoint will be created without MNP compression.

STATE MACHINES AND INPUTSPECS

So far this article has dealt with a single state state machine, in which all incoming data is expected to be of the same type (strings), and terminated with the same character (a carriage return). In Newton communications, each state is represented by a separate `inputSpec`. An `inputSpec` is a frame which is usually kept as a slot in the endpoint and which defines how incoming data is to be decoded, terminated and handled upon receipt. As an `inputSpec` handles its received data, it can change the state to another `inputSpec`, so that the next input stream will be handled in a different fashion.

To demonstrate how multiple states can interact, an example created by Apple PIEDTS is included. The example application is called Serial Protocol; it handles a handshake protocol, a command state and a message-receiving mode.

```
ep := {
  _proto: protoEndpoint,
  _parent: self,
  ....
  // this is just the initial handshaking part to make sure that both ends are alive
  // this would be set as the starting inputSpec in the connect method
  // ep:SetInputSpec(ep.waitForACK);

  waitForACK:
  {
    InputForm: 'string,
    discardAfter: 4, // only expecting 3 characters and the ?
    endCharacter: $?, // ACK? expected

    InputScript: func(endpoint, s) begin
      if (StrPos(s, "ACK?", 0)) then // was it ACK?
        begin
          endpoint:SetInputSpec(endpoint.waitForFUNCTION);
          // the main state
          endpoint:Output("ACK", nil); // send response
          endpoint:FlushOutput();
        end
      end,
    end,
  }

  // This is the generic dispatcher state, other end sends something
  // ending with ! and the Newton will serve.
```

```
waitForFUNCTION:
{
  InputForm: 'string,
  discardAfter: 10,
  endCharacter: $!, // expects a '!' as end of the command

  InputScript: func(endpoint, s)
  begin
    if (StrPos(s, "CARD!", 0)) then // card function
      begin
        endpoint:Output(endpoint:DumpNameSoup(), nil);
        // Send names in cards
        endpoint:Output(unicodeCR, nil);
        endpoint:Output(unicodeLF, nil);
        endpoint:FlushOutput();
      end;

    if (StrPos(s, "NAME!", 0)) then // name function
      begin
        // Call the Name function (just a wrapper around
        // userConfiguration.name)
        endpoint:Output(endpoint:DumpName(), nil);
        endpoint:Output(unicodeCR, nil);
        endpoint:Output(unicodeLF, nil);
        endpoint:FlushOutput();
      end;

    if (StrPos(s, "SIZE!", 0)) then // size function
      begin
        //Send size of internal storage
        endpoint:Output(endpoint:DumpSize(), nil);
        endpoint:Output(unicodeCR, nil);
        endpoint:Output(unicodeLF, nil);
        endpoint:FlushOutput();
      end;

    if (StrPos(s, "SEND!", 0)) then //String sending fcn
      begin
        // switch to receive string state
        endpoint:SetInputSpec(endpoint.waitForSTRING);
      end;

    if (StrPos(s, "BYE!", 0)) then // bye function
      begin
        endpoint:Output("Bye", nil); // send response
        endpoint:FlushOutput();
      end;
    end,
  }

  // our special string handling state for receiving messages
  waitForSTRING:
  {
    InputForm: 'string,
    discardAfter: 200, // maximum length of message
    endCharacter: $>, // terminate message with >

    InputScript: func(endpoint, s)
    begin
      // back to the main
      endpoint:SetInputSpec(endpoint.waitForFUNCTION);
      endpoint:SendOverInfo(s); // handle the message sent
      endpoint:Output("OK!", nil); // send response
      endpoint:FlushOutput();
    end,
  }
}
```

After making a successful connection, the `inputSpec` would be set to `waitForACK`. As this `inputSpec` is triggered by the incoming data stream, it switches the `inputSpec` to `waitForFunction`. This `inputSpec` checks the content of the command that was sent and executes an appropriate method based on the command received.

Several of these messages, such as `DumpNameSoup` are being sent to the endpoint (`endpoint:DumpNameSoup`

), but these methods are not in the endpoint.

To find them, you have to look further back in the endpoint definition, where there is a `_parent` slot defined as `self (_parent: self)`. Since the endpoint is usually defined in the context of the base view, the parent of the endpoint is therefore the base view. Now, when messages are sent to the endpoint and are not recognized, they are passed on to its parent, the base view, where all the data-handling methods are defined.

UPLOAD/DOWNLOAD FILES/SOUPS

In the desktop world, data files tend to have a structure with field names and records which are often converted to tab-delimited text. The Newton stores data in soups using entries of frames with slots for the field names.

This means that to pass files generically between these two environments, the first state you must define is the interchange of field and slot names. The second state would be to pass the number of records or entries which you intend to transfer. The third state to define is the actual passing of a record or entry. The final state would be some form of confirmation that all the data has been transferred successfully.

Trace the flow of the code to see how the state is changed from one `inputSpec` to another.

```
getFieldNames:
{
  inputForm: 'string,
  endCharacter: unicodeETX, // not a character found in a field name

  inputScript: func(endpoint, s)
  begin
    endpoint:SetInputSpec(getNumberOfRecords);
    endpoint:SetUpSoup(s); // create soup frame based on field names
  end,
}

getNumberOfRecords:
{
  inputForm: 'byte,
  byteCount: 4,

  inputScript: func(endpoint, b)
  begin
    endpoint:SetInputSpec(getRecords);
    //assemble integer of number of records
    local num := b[0]<<24 + b[1]<<16 + b[2]<<8 + b[3];
    endpoint:SetMaxCounter(num); // note number in a baseview slot
  end,
}

getRecords:
{
  inputForm: 'string,
  endCharacter: unicodeCR // end of record

  inputScript: func(endpoint, r)
  begin
    endpoint:AddRecord(r); // make a soup entry
    if endpoint:GetMaxCounter() then // check number of entries
      endpoint:SetInputSpec(checkTransfer);
    end,
  }

  checkTransfer:
  {
    inputForm: 'byte,
    byteCount: 4,
```

```
inputScript: func(endpoint, b)
begin
  //assemble integer of total bytes sent
  local num := b[0]<<24 + b[1]<<16 + b[2]<<8 + b[3];
  if endpoint:CheckBytesSent(num) then
    endpoint:Output("OK",nil)
  else
    endpoint:Output("NOK",nil);
  end,
}
```

In this case, the state machine starts out expecting to receive a string terminated by the ETX character. This string is expected to have the names of the slots in the soup which is being downloaded. When this string is received, you change the input state machine to use the `inputSpec` `getNumberOfRecords`. This expects a stream of 4 bytes that can be assembled into an integer describing how many records will be downloaded. After these bytes are received, the state machine transitions to the `getRecords` `inputSpec`, which will successively accept carriage-return-terminated strings. These are expected to hold the tab-delimited fields which will be put into the appropriate slots. This `inputSpec` will remain in use until the expected number of records are received, after which you transition to the `transferCheck` `inputSpec`. This `inputSpec` expects a 4-byte number describing the number of records sent; these should match the number-of-records value received earlier.

This example shows the way to build a state machine, but obviously it is only the tip of the iceberg. For example, no provision is made to detect the type of the fields and no error handling is in place. It does show a typical multi-state state machine and how it is built.

MODEM COMMUNICATIONS

The modem endpoint is almost identical to the serial endpoint, since the underlying medium is still serial.

```
myModemEndpoint := {
  _proto:protoEndPoint,
  configOptions: [
    { label: kCMSModemID,
      type: 'service,
      opCode: opSetRequired },

    { label: kCMOSerialIOParms,
      type: 'option,
      opCode: opSetNegotiate,
      data: { hps: k9600bps,
              dataBits: k8DataBits,
              stopBits: k1StopBits,
              parity: kNoParity },
    }
  ]
}
```

Modem endpoints support all the compression options discussed so far. An additional option that can be specified is error control; this is done using either the internal MNP (class 4) standard, or the external modem's built-in error control. Note that the `opCode` is set as required, so if you want the endpoint defined even when error correction is not available, you must also specify the `none` option.

```
{ label: kCMOModemECType,
  type: 'option,
```

```

    opCode:    opSetRequired,
    data:      kModemECProtocolMNP + kModemECProtocolNone
}
}
addressData: "9,555-1212" }
}

```

The `kCMOModemEType` option specifies which error correction method to use; possible data slot values are as follows:

CONSTANT	MEANING
<code>kModemECProtocolNone</code>	connect with no error correction
<code>kModemECProtocolMNP</code>	connect with internal MNP (class 4)
<code>kModemECProtocolExternal</code>	connect using external modem's built in error control

Other options specific to modems are dialing preferences. You specify these preferences by adding the `kCMOModemDialing` option to the `configOptions` array.

```

{ label: kCMOModemDialing,
  type: 'option,
  opCode: opSetRequired,
  data: { speakeron: nil,
         detectdialtone: true,
         waitforcarrier: 60, },
}

```

The first three preferences in the table can be set by a user in the Modem section of the Newton preferences. The rest have the default values shown below:

SLOT NAME	DEFAULT VALUE
<code>speakerVolume</code>	preferences
<code>detectDialtone</code>	preferences
<code>dtmfToneDialing</code>	preferences
<code>speakerOn</code>	true
<code>detectBusy</code>	true
<code>manualDial</code>	nil
<code>waitForCarrier</code>	55 (seconds)
<code>waitBeforeBlindial</code>	3 (seconds)
<code>commaDelay</code>	1 (second)
<code>ringToAnswerAfter</code>	2 (seconds, if listening)

Note that "default value" does not mean they are automatically set. Instead, these are the defaults that Apple uses for its applications. For the preferences in particular it is necessary to fetch the appropriate values from the system soup and use them to set modem options.

The last slot name is used when the endpoint has been sent the message `Listen`, instead of `Connect`. This will allow the Newton to answer an incoming call. The more usual use is to connect by making an outgoing call for which you will have to specify a phone number. The phone address is specified in an address option frame.

```

modemAddress: {
  label: kCMARouteLabel,
  type: 'address,
  opCode: opSetRequired,
  data: { addressType: kPhoneNumber,
}

```

This address option frame may be stored anywhere, but for convenience sake, it is usual to keep it in the endpoint. To make a modem connection, you provide the address as an argument to the `Connect` method.

```
ep:Connect(ep.modemAddress,nil);
```

The modem endpoint currently supports only the Newton modem. It is anticipated that in the future, other modems will also be supported.

Connecting a Newton to an outside service through a modem leaves you vulnerable to problems with external equipment over which you have limited control. This possibility can hang your Newton if the `Connect` method never returns. If the hang is in the `buttonClickScript`, this usually manifests as a highlighted button, leading a user to think the machine is gone forever. To avoid this problem, you should start the connection through a deferred action method and check on its progress using a delayed action method.

Connect as follows:

```
AddDeferredAction(func(ep) ep:Connect(ep.modemAddress),
                  [ ep ]);
```

To check on the progress of the connection, use an endpoint variable `vConnected` set by `Connect`:

```
AddDelayedAction(func(endpoint) begin
  if not endpoint.vConnected then
    GetRoot:Confirm("", "Problems connecting,
                      Continue?",
                      endpoint, 'MAbortConnection);
end,
[ ep ],
10000 ); // give it 10 seconds to make connection
```

Then add a method `MAbortConnection` to the endpoint, which `Confirm` will call. Remember that after aborting the connection, you must wait for a short pause before disposing of the



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 or AppleLink: PIESYSOP



Newton Developer Programs

Apple offers two programs for Newton developers—the Newton Associates Program and the Newton Partners Program. The Newton Associates Program is a low cost, self-help development support program. The Newton Partners Program is designed for developers who need expert-level development support via electronic mail. Both programs provide focused Newton development information and discounts on development hardware, software, and tools—all of which can reduce your organization's development time and costs.

Newton Associates Program

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

Self-Help Technical Support

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

Newton Developer Mailing

- *Newton Technology Journal*
- *Newton Developer CD*, which may include:
 - Newton Sample Code
 - Newton System Software
 - Newton tools and utilities
 - Marketing and business information
- *Apple Directions—The Developer Business Report*

Savings on Hardware, Tools, and Training

- Discounts on certain 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*
- Starcore Affiliate Label Program availability



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 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.

Expert Newton Programming-level Support

- One-to-one technical support via e-mail

Apple Newton Hardware

- Discounts on five additional Newton development units

Pre-release Hardware and Software

- Consideration as a test site for pre-release Newton products

Marketing Activities

- Participation in select Apple-sponsored marketing and PR activities

All Newton Associates Program Features:

- Developer Support Center Services
- Self-help technical support
- Newton Developer mailing

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