Newton 2.0 Messaging Enabler - Get Your Messages Movin'

by Jason Rukman Apple Computer, Inc.

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

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

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

Your specialty may be developing communications software. The idea behind the Messaging Enabler is to remove the user interface requirement from the Original Equipment Manufacturer (OEM). This allows a consistent user interface for all wireless messaging products that use the Messaging Enabler.

Is it for you?

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

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

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

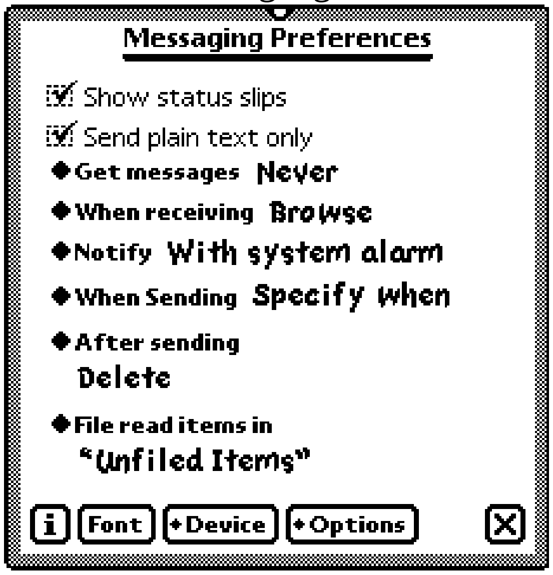
Message modules have many options that may be customized so you can support features specific to the messaging device being used.

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

Features

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

- Standardized preferences for wireless messaging.



Messaging Preferences

☑ Show status slips

☑ Send plain text only

◆Get messages Never

◆When receiving Browse

◆Notify With system alarm

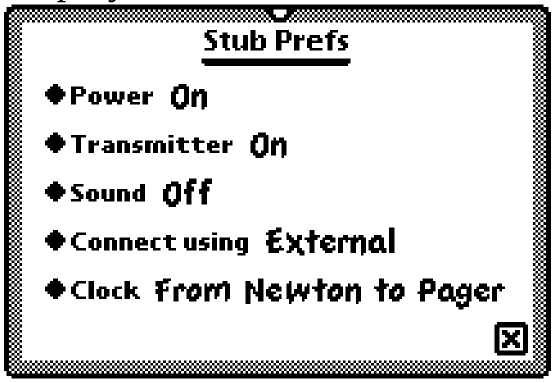
◆When Sending Specify when

◆After sending Delete

◆File read items in "Unfiled Items"

Figure 1: Messaging Enabler Preferences

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



Stub Prefs

◆Power On

◆Transmitter On

◆Sound Off

◆Connect using External

◆Clock From Newton to Pager

Figure 2: Device Specific Preferences

- Send routing slip(s).

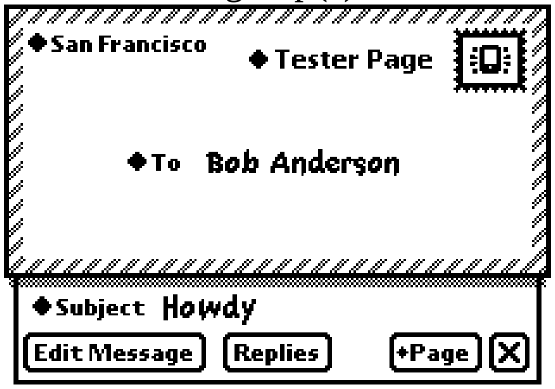


Figure 3: Paging Routing Slip

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

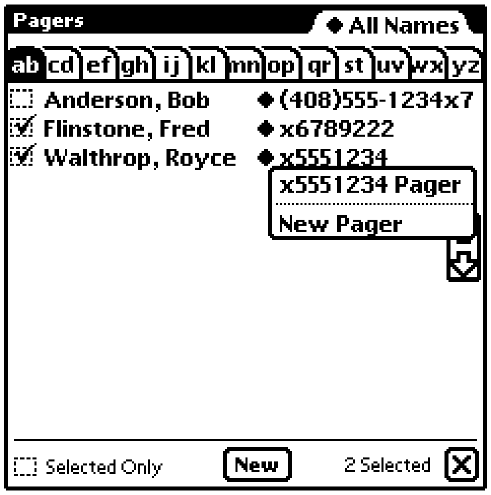

Pagers
◆ All Names
ab cd ef gh ij kl mn op qr st uv wx yz
Anderson, Bob ◆(408)555-1234x7
Flinstone, Fred ◆x6789222
Walthrop, Royce ◆x5551234
x5551234 Pager
New Pager
Selected Only
New
2 Selected

Figure 4: Pager Address Listpicker

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

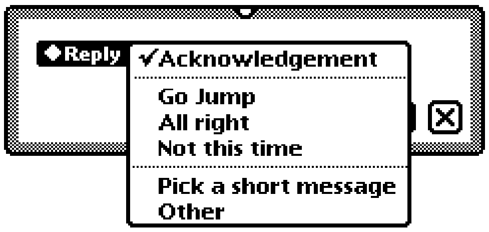
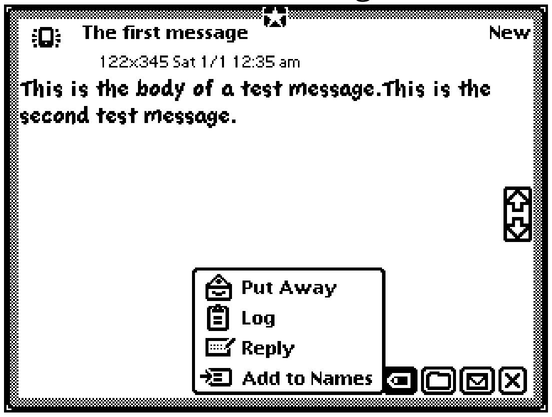


Figure 5: Message Reply Picker

- Automatic message retrieval.
- Automatic control and combination of multi-part messages.
- Text message viewing/editing/and put away.

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



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

Figure 6: Displayed In Box Message

- Manage and control the I/O Box

- User status feedback.

How it ticks

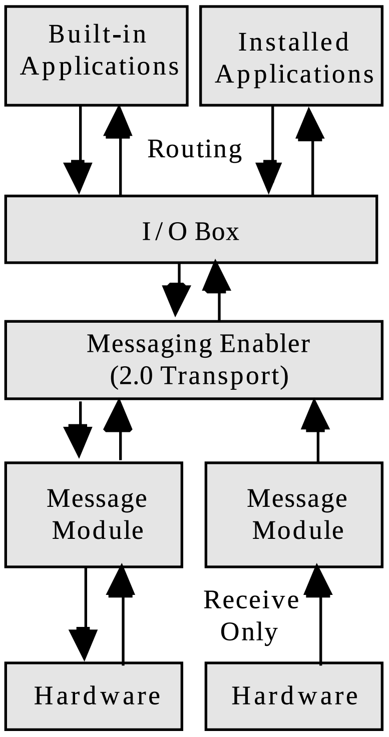


Figure 7: Messaging Enabler Hierarchy

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

Installing a message module

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

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

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

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

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

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

Working with callbacks and events

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

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

end;   // GetNextMessage
```

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

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

Need to send messages?

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

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

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

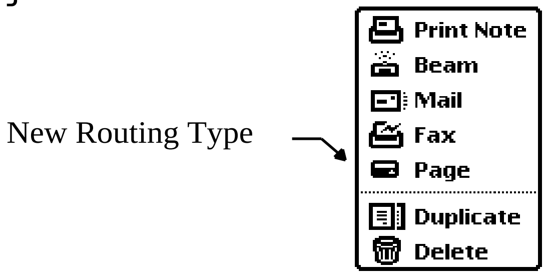


Figure 8: Notepad Routing Picker

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

Programmers Guide: Communications.

What's your preference?

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

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

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

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

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

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

Controlling the Messaging Enabler.

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

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

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

This does the following:

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

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

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

Give me those In Box items!

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

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

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

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