N e w t o n A p p l i c a t i o n
D e v e l o p m e n t

Newton Internet Enabler

Version 1.0

Contents

Newton Internet Enabler

This chapter describes the NewtonScript interface to Newton Internet Enabler, a collection of built-in software that allows Newtons to interface with the Internet. This chapter describes:

- how the components of Newton Internet Enabler work together
- how to use Newton Internet Enabler in your NewtonScript applications
- the constants, protos, functions, and methods that you use with Newton Internet Enabler
- the options that you use to control and configure Newton Internet Enabler

For information about the user interface to Newton Internet Enabler, see the *User's Guide to Newton Internet Enabler.*

To use Newton Internet Enabler, you must understand how to use endpoints to perform communications operations on the Newton. To learn about endpoints, see the chapter "Endpoint Interface" in *Newton Programmer's Guide*.

About Newton Internet Enabler

With Newton Internet Enabler, you can establish a link to the Internet to allow your applications to communicate over the net. Newton Internet Enabler supports one link to the Internet at any time; however, several Newton applications can use that link to perform communications.

Newton Internet Enabler consists of an application named Internet Setup, with which users can configure their Internet access, and a NewtonScript application programming interface (API), with which you can send and receive data, determine status, and modify your configuration parameters.

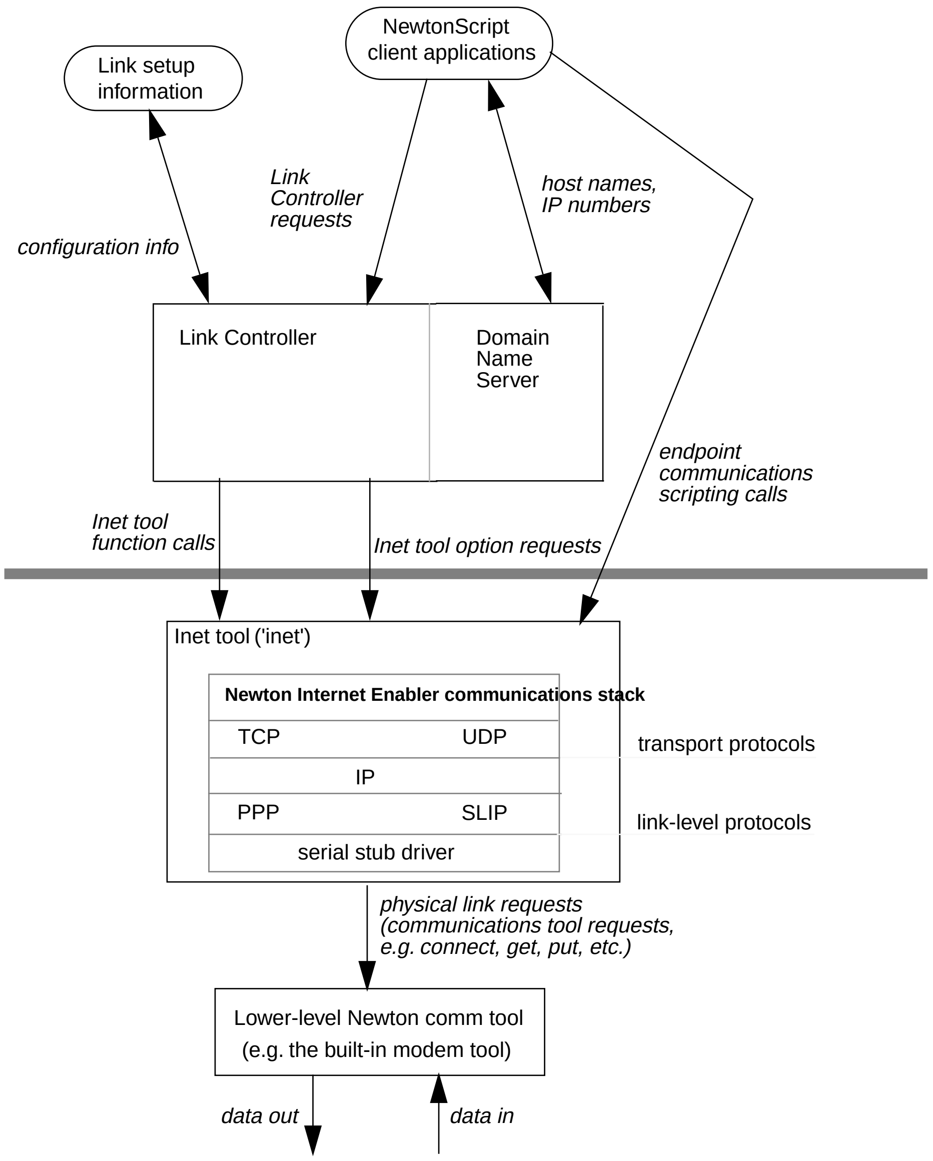
Underneath the NewtonScript API is the Inet tool, which is a built-in Newton communications tool that is used by the other Newton Internet Enabler components to provide the standard Newton communications operations.

The Newton Internet Enabler interface components are:

- The Inet communications tool, which provides the capability to establish and use links to the Internet via the TCP/IP family of protocols. This tool implements a streams interface with a communications stack, providing the capability to use various transport-level and link-level protocols. You can use standard Newton communications endpoint methods and options to control the operations provided by the Inet tool.
- The Link Controller, which uses the Inet tool and maintains net links at a higher level, providing a convenient NewtonScript interface for applications. You use the link controller by calling the global functions that it provides.
- The Domain Name Service interface, which provides functions for converting between Internet host names and their corresponding IP Numbers. You use the domain name service by calling the global functions that it provides.

Figure 1-1 shows the relationship of the Newton Internet Enabler components.

Figure 1-1 The Newton Internet Enabler layers and components


NewtonScript client applications
Link setup information
Link Controller requests
host names, IP numbers
configuration info
Link Controller
Domain Name Server
endpoint communications scripting calls
Inet tool function calls
Inet tool option requests
Inet tool ('inet')
Newton Internet Enabler communications stack
TCP
UDP
transport protocols
IP
PPP
SLIP
link-level protocols
serial stub driver
physical link requests (communications tool requests, e.g. connect, get, put, etc.)
Lower-level Newton comm tool (e.g. the built-in modem tool)
data out
data in

The Inet Tool Layer

The Inet communications tool provides a configurable stack of protocols at and below the TCP/IP level. The Inet tool is a standard Newton communications tool, which means that it provides all of the endpoint services that are provided by other built-in communications tools, including the built-in modem tool and the built-in serial tool.

Like the other communications tools, you can control the configuration of the Inet tool with communications options. The options that you use with the Inet tool all use the `'inet'` service identifier and are described in this chapter.

For more information on how to use the Newton endpoint interface, Newton communications options, and the other built-in communications tools, see *Newton Programmer's Guide*.

The Inet tool supports physical links using the built-in serial tool or the built-in modem tool. You are currently limited to the use of one physical link at any point in time.

The Inet tool can run various link-level protocols that are provided with the Newton system software. These currently include PPP and SLIP.

The Inet tool can establish links using various low-level communications services. Each communications service is provided by a Newton communications tool such as the built-in modem tool.

The Link Controller

You can use the Link Controller to create and manage a link between the Newton and the Internet. The Link Controller can manage a single link for multiple applications simultaneously. This means that one application establishes the link and other applications use the same link.

The Link Controller uses and provides a higher-level interface to the link control functions and options supported by the Inet tool. The Link Controller functions are global functions built into the Newton operating system. The use of the link controller functions is explained in the section "Using the Newton Internet Enabler Link Controller" beginning on page 1-11.

The first grab of a link can be expensive in terms of time: typically, the Inet tool software dials the Newton modem and negotiates the connection to establish an Internet session. The Inet tool then performs whatever login and initialization procedures are required, which the user has configured with the Internet Setup application. All of this can take a substantial amount of time.

Since it can take so much time to grab a new link, Newton Internet Enabler makes it easy for another application to grab a link that has already been established. Whenever an application grabs a link, the link controller increments its count of users of that link. The physical link is dropped only after all users have released the link (when the count becomes `0`).

The following is an example of a typical flow of operations that occur during an Internet session:

1. An application ("Application_1") calls `InetOpenConnectionSlip` to allow the user to customize the link settings and then issues a call to the `InetGrabLink` function. The link controller dials the modem and begins an Internet session with an Internet provider.
2. Application_1 instantiates and binds one or more endpoints to use over that link. Each endpoint can use either the TCP or UDP transport services. And you bind each endpoint either to initiate an outgoing connection (*connect*) or to listen for an incoming connection (*listen*).
3. Application_1 uses its endpoint(s) to perform communications operations.
4. Another application ("Application_2") calls `InetOpenConnectionSlip` to allow the user to customize the link settings and then issues a call to the `InetGrabLink` function to use the same service provider as Application_1. The Inet tool returns the same link that it established in step 1.
5. Application_2 creates and uses endpoint(s) to perform communications operations.
6. Application_2 finishes its use of the link and calls the `InetReleaseLink` function. The link controller decrements its count of users of the link.

7. Application_3 grabs the link, creates endpoints to use over the link, and releases the link.
8. Application_1 finishes its use of the link and calls the `InetReleaseLink` function. The link controller decrements its count of link users. The count becomes 0, so the link is dropped: the Internet session ends, the modem is hung up, and any resources used for the link are released.

Note

Only one Internet session can be active at any time. This means that if an application requests a link to a different Internet service provider when a session is in progress, the `InetGrabLink` call will fail and generate an error. ◆

The Domain Name Service Interface

The domain name service interface builds on top of the Inet tool to provide Newton applications with the ability to translate Internet domain names into IP numbers and vice-versa. The domain name service (DNS) functions are also global functions in the Newton operating system. You can use these functions to:

- translate a domain name into its corresponding Internet address
- translate a a domain name into the Internet address for a mail server that serves that domain
- translate a domain name into the domain name for a mail server that serves that domain
- translate an Internet address into its corresponding domain name

IMPORTANT

The Newton Internet Enabler implements what is known as a “stub domain name service resolver.” The NIE domain name resolver does not re-query the server based on what type of response it received. This means that users must specify a recursive name server for their DNS server.

The use of the domain name server functions is explained in the section "Using the Domain Name Service Interface" beginning on page 1-27.

Using Newton Internet Enabler

This section helps you understand how to use Newton Internet Enabler in your NewtonScript application programs. To use Newton Internet Enabler, you need to know about the options and functions provided by the Newton Internet Enabler API. You also need to understand the callback functions that many of the functions use to communicate results back to your application. And finally, you need to know the proper sequence of steps to take when using Newton Internet Enabler.

The next section, "Using Endpoints With Newton Internet Enabler Links," describes the relationship between the links that Newton Internet Enabler maintains for you and Newton communications endpoints.

The section, "Newton Internet Enabler and Callback Functions" beginning on page 1-9 describes the format and use of callback functions with Newton Internet Enabler.

The section "Using the Newton Internet Enabler Link Controller" beginning on page 1-11 outlines the basic sequence of steps that you need to perform to use the link controller with a Newton communications endpoint. The subsections of "Using the Newton Internet Enabler Link Controller" describe each step in detail.

The section "Using the Domain Name Service Interface" beginning on page 1-27 describes how to use the domain name service API to translate between Internet addresses and domain name strings.

The section "Using the Newton Internet Enabler Options" beginning on page 1-29 provides detailed information about using Newton communications options to configure and control your Newton Internet Enabler sessions.

Note
You can override the default Inet icon by defining the `'Icon` slot in your application's base view. ◆

Using Endpoints With Newton Internet Enabler Links

You use Newton Internet Enabler in your applications in much the same way that you use the other built-in Newton communications tools: you instantiate endpoints to use with Newton Internet Enabler and perform your communications operations with those endpoints. To learn about endpoints, see the chapter "Endpoint Interface" in *Newton Programmer's Guide*.

You specify the Newton Internet Enabler service identifier (`'inet'`) in your service options, and configure Newton Internet Enabler by passing options in options frames to your endpoint methods. See the section "Using the Newton Internet Enabler Options" beginning on page 1-29 for information about which options to use with each of your endpoint methods.

You can use a Newton Internet Enabler link with more than one endpoint. In fact, it makes sense to reuse your link with multiple endpoints over the life of your application: the first application to grab the link establishes an Internet session by dialing a modem and negotiating the low-level connection, and subsequent applications can use the same session without having to pay that price.

Your application can use several endpoints with the same Newton Internet Enabler link. Each endpoint, however, requires a significant amount of memory. And the Newton system software restricts the total number (for all applications) of endpoints that can be active.

To use multiple endpoints in your application, follow this sequence of operations:

1. Grab the link, as described in the section "Grabbing a Link" beginning on page 1-12.
2. Instantiate your first endpoint for use with Newton Internet Enabler. Use this endpoint to perform communications and then dispose of the endpoint.

3. Instantiate, use, and dispose of other endpoints.
4. Release your Newton Internet Enabler link, as described in the section “Releasing Your Link” beginning on page 1-26.

Using Multiple Endpoints With a Link

Several Newton applications can use endpoints at the same time. In fact, multiple applications can use multiple endpoints with a single Newton Internet Enabler link. However, there are certain restrictions around endpoint usage that you must beware of:

- Each endpoint requires a significant amount of memory
- The number of endpoints that can be open on the Newton at any time is limited. The exact limit depends on your hardware configuration.
- The limit on opened endpoints applies to the Newton as a whole. This means that if the machine limit is 4 endpoints and one application is using 2 endpoints, all other applications will be restricted to the possibility of using 2 endpoints.

Newton Internet Enabler and Callback Functions

Many of the Newton Internet Enabler functions require you to provide a **callback function**, which is a function that the Inet tool calls during and/or after the performance of the operation that you requested. The callback function receives status and error information.

For example, the `InetCancelLink` function calls the callback function that you provide after it finishes its operation. Your callback function for `InetCancelLink` can determine if an error occurred and can determine the current status of the link that you wanted cancelled.

Some operations call your callback function more than once. For example, the `InetGrabLink` function calls the callback function you provide many times during its operations. You can use your `InetGrabLink` callback function to monitor the progress of the grab, since each call to it provides you with the current status.

When a function requires that you specify a callback function, you do so by providing a context frame and the symbol of the function defined in that frame that you want to use as the callback function. For example, the `InetGrabLink` function takes three parameters and is declared as follows:

`InetGrabLink(`*linkID, clientContext, clientCallback*`);`

When you call InetGrabLink, you must specify a frame (or your application frame) as the value of *clientContext*, and you must specify a function defined in the frame as the value of *clientCallback*.

You might create a callback function for your `InetGrabLink` calls that looks like the following:

```
myApp.GrabLinkCallback := func(linkID, stat, err)
   begin
   if err=nil and stat.linkStatus <> 'connected then
      ;     // display status
   if err then
      ;     //handle the error
   // link established, so resolve the address
   end;
```

Then, when you call the `InetGrabLink` function in your application, you pass it the name of your callback function. For example:

```
myApp.TestGrab := func()
   begin
   myStatusView := InetStatusDisplay(nil, nil, nil);
   InetGrabLink(nil, self, 'GrabLinkCallback);
   ...
   end;
```

This function first calls the `InetStatusDisplay` function to create and display the status view. The call to `InetGrabLink` uses the default link ID and specifies `self` (the application frame) as the value of the *clientContext* parameter, and `'GrabLinkCallback` (the symbol for the callback function) as the value of the *clientCallback* parameter. The `GrabLinkCallback` function will be called repeatedly while the system is attempting to grab the link, until either the status is `'connected` or an error occurs.

The section "Grabbing a Link" beginning on page 1-12 provides a complete example and explanation of a callback function for the `InetGrabLink` function.

The section "Retrieving and Displaying Link Status Information" beginning on page 1-15 describes how to use the `InetStatusDisplay` function to display the progress of your `InetGrabLink` operation.

The section "Newton Internet Enabler Function Parameter Information" beginning on page 1-43 provides a detailed description of the *clientContext* and *clientCallback* parameters that you use in your Newton Internet Enabler function calls.

Using the Newton Internet Enabler Link Controller

The following is the sequence of steps that you need to perform when using Newton Internet Enabler in your application. Each step is described in detail in the sections that follow.

1. Grab a link by calling the `InetGrabLink` function. Before calling `InetGrabLink`, call `InetOpenConnectionSlip` to allow the user to modify the default link settings. While the link is being grabbed, call the `InetDisplayStatus` function to report the current link status to the user. After the link is grabbed, store the returned link ID into a variable for future access.
2. Instantiate an endpoint, passing down the Inet configuration options.
3. Bind the endpoint. The options that you pass down depend on whether you are using TCP or UDP, and on whether you are binding to initiate an outgoing connection (*connect*) or to wait for an incoming connection (*listen*).
4. Connect the endpoint. If you are using a TCP link, pass down the TCP remote socket option.
5. Set up the input spec for your endpoint.

6. Set up the output spec for your endpoint.
7. Send and receive data.
8. Disconnect and dispose of your endpoint.
9. Release the link by calling the `InetReleaseLink` function.

Grabbing a Link

To get started, you need to establish (grab) a link. To establish a link, you need to call the `InetGrabLink` function. Before calling InetGrabLink, you should call the `InetOpenConnectionSlip` function to allow the user to modify the default link settings. This function also resets the default link ID.

`InetOpenConnectionSlip(`*linkID, clientContext, clientSlipCallback*`)`

After `InetOpenConnectionSlip` finishes, it calls your callback function to let you know whether or not to proceed with the connection process. `InetOpenConnectionSlip` passes one parameter, a symbol, to your callback function. If the symbol is `'connect`, you should proceed with the connection; if not, the user cancelled the connection,

After calling `InetOpenConnectionSlip`, you call `InetGrabLink` with a link ID, a callback function, and a callback context frame:

`InetGrabLink(`*linkID, clientContext, clientCallback*`)`

For the linkID, you can tell `InetGrabLink` to use the default link by using `nil` or you can use an identifier returned by the `InetAddNewLinkEntry` function as the value of this parameter. When you specify `nil`, the system software uses the link ID that has been established as the default link ID. This is the ID established by the user in the connection slip. You almost always use `nil` as the value of this parameter to specify the default link ID, especially if you have first called `InetOpenConnectionSlip`.

The `InetGrabLink` operation can take some time to complete. After you call the `InetGrabLink` function, the Newton system software repeatedly calls your callback function to report the current status of grabbing the link. `InetGrabLink` calls your callback function until either an error occurs or until the status becomes 'connected.

The status value in your callback is a status frame, as described in the section “The Link Controller Status Frame” beginning on page 1-45. This frame contains the current link status value and (possibly) other information. In your callback, you can use the `InetDisplayStatus` function to show the current status to the user. The next section, “Retrieving and Displaying Link Status Information,” describes how to display status to the user.

Here is an example of a callback function for the InetOpenConnectionSlip function:

```
mySlipCallback := func(action)
   if action = 'connect then
      InetGrabLink(nil, self, 'GrablinkCallback);
```

Here is an example of a callback function for the `InetGrabLink` function:

```
myApp.GrabLinkCallback := func(linkID, stat, err)
   begin
   myLinkID := linkID;
   if err then
      begin
               // handle the error
      GetRoot():Notify(kNotifyAlert, kAppName,
                     call kGetInetError with (err));
               // close the status dialog
      InetDisplayStatus(nil, myStatusView, nil);
      end
   else if status.linkStatus <> 'connected then
         begin
               // just update the status dialog
         InetDisplayStatus(nil, myStatusView, status);
         end
   else  begin
               // now connected, soclose status dialog
               //  or start sending your own status
         local myStatus := {
               statusText:"Resolving Internet Address",};
         InetDisplayStatus(nil, myStatusView, myStatus);
               // resolve our host name address
         DNSGetAddressFromName("apple.com",self,
                                          DNSCallback);
         end;
   end;
```

The first statement, `myLinkID:=linkID`, saves the ID of the link that `InetGrabLink` is in the process of grabbing in one of your variables. You might want to store the link ID for use in other portions of your application.

If grabbing of the link is progressing without errors, your callback function gets called to report the progress. You can call the `InetDisplayStatus` function, as shown in the above example. The `myStatView` view used in the this example was created before the grab of the link was initiated, as shown in the next section, "Retrieving and Displaying Link Status Information."

The grab of the link terminates when the connection is made or when an error occurs. In either case, you can remove the status display view at that point. To do so, call the `InetDisplayStatus` function with `nil` as the value of the status parameter.

If `InetGrabLink` encounters an error, the error code will be a non-zero value and your application has to do something with that error. In the example function, a message is displayed and the connection attempt is terminated.

If `InetGrabLink` succeeds, the callback receives `'connected` as the value of `linkStatus`. At that point, you can perform any operations that are appropriate. The example function takes this opportunity to convert its remote echo host name into an IP address, which is saved in a local variable by the `DNSCallback` function. While the name resolution is taking place, the example updates the status display with its own message.

Retrieving and Displaying Link Status Information

Many applications want to display status to the user while a net connection is being established. Newton Internet Enabler makes this easy for you with the `InetDisplayStatus` function, which displays link status information on the Newton screen. Here is the declaration of the function:

statusView `InetDisplayStatus(`*linkID, statusView, status*`)`

You can use the `InetDisplayStatus` function in three ways, as follows:

- to create a new status view, pass `nil` as the value of each parameter:
```
myStatusView := InetDisplayStatus(nil, nil, nil);
```
- to display status for a link in an existing status view, pass in the link ID, the status view, and the status frame that was sent to your callback function:
```
InetDisplayStatus(myLinkID, myStatusView, myStatus);
```
- to remove and dispose of the status view, pass `nil` as the value of the status frame:
```
InetDisplayStatus(myLinkID, myStatusView, nil);
```

The `InetStatusDisplay` function creates and uses a view that is based on `protoStatusTemplate`. For information about this proto, see the chapter "Additional System Services" in *Newton Programmer's Guide*.

To initiate the status display, you need to open the status view. The most convenient place to do this is just before your call to the `InetGrabLink` function. For example, the following function creates the status view, stores it in `myStatView` for subsequent use, and then calls the `InetGrabLink` function:

```
DoGrabLink := func()
   begin
   myStatView := InetDisplayStatus(nil, nil, nil);
   InetGrabLink(nil, self, 'GrabLinkCallback);
   end;
```

While the grab operation is in progress, you can update the status display whenever your callback function gets called. For example, the following code segment from a grab link callback function updates the status display if no errors have occurred and if the link status has not yet become `'connected`:

```
   if err = nil and status.linkStatus <> 'connected then
      InetDisplayStatus(linkID, myStatView, stat);
```

When the grab operation is done, you can remove the status display. The following code segment from a grab link callback function removes the status display when the link status becomes `'connected`:

```
if err = nil and status.linkStatus = 'connected then
   InetDisplayStatus(linkID, myStatView, nil);
```

The view displayed by the `InetDisplayStatus` function contains a button that the user can tap to call the `InetCancelLink` function.

Configuring Newton Internet Enabler for Your Endpoint

After grabbing your Newton Internet Enabler link, you need to instantiate your endpoint. You send the `Instantiate` message to your endpoint with the options required to configure Newton Internet Enabler for your application.

You must set three options in your `Instantiate` message:

- The `'inet` service identifier option, which tells the Newton system software to use Newton Internet Enabler with your endpoint.
- The Inet tool physical link (`'ilid'`) option, which tells Newton Internet Enabler which link ID to use for your endpoint. Use the link ID that was returned by the `InetGrabLink` function.
- The Inet tool transport service type (`'itsv'`) option, which tells Newton Internet Enabler which transport type (for example, UDP or TCP) to use for your endpoint.

Binding Your Endpoint with Newton Internet Enabler

After you instantiate your endpoint, you need to bind it to an address. You either bind your endpoint to *connect* (initiate an outgoing connection), or to *listen* for an incoming connection. If you are binding an endpoint that is going to listen, you always need to pass the Inet local port (`'ilpt'`) option when you send the `Bind` message to your endpoint. If you are binding an endpoint that is going to connect, you need to pass the Inet local port option for UDP links, but not for TCP links.

The Inet local port option has two data slots that you specify: a short value, `InetPortNumber`, and a Boolean value, `useDefaultPort`. The `useDefaultPort` value only applies when you are binding an endpoint to connect over a UDP link. Assign the `InetPortNumber` a value as shown in Table 1-1 when sending the local port option with a `Bind` request:

Table 1-1 Local port numbers for binding with Newton Internet Enabler

Bind type	Transport service type	Local port number
For connect	TCP	The system always selects the local port number, so don't set this option. You can, however, send a get (`opGetCurrent`) of this option with your `Bind` to retrieve the port number that the system assigned.

Table 1-1 Local port numbers for binding with Newton Internet Enabler (continued)

Bind type	Transport service type	Local port number
For connect	UDP	If you specify `true` for `useDefaultPort`, Newton Internet Enabler will select the local port to use and will return its value in the option. If you specify `nil` for `useDefaultPort`, you must supply a port number that is not in use or the `Bind` will fail.
For listen	TCP	Specify a port number to listen on as defined by the *IETF Assigned Numbers RFC* document.
For listen	UDP	Specify a port number to listen on as defined by the *IETF Assigned Numbers RFC* document.

Connecting Your Endpoint with Newton Internet Enabler

After instantiating and binding your endpoint, you need to connect it. If you are using a TCP link, you need to pass the TCP remote socket (`'itrs'`) option when you send the `Connect` message to your endpoint. This option sets the host address with which TCP connects. You can use the domain name server to get this address, as described in the section "Using the Domain Name Service Interface" beginning on page 1-27.

If you are using a UDP link, you do not need to pass any options in your `Connect` message.

If you are sending the `Listen` message to your endpoint, you do not need to send any options with that message.

Sending Data

You use Newton Internet Enabler to send data just as you would with any Newton communications tool. You can set up an output specification frame and send the `Output` message to your endpoint after you have established a connection.

Note
When you are sending data over the Internet, you usually need to insert a linefeed (`unicodeLF`) character in your data. Most Internet data uses linefeed-carriage return pairs, while the Newton uses only carriage returns. ◆

Sending Data With a UDP Connection

For UDP connections, you need to include the Inet UDP destination socket (`'iuds'`) option to establish the destination of the UDP datagram. Your UDP output specification must include two flags in the `sendFlags` slot: the `kPacket` and `kEOP` flags. For example, the following code segment sends the string “Hello World!” out over a UDP link.

```
local myUDPstreamOutputSpec := {
   form:          'string,
   sendFlags:     'kPacket+'kEOP,
}

local myUDPOptions :=
[{
   label:      "iuds",
   type:       'option,
   opCode:     opSetCurrent,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
   [
      [130,43,2,2],  // host address
      7,             // destination port number
   ],
      typelist:
      [
         'struct,
         ['array, 'byte, 4],
         'short
      ]
   }
}];

try
   ep:Output("Hello World!", myUDPOptions,
                                 myUDPstreamOutputSpec);
onexception |evt.ex.comm| do
   return :DoDisconnect();
```

Sending Data With a TCP Connection

For TCP links, you do not need to include any options in your `Output` message, nor do you need to specify any `sendFlags` values in the output specification frame. For example, the following code segment sends the string "Hello World!" out over a TCP link.

```
local myTCPstreamOutputSpec := {
   form:          'string,
}

try
   ep:Output("Hello World!", nil, myTCPstreamOutputSpec);
onexception |evt.ex.comm| do
   return :DoDisconnect();
```

The above example calls the application's `DoDisconnect` function if any communication exception occurs while sending the data.

You can also send **expedited data** over a TCP link. Expedited data is a single byte of data that gets sent immediately. The data byte gets inserted in front of any data on the remote end that has been received but not yet processed. For example, you might need to send out a break character in the middle of transmitting a large amount of data. To do so, you use the Inet expedited data option with your `Output` message. The expedited data option is described on page 1-67.

See the chapter "Endpoint Interface" in *Newton Programmer's Guide* for detailed information about output specification frames and the `Output` method.

Receiving Data

You use Newton Internet Enabler to receive data just as you would with any Newton communications tool. Typically, this means that you set up an input specification frame and send the `SetInputSpec` message to your endpoint.

Note
When you are receiving data from the Internet, you usually need to strip the linefeed (`unicodeLF`) characters from your data. Most Internet data uses linefeed-carriage return pairs, while the Newton just uses the carriage return. ◆

Receiving Data With UDP

For UDP links, your input specification frame must include the `kPacket` receive flag and must include `useEOP:true` in the termination slot. In addition, you can include two options in the `rcvOptions` slot if you want to: include the UDP source socket option to retrieve the address of the datagram sender, and include the UDP destination socket option if you want to retrieve the exact address to which the packet you received was sent. The destination address might be other than your local address if the packet was sent to a broadcast address.

The following code segment receives a datagram packet over a UDP link.

```
local streamInputSpec := {
   form:          'string,
   termination:   {useEOP: true},
   discardAfter:  565,
   rcvFlags:      kPacket,
   rcvOptions:    {
                     label:   "iuss",
                     type:    'option,
                     opCode:  opGetCurrent,
                     result:  nil,
                     form:    'template,
                     data: {
                           arglist:
                           [
                              [0,0,0,0],  // host address
                              0, // host port number
                           ],
                           typelist: kPortAddrStruct,
                           [
                              'struct,
                              ['array, 'byte, 4],
                              'short
                           ]
                     }
                  },

   inputScript: func(ep, data, terminator, options)
      begin
      // do something with data
      end,

   completionScript: func(ep, options, result)
      begin
```

```
            // skip error handling for cancelled requests
        if result <> kCommAbortErr then
            begin
            print("Error: " && result);
            ep:DoDisconnect();
            end;
        end,
}

try
    ep:SetInputSpec(streamInputSpec);
onexception |evt.ex.comm| do
    return :DoDisconnect();
```

The example input specification frame above tells Newton Internet Enabler to receive a packet of data from the UDP link and provides two scripts: the `inputScript` function to process normal completion of data reception and the `completionScript` function to process unexpected termination of data reception. In addition, this input spec includes a "get" of the UDP source socket address, which will be filled in with the IP address of the host that sent the datagram to your application.

WARNING

If you attempt to send or receive a packet larger than the value specified in the discardAfter slot of your input specification frame, a `kCommErrBufferOverflow` error results and some data may be lost. The maximum size of a UDP packet for Newton Internet Enabler is `565`. ▲

Receiving Data With TCP

For TCP links, you do not need to include any options or specify any receive flags in your input specification frame. For example, the following code segment receives a carriage return-terminated string from a TCP connection.

```
local streamInputSpec := {
   form:          'string,
   termination:   {endSequence: UnicodeCR},

   inputScript: func(ep, data, terminator, options)
      begin
      // do something with data
      end,

   completionScript: func(ep, options, result)
      begin
         // skip error handling for cancelled requests
      if result <> kCommAbortErr then
         begin
         print("Error: " && result);
         ep:DoDisconnect();
         end;
      end,
}

try
   ep:SetInputSpec(streamInputSpec);
onexception |evt.ex.comm| do
   return :DoDisconnect();
```

The example input specification frame above tells Newton Internet Enabler to terminate input upon receiving a Unicode carriage return character and provides two scripts: the `inputScript` function to process normal completion of data reception and the `completionScript` function to process unexpected termination of data reception.

WARNING
Do not use the `discardAfter` slot in your input specification for TCP connections. If you want to limit the size of the data packet, specify the `byteCount` value in your termination frame. ▲

You can also receive expedited data over a TCP link. When expedited data arrives, your application is immediately notified: the link controller sends an application event frame. The `eventCode` slot of this event frame has the value `kEventToolSpecific` and the `data` slot is the byte that was received.

See the chapter “Endpoint Interface” in *Newton Programmer’s Guide* for detailed information about input specifications, the `SetInputSpec` method, handling communications events, and other styles of receiving data with an endpoint.

Disconnecting Your Endpoint

When you have finished using your endpoint, you need to disconnect, unbind, and dispose of it. The following function shows you an example of finishing your use of an endpoint.

```
MyApp.DoDisconnect := func()
   begin
   if ep then begin// ignore all disconnect errors
      try
            ep:Disconnect(true, nil);
      onexception |evt.ex.comm|  do
            nil;

      try
            ep:UnBind(nil)
      onexception |evt.ex.comm| do
            nil;

      try
```

```
            ep:Dispose()
      onexception |evt.ex.comm| do
            nil;
      end;
   end;
```

Releasing Your Link

After your application is completely done with the link, or whenever you will not be using the link for a long period of time (approximately 15 minutes or longer), you need to release it by calling the `InetReleaseLink` function. If no other applications are using the link, the Newton system software shuts it down.

You need to provide `InetReleaseLink` with a link ID, a callback function, and a callback context frame:

`InetReleaseLink(`*linkID, clientContext, clientCallback*`)`

IMPORTANT
The *clientContext* value that you specify in your `InetReleaseLink` call must match the *clientContext* value that you previously specified in your call to the `InetGrabLink` function for this link. Otherwise, an error will occur. ▲

The status value in your callback is a status frame, as described in the section "The Link Controller Status Frame" beginning on page 1-45. This frame contains the current link status value and other information. In your callback, you can determine the current status of the link after your release of it has completed, which will depend upon its use by other applications.

Here is an example of a callback function for the `InetReleaseLink` function:

```
myApp.ReleaseLinkCallback := func(linkID, stat, err)
   begin
   if stat.linkStatus = 'idle then
```

```
      ; // do something here
   end;
```

Power Management and Internet Links

Whenever a link is active or a grablink is in progress, the Newton Internet Enabler modifies the handling of power-down requests in the following two ways:

- The standard “power-down when idle” feature of the Newton is disabled.
- If the user tries to power down with the power switch, the Newton Internet Enabler displays a dialog asking the user if he or she really wants to drop the link. If the user taps OK, the link is dropped and the Newton is powered down. If the user taps Cancel, the link is retained and the Newton remains powered.

Using the Domain Name Service Interface

You can use the Newton Internet Enabler domain name service functions to translate between host name and Internet address representations. Newton Internet Enabler provides the following domain name service global functions:

- the `DNSCancelRequests` function cancels any pending DNS requests.
- the `DNSGetAddressFromName` function translates a domain name into its corresponding Internet address.
- the `DNSGetMailAddress` function translates a domain name into the Internet address for a mail server that serves that domain.
- the `DNSGetMailServerNameFromDomainName` function translates a domain name into the domain name for a mail server that serves that domain.
- the `DSNGetNameFromAddress` function translates an Internet address into its corresponding domain nam.

Each of the DNS global functions is described in the section “Domain Name Service Functions and Methods” beginning on page 1-61.

You must supply a *clientContext* and *clientCallback* parameter to each of the DNS functions, just as you do for the link controller functions. However, the DNS callback functions are called with different parameters than are the link controller functions.

The callback function for `DNSCancelRequests` receives no parameters.

The callback function for all of the other DNS functions receives two parameters: an array of DNS results frames and a result code. Each results frame contains a number of slots that describe the DNS operation that was performed. The format of these parameters is described in the section "The Domain Name Service Callback Function Format" beginning on page 1-50.

For example, the `DNSGetAddressFromName` function is declared as follows:

`DNSGetAddressFromName(`*addr, clientContext, clientCallback*`)`

An example of a callback for this function is shown here:

```
myApp.DNSGetAddrcallback := func(results, error)
   begin
   if error or length(results) < 1 then
      begin
      print("DNS error: " && error);
      // do something with the error
      return;
      end;

      // save the resolved address
   myRemoteIpAddr := results[0].resultIPAddress;
   end;
```

Using the Newton Internet Enabler Options

Table 1-2 describes the Newton Internet Enabler options. Each of these options is described in detail in the section "Newton Internet Enabler Options" beginning on page 1-67.

Table 1-2 Newton Internet Enabler options

Option name	**Description**	**When to use**
Expedited data transfer (`'iexp'`)	For expedited transmission of data over a TCP link.	Set this option with an `Output` call to transfer data on a TCP endpoint.
Physical link identifier (`'ilid'`)	To identify the link ID to use.	Set this option at endpoint instantiation time.
Local port (`'ilpt'`)	To set the local port number for TCP binds.	Set this option if you are binding to do a `Listen` (at endpoint instantiation or bind time). You don't need to set this option for a `Connect`.
	To set the local port number for UDP binds.	Set this option at endpoint instantiation or bind time.
	To retrieve the local port number used for TCP or UDP.	Retrieve the value of this option when you are connecting, sending, or receiving data.
Inet profile (`'iprf'`)	To retrieve the local host and gateway host IP addresses.	At any time after the link is established.
TCP remote socket (`'itrs'`)	To set the socket to which TCP connects.	Set the value of this option at before using the connection (at endpoint instantiation, bind, or connect time).

Table 1-2 Newton Internet Enabler options (continued)

Option name	Description	When to use
	To retrieve the sender address for data received over a TCP link.	Get the value of this option when listening for data on a TCP connection.
Transport service type (`'itsv'`)	To set the transport service type (TCP or UDP).	Set this option at endpoint instantiation time.
UDP destination socket (`'iuds'`)	To set the destination address for data being sent over a UDP link.	Set this option when sending data with a UDP connection.
	To retrieve the destination address for data received over a UDP link.	Get this option when listening for data on a UDP connection.
UDP source socket (`'iuss'`)	To retrieve the source address for data received over a UDP link.	Get the value of this option when listening for data on a UDP connection.

Newton Internet Enabler Interface Reference

This section describes the constants, data types, methods, and functions that you use with Newton Internet Enabler.

Constants

This section describes the constants that you use with Newton Internet Enabler.

Status Code Constants

The Newton system software uses the following status code symbols to convey the status of a Newton Internet Enabler operation:

```
'idle
'initializing
'connecting
'login
'connected
```

Constant descriptions

`'idle`	No link is established.
`'initializing`	Setting up for connecting.
`'connecting`	In the process of connecting.
`'login`	Performing login script.
`'connected`	The link is established.

Transport Service Type Constants

You can use the following constants to specify transport service types in Inet tool option requests:

```
constant kTCP  := 1;
constant kUDP  := 2;
```

Constant descriptions

`kTCP`	Use TCP transport service.
`kUDP`	Use UDP transport service.

Link Controller Error Codes

The high-level link controller can generate the errors described in this section.

```
constant kInetErrNoSuchLinkID       := -60501;
constant kInetErrLinkDisconnected   := -60504;
constant kInetErrConnectLinkFailed  := -60505;
```

Constant descriptions

`kInetErrNoSuchLinkID`	The specified link identifier does not exist.
`kInetErrnLinkDisconnected`	The link has been disconnected.
`kInetErrConnectLinkFailed`	The link could not connect.

DNS Error Codes

This section describes the error codes that the DNS tool can generate.

```
constant kDNSErrNoAnswerFoundYet          -60751;
constant kDNSErrInternalErr               -60752;
constant kDNSErrNameSyntaxErr             -60791;
constant kDNSErrNoNameServer              -60794;
constant kDNSErrAuthNameErr               -60795;
constant kDNSErrNoAnswerErr               -60796;
constant kDNSErrNonexistentDomain         -60797;
constant kDNSErrOutOfMemory               -60798;
constant kDNSErrCouldNotContactServer     -60800;
constant kDNSErrNoServersAvailable        -60801;
constant kDNSErrRequestFormatErr          -60802;
constant kDNSErrServerInternalErr         -60803;
constant kDNSErrServerNotImplemented      -60804;
constant kDNSErrServerRefused             -60805;
constant kDNSErrUnknownServerErr          -60806;
constant kDNSErrNoResponseFromServer      -60814;
constant kDNSErrNoResponseFromAnyServer   -60815;
```

Constant descriptions

`kDNSErrNoAnswerFoundYet`	The answer for the question has not yet been found.
`kDNSErrInternalErr`	Internal DNS tool error.
`kDNSErrNameSyntaxErr`	The name in the DNS request is not valid.
`kDNSErrNoNameServer`	The option specification does not contain a name server.
`KDNSErrAuthNameErr`	The domain does not exist.
`kDNSErrNoAnswerErr`	No answers available for request; this could be due to a domain that does not exist.
`kDNSErrNonexistentDomain`	The domain name does not exist.
`kDNSErrOutOfMemory`	DNS tool out of memory.
`kDNSErrCouldNotContactServer`	Could not connect to the current DNS server
`kDNSErrNoServersAvailable`	Could not connect to any of the listed DNS servers.
`kDNSErrRequestFormatErr`	The DNS server did not like the format of the request, which could indicate an invalid domain name.
`kDNSErrServerInternalErr`	An internal error occurred in the DNS server.
`kDNSErrServerNotImplemented`	The DNS server does not support the specified type of request.
`kDNSErrServerRefused`	The DNS server refused to answer the client's query.
`kDNSErrUnknownServerErr`	The DNS server returned an error code that is not recognized.

`kDNSErrNoResponseFromServer`
: No response from the current server

`kDNSErrNoResponseFromAnyServer`
: No response from any of the available DNS servers.

Newton Internet Enabler Lower-Level Tool Errors

The Newton Internet Enabler lower-level erors are separated into five sections:

- Inet tool errors
- application-related errors
- UDP errors

Inet Tool Errors

This section describes the errors that the built-in Inet tool can generate.

Note
Some of the errors in this section are internal communications stack configuration or operation errors. If you receive one of these errors, please contact the Newton Developer Technical Support team. ◆

```
constant kInetToolErrBindFailed               := -60001;
constant kInetToolErrIPBindFailed             := -60002;
constant kInetToolErrPushModule               := -60004;
constant kInetToolErrIlink                    := -60005;
constant kInetToolErrNetActivateReq           := -60006;
constant kInetToolErrTCPBind                  := -60007;
constant kInetToolErrGetRequest               := -60008;
constant kInetToolErrPutRequest               := -60009;
constant kInetToolErrConnect                  := -60010;
constant kInetToolErrDlAttach                 := -60011;
constant kInetToolErrBind                     := -60012;
constant kInetToolErrOpenLink                 := -60013;
constant kInetToolErrUnlink                  := -60014;
constant kInetToolErrOutOfPhase              := -60015;
constant kInetToolErrAddRoute                := -60016;
constant kInetToolErrListen                  := -60017;
constant kInetToolErrLinkNotOpened           := -60018;
constant kInetToolErrDriverNotOpened         := -60019;
constant kInetToolErrStreamNotOpened         := -60020;
constant kInetToolErrBindReqFailed           := -60021;
constant kInetToolErrConnResReqFailed        := -60022;
```

Constant descriptions

`kInetToolErrBindFailed`
: The bind operation failed at the lowest level.

`kInetToolErrIPBindFailed`
: The IP layer bind operation failed.

`kInetToolErrPushModule`
: Internal communications stack configuration error.

`kInetToolErrIlink`
: Internal communications stack configuration error.

`kInetToolErrNetActivateReq`
: Internal communications stack configuration error.

`kInetToolErrTCPBind`
: The TCP layer bind operation failed.

`kInetToolErrGetRequest`
: The `get` request resulted in an error.

`kInetToolErrPutRequest`
: The `put` request resulted in an error.

`kInetToolErrConnect`
: The `connect` request resulted in an error.

`kInetToolErrDlAttach`
: Internal communications stack configuration error.

`kInetToolErrBind`
: Internal communications stack configuration error.

`kInetToolErrOpenLink`
: Internal communications stack configuration error.

`kInetToolErrUnlink`
: Internal communications stack configuration error.

`kInetToolErrOutOfPhase`
: Internal communications stack configuration error (the stack layers are out of sync).

`kInetToolErrAddRoute`
: Internal communications stack configuration error.

`kInetToolErrListen`
: Internal communications stack configuration error.

`kInetToolErrLinkNotOpened`
: Internal communications stack configuration error.

`kInetToolErrDriverNotOpened`
: Internal communications stack configuration error.

`kInetToolErrStreamNotOpened`
: Internal communications stack configuration error.

`kInetToolErrBindReqFailed`
: The bind request failed.

`kInetToolErrConnResReqFailed`
: Internal communications stack configuration error.

Application-related Errors

This section describes the Inet errors that are related to your application.

Note

Some of the errors in this section are internal communications stack configuration or operation errors. If you receive one of these errors, please contact the Newton Developer Technical Support team. ◆

```
constant kInetToolErrMemAlloc                := -60023;
constant kInetToolErrMsgType                 := -60024;
constant kInetToolErrNoDevice                := -60025;
constant kInetToolErrllegalOpenOnStream      := -60026;
constant kInetToolErrReqInInvalidState       := -60027;
constant kInetToolErrPrimitiveTooSmall       := -60028;
constant kInetToolErrPrimitiveOutOfRange     := -60029;
constant kInetToolErrPrimitiveOnInvalidStr   := -60030;
constant kInetToolErrMessageTooLong          := -60031;
constant kInetToolErrNetworkAlreadyActive    := -60032;
constant kInetToolErrNetworkNumberInvalid    := -60033;
constant kInetToolErrUnsupportedIoctl        := -60034;
constant kInetToolErrStreamAlreadyAttached   := -60035;
constant kInetToolErrUnknownMuxIndex         := -60036;
constant kInetToolErrNetworkIsInactive       := -60037;
constant kInetToolErrBogusConnection         := -60038;
constant kInetToolErrInvalidBillingMode      := -60039;
constant kInetToolErrNoTrigSelectedInAlarm   := -60040;
constant kInetToolErrInvalidTrigSize         := -60041;
constant kInetToolErrInvalidConnectionRef    := -60042;
constant kInetToolErrIlegalMdataInPrim       := -60043;
constant kInetToolErrMissingMdataInPrim      := -60044;
constant kInetToolErrInvalidSegmentedPrim    := -60045;
constant kInetToolErrInvalidNPIVersion       := -60046;
constant kInetToolErrInvalidAddress          := -60047;
constant kInetToolErrOutOfTCPPortNumbers     := -60048;
constant kInetToolErrSocketInUse             := -60049;
constant kInetToolErrReservedPortNumber      := -60050;
constant kInetToolErrExpDataNotSupported     := -60051;
```

Constant descriptions

`kInetToolErrMemAlloc`	Requested memory could not be allocated.
`kInetToolErrMsgType`	Internal communications stack configuration error.
`kInetToolErrNoDevice`	Internal communications stack configuration error: no stream available.
`kInetToolErrllegalOpenOnStream`	Internal communications stack configuration error.
`kInetToolErrReqInInvalidState`	Internal communications stack configuration error.
`kInetToolErrPrimitiveTooSmall`	Internal communications stack configuration error.
`kInetToolErrPrimitiveOutOfRange`)	Internal communications stack configuration error.
`kInetToolErrPrimitiveOnInvalidStr`	Internal communications stack configuration error.
`kInetToolErrMessageTooLong`	Internal communications stack configuration error.
`kInetToolErrNetworkAlreadyActive`	Internal communications stack configuration error.
`kInetToolErrNetworkNumberInvalid`	Internal communications stack configuration error.
`kInetToolErrUnsupportedIoctl`	Internal communications stack configuration error.
`kInetToolErrStreamAlreadyAttached`	Internal communications stack configuration error.
`kInetToolErrUnknownMuxIndex`	Internal communications stack configuration error.
`kInetToolErrNetworkIsInactive`	Internal communications stack configuration error.
`kInetToolErrBogusConnection`	Internal communications stack configuration error.

`kInetToolErrInvalidBillingMode`
Internal communications stack configuration error.

`kInetToolErrNoTrigSelectedInAlarm`
Internal communications stack configuration error.

`kInetToolErrInvalidTrigSize`
Internal communications stack configuration error.

`kInetToolErrInvalidConnectionRef`
Internal communications stack configuration error.

`kInetToolErrIlegalMdataInPrim`
Internal communications stack configuration error.

`kInetToolErrMissingMdataInPrim`
Internal communications stack configuration error.

`kInetToolErrInvalidSegmentedPrim`
Internal communications stack configuration error.

`kInetToolErrInvalidNPIVersion`
Internal communications stack configuration error.

`kInetToolErrInvalidAddress`
Internal communications stack configuration error.

`kInetToolErrOutOfTCPPortNumberss`
Internal communications stack configuration error.

`kInetToolErrSocketInUse`
Internal communications stack configuration error.

`kInetToolErrReservedPortNumber`
Internal communications stack configuration error.

`kInetToolErrExpDataNotSupported`
Internal communications stack configuration error.

UDP Errors

This section describes the UDP-related errors that the built-in Inet tool can generate.

Note
Some of the errors in this section are internal communications stack configuration or operation errors. If you receive one of these errors, please contact the Newton Developer Technical Support team. ◆

```
constant kInetToolErrRedundentRequest        := -60052;
constant kInetToolErrUnexpectedDLPrim        := -60053;
constant kInetToolErrUnexpectedTPIPrim       := -60054;
constant kInetToolErrUnexpectedNPIPrim       := -60055;
constant kInetToolErrUnknownTPIErrorCode     := -60056;
```

Constant descriptions

`kInetToolErrRedundentRequest`
Internal communications stack configuration error.

`kInetToolErrUnexpectedDLPrim`
Internal communications stack configuration error.

`kInetToolErrUnexpectedTPIPrim`
Internal communications stack configuration error.

`kInetToolErrUnexpectedNPIPrim`
Internal communications stack configuration error.

`kInetToolErrUnknownTPIErrorCode`
Internal communications stack configuration error.

Inet Tool-Specific Errors

This section describes the errors that relate to the Inet tool disconnecting unexpectedly.

```
constant kInetErrStreamInoperative           := -60057;
```

Constant descriptions

`kInetErrStreamInoperative`
The communications connection shut down due to a fatal error.

Newton Internet Enabler Events

The Newton Internet Enabler generates some events that you can handle in the `EventHandler` method of your endpoints. The `EventHandler` method, which is described in the chapter "Endpoint Interface" of *Newton Programmer's Guide*, receives a single parameter, which is an event frame. The slots of the event frame are as follows:

`eventCode`	An integer event code.
`data`	An integer representing event data.
`serviceId`	A string representing the communication tool that originated the event. For example, "mods" identifies the modem tool.
`time`	An integer representing the time when the event occurred. This is the number of ticks since the system was last restarted, not including time when it was turned off.

When the Newton Internet Enabler sends an event to your application, the `serviceId` is '`inet`'. The event codes and data that Newton Internet Enabler can send are shown in Table 1-3.

Table 1-3 Newton Internet Enabler Application Events

Event code	Data value	Description
`kEventToolSpecific`	An error code; currently, this is always the value: `kInetErrStreamInoperative`	Sent to your endpoint when the connection becomes inoperative due to a fatal error.
`kEventToolSpecific`	A byte of data received for the application.	Sent to your endpoint when an expedited data byte arrives. This is only applicable to endpoints using TCP.
`kCommToolEventDisconnected`	none	Sent to your endpoint if the remote side disconnects a TCP connection.

Since the `eventCode` slot has the same value (`kEventToolSpecific`) in the first two cases shown in Table 1-3, you must check the value of the data slot to determine what has happened:

- if the `data` slot value is less than zero, you know that your connection has closed down due to an error. The `data` slot value is the error code.
- otherwise, you know that your application has just received a byte of expedited data. The `data` slot value is the byte of expedited data that has just been received.

For more information on receiving expedited data, see the section "Receiving Data With TCP" beginning on page 1-23.

Handing TCP Disconnect Events

If the remote sides disconnects a TCP connection, your endpoint generates a disconnect event. However, you might still need to process data that has been received and buffered prior to the disconnect. To process the buffered data, you need to post input specifications until you get the error `kCommErrNotConnected`.

Note
You must disconnect and unbind your endpoint event if you receive a disconnect (`kCommToolEventDisconnected`) event. ◆

Newton Internet Enabler Function Parameter Information

Many of the Newton Internet Enabler functions use one or more of the parameter types described here.

The Link Identifier Parameter

The link identifier parameter, which is used by a number of the link controller functions, defines the link that you want to use for a link controller operation. If you specify `nil` for the link identifier, Newton Internet Enabler substitutes the default link identifier.

Users define the default link ID to use in the Internet Setup application. Whenever you successfully call the `InetGrabLink` function, the link ID that you supplied to that function will automatically become the default link ID. Under most circumstances, you should specify `nil` as the value of the link identifier to use the default link ID as established by the user.

The Client Context Parameter

The client context parameter, which is used by a number of the link controller functions, is a reference to a frame. The referenced frame must contain the callback function that is defined in the client callback parameter.

Note

You must provide the same client context frame to some of the Newton Internet Enabler functions that are called in pairs. For example, the `InetCancelLink` function can only cancel a link if the *clientContext* parameter that you pass into `InetCancelLink` matches the *clientContext* parameter that you previously passed into `InetGrabLink`. ◆

The Client Callback Parameter

The client callback parameter, which is used by a number of the Newton Internet Enabler functions, is the symbol for a function. This function must be defined in the frame that is defined by the client context parameter.

In most cases, the Newton system software calls your callback function upon completion of its operations. In some cases, notably the `InetGrabLink` function, the Newton system software calls your callback function repeatedly during the operation to keep you informed of the status.

If you are calling a link controller function, the declaration of your callback function must match the specification described in the next section, “The Link Controller Callback Function Format.”

If you are calling a domain name service function, the declaration of your callback function must match the specification in the section “The Domain Name Service Callback Function Format” beginning on page 1-50.

The Link Controller Callback Function Format

The callback function that you supply to each link controller call must be declared with the following format:

`func callBackFcn(`*linkId, statusFrame, err*`);`

The system software calls `callBackFcn` after the Newton Internet Enabler function has completed.

statusFrame	A frame with the format shown in the next section, "The Link Controller Status Frame." The *statusFrame* provides information on the current state of the link.
err	The operation result. If the value is `nil`, the operation was successful; otherwise, the value is one of the error codes shown in the section "Newton Internet Enabler Lower-Level Tool Errors" on page 1-34.

The Link Controller Status Frame

The link controller status frame contains a slot that conveys link status information to you. This frame is designed to work closely with views based on the `protoStatusTemplate` view, which is described in the Transport Interface chapter of *Newton Programmer's Guide*.

Note that the status frame can have other slots in it; however, only the *linkStatus* slot is available for application use.

Slot descriptions

linkStatus	A symbol that describes the current status of the link. This is one of the values described in the section "Status Code Constants" on page 1-31.

The following is an example of a link controller status frame:

```
{
   linkStatus: 'connected
}
```

The Link Entry Information Frame

The Newton Internet Enabler accesses link entry information that is stored in an internal soup. Each entry in this soup defines link configuration information and is represented by a link entry information frame. An example of a link entry information frame follows:

```
{
   linkID:               0,
   tags:                 ['Inet],
   setupName:            "My dial-up",
   physicalLayer:        'modem,
   linkLayer:            'PPP,
   localAddress:         [205,149,167,179],
   localAddressFixed:    nil,
   gatewayAddress:       [204,156,128,1],
   gatewayAddressFixed:  nil,
   phoneNumber:          "(408) 555-1234",
   userName:             "",
   passWord:             "",
   dnsServerAddress:     [204,156,128,1],
   defaultDomain:        ".",
   loginInfo:            {interpreterSymbol:  'default,
                          loginScriptTimeout: 60,
                          loginInstructions: [
                            {type: 'waitFor, data: "ogin:"},
                            {type: 'userName},
                            {type: 'sendCR},
                            {type: 'waitFor, data: "word"},
                            {type: 'password},
                            {type: 'sendCR},
                            {type: 'waitFor, data:"ing..."},
                           ],
                         }
}
```

Slot descriptions

linkID	The (integer) ID for this link. This ID is generated by the link controller.
tags	Reserved for internal use. Do not modify.
setupName	The name of this link setup entry.
physicalLayer	The symbol for which physical layer to use when connecting. Use either `'modem` or `'serial`.
linkLayer	The symbol for which link-layer protocol to use for this link. Use either `'PPP` or `'SLIP`.
localAddress	Optional. The manually-entered local IP address of the Newton.
localAddressFixed	`True` if a fixed local address is required, `nil` if not.
gatewayAddress	Optional. The IP address of the gateway host to which the Inet tool connects.
gatewayAddressFixed	`True` if a fixed host address is required, `nil` if not.
phoneNumber	Optional. The phone number for the modem to dial.
userName	Optional. The user name to use as the account name for login scripts.
passWord	Optional. The password to use as the account password for login scripts.
dnsServerAddress	The DNS server IP address.
defaultDomain	Optional. The default DNS domain
loginInfo	A frame specifying the login information. This frame contains the following three slots:

interpreterSymbol	A symbol identifying which interpreter to use. Use `'default` for the default interpreter.
loginScriptTimeout	The number of seconds to wait for input.
loginInstructions	An array of frames interpreted by the default interpreter. The frames are described in the next section.

Login Script Frames

The default login script interpreter accepts an array of frames, each of which must have a `'type` slot that specifies the type of information contained in the frame. Table 1-4 shows the frame types that you can use in your login scripts.

Table 1-4 Login script frames

Frame format	Description

`{type: 'waitFor,`
` data: "data to wait for"}`
	Waits for the string in the `data` slot. The string can contain any Unicode character. The string is converted to ASCII using the standard Macintosh encoding.

`{type: 'send,`
` data: "data to send"}`
	Sends the string specified in the `data` slot. The string can contain any Unicode character. The string is converted to ASCII using the standard Macintosh encoding. Note that a newline is not automatically sent.

`{type: 'sendCR}`	Sends a newline (0x10).
`{type: 'pause, data:1}`	Pauses the input script for the number of seconds specified in the `data` slot.
`{type: 'userName}`	Sends the string in the `userName` slot of the link entry information frame. Note that a newline is not automatically sent.

Table 1-4 Login script frames (continued)

Frame format	Description
`{type: 'password}`	Sends the string in the `password` slot of the link entry information frame. Note that a newline is not automatically sent.
`{type: 'localAddress}`	Waits for and reads in an IP address. The value of the link entry frame's `localAddress` slot is overridden with this address value. Use with SLIP connections to obtain the dynamically assigned address.
`{type: 'gatewayAddress}`	Waits for and reads in an IP address. The value of the link entry frame's `gatewayAddress` slot is overridden with this address value. Use with SLIP connections to obtain the dynamically assigned address.

The Domain Name Service Callback Function Format

The callback function for the `DNSCancelRequests` function does not receive any parameter values.

The callback function that you supply to each of the other domain name service calls must be declared with the following format:

```
func callBackFcn(resultsArray, resultCode);
```

The system software calls `callBackFcn` after the Newton Internet Enabler function has completed.

resultsArray	An array of zero or more DNS results frames. The format of a DNS results frame is described in the next section, "The DNS Results Frame."
resultCode	One of the result codes described in the section "DNS Error Codes" on page 1-32.

The DNS Results Frame

The DNS results frame contains a number of slots that describe the DNS operation. Different slot values are filled in by each of the DNS operations. The possible slot types are shown here:

```
{
   type resultType,
   targetDomainName domainNameString,
   resultDomainName domainNameString,
   targetIPAddress addressArray,
   resultIPAddress addressArray
}
```

Each results frame contains a `type` slot and at least one result slot. Most results frames contain the `targetDomainName` slot; however, this is not guaranteed. Table 1-5 shows which slot is guaranteed to be valid for each DNS operation.

Table 1-5 Result slots for each DNS operation

DNS operation	Results frame slot
`DNSGetAddressFromName`	`resultIPAddress`
`DNSGetNameFromAddress`	`resultDomainName`
`DNSGetMailServerNameFromDomainName`	`resultDomainName`
`DNSGetMailAddressFromName`	`resultIPAddress`

As you can see in Table 1-5, if the operation results in a domain name, the results frame contains one or more `resultDomainName` slots. If the operation results in an IP address, the results frame will instead contain one or more `resultIPAddress` slots.

For example, the `DNSGetAddressFromName` function returns a results array that looks something like this:

```
[ {
type kDNSAddressType,
targetDomainName "newton.apple.com.",
resultIPAddress [155,227,54,3]
} ]
```

In contrast, the `DNSGetNameFromAddress` function returns a results array that looks something like this

```
[{
type kDNSDomainNameType,
targetDomainName "newton.apple.com.",
resultIPAddress [155,227,54,3]
} ]
```

Some DNS operations return a results array that contains more than one results frame. For example, a mail exchange operation can generate multiple mail exchange results frames.

The value types for each frame type are described below.

Slot descriptions

resultType	The type of result contained in the frame. This is either `kDNSAddressType` or `kDNSDomainNameType`. If the value is `kDNSAddressType`, the operation that generated this result frame resulted in an IP address; for example, the `DNSGetAddressFromName` function. If the value is `kDNSDomainNameType`, the operation that generated this result frame resulted in a domain name string; for example, the `DNSGetNameFromAddress` function.
domainNameString	The domain name used or resulting from the operation. For example `"newton.apple.com."`.
addressArray	The IP address, specified as an array of four bytes. For example, `[155,227,54,3]`.

Link Controller Functions and Methods

This section describes the link controller functions and methods.

InetAddNewLinkEntry

linkID `InetAddNewLinkEntry(`*newLinkInfo*`)`

Installs a new link information entry on the user's Newton and returns the ID assigned to the entry.

newLinkInfo	A link entry information frame, as described in "The Link Entry Information Frame" on page 1-46.

You can use the `InetAddNewLinkEntry` function to install a new link information entry in the Internet setup soup. The new link entry is created using the information supplied in the *newLinkInfo* frame. A new linkID is allocated for the entry. That link ID is returned as the function value.

InetCancelCurrentRequest

`InetCancelCurrentRequest(`*linkId*`)`

Cancels any active request on the specified link.

linkId	The ID of the link that you want to cancel. Specify `nil` to use the default link ID.

The `InetCancelCurrentRequest` function cancels any active requests on the link specified by *linkId*.

InetCancelLink

`InetCancelLink(`*linkId, clientContext, clientCallback*`)`

Cancels an `InetGrabLink` operation that is in progress.

linkID	The ID of the link that you want to cancel. Specify `nil` to use the default link ID.
clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.

clientCallback	The client callback function, as described in the section "The Link Controller Callback Function Format" on page 1-45.

The `InetCancelLink` function cancels an `InetGrabLink` request that is in progress.

The `InetGrabLink` function is described on page 1-58.

InetDisconnectLink

`InetDisconnectLink(`*linkId, clientContext, clientCallback*`)`

Disconnects a link no matter how many clients are using the link.

linkID	The ID of the link that you want to cancel. Specify `nil` to use the default link ID.
clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.
clientCallback	The client callback function, as described in the section "The Link Controller Callback Function Format" on page 1-45.

The `InetDisconnectLink` function disconnects a link directly without needing to close all of the applications that are using it.

WARNING
The `InetDisconnectLink` function is intended for use only by special purpose utility programs. Do not use this function unless you are certain that it the right thing to do. ▲

InetDisplayStatus

statusView `InetDisplayStatus(`*linkId, statusView, status*`)`

Displays status information about a link on the user's Newton screen.

linkID	The ID of the link for which you want to display status information. This must be an active link. Specify `nil` to use the default link ID.
statusView	The view to use for displaying the status. The view must be based on `protoStatusTemplate`. If you provide a template, `InetDisplayStatus` opens the view for your. If you supply `nil` as the value of this parameter, `InetDisplayStatus` creates and opens a new view for you.
status	A status frame, such as the one passed to your `InetGrabLink` callback function. The format of this frame is described in the section “The Link Controller Status Frame” beginning on page 1-45. If you supply `nil` as the value of this parameter and *statusView* is not `nil`, `InetDisplayStatus` removes the status display from the screen.

The `InetDisplayStatus` function displays link status information on the Newton screen. The view also contains a Stop button that the user can tap to cancel a grab link operation. If the grab gets cancelled, you can keep the view open and continue to display your own application status. To do so, follow these steps:

1. Set the value of the `appSymbol` slot in the status view to reference your application. For example:

```
myStatusView.appSymbol := kAppSymbol;
```

2. Implement a cancel script in your base view. When the user taps the Stop button, the status view calls a `'CancelScript(reason)` script in your base view. You can implement this script to respond to the stop button.

The `InetDisplayStatus` function returns the view that it used to display the status. You can use this for future calls to this function.

Note

To create a new status view for display on the screen, specify nil as the value of the *statusView* parameter. Newton Internet Enabler will create the view and return it as the function value.

To remove the status view from the screen, specify `nil` as the value of the *status* parameter and specify a view (not `nil`) as the value of the *statusView* parameter. ◆

InetGetAllLinksStatus

linksStatusFrame `InetGetAllLinksStatus()`

The `InetGetAllLinksStatus` function returns a frame that specifies the status of all known links. If there are no available links, the `InetGetAllLinksStatus` function returns `nil` or a frame containing empty arrays.

WARNING

Since the user can add or remove available links at any time by using the Internet Setup application, you must be careful about caching the results of the `InetGetAllLinksStatus` function. ▲

The *linksStatusFrame* contains three slots, each of which is an array with one entry for each available link.

Slot descriptions

LinkIDS	An array that contains the ID of each available link.
statuses	An array that contains the status code for each available link. Each entry in this array is a status code value, as described in the section “Status Code Constants” on page 1-31.
names	An array that contains the name for each available link. Each entry in this array is the string name that the user specified when defining the link configuration in the Internet Setup application.

The following is an example of the *linksStatusFrame*:

```
{
   LinkIDs [1,2,3]
   statuses ['connected, 'idle, 'idle],
   names ["Best", "Scruznet", "Compuserve"]
}
```

Note
The `InetGetAllLinksStatus` function is a synchronous call that raises an exception if it encounters any problems. ◆

InetGetDefaultLinkID

linkID `InetGetDefaultLinkID()`

The `InetGetDefaultLinkID` function returns the ID of the default link. This is the ID that the Newton system software uses when you specify `nil` as the value of the *linkID* parameter for one of the other link controller functions.

InetGetLinkEntry

linkEntry `InetGetLinkEntry(`*linkID*`)`

Returns the link entry information frame associated with *linkId*.

linkID	The ID of the link. Specify `nil` to use the default link ID.

The `InetGetLinkEntry` function returns the link entry information frame for the link *linkID*. For information about the format of link entry information frames, see “The Link Entry Information Frame” on page 1-46.

InetGetLinkStatus

`InetGetLinkStatus(`*linkID*`)`

Returns the status of a link.

linkID	The ID of the link. Specify `nil` to use the default link ID.

The `InetGetLinkStatus` function returns the status of the link *linkID*. The status value is one of the status code constants, as described in the section "Status Code Constants" on page 1-31..

Note
The `InetGetLinkStatus` function is a synchronous function that raises an exception if it encounters any problems. ◆

InetGrabLink

`InetGrabLink(`*linkId, clientContext, clientCallback*`)`

Provides access to a link.

linkID	The ID of the link that you want to access. Specify `nil` to use the default link ID.
clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.
clientCallback	The client callback function, as described in the section "The Link Controller Callback Function Format" on page 1-45.

The `InetGrabLink` function attempts to access a link and calls your callback function with the status and error code for that operation.

The Newton system software calls your *clientCallback* function repeatedly while attempting to connect, supplying you with the current status. The `InetGrabLink` operation does not complete until the returned status is `'connected`.

If the `InetGrabLink` operation fails, the `err` parameter to the *clientCallback* function indicates the reason. Otherwise, the value of the `err` parameter is `nil`.

InetOpenConnectionSlip

view `InetOpenConnectionSlip(`*linkId, clientContext, clientCallback*`)`

Opens a connection slip for the link. The user can change link information with the slip.

linkID	The ID of the link for which you want to open a connection slip. Specify `nil` to use the default link ID.
clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.
clientSlipCallback	A client callback function that receives a single parameter. This parameter is a symbol with one of the following values: `'connect` or `'close`.

The `InetOpenConnectionSlip` function displays a connection slip on the screen to allow the user to change to a different link or override the current worksite or phone number information. The user can complete the slip by tapping the Connect button or by tapping the Close box.

Your *clientCallback* function is called after the slip has been completed by the user. If your callback is called with the `'close` symbol, it means that the user has canceled the connection by tapping the Close box in the slip. If your callback is called with the `'connect` symbol, it means that the user has tapped the Connect button in the slip and that you should call `InetGrabLink` to proceed with the connection process.

If the link is already open, `InetOpenConnectionSlip` does not open a slip. In this case, `InetOpenConnectionSlip` calls your callback function with `'connect`, which means that you can proceed as if the user had tapped the Connect button.

The `InetOpenConnectionSlip` function returns the open slip view or nil if it does not open a slip.

IMPORTANT
Apple recommends that you call `InetOpenConnectionSlip` before calling `InetGrabLink` to allow users the opportunity to change the connection settings. ▲

InetReleaseLink

`InetReleaseLink(`*linkId, clientContext, clientCallback*`)`

Relinquishes access to a link.

linkID	The ID of the link that you want to release. Specify `nil` to use the default link ID.
clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.
clientCallback	The client callback function, as described in the section "The Link Controller Callback Function Format" on page 1-45.

The `InetReleaseLink` function releases your access to a link. If the link does not have any additional clients, it may close down.

Note
If the user has enabled a release link timeout in the link entry, then `InetReleaseLink` does not necessarily drop the link, even if it does not have any additional clients. When the count goes to zero and a release link timeout has been enabled, the Newton Internet Enabler software displays an indicator on the user's screen (a blinking star at the top center of the screen). The user can tap that indicator to open a slip that can be used to release the link. If another `InetGrabLink` call is issued for the link before the timeout completes, the link is not released. ◆

Your *clientCallback* function is called after the link has been released. The status of the link at that time will depend on its use by other applications.

IMPORTANT

Apple recommends that you call `InetReleaseLink` whenever you no longer need to perform communications over the link for fifteen minutes or longer. ▲

InetSetDefaultLinkID

`InetSetDefaultLinkID(`*linkId*`)`

Establishes which link is the default link.

linkID	The ID of the link that you want to become the default link ID.

The `InetSetDefaultLinkID` function establishes linkId as the default link ID. The Newton Internet Enabler software uses the default link ID whenever you pass `nil` as the value of a linkID parameter. For more information about the default link ID, see "The Link Identifier Parameter" on page 1-43.

Domain Name Service Functions and Methods

This section describes the functions and methods that you can use to access the domain name service.

DNSCancelRequests

`DNSCancelRequests(`*clientContext, clientCallback*`)`

Cancels outstanding domain name server requests.

clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.
clientCallback	The client callback function. This function is called with no parameters, in contrast to the other DNS callback functions.

The `DNSCancelRequests` function cancels any outstanding domain name service requests that have been made by the client with context *clientContext*.

You must supply the same value for *clientContext* as you did when you made the DNS request.

Newton Internet Enabler will not call the callback functions for any DNS requests that get cancelled.

WARNING
The callback function that you supply to `DNSCancelRequests` does not have any parameters. This is different from the other DNS callback functions. ◆

DNSGetAddressFromName

`DNSGetAddressFromName(`*nameString*`,` *clientContext*`,`*clientCallback*`)`

Translates a domain name into its corresponding Internet address.

nameString	An Internet domain name string.
clientContext	The client context, as described in the section "The Client Context Parameter" on page 1-44.
clientCallback	The client callback function, as described in the section "The Domain Name Service Callback Function Format" on page 1-50.

The `DNSGetAddressFromName` function resolves the domain name *nameString* into an IP address. `DNSGetAddressFromName` fills in the `resultIPAddress` slot in a DNS results frame, as described in the section "The Domain Name Service Callback Function Format" beginning on page 1-50, and calls your callback function with the result code and that frame as parameters. The following is an example of a results frame for the `DNSGetAddressFromName` function:

```
{
type kDNSAddressType,
targetDomainName "newton.apple.com.",
resultIPAddress [155,227,54,3],
}
```

The result code is `nil` if the function succeeded; otherwise, the result code is one of the error codes described in the section “DNS Error Codes” beginning on page 1-32.

If *nameString* ends with the period (`'.'`) character, the string is assumed to be in conformance with RFC 1123. If *nameString* does not end with the period character, `DNSGetAddressFromName` attempts to complete the name as follows:

- `DNSGetAddressFromName` first appends a period to a copy of *nameString* and attempts name resolution with that string.
- If that request is not successful, `DNSGetAddressFromName` appends the local domain name and a period to a copy of *nameString* and attempts name resolution with that string.

DNSGetMailAddressFromName

`DNSGetMailAddressFromName(`*nameString*`,`*clientContext*`,`*clientCallback*`)`

Translates a domain name into the Internet address for a mail server that serves that domain.

nameString	An Internet domain name string.
clientContext	The client context, as described in the section “The Client Context Parameter” on page 1-44.
clientCallback	The client callback function, as described in the section “The Domain Name Service Callback Function Format” on page 1-50.

The `DNSGetMailAddressFromName` function returns the IP address of a mail server that serves the domain specified by *nameString*. `DNSGetMailAddressFromName` fills in the `resultIPAddress` slot in a DNS results frame, as described in the section “The Domain Name Service Callback Function Format” beginning on page 1-50, and calls your callback function with the result code and that frame as parameters. The following is an example of a results frame for the `DNSGetMailAddressFromName` function:

```
{
type kDNSAddressType,
targetDomainName "mail.newton.apple.com.",
resultIPAddress [155,227,54,3],
}
```

The result code is `nil` if the function succeeded; otherwise, the result code is one of the error codes described in the section “DNS Error Codes” beginning on page 1-32.

If *nameString* ends with the period (`'.'`) character, the string is assumed to be in conformance with RFC 1123. If *nameString* does not end with the period character, `DNSGetAddressFromName` attempts to complete the name as follows:

- `DNSGetMailAddressFromName` first appends a period to a copy of *nameString* and attempts name resolution with that string.
- If that request is not successful, `DNSGetMailAddressFromName` appends the local domain name and a period to a copy of *nameString* and attempts mail server resolution with that string.

If there is more than one mail server for the domain specified by *nameString*, the results array contains multiple `resultDomainName` frames, one for each mail server. The frames are ordered in the results frame according to the preference order of the mail exchange resource record.

DNSGetMailServerNameFromDomainName

`DNSGetMailServerNameFromDomainName(` *nameString*`,` *clientContext*`,`*clientCallback*`)`

Translates a domain name into the domain name for a mail server that serves that domain.

nameString	An Internet domain name string.
clientContext	The client context, as described in the section “The Client Context Parameter” on page 1-44.

clientCallback	The client callback function, as described in the section "The Client Callback Parameter" on page 1-44.

The `DNSGetMailServerNameFromDomainName` function returns the domain name for a mail server that serves the domain specified by *nameString*. `DNSGetMailServerNameFromDomainName` fills in the `resultDomainName` slot in a DNS results frame, as described in the section "The Domain Name Service Callback Function Format" beginning on page 1-50, and calls your callback function with the result code and that frame as parameters. The following is an example of a results frame for the `DNSGetMailServerNameFromDomainName` function:

```
{
type kDNSDomainNameType,
targetDomainName "newton.apple.com.",
resultDomainName "mail.newton.apple.com.",
}
```

The result code is `nil` if the function succeeded; otherwise, the result code is one of the error codes described in the section "DNS Error Codes" beginning on page 1-32.

If *nameString* ends with the period (`'.'`) character, the string is assumed to be in conformance with RFC 1123. If *nameString* does not end with the period character, `DNSGetMailServerNameFromDomainName` attempts to complete the name as follows:

- `DNSGetMailServerNameFromDomainName` first appends a period to a copy of *nameString* and attempts name resolution with that string.
- If that request is not successful, `DNSGetMailServerNameFromDomainName` appends the local domain name and a period to a copy of *nameString* and attempts name resolution with that string.

If there is more than one mail server for the domain specified by *nameString*, the results array contains multiple `resultDomainName` frames, one for each mail server. The frames are ordered in the results frame according to the preference order of the mail exchange resource record. This is the order in which a mail application should attempt to connect to the SMTP ports of these servers.

DNSGetNameFromAddress

`DNSGetNameFromAddress(`*address*`,` *clientContext*`,`*clientCallback*`)`

Translates an Internet address into its corresponding domain name.

address	An Internet IP address, specified as a NewtonScript array of four integer values. For example: `[155, 227, 54, 3].`
clientContext	The client context, as described in the section “The Client Context Parameter” on page 1-44.
clientCallback	The client callback function, as described in the section “The Client Callback Parameter” on page 1-44.

The `DNSGetAddressFromName` function finds the domain name string for the IP *address*. `DNSGetNameFromAddress` fills in the `resultDomainName` slot in a DNS results frame, as described in the section “The Domain Name Service Callback Function Format” beginning on page 1-50, and calls your callback function with the result code and that frame as parameters. The following is an example of a results frame for the `DNSGetAddressFromName` function:

```
{
type kDNSDomainNameType,
targetIPAddress [155,227,54,3],
resultDomainName "newton.apple.com.",
}
```

The result code is `nil` if the function succeeded; otherwise, the result code is one of the error codes described in the section “DNS Error Codes” beginning on page 1-32.

If there is more than one domain name for the address specified by *address*, the results array contains multiple results frames, each with a `resultDomainName` slots.

Newton Internet Enabler Options

This section describes the options that you can use to control Newton Internet Enabler.

Inet Tool Expedited Data Transfer ('iexp') Option

The Inet Tool expedited data transfer option is used for the expedited transmission of an unsigned data byte. You can use this option with an `Output` request to your endpoint to cause the data in that request to be sent immediately. You typically use this to send a break character or a similar indicator.

Here is an example of this option:

```
{
   label:      "iexp",
   type:       'option,
   opCode:     opSetRequired,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
      [
         15          // expeditedData byte
      ],
      typelist:
      [
         'struct,
         'byte
      ],
   },
}
```

The data slots in the expedited data transfer option frame are described in Table 1-6.

Table 1-6 Inet tool expedited data transfer option data slots

Option field	Description
`expeditedData`	The data byte that was received or is to be sent.

Inet Tool Physical Link Identifier ('ilid') Option

The Inet physical link identifier option is used to set or retrieve the physical link identifier.

Here is an example of using this option to set the physical link identifier:

```
{
   label:      "ilid",
   type:       'option,
   opCode:     opSetRequired,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
      [
         linkID
      ],
      typelist:
      [
         'struct,
         'ulong
      ],
   },
}
```

The data slots in the physical link identifier option frame are described in Table 1-7.

Table 1-7 Inet tool physical link identifier option data slots

Option field	Description
`linkID`	The link identifier.

Inet Tool Local Port ('ilpt') Option

The Inet Tool local port option is used to set or retrieve the Internet port number for a transport service. The rules shown in Table 1-9 apply to port number assignments.

Here is an example of this option to set the port number:

```
{
   label:      "ilpt",
   type:       'option,
   opCode:     opSetRequired,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
      [
         7,             // inetPortNumber
         nil,           // useDefault
      ],
      typelist:
      [
         'struct,
         'short,
         'boolean
```

```
        ],
    },
}
```

The data slots in the local port option frame are described in Table 1-8.

Table 1-8 Inet tool local port option data slots

Option field	Description
`inetPortNumber`	The reserved port number for this service. This value is used as described in Table 1-9
`useDefault`	A Boolean value that applies only to *connect* binds for the UDP transport service type. If `useDefault` is `true`, the default UDP port number is used.

Table 1-9 shows the use of the `inetPortNumber` slot, based on the service type and operation.

Table 1-9 Use of the port number by the Inet tool

Service type and operation	Description
Connect over TCP link	TCP picks this port; no need to set.

Table 1-9 Use of the port number by the Inet tool (continued)

Service type and operation	Description
Listen over TCP link	The port on which to listen. Specify `0` to indicate listening on all ports or use one of the port numbers as specified in *IEFT Assigned Numbers RFC*.
Connect over UDP link	The port to bind to locally. Specify `useDefault:true` to indicate that Newton Internet Enabler should choose the port number for you, in which case the assigned value will be returned in the option.
Listen over UDP link	The port on which to listen. Specify `0` to indicate listening on all ports or use one of the port numbers as specified in *IEFT Assigned Numbers RFC*.

Inet Profile ('iprf') Option

The Inet profile option is used to retrieve the the local and gateway IP addresses used by your endpoint.

Here is an example of this option to get the IP addresses:

```
{
   label:      "iprf",
   type:       'option,
   opCode:     opGetCurrent,
   data:
   {
      arglist:
      [
         [0,0,0,0],     // local host (Newton) IP address
         [0,0,0,0],     // gateway host IP address
      ],
      typelist:
      [
         'struct,
         ['array, 'byte, 4],
         ['array, 'byte, 4],
      ],
   },
}
```

Inet Tool TCP Remote Socket ('itrs') Option

The Inet Tool TCP remote socket option is used to set or retrieve the parameters of the remote host. If you are sending a `Connect` request over a TCP link, you must use this option to retrieve the remote socket address; if you are sending a `Listen` request over a TCP link, you can use this option to retrieve the address of the sender of the data.

Here is an example of using this option to set the TCP remote socket:

```
{
   label:      "itrs",
   type:       'option,
   opCode:     opSetRequired,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
      [
         [130,43,2,2],        // hostAddress
         7,                   // InetPortNumber
      ],
      typelist:
      [
         'struct,
         ['array, 'byte, 4],
         'short
      ]
   },
}
```

The data slots in the TCP remote socket option frame are described in Table 1-10.

Table 1-10 Inet tool TCP remote socket option data slots

Option field	Description
`hostAddress`	Internet address of remote host IP address, expressed as four single-byte values.
`InetPortNumber`	Reserved Internet port identifier.

Inet Tool Transport Service Type ('itsv') Option

The Inet Tool transport service type option is used to specify the transport service type associated with a link.

Here is an example of using this option to set the transport service type:

```
{
   label:      "itsv",
   type:       'option,
   opCode:     opSetRequired,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
      [
         kTCP           // transportService
      ],
      typelist:
      [
         'struct,
         'ulong
      ],
   },
}
```

The data slots in the link service type option frame are described in Table 1-11.

Table 1-11 Inet tool link service type option data slots

Option field	Description
`transportService`	The transport service type. Use one of the constants described in the section “Transport Service Type Constants” beginning on page 1-31.

Inet Tool UDP Destination Socket ('iuds') Option

The Inet Tool UDP destination socket option is used to set or retrieve the Internet destination host Internet socket address that is used for data transmission over a UDP link.

Here is an example of using this option to retrieve the current UDP destination address:

```
{
   label:      "iuds",
   type:       'option,
   opCode:     opSetRequired,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
```

```
   [
      [0,0,0,0]      // hostAddress
      0,             // InetPortNumber
   ],
      typelist:
      [
         'struct,
         ['array, 'byte, 4],
         'short
      ]
   }
}
```

The data slots in the UDP destination socket option frame are described in Table 1-12.

Table 1-12 Inet tool UDP destination socket option data slots

Option field	Description
`hostAddress`	The destination IP address, expressed as four single-byte values.
`InetPortNumber`	The reserved Internet port identifier.

Inet Tool UDP Source Socket ('iuss') Option

The Inet Tool UDP source socket option is used to retrieve the host Internet socket address that sent a datagram received by your application.

Here is an example of using this option to retrieve the UDP source socket:

```
{
   label:      "iuds",
   type:       'option,
   opCode:     opGetCurrent,
   result:     nil,
   form:       'template,
   data:
   {
      arglist:
      [
         [0,0,0,0],           // hostAddress
         port,                // InetPortNumber
      ],
      typelist:
      [
         'struct,
         ['array, 'byte, 4],
         'short
      ]
   },
}
```

The data slots in the UDP source socket option frame are described in Table 1-13.

Table 1-13 Inet tool UDP source socket option data slots

Option field	Description
`hostAddress`	The source IP address, expressed as four single-byte values.
`InetPortNumber`	The reserved Internet port identifier.

Newton Internet Enabler Exceptions

Any of the Newton Internet Enabler functions that receive a link identifier as a parameter can throw an exception if the link ID is not valid. The exception frame is as follows:

```
{
   type: |evt.ex.comm|,
   error: kInetErrNoSuchLinkID,
}
```

This exception is raised in response to two conditions:

- You passed in a link ID that does not exist.
- You passed in `nil` as the link ID and there are not any links defined.

Summary of Newton Internet Enabler

Link Controller Errors

```
constant kInetErrNoSuchLinkID                 := -60501;
constant kInetErrLinkDisconnected             := -60504;
constant kInetErrConnectLinkFailed            := -60505;
```

DNS Errors

```
constant kDNSErrNoAnswerFoundYet              := -60751;
constant kDNSErrInternalErr                   := -60752;
constant kDNSErrNameSyntaxErr                 := -60791;
constant kDNSErrNoNameServer                  := -60794;
constant kDNSErrAuthNameErr                   := -60795;
constant kDNSErrNoAnswerErr                   := -60796;
constant kDNSErrNonexistentDomain             := -60797;
constant kDNSErrOutOfMemory                   := -60798;
constant kDNSErrCouldNotContactServer         := -60800;
constant kDNSErrNoServersAvailable            := -60801;
constant kDNSErrRequestFormatErr              := -60802;
constant kDNSErrServerInternalErr             := -60803;
constant kDNSErrServerNotImplemented         := -60804;
constant kDNSErrServerRefused                := -60805;
constant kDNSErrUnknownServerErr             := -60806;
constant kDNSErrNoResponseFromServer         := -60814;
constant kDNSErrNoResponseFromAnyServer      := -60815;
```

Newton Internet Enabler Lower-Level Tool Errors

```
constant kInetToolErrBindFailed              := -60001;
constant kInetToolErrIPBindFailed            := -60002;
constant kInetToolErrPushModule              := -60004;
constant kInetToolErrIlink                   := -60005;
constant kInetToolErrNetActivateReq          := -60006;
constant kInetToolErrTCPBind                 := -60007;
constant kInetToolErrGetRequest              := -60008;
constant kInetToolErrPutRequest              := -60009;
constant kInetToolErrConnect                 := -60010;
constant kInetToolErrDlAttach                := -60011;
constant kInetToolErrBind                    := -60012;
constant kInetToolErrOpenLink                := -60013;
constant kInetToolErrUnlink                  := -60014;
constant kInetToolErrOutOfPhase              := -60015;
constant kInetToolErrAddRoute                := -60016;
constant kInetToolErrListen                  := -60017;
constant kInetToolErrLinkNotOpened           := -60018;
constant kInetToolErrDriverNotOpened         := -60019;
constant kInetToolErrStreamNotOpened         := -60020;
constant kInetToolErrBindReqFailed           := -60021;
constant kInetToolErrConnResReqFailed        := -60022;

constant kInetToolErrMemAlloc                := -60023;
constant kInetToolErrMsgType                 := -60024;
constant kInetToolErrNoDevice                := -60025;
constant kInetToolErrllegalOpenOnStream      := -60026;
constant kInetToolErrReqInInvalidState       := -60027;
constant kInetToolErrPrimitiveTooSmall       := -60028;
constant kInetToolErrPrimitiveOutOfRange     := -60029;
constant kInetToolErrPrimitiveOnInvalidStr   := -60030;
constant kInetToolErrMessageTooLong          := -60031;
constant kInetToolErrNetworkAlreadyActive    := -60032;
constant kInetToolErrNetworkNumberInvalid    := -60033;
constant kInetToolErrUnsupportedIoctl        := -60034;
constant kInetToolErrStreamAlreadyAttached   := -60035;
constant kInetToolErrUnknownMuxIndex         := -60036;
constant kInetToolErrNetworkIsInactive       := -60037;
constant kInetToolErrBogusConnection         := -60038;
constant kInetToolErrInvalidBillingMode      := -60039;
constant kInetToolErrNoTrigSelectedInAlarm   := -60040;
constant kInetToolErrInvalidTrigSize         := -60041;
constant kInetToolErrInvalidConnectionRef    := -60042;
constant kInetToolErrIlegalMdataInPrim       := -60043;
constant kInetToolErrMissingMdataInPrim      := -60044;
constant kInetToolErrInvalidSegmentedPrim    := -60045;
constant kInetToolErrInvalidNPIVersion       := -60046;
constant kInetToolErrInvalidAddress          := -60047;
constant kInetToolErrOutOfTCPPortNumbers     := -60048;
constant kInetToolErrSocketInUse             := -60049;
constant kInetToolErrReservedPortNumber      := -60050;
constant kInetToolErrExpDataNotSupported     := -60051;

constant kInetToolErrRedundentRequest        := -60052;
constant kInetToolErrUnexpectedDLPrim        := -60053;
constant kInetToolErrUnexpectedTPIPrim       := -60054;
constant kInetToolErrUnexpectedNPIPrim       := -60055;
constant kInetToolErrUnknownTPIErrorCode     := -60056;
constant kInetErrStreamInoperative          := -60057;
```

Link Controller Functions and Methods

linkId `InetAddNewLinkEntry(`*newLinkInfo*`);`
`InetCancelCurrentRequest(`*linkId*`);`
`InetCancelLink(`*linkId, clientContext, clientCallback*`);`
`InetDisconnectLink(`*linkId, clientContext, clientCallback*`);`
statusView `InetDisplayStatus(`*linkId, statusView, status*`);`
linksStatusFrame `InetGetAllLinksStatus();`
linkID `InetGetDefaultLinkID();`
linkEntry `InetGetLinkEntry(`*linkID*`);`
`InetGetLinkStatus(`*linkID*`);`
`InetGrabLink(`*linkId, clientContext, clientCallback*`);`
view `InetOpenConnectionSlip(`*linkId, clientContext, clientCallback*`);`
`InetReleaseLink(`*linkId, clientContext, clientCallback*`);`
`InetSetDefaultLinkID(`*linkId*`);`

DNS Functions and Methods

`DNSCancelRequests(`*clientContext, clientCallback*`)`
`DNSGetAddressFromName(`*nameString*`,` *clientContext*`,`*clientCallback*`);`
`DNSGetMailAddressFromName(`*nameString, clientContext, clientCallback*`);`
`DNSGetMailServerNameFromDomainName(`*nameString*`,`
clientContext`,`*clientCallback*`);`
`DNSGetNameFromAddress(`*address, clientContext, clientCallback*`);`

Exceptions

`|evt.ex.comm|`

Index