Messaging Enabler

version 1.2

Table of Contents

Figures and Tables

About This Document

This document describes how to use the Newton Messaging Enabler version 1.2, and how to write message modules. Message modules are plug-in modules which represent a particular messaging device to the Messaging Enabler.

Related Books

This book is one in a set of books available for Newton programmers. You'll also need to refer to these other books in the set:

- *Newton Programmer's Guide*. This book is the definitive guide to Newton programming, providing conceptual information and instructions for using the Newton application programming interfaces.
- *Newton Programmer's Reference*. This online book is the comprehensive reference to the Newton programming interface. It documents all routines, prototypes, data structures, constants, and error codes defined by the Newton system for use by NewtonScript developers.
- *Newton Toolkit User's Guide*. This book comes with the Newton Toolkit development environment. It introduces the Newton development environment and shows how to develop applications using Newton Toolkit. You should read this book first if you are a new Newton application developer.
- *The NewtonScript Programming Language*. This book comes with the Newton Toolkit development environment. It describes the NewtonScript programming language.

- *Newton Book Maker User's Guide*. This book comes with the Newton Toolkit development environment. It describes how to use Newton Book Maker and Newton Toolkit to make Newton digital books and to add online help to Newton applications.
- *Newton 2.0 User Interface Guidelines*. This book contains guidelines to help you design Newton applications that optimize the interaction between people and Newton devices.

Sample Code

There is sample code that deals specifically with the Messaging Enabler.

The Newton Toolkit development environment, from Apple Computer, includes many sample code projects. You can examine these samples, learn from them, and experiment with them. These sample code projects illustrate most of the topics covered in this book. They are an invaluable resource for understanding the topics discussed in this book and for making your journey into the world of Newton programming an easier one.

The Newton Developer Technical Support team continually revises the existing samples and creates new sample code. The latest sample code is included each quarter on the Newton Developer CD, which is distributed to all Newton Developer Program members and to subscribers of the Newton monthly mailing. Sample code is updated on the Newton Development side on the World Wide Web (`http://devworld.apple.com/dev/newtondev.shtml`) shortly after it is released on the Newton Developer CD. For information about how to contact Apple Computer regarding the Newton Developer Program, see the section "Developer Products and Support," on page xi.

The code samples in this book show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code samples have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application.

To make the code samples in this book more readable, only limited error handling is shown. You need to develop your own techniques for detecting and handling errors.

Conventions Used in This Book

This book uses the following conventions to present various kinds of information.

Special Fonts

This book uses the following special fonts:

- **Boldface**. Key terms and concepts appear in boldface on first use.
- `Courier typeface`. Code listings, code snippets, and special identifiers in the text such as predefined system frame names, slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text. If you are programming, items that appear in Courier should be typed exactly as shown.
- *Italic typeface*. Italic typeface is used in code to indicate replaceable items, such as the names of function parameters, which you must replace with your own names. The names of other books are also shown in italic type, and *rarely*, this style is used for emphasis.

Tap Versus Click

Throughout the Newton software system and in this book, the word “click” sometimes appears as part of the name of a method or variable, as in `ViewClickScript` or `ButtonClickScript`.

This may lead you to believe that the text refers to mouse clicks. It does not. Wherever you see the word "click" used this way, it refers to a tap of the pen on the Newton screen (which is somewhat similar to the click of a mouse on a desktop computer).

Frame Code

If you are using the Newton Toolkit (NTK) development environment in conjunction with this book, you may notice that this book displays the code for a frame (such as a view) differently than NTK does.

In NTK, you can see the code for only a single frame slot at a time. In this book, the code for a frame is presented all at once, so you can see all of the slots in the frame, like this:

```
{  viewClass: clView,
   viewBounds: RelBounds( 20, 50, 94, 142 ),
   viewFlags: vNoFlags,
   viewFormat: vfFillWhite+vfFrameBlack+vfPen(1),
   viewJustify: vjCenterH,

   ViewSetupDoneScript: func()
      :UpdateDisplay(),

   UpdateDisplay: func()
      SetValue(display, 'text, value);
   };
```

If while working in NTK, you want to create a frame that you see in the book, follow these steps:

1. On the NTK template palette, find the view class or proto shown in the book. Draw out a view using that template. If the frame shown in the book contains a `_proto` slot, use the corresponding proto from the NTK template palette. If the frame shown in the book contains a `viewClass` slot instead of

a `_proto` slot, use the corresponding view class from the NTK template palette.

2. Edit the `viewBounds` slot to match the values shown in the book.
3. Add each of the other slots you see listed in the frame, setting their values to the values shown in the book. Slots that have values are attribute slots, and those that contain functions are method slots.

Developer Products and Support

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order product or to request a complimentary copy of the *Apple Developer Catalog* contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com
World Wide Web	http://www.devcatalog.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For Newton-specific information, see the Newton developer World Wide Web page at:

`http://devworld.apple.com/dev/newtondev.shtml`

Undocumented System Software Objects

When browsing in the NTK Inspector window, you may see functions, methods, and data objects that are not documented in this book. Undocumented functions, methods, and data objects are not supported, nor are they guaranteed to work in future Newton devices. Using them may produce undesirable effects on current and future Newton devices.

C H A P T E R 1

Messaging Enabler Interface

This chapter describes the Messaging Enabler verion 1.2 interface. The Messaging Enabler, along with one or more message modules, provides transport-level communication between an application and a messaging device. This makes sending a page as easy as printing or faxing.

This chapter describes how to

- Create a message module to interface between the Messaging Enabler and a particular messaging device.
- Customize the user interface to your message module.
- Install and remove a message module.
- Interact with the Messaging Enabler from an installed application.

This text assumes a basic familiarity with programming for the Newton. In particular, you should be familiar with the NewtonScript programming language, building and downloading projects with the Newton Toolkit, and Chapter 22, "Transport Interface," of *Newton Programmer's Guide*. It is also recommended that you understand the information in Chapter 3, "Views," Chapter 21, "Routing Interface," and Chapter 23, "Endpoint Interface," of *Newton Programmer's Guide*.

About the Messaging Enabler

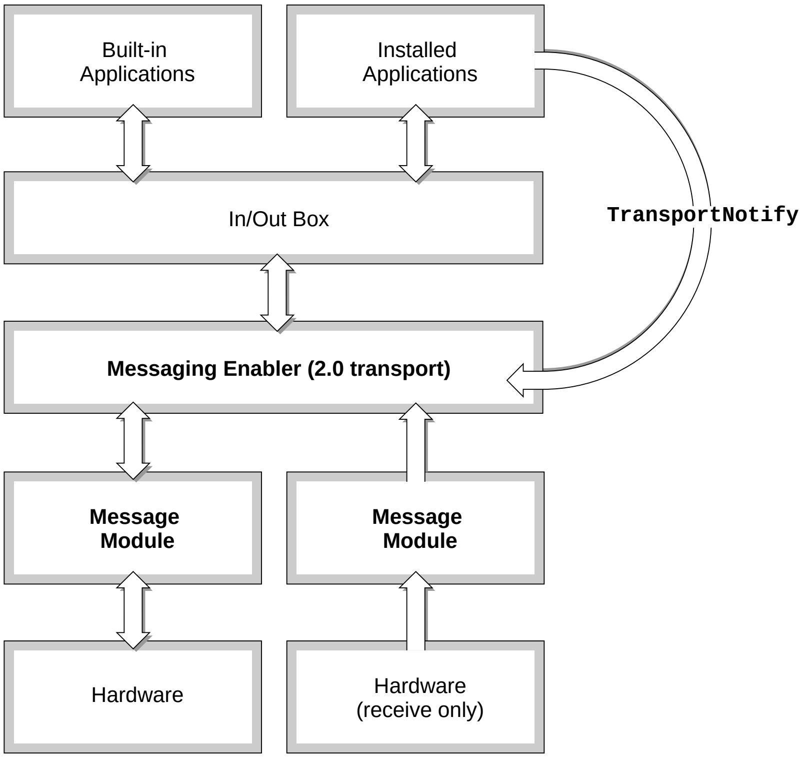
The Messaging Enabler is intended to provide high-level support for a wide variety of **messaging devices**. Messaging devices are both one-way and two-way pagers, or more generally, any device that can receive (and maybe send) short, usually less than 5 KB, wireless messages. The Messaging Enabler provides much of the functionality of a transport that would be necessary to support these devices. The particular routines needed to support a messaging device are implemented in a plug-in **message module** (msgModule) created from `protoMsgModule`. The msgModule comes between the Messaging Enabler and the **message channel**. The message channel is considered to be everything from the Messaging Enabler to the outside world. That is, the message channel starts at the msgModule, continues through the messaging device, and to the outside world from there.

The Messaging Enabler provides the user interface, and handles the interactions with installed applications, by way of the In/Out Box. The msgModule creator can thus concentrate on writing communications code. Another advantage of this setup is that the user sees a consistent interface with any type of messaging device.

The Messaging Enabler and msgModules

A msgModule’s only contact with the system is through the Messaging Enabler. In turn, the Messaging Enabler’s only contact with the system is through the In/Out Box; this is true of all transports in the Newton operating system. From the other end, the user’s interactions with the message channel are similarly limited to the In/Out Box. The user creates a message in some application that supports routing, and the message is routed to the In/Out Box. The Messaging Enabler takes over from here. This arrangement is schematized in Figure 1-1.

Figure 1-1 Messaging Enabler hierarchy



The Messaging Enabler and Applications

Applications may interact with the Messaging Enabler by using the global function `TransportNotify`. An application may use this function to change a configuration setting, and optionally to disable user preferences. This is intended for vertical applications to properly configure the Messaging Enabler to known parameters.

An application may also perform msgModule commands, if the msgModule supports this. This functionality is up to the msgModule designer to provide.

The Messaging Enabler simply passes messages from the application to the msgModule. This can be useful if an application is being distributed with the msgModule.

Functionality of a msgModule

The Messaging Enabler is intended to work with a wide variety of messaging devices. These devices have different capabilities, and the Messaging Enabler has been designed to take advantage of the capabilities of particular messaging devices, while still supporting messaging devices that provide only a minimal functionality.

All msgModules must implement the following minimal functionality to work with the Messaging Enabler:

- Describe its functionality to the Messaging Enabler; this is done by setting certain Boolean slots.
- Initiate and terminate a connection with the message channel.
- Receive and delete messages from the message channel.

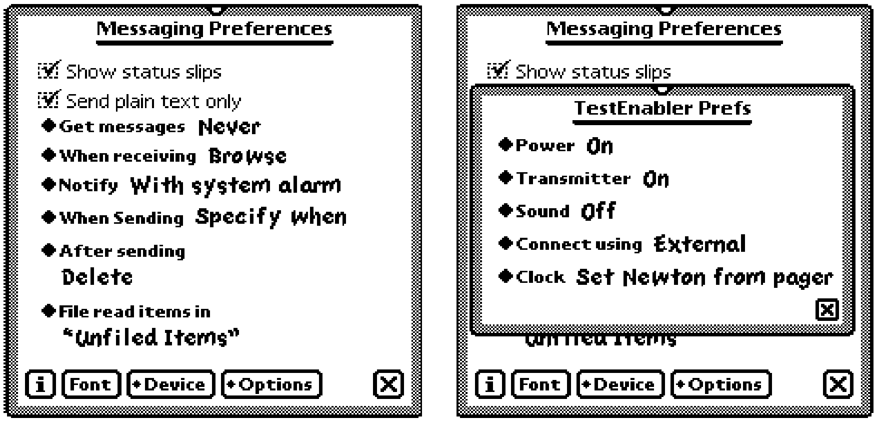
A msgModule may also provide the following functionality:

- Send messages.
- Retrieve a directory of incoming messages.
- Detect the presence of either the messaging device or a new message.
- Provide parameters to the built-in configuration options.
- Provide custom preferences.
- Provide a custom routing slip.
- Customize the About slip.
- Accept commands from an application

Preference Slips

The Messaging Enabler provides a preference slip with choices which are not specific to a particular msgModule, such as when to poll for messages, whether to display status slips, and other such preferences. The Messaging Enabler's preference slip, shown to the left in Figure 1-2, is accessed through the information ("i") button of the In/Out Box. The Messaging Enabler also provides a slip with common hardware configuration options for a particular msgModule and messaging device. The hardware preference slip, shown to the right in Figure 1-2, is accessed through the Device button in the Messaging Enabler's preference slip. The Device button provides a picker for all installed msgModules.

Figure 1-2 Messaging Enabler and a msgModule hardware preference slips


Messaging Preferences

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

TestEnabler Prefs

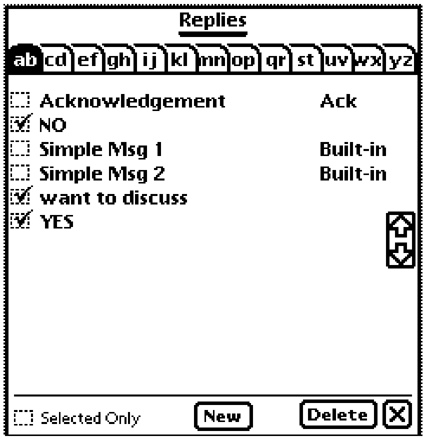
◆Power On
◆Transmitter On
◆Sound Off
◆Connect using External
◆Clock Set Newton from pager

You may also create a preference slip of options which are specific to your msgModule. This slip is accessed through the Options button in the Messaging Enabler's preference slip. If no installed msgModules have a custom slip, the Options button is not included.

Replies

The Messaging Enabler allows the user both to reply to received messages, and to attach possible replies to outgoing messages, if the msgModule supports this. The user replies to an incoming message by choosing "Reply" from the Tag button in the In/Out Box. Replies to outgoing messages are created from the Replies button of the routing slip. Tapping this button brings up the replies slip shown in Figure 1-3. These replies are stored in a soup, which your msgModule may manipulate.

Figure 1-3 The replies slip



☐	Acknowledgement	Ack
✓	NO	
☐	Simple Msg 1	Built-in
☐	Simple Msg 2	Built-in
✓	want to discuss	
✓	YES	

Multipart Messages

Some messaging devices allow multipart messages in order to overcome message length restrictions. The Messaging Enabler provides the ability to combine these message segments. The msgModule can simply retrieve the individual message parts. The message is integrated by the Messaging Enabler, and the user sees a single message in the In Box. If only part of the multipart message is available at any time, individual message segments are displayed prefixed with `[a:b]`, where *a* is the segment number, and *b* is the total number of segments.

Compatibility

The Messaging Enabler is a Newton 2.x transport. It is incompatible with Newton system software 1.x versions. Messaging Enabler 1.0 does not send a `ProcessCombinedMessage` method, this functionality is introduced in version 1.2.

Using the Messaging Enabler

This section describes how to

- create a msgModule
- implement the msgModule methods
- receive messages
- send messages
- use name references
- control the Reply soup
- provide preferences
- customize the About slip
- install and remove a msgModule
- alter both the Messaging Enabler and a msgModule's configuration settings and send a msgModule a command from an application

Creating a msgModule

A msgModule object is a frame based on the `protoMsgModule` proto. To create a msgModule, create a layout in NTK containing a single `protoTextButton` and create an `AfterScript` with the following line:

```
thisView._proto := protoMsgModule
```

Note
The `protoMsgModule` frame is defined in the file "pMsgMod.stream" file. ◆

The msgModule must set a number of slots to inform the Messaging Enabler of its functionality. The msgModule slots of interest can be grouped as follows:

- Convenience and miscellaneous slots:

`deviceTitle`	The user-visible name of the messaging device.
`title`	The user-visible name of the msgModule.
`version`	The version number.
`declareSelf`	A reference to your msgModule.
`pfVars`	A convenience frame for your data.
`icon`	An icon identifying the msgModule.
`about`	What to display in the About slip; see "Customizing the About Slip" (page 1-25).

- Slots related to user preferences; these are described in "Providing Preferences" (page 1-22):

`prefsTemplate`	A view template for a custom preference slip.
`timeStrings`	The choices available for the "Clock" picker.
`transmitterStrings`	The choices available for the "Transmitter" picker.
`soundStrings`	The choices available for the "Sound" picker.
`portStrings`	The choices available for the "Connect Using" picker.
`powerStrings`	The choices available for the "Power" picker.

- Boolean slots which inform the Messaging Enabler of the message channel's functionality:

`dirSupport`	Does the message channel support returning a directory of available messages? If so, you must supply a `GetDir` method (page 2-30).
`cannedSupport`	Does the messaging channel support returning canned replies?
`persistConnection`	Should the message channel be left open after receiving messages?

- Slot which describes the messaging channel's sending ability:

`sendOptions`	The addressing options that the message channel supports; see "Sending Messages" (page 1-17).

- Slots which determine if and when the Messaging Enabler should poll the message channel:

`pollMessage`	How often to send either a `GetDir` (page 2-30) or a `GetNextMessage` (page 2-32) message, depending on the value of `dirSupport`.
`pollHardware`	How often to send an `Open` message (page 2-33).
`pollAlive`	How often to send an `Alive` message (page 2-16).

For a full description of these slots see "protoMsgModule" beginning on page 2-9.

Implementing msgModule Methods

Since the msgModule methods are called asynchronously, they must be structured in a slightly idiosyncratic manner. The Messaging Enabler cannot use the return values of the msgModule methods, therefore these methods must report their outcome with a function call. The function they must call is the proto-supplied msgModule method `DoCallBack` (page 2-21).

Furthermore, if the msgModule needs to communicate with the Messaging Enabler, it must do so by sending it events. This is done by calling the proto-supplied msgModule method `DoEvent` (page 2-22).

Note that the Messaging Enabler wraps all calls to msgModule methods in a `try`...`onexception` block. If a msgModule method throws an exception the Messaging Enabler will catch the exception and close the msgModule.

Using Callback Functions

All the msgModule methods that are called asynchronously must call the proto-supplied msgModule method `DoCallBack` (page 2-21) after they have completed their operation. These methods are passed a *callBack* parameter, which must be passed on to the `DoCallBack` method. You should not alter the value of this parameter, nor can you rely on its value.

In addition to *callBack*, the `DoCallBack` function accepts two other parameters: *complCode*, which indicates the success of the function; and *complFrame*, which is used to return data. Result codes to use for the *complCode* parameter are discussed in "Result Codes" beginning on page 1-11. The values for the *complFrame* parameter depend on the particular method. These values are described under the particular method in Chapter 2, "Messaging Enabler Interface Reference."

The following sample code demonstrates how these methods should be structured:

```
msgModule.MethodName := func (callBack , ...)
begin
   // In this sample, the DoTheWork method performs
   // whatever is required of MethodName, and returns an
   // integer result code. Normally this work would be
   // performed in the body of MethodName.

   local theResult := :DoTheWork();

   :DoCallBack(callBack, theResult, nil);
end;
```

Result Codes

Most msgModule methods must return an integer result code indicating the success of the operation. This result code is returned through the *complCode* parameter to the `DoCallBack` method. It is used both to indicate whether the operation was successful and to map to error strings if necessary.

A number of result codes are built in to the Messaging Enabler. These are listed in Table 2-1 "Built-in error codes" on page 2-2. Table 2-1 also lists the string used in alerts corresponding to each error. However, before displaying one of these strings, the Messaging Enabler calls your msgModule's `TranslateError` method (page 2-38), if it has one. You may return a string from this method to use in the alert, instead of the Messaging Enabler supplied string.

You may also define your own error codes. The built-in error codes are all negative integers. You may define both negative and positive result codes. The negative codes are treated just like the built-in error codes; these should be integers below the constant `kRES_MMBASE`. Positive result codes indicate to the Messaging Enabler a noncritical event. In this case, the Messaging Enabler displays an alert, with the string your `TranslateError` method returned, and continues.

Some of the method descriptions in Chapter 2, "Messaging Enabler Interface Reference," explicitly mention specific result codes. You should use these codes if they apply, as the Messaging Enabler treats this code in a particular manner. For example, the result code `kRES_PARTIAL`, when returned from the `GetDir` method (page 2-30) indicates that there are more messages, and `GetDir` needs to be called again.

Otherwise, if no result code is mentioned explicitly in the method description, you should return the code that best describes the state of the operation. The method description list a number of suggested error codes for each particular method. These are only suggestions, you should return the code that best describes the status of the operation. When returning error codes, you should consider them from the point of view of what string the user sees.

Sending Events

Aside from the call to `DoCallBack` within a msgModule method, the only way for a msgModule to communicate with the Messaging Enabler is by sending events. For example, if the msgModule detects that there is a new message on the message channel, it should notify the Messaging Enabler rather than wait to be polled.

The events your msgModule can send are summarized in Table 1-1.

Table 1-1 Messaging Enabler events

Constant for event name	Event description
`kEV_BATTERY`	The battery level of the messaging device has changed.
`kEV_CLOSED`	The messaging channel is no longer available.
`kEV_DETECTED`	The messaging channel is available.
`kEV_MEMORY`	The messaging device is either out of memory, or no longer out of memory.
`kEV_MESSAGE`	There is a change in the messages available.
`kEV_PROGRESS`	There has been progress on a current operation. This allows progress slips to be updated.
`kEV_SERVICE`	The message channel is temporarily unavailable, or no longer so.

Events are sent to the Messaging Enabler by calling the proto-supplied msgModule `DoEvent` method (page 2-22). For further information about these events and how the Messaging Enabler reacts to each of them, see the description of the `DoEvent` method.

msgModules and Endpoints

In most cases, the connection to the messaging device is created with an endpoint. For instance you could instantiate an endpoint in your msgModule's `Open` method (page 2-33), and close, unbind, and dispose of it in your `Close` method (page 2-18). However, it is not necessary that your msgModule use endpoints.

For information on endpoints, see Chapter 23, "Endpoint Interface," in *Newton Programmer's Guide*.

Receiving Messages

There are three ways messages can be received through the message channel. If the msgModule `pollMessage` slot is non-`nil`, the Messaging Enabler periodically sends the msgModule either a `GetDir` (page 2-30) or a `GetNextMessage` (page 2-32) message, depending on the value of the msgModule `dirSupport` slot.

If the message channel is open and the msgModule detects the arrival of a message, it can send a `kEV_MESSAGE` event; see "Sending Events" (page 1-12). This causes the Messaging Enabler to send either a `GetDir` or a `GetNextMessage` message.

The user may also poll for messages at any time via the Receive button in the In Box.

Messages are represented as In/Out Box item frames. The section "Item Frame" beginning on page 2-4 describes the slots in an item frame that are used particularly by the Messaging Enabler and your msgModule. For information about the standard set of slots used by all In/Out Box item frames, see Chapter 18, "Routing Interface Reference," in *Newton Programmer's Reference*.

For information on how an application may retrieve items from the In Box, see Chapter 22, "Transport Interface," in *Newton Programmer's Guide*. In particular, you should read about the `RegInBoxApps`, `RegAppClasses`, `PutAway`, and `AutoPutAway` functions. Applications receiving Message Enabler items from the In Box should check whether the item has a `segment` slot, as this indicates that it is not a complete message.

Sample Message Receiving Method

The following sample code shows how a number of item frame slots should be set:

```
// This method is only necessary if we do not have directory support
msgModule.GetNextMessage := func( callBack )
begin
   local message := nil; //NOTE: if we DON'T return a nil message,
                        //this function will be called again

   //For the sake of this example, we assume that the msgModule
   //has a GetMessageFromPager function returns the message.
   //
   //Note that GetMessageFromPager MUST remove the message
   //from the message channel.
   local rawMessage := nil;

   try
      rawMessage := :GetMessageFromPager();
   onexception |evt.ex.outOfMem| do
   begin
      :DoCallBack (callBack,kRES_MEMORY,nil);
      return;
   end;
   onexception |evt.ex| do
   begin
      :DoCallBack (callBack,kRES_FAILED,nil);
      return;
   end;
```

```
if rawMessage then
begin
   message := {
               complete: true,
               body :   {
                           class: 'text,
                           text: rawMessage.text
                        },
               title :  rawMessage.title
            };

   // create a fromRef frame with the appropriate information
   local theEmailString := Clone (rawMessage.email);
   SetClass ( theEmailString, '|string.email.internet|);
   local fromRefTemplate := {
                           name: {
                                 first: rawMessage.firstName,
                                 last: rawMessage.lastName
                                 },
                           email: theEmailString,
                           country : nil,
                        };
   local addressType := GetDataDefs('|nameRef.email|);
   message.fromRef := addressType: MakeNameRef( fromRefTemplate,
                                                '|nameRef.email|
                                               );

   //Set the ID. Here, msgNum is slot in base view. We set
   //msgNum to 0 when the msgModule is installed.
   message.ID := msgNum := msgNum +1;
```

```
        //If this message is intended for a specific application, we
        //may want to set some extra slots for that application
        message.appSymbol := 'theAppsSymbol;
        message.appSpecificSlot := nil;

    end; // if rawMessage

    :DoCallBack (callBack, kRES_SUCCESS, message);
end
```

Receiving Multipart Messages

The Messaging Enabler provides the ability to combine multi-part messages. The msgModule needs only set the `segment`, `totalSegments`, and `groupID` slots to the appropriate numbers in the item frame. If all parts are present, the Messaging Enabler will combine these and the user will see a single message in the In Box. If all message segments are not present, the individual segments are prefixed with `[`*segment*`+1:`*totalSegments*`]`.

If using Message Enabler version 1.2 or later, it conditionally sends your msgModule a `ProcessCombinedMessage` (page 2-38) message when all parts are in, but before posting the message to the In box. You can do any porcessing you wish on the combined message here.

Text and Frame Messages

The actual message data is stored in the `body` slot of the item frame. When a text message is received, the `body` slot should be set to a frame with a `class` slot of `'text` and a `text` slot which contains the message string. Otherwise, the format of this frame depends on what the application which is managing this data expects.

If the message contains a frame that has been flattened, you should unflatten it using the `Translate` global function, described in Chapter 23, "Endpoint Interface,"in *Newton Programmer's Reference*. However, if the message contains only part of the flattened frame, you can treat it as a multipart message. The Messaging Enabler will collect the individual parts, call `Translate`, and set the `body` slot to the resulting frame. If returning part of a flattened frame the `body` slot should be a frame with a `class` slot set to `'|file.frameSegment|` and a `data` slot containing the binary data.

Sending Messages

The user originates a message, just like any other transport-related operation, by tapping the Action button (the envelope button). This routes the message to the Out Box. The Messaging Enabler then passes this message on by calling your msgModule's `SendMessage` method (page 2-35). The `SendMessage` method is passed a *message* parameter which consists of an In/Out Box item frame.

To support sending messages, the msgModule must have a non-`nil` `sendOptions` slot and must implement the `SendMessage` method. For a description of the slots in an item frame, see "Item Frame" (page 2-4).

The `sendOptions` slot contains an array of **send options frames**. A send option frame specifies the following:

- The routing slip to use; either a custom template, or a name reference describing the type of addressing to use in the Messaging Enabler-supplied routing slip.
- The routing group this msgModule belongs to; predefined groups are `'print`, `'mail`, `'fax`, and `'page`. You may also define your own routing group, but these must be registered with Newton Developer Technical Support.
- The title and icon shown in the Action button; these should correspond to the routing group.
- The type of data that can be sent; sending text and frame data is currently supported.
- The types of reply frames that can be attached to a message; see "Reply Frames" (page 2-7).

For a full description of send option frames, see the description of the `sendOptions` slot in "protoMsgModule" beginning on page 2-9.

It is up to your msgModule's `SendMessage` (page 2-35) method to detect if the size of an outgoing message is larger than the messaging channel can handle. If so, it must partition the message and send the individual segments. This is true regardless of whether the message contains text or a frame.

Using Name References with the Messaging Enabler

A name reference is a reference to a Names soup entry, or a frame created on the fly that has some of the slots of a Names soup entry. Name references are used by the Messaging Enabler to specify the originator and receiver of a message (these are stored in the message's `toRef` and `fromRef` slots) and the type of routing slip to display for outgoing messages.

You need to create a name reference only when setting the `fromRef` slot of an incoming messages. Outgoing messages will have the `toRef` and `fromRef` slots set by the Messaging Enabler, and incoming messages do not need a `toRef` slot. If your msgModule supports sending messages, you will also need to extract information from the `toRef` slot to use in your `SendMessage` method (page 2-35).

Creating the fromRef Slot of an Incoming Message

To create a name reference, call the `MakeNameRef` method of the data definition. The call to `MakeNameRef` requires either a Names soup entry, an alias to one, or a frame with some of the slots of a Names soup entry. The Names soup format is described in Chapter 16, "Built-in Applications and System Data Reference," in *Newton Programmer's Reference*.

You should call `MakeNameRef` with a frame created on the fly containing only a subset of a Names soup entry's slots. The slots to include depend on the type of name reference, as summarized in Table 1-2. If the value of any of these slots is unavailable to you, you may either not include this slot or set it to `nil`.

Table 1-2 Slots to include in name references

Name reference	Slots to include
`'|nameRef.people|`	`name`
`'|nameRef.phone|`	`name`, `phone`, `country`
`'|nameRef.fax|`	`name`, `phone`, `country`
`'|nameRef.email|`	`name`, `email`, `country`
`'|nameRef.people.pager|`	`name`, `pagers`

The following sample code sets the `fromRef` slot of an incoming message:

```
msgModule.GetNextMessage := func (callBack)
begin
   ...
   local item := {};
   local myData := {};

   myData.name := {
                    first :sendersFirstName,
                    last  :sendersLastName
                 };
   myData.email := SetClass(
                             sendersEmailAdress,
                             '|string.email.internet|
                          );

   item.fromRef := GetDataDefs('|nameRef.email|)
               :MakeNameRef( myData,'|nameRef.email|);
   ...

   :DoCallBack(callBack, kRES_SUCCESS, item);
end
```

To set the class of a pager, use either one of the predefined class symbols, or define your own class symbol using the global function `RegPagerType`. The following are the predefined class symbols:

```
'|string.pager|
'|string.pager.skytel|
'|string.pager.mobilecomm|
'|string.pager.embarc|
```

For information about the `RegPagerType` function, and its companion `UnRegPagerType`, see Chapter 26, "Utility Functions," in *Newton Programmer's Reference*.

Obtaining Information from the toRef Slot of an Outgoing Message

When sending a message, you are passed an item frame which contains a `toRef` slot. This slot holds an array of name references for the recipient(s) of the message. To obtain information from these name references to use when sending the message out the message channel, call the name reference's `GetRoutingInfo` method. `GetRoutingInfo` returns an array of routing information frames. Routing information frames contain a `name` slot and certain slots with routing information, depending on which name reference's `GetRoutingInfo` method is called.

The following code retrieves the `pager` slot of an array of name references:

```
msgModule.SendMessage := func (callBack, message)
begin
   ...
   dataDef := GetDataDefs ('|nameRef.people.pager|);
   pagersArray := dataDef:GetRoutingInfo(message.toRef);
   ...
end;
```

The variable `pagersArray` will contain an array of frames with the following format:

```
{
   pagerNum: pagerNumberString,
   pagerPin: pagerPINString
}
```

Note that the string *pagerNumberString* will have the same class as the pager, that is, `'|string.pager|` or a subclass of it. Pager classes are discussed in "Creating the fromRef Slot of an Incoming Message" (page 1-18).

To retrieve the `email` slot use code such as:

```
msgModule.SendMessage := func (callBack, message)
begin
   ...
   dataDef := GetDataDefs ('|nameRef.email|);
   emailsArray := dataDef:GetRoutingInfo(message.toRef);
   ...
end;
```

The Reply Soup

Replies are stored in the Reply soup. A list of stored replies is shown to the user when she taps the Replies button in the Routing slip. Entries in this soup are reply frames; the format of these frames is described in "Reply Frames" (page 2-7). The soup's name is `kMsgEnablerReplySoupName`.

A msgModule may add or delete entries in this soup. It is important that any changes are made with functions that transmit the change (these functions end in "Xmit"). The msgModule `InstallScript` method (page 2-33) is a handy place to make these changes. For more information on manipulating soups, see Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*.

This soup contains canned replies provided by the msgModule, user-created replies, and the acknowledgment reply.

The list of canned messages is retrieved by the Messaging Enabler by sending the msgModule a `GetDir` message (page 2-30). This is usually done once, when the msgModule is installed. The Messaging Enabler retrieves the list of canned messages whenever a msgModule is opened with a new serial number; if it has no serial number the version string is used. The serial number is obtained by sending the msgModule a `GetConfig` message (page 2-26); the version string is in the msgModule `version` slot. The msgModule may also request that the Messaging Enabler update the list of canned messages by sending a `kEV_MESSAGE` event; see "Sending Events" beginning on page 1-12.

The list of user replies is maintained by the user, through the New and Delete buttons of the Reply slip.

Note

If the user attempts to add a reply to a message and the Reply soup is empty, the Message Enabler populates the soup with the acknowledgment reply and user replies. These user reply frames have the following strings in their `text` slot:

`"Yes/OK"`, `"No"`, `"Will call later"`, `"Call me"`, `"On my way"`, `"Running late"`, `"Need more info"`, `"Send # to call"`, `"Call home"`, `"Where are you"`, `"Will arrive 15m"`, `"Will arrive 30m"`, `"Traffic delay"`, `"Pick me up"`, `"Busy"`, and `"Finished"`. ◆

Providing Preferences

The Messaging Enabler supplies a preference slip, accessed through the information ("i") button in the In/Out Box. This slip provides a Device picker which brings up the hardware preference slip. You may also supply a custom preference slip. If any installed msgModules supply a custom preference slip, the Messaging Enabler's preference slip will include an Options button to allow access to these slips.

Providing Hardware Preferences

The hardware preference slip consists of up to five pickers for the typical hardware configuration options that may be set in a messaging device. The labels of these pickers are set by the Messaging Enabler, but it is up to the msgModule to provide the menu choices (`labelCommands`) for each picker. Table 1-3 lists the msgModule slots that specify the menu choices for these pickers.

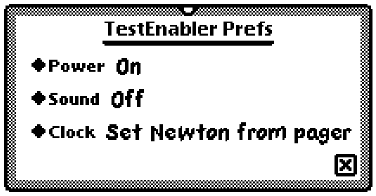
Table 1-3 msgModule slots controlling hardware preference

msgModule slot	**Picker label**
`timeStrings`	"Clock"
`transmitterStrings`	"Transmitter"
`soundStrings`	"Sound"
`portStrings`	"Connect Using"
`powerStrings`	"Power"

Each one of these msgModule slots should contain either an array of strings or `nil`. If the slots holds the value `nil`, or is not present, then the corresponding picker is not included in the preference slip. For example, the following msgModule creates the preference slip shown in Figure 1-4 (page 1-24):

```
myMsgModule := {
   _proto : protoMsgModule,
   ...
   timeStrings := ["Set Newton from pager",
                   "Set pager from Newton" ],
   soundStrings := ["Off", "Quiet", "Loud"],
   powerStrings := ["On", "Off"],
   ...
}
```

Figure 1-4 msgModule hardware preference slip



TestEnabler Prefs

◆Power On

◆Sound Off

◆Clock Set Newton from pager

IMPORTANT

The first element in each of these arrays is the default for the picker. Pick the first element with forethought. ▲

The Messaging Enabler calls your msgModule's `SetConfig` method (page 2-36) when one of these settings is changed if the msgModule is open. Otherwise, this new value is simply used next time the msgModule is opened.

Providing a Custom Preference Slip

You may supply a template for a custom preference slip by including a `prefsTemplate` slot in your msgModule. If this slot exists, the Messaging Enabler's preference slip includes an Options button. This button provides access to the slip in the `prefsTemplate` slot.

You must ensure that this slip is rotatable, for consistency with the Messaging Enabler preference slips. To accomplish this, you must supply a `ReorientToScreen` method in your view template. You may set the `ReorientToScreen` slot to the ROM constant `ROM_DefRotateFunc`. For more information on this, see the description of the `ReorientToScreen` method in Chapter 2, "Views Reference," of *Newton Programmer's Reference*.

The proto-supplied msgModule methods `SavePreference` (page 2-35) and `GetPreference` (page 2-33) can be useful when writing the methods of this template. These methods are described in "Message Module Methods for Storing Data" (page 1-25).

Message Module Methods for Storing Data

The proto-supplied msgModule methods `GetPreference` (page 2-33) and `SavePreference` (page 2-35) are designed to save you the effort of writing soup manipulation code. You may store any information by calling the `SavePreference` method. This method takes two parameters, a symbol for the variable name and the actual data. This data is stored in a soup. The following sample code stores an array associated with the symbol `'encryptionScheme`:

```
//Note, 'msgModule is the value in the declareSelf slot
msgModule:SavePreference(
                           'encryptionScheme,
                           ['orange,'red]
                        );
```

The `GetPreference` method is used to retrieve data stored with `SavePreference`, as in the following code sample:

```
encryption := msgModule:GetPreference('encryptionScheme);
```

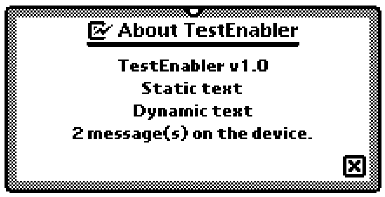
The `SavePreference` method should also be used to alter the value of any symbol previously stored with `SavePreference`.

The msgModule slot `pfVars` is also provided to aid in data storage. Any compile time data stored in this slot is kept in the package, and values which have been changed at run time are stored in the heap. See the description of this slot in “protoMsgModule” (page 2-9).

Customizing the About Slip

The msgModule `about` slot is used to provide information for the About slip, shown in Figure 1-5. The About slip is accessed through the information (“i”) button in the Messaging Enabler’s preference slip. This slip contains both static and dynamic information.

Figure 1-5 msgModule About slip



About TestEnabler

TestEnabler v1.0
Static text
Dynamic text
2 message(s) on the device.

The `about` slot can contain a string, a view template, or the value `nil`. If a view template is provided, this view replaces the Messaging Enabler-supplied About slip. This view should look like the built-in About slips, and should be rotatable. To make the view rotatable, you must supply a `ReorientToScreen` method in your view template. You may set the `ReorientToScreen` slot to the ROM constant `ROM_DefRotateFunc`. For more information on this, see the description of the `ReorientToScreen` method in Chapter 2, "Views Reference," of *Newton Programmer's Reference*.

If this slot contains a string or the value `nil`, the About slip will contain the following objects:

- The strings in the msgModule slots `title` and `version`.
- The string in the `about` slot, if provided (in Figure 1-5 this is the string `"Static text"`). If the `about` slot holds the value `nil`, the other information is still displayed.
- A string returned by the msgModule `GetConfig` method (page 2-26), if defined (in Figure 1-5 this is the string `"Dynamic text"`).
- An integer message count returned by `GetConfig`, if defined (in Figure 1-5 this is the number 2).

Installing and Removing a msgModule

A msgModule is installed using the `RegMsgModule` function (page 2-39). It should be called from your part's `InstallScript`; for information on auto part's `InstallScript`, see Chapter 2, "Getting Started," in *Newton*

Programmer's Guide. The `RegMsgModule` function takes two parameters, a msgModule and a symbol which is considered your msgModule's appSymbol. The following code registers a msgModule in a file named "myMessageModule":

```
SetPartFrameSlot( 'msgModuleTemplate,
                  GetLayout("myMessageModule")
                );

InstallScript := func( partFrame, removeFrame )
begin
   call kRegMsgModuleFunc
      with (kAppSymbol, partFrame.msgModuleTemplate );
end;
```

Note

The `call k`*FunctionName*`Func with (`*args*`)` syntax is used because this function is defined in the stream file "pMsgMod.stream." The `DeleteMsgModule` and `UnRegMsgModule` functions discussed below must also be called using this syntax. ◆

To remove your msgModule you need to supply both a `DeletionScript` and a `RemoveScript`. For information on the `DeletionScript` and the `RemoveScript`, see Chapter 2, "Getting Started," in *Newton Programmer's Guide*. The `DeletionScript` function should call the `DeleteMsgModule` function (page 2-39), as in the following code sample:

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

The RemoveScript function should call UnRegMsgModule function (page 2-40), as in the following code sample:

```
RemoveScript := func( removeFrame )
begin
   call kUnRegMsgModuleFunc with ( kAppSymbol );
end;
```

Application Program Interface to a msgModule

There are two ways an installed NewtonScript application can interact with the Messaging Enabler. It may change configuration options and send a msgModule a command (if the msgModule supports this). The Messaging Enabler provides the two methods ChangeConfig (page 2-41) and SendDirectCommand (page 2-45) to accomplish this. To call these methods you must use the global function TransportNotify. The TransportNotify function is described in Chapter 18, "Routing Interface Reference," in *Newton Programmer's Reference*.

Using the ChangeConfig Method

The ChangeConfig method (page 2-41) can be used to change both the Messaging Enabler's and a msgModule's preferences. The Messaging Enabler preferences that may be set are those available to the user in its preference slip, shown in Figure 1-2 on page 1-5, as well as a few preferences that the user can't set. The msgModule preferences that may be set with ChangeConfig are exactly those that the user may set in the msgModule's hardware preference slip, also shown in Figure 1-2 on page 1-5.

The following sample code demonstrates a call to ChangeConfig:

```
TransportNotify (
   'msgEnabler,   //The Messaging Enabler's transport sym
   'ChangeConfig,
   [
      callBackFn, //fn. to call when ChangeConfig is done
      {  //Messaging Enabler preferences
         disable:true,        //don’t let user set prefs
         autoStatus:nil,      //don’t show status slips
         pollPeriod:'syCheck5,//get msgs every 5 mins
         nowOrLater:'now,     //send msgs immediately
      },
      {  //msgModule preferences
         deviceSym:'|msgModuleSym:Sig|, //which msgModule
         portIndex:0, //1st elem. in portStrings array
                     //Note that the 1st elem always
                     //holds the default string
      }
   ]
);
```

Using the SendDirectCommand Method

The `SendDirectCommand` method (page 2-45) calls a msgModule’s `DirectCommand` method (page 2-20). The `DirectCommand` method is intended for a msgModule to implement an application program interface (API). The *command* parameter to the `DirectCommand` method may be any NewtonScript object. It is through this parameter that the application informs the `DirectCommand` method of both the action to perform and the data to use.

If you as the creator of the msgModule want a third party application to take advantage of the functionality provided by your `DirectCommand` method, it is up to you to document the possible values the *command* parameter can take.

Summary

Constants

Built-in Result Codes

```
kRES_SUCCESS
kRES_INUSE
kRES_MEMORY
kRES_FAILED
kRES_NOTCONNECTED
kRES_UNSUPPORTED
kRES_COMMS
kRES_TIMEOUT
kRES_SYNCHRONIZE
kRES_INVPORT
kRES_INVDEVICE
kRES_READY
kRES_NOCANCEL
kRES_NOTREADY
kRES_UNCHANGED
kRES_NOTFOUND
kRES_BATTERY
kRES_NOACCESS
kRES_MSGSIZE
kRES_INVADDRESS
kRES_INVMESSAGE
kRES_MSGDATA
kRES_MSGEMPTY
kRES_NOSERVICE
```

```
kRES_PREFSCHANGED
kRES_BATTERYWARN
kRES_CFGINVALID
kRES_PARTIAL
kRES_NOSEND
kRES_NORECEIVE
kRES_STORE
kRES_NOMESSAGE
```

Data Structures

Item Frame

```
anItemFrame := {
   deviceSym   : symbol,  //symbol identifying msgModule
   reply       : frame,   //A reply frame
   ID          : integer, //unique message ID
   complete    : Boolean, //is this the whole message?
   title       : string,  //message title
   body        : frame,   //message data
   fromRef     : nameRef, //sender
   toRef       : array,   //array of recipients
   appSymbol   : symbol,  //application to receive this msg
   replies     : array,   //acceptable replies to this msg
   replyIndex  : integer, //index to sendOptions array
   segment     : integer, //message part number
   totalSegments: integer,//total number of segments
   groupID     : integer, //id of group of segments
   error       : integer, //integer result code
   ...
}
```

Reply Frame

```
aReplyFrame := {
   type        : symbol,  //reply type
   deviceSym   : symbol,  //symbol for msgModule
   text        : string,  //the reply
   replyID     : integer, //integer ID
   complete    : Boolean, //is this the whole reply?
   ...
}
```

Proto

protoMsgModule

```
aMsgModule := {
   _proto      : protoMsgModule,//proto msgModule object
   about       : stringViewTemplateOrNil,//info in About slip
   cannedSupport: Boolean,//support canned replies?
   declareSelf : symbol,  //msgModule base view
   deviceTitle : string,  //messaging device name
   dirSupport  : Boolean, //support directory of messages?
   icon        : bitmap,  //msgModule icon
   persistConnection: Boolean,//keep connection open
   pfVars      : frame,   //for your data
   pollAlive   : integer, //when to poll for aliveness
   pollHardware: integer, //when to poll for openess
   pollMessage : integer, //when to poll for messages
   portStrings : array,   //strings in port picker
   powerStrings: array,   //strings in power picker
   prefsTemplate: viewTemplate,//custom prefs. slip
   sendOptions : array,   //ways to send data
   soundStrings: array,   //strings in sound picker
   timeStrings : array,   //strings in time picker
   title       : string,  //type of messaging
   transmitterStrings:array,//strs in transmitter picker
   version     : string,  //verison string
   Alive       : function, //is msg. channel avialable?
   Cancel      : function, //cancel present operation
   Close       : function, //close msg channel
   DeleteMessageFromDir: function,//delete msg. from dir.
   DirectCommand: function,//perform command from app.
   DoCallBack  : function, //execute callBack function
   DoEvent     : function, //process event
   GetConfig   : function, //get configuration info
   GetDir      : function, //get dir of msgs/canned replies
   GetMessageFromDir: function,//get next msg. from dir.
   GetNextMessage: function,//get next message
   GetPreference: function,//returns pref. value
   InstallScript: function,//called when installed
   Open        : function, //open the msg. channel
   RemoveScript: function, //called when removed
   SavePreference: function,//saves a preference value
   SendMessage : function, //send a message
   SetConfig   : function, //set configuration info
   TranslateError: function,//map result code to string
   ProcessCombinedMessage: function, //process combined msg
...
}
```

Functions and Methods

Global Functions

```
DeleteMsgModule(symbol) //deletes a msgModule
RegMsgModule(symbol, msgModule) //registers a msgModule
UnRegMsgModule(symbol) // unregisters a msgModule
```

Messaging Enabler Methods

```
messagingEnabler:ChangeConfig(callBack, cfgMsgEnabler, cfgMsgModule)
   //change either Enabler or msgModule configuration
messagingEnabler:SendDirectCommand(callBack,options,command)
   //call a msgModule’s DirectCommand method
```

Messaging Enabler Interface Reference

This chapter lists all constants, data structures, protos, functions, and methods used by the Messaging Enabler.

Constants

This section described the constants used by the Messaging Enabler.

Built-in Result Codes

Table 2-1 describes the built-in error codes. You may customize the error messages used by your msgModule by defining a `TranslateError` method (page 2-38). For information about result codes see "Result Codes" (page 1-11).

Table 2-1 Built-in error codes

Result code	Error string
`kRES_SUCCESS`	Success.
`kRES_INUSE`	The required communication port for the *deviceTitle* is currently in use.\n Try resetting your Newton PDA. Refer to the handbook for information on resetting
`kRES_MEMORY`	There is not enough memory to complete the operation.\n Try resetting your Newton PDA. Refer to the handbook for information on resetting.
`kRES_FAILED`	An error has occurred. The last operation failed.\n You may retry the operation.
`kRES_NOTCONNECTED`	An error has occurred. The *deviceTitle* has not been opened. Try resetting your Newton PDA. Refer to the handbook for information on resetting.
`kRES_UNSUPPORTED`	An error has occurred. The function requested is not supported by the installed message module.
`kRES_COMMS`	Communication with the *deviceTitle* has ended.
`kRES_TIMEOUT`	The *deviceTitle* is not responding to requests.\n Please check the connection to the *deviceTitle*.
`kRES_SYNCHRONIZE`	The *deviceTitle* is not communicating correctly. (Data Error)
`kRES_INVPORT`	The *deviceTitle* could not be found at the required location.
`kRES_INVDEVICE`	The connected device does not appear to be a *deviceTitle*.
`kRES_READY`	There is no operation to cancel.

Table 2-1 Built-in error codes

Result code	Error string
`kRES_NOCANCEL`	The current operation can't be stopped right now.\n You may try again later.
`kRES_NOTREADY`	The *deviceTitle* is busy and can't do anything else right now.\n Wait until the current operation completes and try again.
`kRES_UNCHANGED`	No new messages were found.
`kRES_NOTFOUND`	The message requested was not found.\n You may have previously deleted the message.
`kRES_BATTERY`	The *deviceTitle* battery is too low to continue.\n You need to replace or recharge the cell and try again.
`kRES_NOACCESS`	The *deviceTitle* is not in range to send.\n Move to a better location and try again.
`kRES_MSGSIZE`	The *deviceTitle* can't send messages of this size.\n Try creating several smaller messages.
`kRES_INVADDRESS`	The destination address is invalid.
`kRES_INVMESSAGE`	The message content can't be sent by the *deviceTitle*.
`kRES_MSGDATA`	Message content contains errors.
`kRES_MSGEMPTY`	Message contains no content.
`kRES_NOSERVICE`	Your *deviceTitle* has not been activated with your service provider.
`kRES_PREFSCHANGED`	The Messaging Enabler preferences have been modified by an installed application.
`kRES_BATTERYWARN`	The *deviceTitle* battery is low. You should change or recharge the cell as soon as possible.
`kRES_CFGINVALID`	The configuration item to set contains an invalid value.

Table 2-1 Built-in error codes

Result code	Error string
`kRES_PARTIAL`	There is not enough memory to retrieve all messages from the *deviceTitle*. Please delete some of the messages to retrieve more.
`kRES_NOSEND`	The *deviceTitle* is unable to send the message at the moment.
`kRES_NORECEIVE`	The *deviceTitle* us unable to receive messages at the moment.
`kRES_STORE`	The Newton PDA does not have enough store memory to complete the operation.\n Try deleting inbox items or storing new items on a card.
`kRES_NOMESSAGE`	There are no messages pending.

Data Structures

This section describes the data structures used by the Messaging Enabler.

Item Frame

The Messaging Enabler extends the standard In/Out Box item frame to include several new slots. The msgModule may also extend an item frame to contain any other specific slots that are needed.

The standard set of slots in item frames are described in Chapter 18, "Routing Interface Reference," in *Newton Programmer's Reference*. Only those extra slots used by the Messaging Enabler are listed here.

Slot descriptions

`deviceSym`	This slot is added by the Messaging Enabler. A symbol indicating the msgModule that owns this item frame. This is the appSymbol that the msgModule registered with. It is for use by applications that need to register to automatically receive Messaging Enabler items from the In Box. Do not modify this slot.
`reply`	This slot is added by the Messaging Enabler. A reply frame; see "Reply Frames" (page 2-7). This is set by the Messaging Enabler to specify the reply to be sent to this item frame. The msgModule should check this slot when sending a message to determine if the message is a reply or a new message. See the description of the msgModule `SendMessage` method (page 2-35). Do not modify this slot.
`ID`	Required. An integer uniquely identifying this message.
`complete`	Optional. A Boolean indicating whether this data is the complete message or if there is more data to be retrieved.
`title`	Required. A string. Set this to a description of the message body.
`body`	Optional. A frame. Set this to the message data. For text messages use the defined `'text` data definition, which is a frame with the following slots: `class` The symbol `'text`. `text` A string that is the message contents. If the message contains a segment of a flattened frame, use a frame with the following slots: `class` The symbol `'|file.fragment|`. `data` The binary data for this part of the flattened frame. If the message is a frame, simply set this `body` slot to that frame. See "Text and Frame Messages" beginning on page 1-16.
`fromRef`	Optional. A name reference or other information that identifies the sender. If this slot is present and the

msgModule slot `sendOptions` is non-`nil`, the user can create a new message to reply to this message. The new message will be sent to the address in this slot. See "Using Name References with the Messaging Enabler" (page 1-18).

`toRef`	Optional. An array containing one or more name references used to identify the recipient(s) of the item. See "Using Name References with the Messaging Enabler" (page 1-18).
`appSymbol`	Optional. Set this slot to an application symbol if your msgModule is intended to receive messages destined for only one application.
`replies`	Optional. An array containing reply type symbols and/or reply frames describing the possible replies to this message.

A reply frame contains a `type` slot, which can have one of four values: `'ack`, `'user`, `'canned`, and `'choice`. For replies of type `'ack`, `'user`, or `'canned`, this symbol is all that needs to be included in this `replies` array. For replies of type `'choice`, the whole reply frame must be included in this `replies` array.

For information on reply frames, see "Reply Frames" (page 2-7).

The following array is a possible value for this slot:

```
[  'ack,
   'user,
   'canned,
   {  type:'choice,
      text:"Reply choice 1",
      replyID:0
   },
   {  type:'choice,
      text:"Reply choice 2",
      replyID:1
```

```
        }
    ]
```

If the msgModule sets this slot, it must supply a `SendMessage` method (page 2-35), even if the msgModule slot `sendOptions` is `nil`.

`replyIndex`	Optional. An integer. Valid only if the msgModule slot `sendOptions` is an array with more than one item. This determines the `sendOptions` item to display to originate the new reply.
`segment`	Optional. An integer. Required if this is a message segment. This is the current block number for the segmented message. Segments are counted beginning at 0.
`totalSegments`	Optional. An integer. Required if this is a message segment. The total number of blocks this message consists of.
`groupID`	Required if this is a message segment. An integer. A unique identifier for the group of segmented messages.
`error`	Optional. Set this to an integer result code if an error occurs. See “Result Codes” (page 1-11).

Reply Frames

Reply frames contain the following slots:

Slot descriptions

`type` Required. A symbol specifying the class of the reply. Currently supported types of replies are:

`'ack`	An acknowledgment.
`'choice`	A reply option extracted from a received message. A reply frame of this type is valid only in the `replies` array of an

	item frame; see "Item Frame" beginning on page 2-4.
`'user`	A reply option from the internal list of user-created replies. These replies are stored in the Reply soup; see "The Reply Soup" (page 1-21).
`'canned`	A reply option supplied by the messaging device. The current list used by the Messaging Enabler is stored in the Reply soup; see "The Reply Soup" (page 1-21).

`deviceSym`	Optional. Required if the `type` slot is set to `'canned`. The symbol stored in the `deviceSym` slot of the msgModule, this is the symbol the msgModule used in the call to `RegMsgModule` (page 2-39).
`text`	Required unless the `type` slot is set to `'ack`. A string that is the actual reply message.
`replyID`	Required if the `type` slot is set to `'canned`. A unique integer identifying this canned reply.
`complete`	Required for replies of class `'canned`. A Boolean. Set this to `nil` if the `text` slot does not contain the entire canned message; that is, it is an abbreviation for the canned message. Currently the Messaging Enabler requires this to be `true`.

Proto

This section describes the single Messaging Enabler proto, `protoMsgModule`, its data slots, and methods.

protoMsgModule

This object is provided as a basis for all msgModules (). A msgModule serves as an interface between a particular messaging device and the Messaging Enabler.

Slot descriptions

`_proto`	Required. Set this slot to `protoMsgModule`.
`about`	Optional. A string, a view template, or `nil`. The default is `nil`. If this slot is nil, an About slip is still available to the user from the `"About` *title*`"` choice of the information ("i") menu. The `title` slot is described on (page 2-15). If this slot contains a string, it is used to display static information about the msgModule; for example, `"Test Pager Driver\n© 1996 Apple Computer."` The following information is also displayed in this view: the `title` and `version` slots, and the `statusText` and `messageCount` slots returned by the `GetConfig` method (page 2-26), if defined. Use the `statusText` and `messageCount` slots to provide dynamic information. If this slot holds a view template, that view is created and displayed as the About slip.
`cannedSupport`	Optional. A Boolean. The default is `nil`. Does the msgModule support canned replies? If `true`, the following msgModule method is required: `GetDir` (page 2-30)
`declareSelf`	Optional. A symbol. The default is `'msgModule`. This symbol can be used to reference the base frame of the msgModule.
`deviceTitle`	Optional. A string. The default is the value of the msgModule `title` slot. The user-visible name of the messaging device.

`dirSupport`	Optional. A Boolean. The default is `nil`. Does the msgModule support retrieving a directory of available messages? If `true`, the following msgModule methods are required: `GetDir` (page 2-30) `GetMessageFromDir` (page 2-31) `DeleteMessageFromDir` (page 2-19) If this slot is set to `true`, the msgModule slot `persistConnection` must also be set to `true`. If `nil`, the following msgModule method is required: `GetNextMessage` (page 2-32)
`icon`	Optional. An icon. If present this icon is used to identify this msgModule and is displayed with In/Out Box items and the routing slip picker. The default is the Messaging Enabler icon.
`persistConnection`	Optional. A Boolean. The default is `true`. Should the message channel be left open after retrieving messages? If `true` then the Messaging Enabler leaves the msgModule open until it receives a `kEV_CLOSED` event, or the msgModule returns a failure to a request from the Messaging Enabler. See “Sending Events” (page 1-12) and “Result Codes” (page 1-11). Set this slot to `nil` if the messaging device consumes a large amount of power while the msgModule is open. The msgModule could save power by opening and closing the message channel for each request if it has some other means of detecting that the messaging device is not available. This slot must be set to `true` if the msgModule slot `dirSupport` is `true`.
`pfVars`	Optional. A frame. This frame can be used to store msgModule-specific variables.

	Before the msgModule's `InstallScript` is called, this slot is assigned to a frame containing only a `_proto` slot. This `_proto` slot points to your original `pfVars` frame. This technique minimizes system heap usage, while allowing this frame to be modifiable.
	Note You may ignore the discussion above. The inheritance mechanism makes these changes transparent to the application developer. ◆
	For information on an alternate way of storing writeable data, see "Message Module Methods for Storing Data" (page 1-25).
`pollAlive`	Optional. An integer. The Messaging Enabler sends your msgModule an `Alive` message (page 2-16) every `pollAlive` seconds. Since this method uses system resources, it is recommended not to set this slot below 30 seconds. The Messaging Enabler does not display a status slip, even if the "Show status slips" preference is selected, for the periodic `Alive` message-sends. This preference allows the Newton device to return an unused system resource to the system if the messaging device is no longer available. If there is another way to detect that the hardware is gone, such as an endpoint event handler, you should use that, since the calls to `Alive` take up system resources.
`pollHardware`	Optional. A Boolean. The default is `nil`. If `true` the Messaging Enabler periodically sends your msgModule the `Open` message (page 2-33) while the msgModule is closed. The rate at which these messages are sent is set by the user in the "Get messages" preference; see "Providing Preferences" (page 1-22). If both the `pollMessage` and `pollHardware` slots are `nil`, the "Get messages" picker is not displayed. If your

msgModule never sends the `kEV_DETECTED` event, you should consider setting this slot to `true`; see “Sending Events” (page 1-12).

The Messaging Enabler does not display a status slip, even if the “Show status slips” preference is selected, for these periodic `Open` message-sends.

`pollMessage`
: Optional. A Boolean. The default is `nil`. If `true` then the Messaging Enabler periodically sends your msgModule either the `GetNextMessage` (page 2-32) or `GetDir` (page 2-30) message, depending on the value of the msgModule `dirSupport` slot. If both `pollMessage` and `pollHardware` are `nil`, the “Get messages” preference is not displayed; see “Providing Preferences” (page 1-22).

  If your msgModule never sends the `kEV_MESSAGE` event, you should consider setting this slot to `true`; see “Sending Events” (page 1-12).

  The Messaging Enabler does not display a status slip, even if the “Show status slips” preference is selected, for these periodic `GetDir` or `GetNextMessage` message-sends.

`portStrings`
: Optional. An array of strings or `nil`. These strings make up the options for the “Connect Using” picker in the hardware preference slip. If this slot is `nil` or not present, the “Connect Using” picker is not displayed. See “Providing Hardware Preferences” (page 1-23).

`powerStrings`
: Optional. An array of strings or `nil`. These strings make up the options for the “Power” picker in the hardware preference slip. If this slot is `nil` or not present, the “Power” picker is not displayed.

  Presence of this slot requires that the following msgModule methods be provided:

  `GetConfig` (page 2-26)

  `SetConfig` (page 2-36)

  See “Providing Hardware Preferences” (page 1-23).

`prefsTemplate` Optional. A view template or `nil`. The default is `nil`. The view template is instantiated when the user taps the Options button in the preference slip.

The following slots are added by the Messaging Enabler to the view before it is instantiated:

`msgModule`	A reference to the your msgModule instance.
`ptTransport`	This slot is used internally; do not override it.

See “Providing a Custom Preference Slip” (page 1-24).

`sendOptions` Optional. A send option frame, an array of send option frames, or `nil`. The default is `nil`.

A send options frame contains the necessary slots to support a particular addressing type for sending messages. If this slot is not present the user cannot originate an outgoing message, since the string in the msgModule `deviceTitle` slot is not listed in any routing pickers. The user may still be able to reply to received messages if a received item frame contains the required slots. For information on the format of a received message see “Item Frame” (page 2-4).

Send options frames contain the following slots:

`routeSlipType`

Required. Either a view template or an installed nameRef data definition symbol. The following nameRefs are predefined:

`'|nameRef.people|`

`'|nameRef.fax|`

`'|nameRef.phone|`

`'|nameRef.email|`

`'|nameRef.people.pager|`

The view template should be a routing slip template; for information on how to create a routing slip template see

		Chapter 22, "Transport Interface," in *Newton Programmer's Guide*.
	`group`	Required. A symbol defining the routing group for this send slip. Currently defined groups are `'print`, `'fax`, `'mail` and `'page`. You may define your own, but should register your group symbol with Newton Developer Technical Support.
	`groupTitle`	Required. A string for the name of the group for sending messages. For the predefined groups, use the strings `"Print"`, `"Fax"`, `"Mail"`, and `"Page"`.
	`groupIcon`	Optional. An icon. This is the icon for the group of this message module. If this slot is not present, the msgModule `icon` slot is used. For the predefined groups you should use the corresponding group icon: `ROM_RoutePrintIcon` `ROM_RouteFaxIcon` `ROM_RouteMailIcon` `ROM_RoutePageIcon`
	`replyTypes`	Optional. An array of reply type symbols. The following reply type symbols are allowed: `'ack`, `'user` and `'canned`. This defines the types of `reply` frames that can be attached to the message to be sent; see "Reply Frames" (page 2-7).
	`dataTypes`	Optional. An array of symbols that specify the types of data that this send frame supports. Possible values are `'frame` and `'text`. The default is `['text]`.
`soundStrings`	Optional. An array of strings or `nil`. These strings make up the options for the "Sound" picker in the hardware preference slip. If this slot is `nil` or not present, the "Sound" picker is not displayed.	

	Presence of this slot requires that the following msgModule methods be provided: `GetConfig` (page 2-26) `SetConfig` (page 2-36) See “Providing Hardware Preferences” (page 1-23).

`timeStrings`	Optional. An array of strings or `nil`. These strings make up the options for the “Clock” menu in the hardware preference slip. If this slot is `nil` or not present, the “Clock” picker is not displayed.
	It is recommended that the following array be used for synchronizing the time to and from the Newton:
	`["Set Newton from` *deviceTitle*`",`
	`"Set` *deviceTitle* `from Newton"]`
	Presence of this slot requires that the following msgModule methods be provided:
	`GetConfig` (page 2-26)
	`SetConfig` (page 2-36)
	See “Providing Hardware Preferences” (page 1-23).

`title`	Optional. A string. The default is `"Message"`. This string is used to identify the particular type of messaging. This string is displayed in the picker for receiving messages, and the titles for the preferences.
`transmitterStrings`	Optional. An array of strings or `nil`. These strings make up the options for the “Transmitter” picker in the hardware preference slip. If this slot is `nil` or not present, the “Transmitter” picker is not displayed. Presence of this slot requires that the following msgModule methods be provided: `GetConfig` (page 2-26) `SetConfig` (page 2-36) See “Providing Hardware Preferences” (page 1-23).
`version`	Optional. A string with the user-visible version number of this particular msgModule. This string is displayed in

the About slip. This is also used to ensure the Reply soup is up to date if the msgModule does not support retrieving the serial number with the `GetConfig` method (page 2-26); see “The Reply Soup” beginning on page 1-21.

IMPORTANT
The `_class`, `new`, `dispose`, `pcEvent`, `boOpen`, `inAlarmKey`, `inAliveKey`, `stSerial`, `syActive`, `arAbleToSend`, `arTransportRegistry`, `fnTryDispatchEvent`, `ptNotifyAction`, `deviceSym`, and `doEvent` slots are reserved internally by `protoMsgModule`. ▲

The methods of interest in `protoMsgModule` are described in the following subsections, in alphabetical order.

Alive

msgModule`:Alive(`*callBack*`)`

Called to check that the message channel is still available.

Required if the msgModule slot `pollAlive` is non-`nil`.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see “Result Codes” (page 1-11). The following values are suggested as reasonable for this method:
	`kRES_SUCCESS`
	`kRES_FAILED`
	`kRES_MEMORY`
	`kRES_TIMEOUT`
	`kRES_COMMS`

complFrame	Pass the value `nil.`

If the message channel is no longer available, you should call `DoCallBack` with *complCode* set to a result code indicating why the message channel is unavailable. The Messaging Enabler does not send a `Close` message (page 2-18) to the msgModule and assumes that all resources have been released back to the system. There is no need to send a `kEV_CLOSED` event as well.

The return value of this method is ignored.

Cancel

msgModule`:Cancel(`*callBack*`)`

Called to cancel any outstanding asynchronous calls from the Messaging Enabler.

This method is optional. If it is not defined the Messaging Enabler assumes that asynchronous methods may not be cancelled. In this case, the Messaging Enabler will stop once the current operation is complete. For example, consider the case where the user taps the Stop button while the Messaging Enabler is polling the msgModule for messages, by calling `GetNextMessage` (page 2-32). If the `Cancel` method is not defined, the Messaging Enabler will wait for the current `GetNextMessage` to return, but will not call `GetNextMessage` again.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:
	`kRES_SUCCESS`
	`kRES_FAILED`
	`kRES_MEMORY`
	`kRES_TIMEOUT`
	`kRES_NOCANCEL`
	`kRES_NOTCONNECTED`

	`kRES_READY`
complFrame	Pass the value `nil.`

You should not close the communications channel because of to a cancel request, unless the cancel occurs while the msgModule is processing an `Open` message (page 2-33). The channel should be properly closed so that any subsequent calls to `Open` succeed. That is, the net effect of canceling an `Open` call should be a closed message channel.

The Messaging Enabler does not expect the operation being cancelled to call `DoCallBack`. For example, if the Messaging Enabler first calls `GetNextMessage` and later calls `Cancel`, `Cancel` is expected to call `DoCallBack`, but `GetNextMessage` isn't.

The return value of this method is ignored.

Close

msgModule`:Close(`*callBack*`)`

Called to close the message channel and free all used resources back to the system.

This method is required.

callBack — A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:
	`kRES_SUCCESS`
	`kRES_FAILED`
	`kRES_MEMORY`
	`kRES_TIMEOUT`
	`kRES_NOTCONNECTED`
	`kRES_READY`
complFrame	Pass the value `nil.`

The msgModule instance is not removed at this point. The Messaging Enabler may send the `Open` message (page 2-33) again after this method, so the msgModule should reset any instance variables needed at this point.

IMPORTANT

The `Close` message can be sent at any time, even if another operation is in progress. This means the msgModule must abort and close if it receives this message, even if data is lost by doing so.

The Messaging Enabler waits for a safe time to send the `Close` message. The user can, however, request that the message channel be closed while the Messaging Enabler is waiting. At this point, the Message Enabler informs the user that data may be lost. Therefore, if you are sent a `Close` message, either no data will be lost, or the user has OK'd the risk. ▲

▲ **WARNING**

You must catch all exceptions from your `Close` method, as it must guarantee return. If your `Close` method does not return, the system may halt. ▲

DeleteMessageFromDir

msgModule`:DeleteMessageFromDir(`*callBack*`,` *arMsgID*`)`

Called to remove the messages specified by the *arMsgID* array from the message channel.

Required if the msgModule slot `dirSupport` is `true`.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:

`kRES_SUCCESS`
`kRES_FAILED`
`kRES_MEMORY`
`kRES_TIMEOUT`
`kRES_NOTFOUND`
`kRES_NOTCONNECTED`
`kRES_NOTREADY`

complFrame	Pass the value `nil`.

arMsgID	An array of message identifiers. This is a unique integer previously returned from the `GetDir` method (page 2-30). The `GetDir` method passed this value as one of the slots in the *complFrame* parameter in its call to `DoCallBack`.

The return value of this method is ignored.

DirectCommand

msgModule`:DirectCommand(`*callBack*`,` *options*`,` *command*`)`

Called by the Messaging Enabler when it receives a `SendDirectCommand` message (page 2-45). You may implement this method to allow an application to send commands directly to the message channel.

This method is optional.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	The Messaging Enabler does not process the return code from this method. This return code is passed directly to the application which called `SendDirectCommand`.

	complFrame	Any valid NewtonScript object. If you need to return data to `SendDirectCommand`, use this value.
options	A frame containing the following slots:	
	`open`	Required. A Boolean. If this slot is `true` and the message channel is not currently open, the MsgModule should attempt to open the message channel before executing the command. The MsgModule should then send a `kEV_DETECTED` event after the command completes if the message channel is left open; see "Sending Events" beginning on page 1-12.
	`port`	`Optional.` If the msgModule slot `portStrings` is non-`nil`, an integer array index to the msgModule slot `portStrings`.
command	Any valid NewtonScript object (usually a frame). Use this parameter to allow the `SendDirectCommand` method both to describe which action the msgModule should take, and to supply the msgModule with any needed data. If you wish to allow applications other than yours to send commands to the message channel, you must document the values this parameter can take.	

The return value of this method is ignored.

DoCallBack

msgModule`:DoCallBack(`*callBack*`,` *complCode*`,` *complFrame*`)`

Executes the function encapsulated in the *callBack* argument. Most of the msgModule methods need to call this function when they complete. See "Using Callback Functions" beginning on page 1-10.

You should not need to override this method, but if you do, you must call the inherited method.

callBack	The *callBack* parameter is passed to the msgModule method which is calling `DoCallBack`.
compCode	An integer result code returned by the msgModule method calling `DoCallBack`. The built-in result codes are listed in Table 2-1 (page 2-2).
complFrame	Any valid NewtonScript object. Allowable values for this parameter depend on which msgModule method is calling `DoCallBack`.

The return value of this method is ignored.

DoEvent

msgModule`:DoEvent(`*evID*`, `*evData*`)`

Sends an event synchronously to the Messaging Enabler. See "Sending Events" beginning on page 1-12.

You should not need to replace this method, but if you do you must call the inherited method.

evID An integer event code. The following codes are supported by the Messaging Enabler.

`kEV_BATTERY`	The battery level of the messaging device has changed.
`kEV_CLOSED`	The message channel is no longer available. The communication link has probably been broken. The Messaging Enabler will either poll the message channel (if `pollHardware` is `true`) or await a `kEV_DETECTED` event.
`kEV_DETECTED`	The message channel is available.
`kEV_MEMORY`	The message channel either has no

memory available, or it now does after reporting a shortage.

`kEV_MESSAGE`
: The message channel has detected that messages are available. This causes the Messaging Enabler to resynchronize to the message channel.

`kEV_PROGRESS`
: There is progress to report. This event allows a msgModule to modify the progress indicator provided by the Messaging Enabler. The Messaging Enabler displays a progress indicator for each message sent to the msgModule.

  This is the only event that is sent synchronously to the Messaging Enabler.

`kEV_SERVICE`
: The message channel is not in range to send or receive messages, or the messaging device is temporarily unavailable. While the message channel is not in service the user cannot send any messages; the About slip displays an alert stating that the message channel is out of service.

evData A frame. The slots that should be in this frame depend on the value of the *evID* parameter.

If *evID* is `kEV_BATTERY`, this frame should contain a single slot:

`state` One of the following symbols: `'Dead`, `'Alive`, `'Warning`.

A state of `'Dead` indicates that the message channel is no longer able to send messages. The message enabler awaits another `kEV_BATTERY` event with a state of `'Alive` before allowing messages to be sent again. A state of `'Warning` issues a message to the user with the string for the `kRES_BATTERYWARN` result code; see "Result Codes" beginning on page 1-11.

If *evID* is `kEV_CLOSED`, this frame should contain a single slot:

`reason`	An integer result code; the built-in result codes are listed in Table 2-1 (page 2-2). A slip is displayed containing the string this error code represents, unless this slot contains the integer `kRES_SUCCESS`. The string `kRES_COMMS` has been designed to be noninvasive in this context.

If *evID* is `kEV_DETECTED`, this frame can contain any number of optional slots. If you have no slots you would like to include in this frame, you may pass the value `nil` for this parameter instead of a frame. Any slots included in this frame, will be passed as the *options* parameter to the `Open` (page 2-33) message. The following two slots have special significance to the Messaging Enabler:

`port`	An integer corresponding to the port the message channel is available on. This is the array index of the array in the msgModule slot `portStrings`.
`open`	A Boolean. If `true`, the Messaging Enabler will not send the `Open` message. You should include this slot if the message channel is open.

If *evID* is `kEV_MEMORY`, this frame should contain a single slot:

`state`	One of the following symbols: `'Full` or `'OK`. The Messaging Enabler does not allow sending messages after it has received this

event with a `state` slot set to `'Full`, until it receives another one of these events with a `state` slot set to `'OK`.

If *evID* is `kEV_MESSAGE`, this frame should contain a single slot:

`state`	One of the following symbols: `'New` or `'Canned`.
	The value `'New` should be used for new messages. The Messaging Enabler will then sends your msgModule the `GetDir` (page 2-30) or `GetNextMessage` (page 2-32) message (depending on the value of the msgModule `dirSupport` slot).
	The value `'Canned` should be used for a new list of canned messages; some messaging service providers send these lists. The Messaging Enabler then sends your msgModule the `GetDir` message (page 2-30).

If *evID* is `kEV_PROGRESS`, this frame should contain the following slots:

`type`	Required. One of the following symbols: `'vGauge`, `'vBarber`, or `'vStatus`.
	The value `'vGauge` specifies that a gauge indicator should be used. Gauge indicators display the percentage of the operation which has completed.
	The value `'vBarber` specifies that a barber indicator should be used. Barber indicators do not display how much of the operation has completed.
	The value `'vStatus` specifies that a slip with a text string should be used.

`value`	Required if the `type` slot is set to `'vGauge`. An integer from 0 to 100 indicating what percentage of the gauge should be filled.
`statusText`	Optional. A string with the status message to display.

If *evID* is `kEV_SERVICE`, this frame should contain a single slot:

`state`	One of the following symbols: `'NoService`, `'Busy`, `'OutService`, or `'InService`. If the `state` slot is `'NoService`, a user alert is displayed immediately to inform the user that the message channel is not registered. The string mapped to the `kRES_NOSERVICE` result code is displayed in the alert; see “Result Codes” beginning on page 1-11. If the `state` slot is `'Busy` or `'OutService` a user alert is displayed when the user attempts to retrieve or send messages. The strings `kRES_NOTREADY` and `kRES_NOACCESS`, respectively, are displayed. Sending this event with a `state` slot set to `'InService` cancels a previous event of any of the other three types.

GetConfig

msgModule`:GetConfig(`*callBack*`,` *cfgFrame*`)`

Called by the Messaging Enabler to retrieve current configuration information. The *cfgFrame* parameter contains a frame with the slots required by the Messaging Enabler. You should set these slots, and return this frame in the *complFrame* parameter to the `DoCallBack` method.

Required if any of the following msgModule slots are non-`nil`: `soundStrings`, `powerStrings`, `transmitterStrings`, or `timeStrings`.

The Messaging Enabler uses this method at start-up to ensure that the message channel's configuration is correct. If any slots are returned with the value `nil`, or with a value that does not agree with the user setting, the Messaging Enabler calls the `SetConfig` method (page 2-36) to set these slots. Setting as many of the *cfgFrame* slots as possible, speeds up the start-up process.

callBack — A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

- *complCode* — An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:

  `kRES_SUCCESS`

  `kRES_FAILED`

  `kRES_MEMORY`

  `kRES_TIMEOUT`

  `kRES_NOTCONNECTED`

  `kRES_NOTREADY`

  `kRES_UNSUPPORTED`

- *complFrame* — Set this slot to the altered *cfgFrame*.

cfgFrame — A frame with the slots required by the Messaging Enabler. This frame has one or more of the slots described below. Set these slots to an appropriate value, and pass this frame as the *complFrame* parameter to the `DoCallBack` method.

- `soundIndex`

  Required only if the msgModule slot `soundStrings` is non-`nil`. Set this slot to an integer for the current device setting of the sound corresponding to the array index of the `soundStrings` array. Leave this set to `nil` if the msgModule is unable to retrieve the current setting.

`powerIndex`
Required only if the msgModule slot `powerStrings` is non-`nil`. Set this slot to an integer for the current device setting of the power corresponding to the array index of the `powerStrings` array. Leave this set to `nil` if the msgModule is unable to retrieve the current setting.

`transmiterIndex`
Required only if the msgModule slot `transmitterStrings` is non-`nil`. Set this slot to an integer for the current device setting of the transmitter corresponding to the array index of the `transmitterStrings` array. Leave this set to `nil` if the msgModule is unable to retrieve the current setting.

`time` Required only if the msgModule slot `timeStrings` is non-`nil`. Set this slot to the device time, an integer for the number of minutes elapsed since midnight, January 1, 1904. If the Messaging Enabler detects that this time is outside a tolerance value of the Newton device time then it sets the time using the current `timeStrings` setting.

`statusText`
Optional. A string to return dynamic device information. This is displayed as the lower part of the About slip. This is to return dynamic information about the messaging device. An example might be, “Ready to send. X bytes of free memory.” This string may contain a newline

character (`/n`), the Messaging Enabler resizes the About slip accordingly. Leave this set to `nil` if you do not support dynamic device information.

`serialNumber`
: Optional. A string for the messaging device serial number. This slot is used to ensure the Reply soup is synchronized; see “The Reply Soup” beginning on page 1-21. If this slot is not present, the Messaging Enabler uses the version string of the msgModule. Leave this set to `nil` if you do not support retrieving the messaging device serial number.

`messageCount`
: Optional. Set this to an integer if you can retrieve the number of messages currently in the messaging device. This information is displayed in the About slip. If you are unable to retrieve the number of messages in the device then leave this `nil`.

The following sample code illustrates the logical structure this method should take:

```
msgModule.GetConfig := func (callBack, cfgFrame)
begin
   ...
   if HasSlot( cfgFrame, 'slot1 ) then
      cfgFrame.slot1 := value1;
   if HasSlot( cfgFrame, 'slot2 ) then
      cfgFrame.slot2 := value2;
   ...
end;
```

The return value of this method is ignored.

GetDir

msgModule`:GetDir(`*callBack*`, ` *dir*`)`

Called to either retrieve the current directory of messages or the current set of canned replies from the message channel.

Required if either of the msgModule slots `dirSupport` or `cannedSupport` is `true`.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:
	`kRES_SUCCESS`
	`kRES_FAILED`
	`kRES_MEMORY`
	`kRES_TIMEOUT`
	`kRES_NOTCONNECTED`
	`kRES_NOTREADY`
	`kRES_PARTIAL`

complFrame	An array of directory entries. If *dir*`.type` is `'userMessages`, these frames should be item frames without a `body` slot; see "Item Frame" (page 2-4). If *dir*`.type` is `'cannedMessages`, these frames should be reply frames; see "Reply Frames" (page 2-7).

dir	A frame describing the type of messages to be retrieved. It contains the following slot:

`type`	A symbol specifying the directory type to retrieve. Either `'userMessages` or `'cannedMessages`.

In some cases of low memory the user will not be able to view or receive all messages at once. When returning an array of messages, the system heap may be exhausted if the message list is large. The msgModule should try to catch this `|evt.ex.outOfMem|` exception, and return the partial list of messages retrieved to the Messaging Enabler, setting *complCode* to `kRES_PARTIAL`. To avoid this the item frames returned should set the `complete` slot to `nil` and return only the title of the message. The system heap may also be exhausted if the list of canned messages is large.

The return value of this method is ignored.

GetMessageFromDir

msgModule`:GetMessageFromDir(`*callBack*`, ` *msg*`)`

Called to completely receive a specific message from the message channel previously retrieved with the `GetDir` method (page 2-30). The item frame for the particular message, returned by the `GetDir` method, is passed in through the *msg* parameter. This method should fill the relevant slots, and pass this frame to the `DoCallBack` method (page 2-21) in the *complFrame* parameter.

Required if the msgModule slot `dirSupport` is `true`.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see “Result Codes” (page 1-11). The following values are suggested as reasonable for this method:

`kRES_SUCCESS`

`kRES_FAILED`

`kRES_MEMORY`

`kRES_TIMEOUT`

`kRES_NOTCONNECTED`

`kRES_NOTREADY`

`kRES_NOTFOUND`

complFrame An item frame with a `body` slot and the `complete` slot set to `true`; see "Item Frame" (page 2-4).

msg The requested item frame returned from a previous call to the `GetDir` method (page 2-30).

The return value of this method is ignored.

GetNextMessage

msgModule`:GetNextMessage(`*callBack*`)`

Called to retrieve the next message from the message channel. The msgModule must remove this message from the message channel before returning the message.

Required if the msgModule slot `dirSupport` is `nil`.

callBack A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:

`kRES_SUCCESS`

`kRES_FAILED`

`kRES_MEMORY`

`kRES_TIMEOUT`

`kRES_NOTCONNECTED`

`kRES_NOTREADY`

complFrame An item frame for the retrieved message, or `nil` if there are no messages; see "Item Frame" (page 2-4).

If `nil` is not returned in this argument, the Messaging Enabler sends your msgModule another `GetNextMessage` message.

The return value of this method is ignored.

GetPreference

msgModule`:GetPreference(`*configSym*`)`

Retrieves the value of a configuration item added with `SavePreference` (page 2-35). See “Message Module Methods for Storing Data” (page 1-25).

Do not override this method.

configSym	A symbol. The variable whose value is needed.

This method returns the value of *configSym*.

InstallScript

msgModule`:InstallScript()`

This method allows the msgModule to execute installation code. For example, the msgModule may need to install a new data definition or view definition into the system, or add replies to the Reply Soup.

This method is optional.

Note
Do not confuse this method with the `InstallScript` for the package. ◆

The Messaging Enabler must be installed for this method to be executed.

The return value of this method is ignored.

Open

msgModule`:Open(`*callBack*`,` *options*`)`

Called by the Messaging Enabler before sending or receiving any messages. The Messaging Enabler calls this method repeatedly at a period specified by the user setting for “Get messages” if the msgModule slot `pollHardware` is `true`, until `Open` eventually calls `DoCallBack` with `kRES_SUCCESS`.

This method is required.

callBack	A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:	
	complCode	An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method: `kRES_SUCCESS` `kRES_FAILED` `kRES_MEMORY` `kRES_TIMEOUT` `kRES_INUSE` `kRES_INVDEVICE` `kRES_INVPORT`
	complFrame	Pass in the value `nil`.
options	A frame with the following slot:	
	`port`	An integer or `nil`. An integer is passed in if the msgModule sent a `kEV_DETECTED` event with the `port` slot set or if the msgModule slot `portStrings` is non-`nil`; otherwise `nil` is passed. The integer is an array index to the msgModule slot `portStrings`.

RemoveScript

msgModule`:RemoveScript()`

This method allows the msgModule to execute removal code. The msgModule is still completely in memory before this method is called. This message is sent only if you call the global function `DeleteMsgModule`. The `UnRegMsgModule` function does not cause this method to be called.

This method is optional.

The return value of this method is ignored.

SavePreference

msgModule`:SavePreference(`*configSym*`,` *configValue*`)`

Changes the value of a configuration symbol, or assigns a value to a new symbol. This data is stored in a soup maintained by the Messaging Enabler. See "Message Module Methods for Storing Data" (page 1-25).

Do not override this method.

configSym	A symbol. The variable whose value is needed.
configValue	Any valid NewtonScript object. The value to assign to *configSym*.

The return the value of this method is unspecified.

SendMessage

msgModule`:SendMessage(`*callBack*`,` *message*`)`

Called to send a message.

Required if the msgModule slot `sendOptions` is non-`nil`, or an item frame is received that has the `replies` slot set.

callBack A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method:

`kRES_SUCCESS`

`kRES_FAILED`

`kRES_MEMORY`

`kRES_TIMEOUT`

`kRES_NOTCONNECTED`

`kRES_NOTREADY`

	`kRES_UNSUPPORTED`
	`kRES_NOSERVICE`
	`kRES_INVADDRESS`
	`kRES_INVMESSAGE`
complFrame	Pass in the value `nil`.

message An item frame for the message to send; see "Item Frame" (page 2-4). If the item frame has a non-`nil` `reply` slot, the message to be sent is a reply to this item frame; that is, the message text is in *message*`.reply.text`, unless the reply is of type `'ack` in which case the *message* argument won't have a `reply` slot. If the `reply` slot is `nil` then this is a new message to be sent; that is, the message content is in the *message*`.body` frame.

Note
The value in this parameter may be larger than the message channel can handle. It is up to you to test for this case, and split up the message if necessary. ◆

The return value of this method is ignored.

SetConfig

msgModule`:SetConfig(`*callBack*`,` *cfgFrame*`)`

Called by the Messaging Enabler to set the configuration items of the message channel. The Messaging Enabler uses this method at start-up and when the user changes their preferences.

Required if any of the following msgModule slots is non-`nil`: `soundStrings`, `powerStrings`, `transmitterStrings`, or `timeStrings`.

callBack A frame to pass to `DoCallBack` (page 2-21) when this method completes. `DoCallBack` takes two additional parameters, *complCode* and *complFrame*. Pass the following values for these parameters:

complCode	An integer error code; see "Result Codes" (page 1-11). The following values are suggested as reasonable for this method: `kRES_SUCCESS` `kRES_FAILED` `kRES_MEMORY` `kRES_TIMEOUT` `kRES_NOTCONNECTED` `kRES_NOTREADY` `kRES_UNSUPPORTED`
complFrame	Pass in the value `nil`.

cfgFrame A frame containing the following slots specifying the configuration items to set:

`soundIndex`	This slot is present only if the msgModule slot `soundStrings` is non-`nil`. An integer index to the `soundStrings` array.
`powerIndex`	This slot is present only if the msgModule slot `powerStrings` is non-`nil`. An integer index to the `powerStrings` array.
`portIndex`	This slot is present only if the msgModule slot `portStrings` is non-`nil`. An integer index to the `portStrings` array.
`transmitterIndex`	This slot is present only if the msgModule slot `transmitterStrings` is non-`nil`. An integer index to the `transmitterStrings` array.
`timeIndex`	This slot is present only if the msgModule slot `timeStrings` is non-`nil`. An integer index to the `timeStrings` array.

The return value of this method is ignored.

TranslateError

msgModule`:TranslateError(`*resultCode*`)`

Called to translate a result code to an error string and return this error string to the Messaging Enabler. This allows your msgModule to customize its error messages, both by mapping built-in integer error codes to different strings, and by allowing you to use your own error codes. The Messaging Enabler calls this method before using one of the default strings specified in Table 2-1 (page 2-2). For more information on result codes, see "Result Codes" beginning on page 1-11.

This method is optional.

resultCode	An integer result code.

This method should return the string to display to the user. Return `nil` to use the predefined error string. Return an empty string to avoid displaying a message for this error.

ProcessCombinedMessage

msgModule`:ProcessCombinedMessage(`*itemFrame*`)`

Called after a multi-part message has been combined, but before it is submitted to the Inbox. This allows your msgModule to perform operations on the item frame before it is submitted to the Inbox.

This method is optional.

itemFrame	An item frame, as described in "Item Frame" (page 2-4).

The return value of this method is ignored.

Functions and Methods

This section list functions and methods used by the Messaging Enabler.

Global Functions

This section lists the global functions provided by the Messaging Enabler.

DeleteMsgModule

`DeleteMsgModule(`*symbol*`)`

Removes information stored in the system that is specific to a msgModule and closes the msgModule if open. Usually you call this function from the `DeletionScript` of your msgModule part. This function sends a `Close` message (page 2-18) to the msgModule if it is open, and sends it a `RemoveScript` message (page 2-34) if it has one defined. For more information and sample code, see “Installing and Removing a msgModule” (page 1-26).

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK stream file “pMsgMod.stream”. Call it using this syntax:

`call kDeleteMsgModuleFunc with (`*symbol*`);`

▲

symbol	The msgModule appSymbol. This is the symbol used in the call to `RegMsgModule` (page 2-39).

The return value of this function is undefined.

RegMsgModule

`RegMsgModule(`*symbol*`, `*msgModule*`)`

Registers a new msgModule with the system. Call this function from the `InstallScript` of your msgModule part. This function creates a new msgModule instance and sends it the `InstallScript` message (page 2-33), if this message is defined. For more information and sample code, see “Installing and Removing a msgModule” (page 1-26).

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK stream file “pMsgMod.stream”. Call it using this syntax:

`call kRegMsgModuleFunc with (`*symbol*`, `*msgModule*`);`

▲

symbol	The msgModule appSymbol.
msgModule	The msgModule template. This template must be based on `protoMsgModule` (page 2-9).

The return value of this function is undefined.

UnRegMsgModule

`UnRegMsgModule(`*symbol*`)`

Unregister a msgModule from the system. Usually you call this function from the `RemoveScript` of your msgModule part. If this function is called while the msgModule is open, the Messaging Enabler becomes unstable and a system alert is displayed. This can happen if the msgModule is on a storage card, which is made unavailable while the msgModule is open. The Messaging Enabler protects against this by marking the msgModule package busy when it is open. For more information and sample code, see “Installing and Removing a msgModule” (page 1-26).

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK stream file “pMsgMod.stream”. Call it using this syntax:

`call kUnRegMsgModuleFunc with (`*symbol*`);`

▲

symbol	The msgModule appSymbol. This is the symbol used in the call to RegMsgModule (page 2-39).

The return value of this function is undefined.

Messaging Enabler Methods

The following two methods allow an application to interact with the Messaging Enabler. See “Implementing msgModule Methods” beginning on page 1-9.

ChangeConfig

messagingEnabler`:ChangeConfig(`*callBack*`,` *cfgMsgEnabler*`,` *cfgMsgModule*`)`

Sets the Messaging Enabler’s and/or a msgModule’s preferences, and, optionally, blocks the user from altering these preferences. The Messaging Enabler preferences are specified in the *cfgMsgEnabler* parameter; msgModule preferences are specified in the *cfgMsgModule* parameter. See also “Using the ChangeConfig Method” (page 1-28).

IMPORTANT

This method should be called with `TransportNotify` function. The `TransportNotify` method is documented in Chapter 19, “Transport Interface Reference,” in *Newton Programmer’s Reference*.

Call it using this syntax:

```
TransportNotify(
          'MsgEnabler,
          'ChangeConfig,
          [callBack, cfgMsgEnabler, cfgMsgModule]
                 )
```

▲

callBack	Optional. A frame or `nil`. If you pass `nil`, you do not receive notification when this method completes. The frame defines a callback function to be executed once this method has completed. This frame must contain the following two slots:

`receiver`	A frame. The receiver to which to send `message`.

`message`	A symbol naming the message to be sent. This method must accept two parameters *complFrame* and *complCode*.
complFrame	A symbol or `nil`. If an error occurs trying to set on of the configuration items, this symbol will be the slot that is incorrect.
complCode	An integer result code; see "Result Codes" beginning on page 1-11. The following result codes are the possible values for this parameter: `kRES_SUCCESS` `kRES_NOTSUPPORTED` `kRES_CFGINVALID`

cfgMsgEnabler A frame or `nil`. Set to `nil` if you do not want to configure any Messaging Enabler preferences. The frame contains the Messaging Enabler preferences to be set. The following slots are configurable:

`disable`	A Boolean. The value `true` means that you want to disable the user access to the Messaging Enabler preferences. The default is `nil`.
`hideItems`	A Boolean. The value `true` means that items are not displayed in the In/Out Box. The default is `nil`.
`covertItems`	A Boolean. The value `true` means items are not logged or saved. The default is `nil`.
`pollPeriod`	A symbol. This sets the "Get messages" preference. The allowed values are `'syCheck1`, `'syCheck5`, `'syCheck10`, `'syCheck30`, and `'syCheckNever`. These constants correspond to the number

of minutes between updates. The default is `'syCheckNever`.

`powerOnCheck`
: A Boolean. The value `true` means the Messaging Enabler checks the message channel when the MessagePad is powered on. The user sets this in the “Get messages” preference. The default is `true`.

`whenReceiving`
: A symbol. This sets the “When receiving” preference. Set this to either `'browse` or `'download`. The default is `'browse`.

`alarm`
: A symbol. This sets the “Notify” preference. Set this to either `'Internal` or `'None`. The default is `'Internal`.

`autoStatus`
: A Boolean. The value `true` means that you want to display the status dialogs. The value `nil` means that only the notifyIcon is shown. This slot corresponds to the “Show status dialogs” preferences check box. The default is `true`.

`outboxLogging`
: Only used if the msgModule slot `sendOptions` is non-`nil`. One of the values `'save`, `'log`, or `nil`. This value determines what’s done with an entry after the send completes successfully. The value `'save` means the item is saved in the Out Box; `'log` means the item is deleted from the Out Box and a log entry is made; and `nil` means the item is deleted from the Out Box. The user sets this in the “After sending” preference. The default is `nil`.

`inboxFiling`
: A symbol indicating the In Box folder in which to file an item when it is received.

Specify a symbol representing a folder name, or `nil` to file incoming items in the Untitled folder. The symbol must represent an existing folder. Note that filing doesn't occur until after the In/Out Box is closed. The user sets this in the "File read items in" preference. The default is `nil`.

`outboxFiling`
A symbol indicating the Out Box folder in which to file an item after it is sent. Specify a symbol representing a folder name, or `nil` to file sent items in the Untitled folder. The symbol must represent an existing folder Note that filing doesn't occur until after the In/Out Box is closed. The default is `nil`.

`nowOrLater`
A symbol indicating what action the Send button in the routing slip should take when the user taps it. Specify the symbol `'now` to force the button always to send items immediately (corresponds to the "Send now" preferences choice). Specify the symbol `'later` to force the button always to send items later (corresponds to the "Send later" preferences choice). Specify `nil` to force the button to display a picker allowing the user to choose now or later each time (corresponds to the "Specify when" preferences choice). The user sets this in the "When Sending" preference. The default is `nil`.

cfgMsgModule A frame or `nil`. Pass in `nil` if you do not want to configure a specific msgModule. The frame contains the msgModule preferences to be set. The required msgModule must be installed in the system. The following slots are configurable:

`deviceSym`	Required. This must be the appSymbol of the msgModule that the msgModule registered with.
`powerIndex`	An integer. An index to the array in the msgModule slot `powerStrings`. If this index is out of range the default will be used. The default is 0.
`transmitterIndex`	An integer. An index to the array in the msgModule slot `transmitterStrings`. If this index is out of range the default is used. The default is 0.
`soundIndex`	An integer. An index to the array in the msgModule slot `soundStrings`. If this index is out of range the default is used. The default is 0.
`portIndex`	An integer. An index to the array in the msgModule slot `portStrings`. If this index is out of range the default is used. The default is 0.
`timeIndex`	An integer. An index to the array in the msgModule slot `timeStrings`. If this index is out of range the default is used. The default is 0.

If the preferences view is open when this message is received, the Messaging Enabler closes the preferences view and displays an alert to the user. The alert displays the string mapped to the `kRES_PREFSCHANGED` result code, see “Result Codes” beginning on page 1-11.

SendDirectCommand

messagingEnabler`:SendDirectCommand(`*callBack*`,`*options*`,`*command*`)`

Calls a msgModule’s `DirectCommand` method (page 2-20). See “Using the SendDirectCommand Method” (page 1-29).

IMPORTANT

This method should be called with `TransportNotify` function. The `TransportNotify` method is documented in Chapter 19, "Transport Interface Reference," in *Newton Programmer's Reference*.

Call it using this syntax:

```
TransportNotify(
         'MsgEnabler,
         'SendDirectCommand,
         [callBack, options, command]
               )
```

▲

callBack	A frame that is the callback to be executed once this method has completed. This contains two slots:	
	`receiver`	The receiver to which to send the message.
	`message`	A symbol naming the message to be sent. This method accepts two parameters *complFrame* and *complCode*.
	complFrame	Specific to the command being sent and may be any type of NewtonScript object. This should be documented by the particular msgModule for each command supported.
	complCode	An integer result code. See "Result Codes" (page 1-11).
options	A frame containing the following slots:	
	`open`	A Boolean. If `true` the msgModule attempts to open the message channel if it is not currently open.
	`deviceSym`	The appSymbol the msgModule registered with.

command	Specific to the particular msgModule. It is up to the msgModule developer to document the format this parameter should take.

The Messaging Enabler sets the *complCode* to the integer result code `kRES_UNSUPPORTED`, if the msgModule installed does not support the `DirectCommand` method. The *complFrame* is set to `nil` and the *complCode* to `kRES_FAILED` if the msgModule generates an error.

Index