Communications

This Chapter Supplement describes how to develop communication tools for Newton based on the Motorola Radio Packet Modem Native Mode Endpoint.

Introduction

The Motorola Radio Packet Modem Native Mode Endpoint provides NewtonScript programmers with Motorola's radio packet modem (RPM) - facilitated access to DataTAC wireless communications networks.

This supplement to the Newton Programmer's Guide is written assuming that the reader is familiar with NewtonScript programming in general and the Newton Programmer's Guide, especially Chapter 14, "Communications", in particular. This Supplement is organized in the same sections as Chapter 14; the information contained in each of these sections is clarified or augmented as is appropriate for proper usage of Motorola Radio Packet Modem Native Mode Endpoint. [This endpoint may also be referred to here forward as the "RPMnative endpoint".]

The RPMnative endpoint supports packet, non-connection oriented communications on DataTAC networks. The radio packet modems, which are currently supported by the RPMnative endpoint, are:

- Motorola's "Eagle" DataTAC PCMCIA, in the Newton's PCMCIA slot.
- Motorola's Infotac, attached to the Newton's DIN8 serial port.

The RPMnative endpoint provides access to a radio packet modem's capability to send and receive messages between itself and wireless "Access Points" in a DataTAC network. Users of the RPMnative endpoint may then implement higher-level protocols, as may be appropriate, to provide their users with services such as "end-to-end" message transfer, and internetworked access to other wired and wireless networks, to mention a few possibilities. For example , access to Ardis™ or RAM™ networks and other subscribers of these networks would be provided by "transport"/higher-level software which would use the RPMnative endpoint.

The RPMnative endpoint also provides a “NewtonScript Frame Conversion Facility”. This facility provides services to “flatten” and “unflatten” NewtonScript objects of class `'frame`. This Conversion Facility is useful in the communications of NewtonScript `'frame` objects; these objects are typically complex with a variety of slots which are of NewtonScript classes that do not permit the complete inspection and manipulation of their underlying “byte representations” via normal NewtonScript object accessor functions. In other words, these complex `'frame` objects and their lengths are not fully accessible as “raw bytes” and thus may not be appropriately prepared for the exclusively packet-oriented data communications supported by the RPMnative endpoint.

“Flattening” refers to the conversion of a NewtonScript `'frame` class object into a NewtonScript object of primitive class `'array,` whose elements are exclusively of primitive class `'integer`, where each element contains an unsigned byte (0..255). “Unflattening” refers to the inverse of this operation.

Endpoints

The RPMnative endpoint is a standard NewtonScript endpoint, with a few additional features designed to provide efficient access to the inherently packet-oriented, connection-less communications facilities of the DataTAC wireless networks being accessed. Control of the Radio Packet Modem transmitter, receiver and other radio-related features are provided through standard endpoint option control.

The RPMnative endpoint is very similar to the other standard NewtonScript endpoints; it maintains the same single prototype — protoEndpoint — and same limitations as other endpoints.

Options

The specifics of the DataTAC wireless network access at hand, including certain communications parameters, such as transmitter control, battery saving control, control of asynchronous notification of network status changes, and so forth, may be set by supplying option frames in the `configOptions` slot of the RPMnative endpoint. Option frames are composed for the RPMnative endpoint as described in base document. Some options for the endpoint are “read-only”; for these options, the `GetOptions` method, described below, may be used to obtain certain information related to the endpoints operation, such as relative RF signal strength, radio packet modem serial number, etc.

It is important to note that for the setting of certain options, the changes desired, based on the values supplied, may take significant time for the endpoint to implement. This endpoint, unlike the Newton OS ROM-based endpoints, generates “events” when the changes requested in a `SetOptions` call are completely made. Thus, `SetOptions` Newton Script calls are effectively “asynchronous” for the RPMnative endpoint. While a `SetOptions` call is still being internally processed by the RPMnative endpoint, it is an error to issue another `SetOptions` call, and, if such a call is made, an exception will be thrown: The `SetOptions` “channel” is effectively busy during this processing.

Specifying the Service

To use the RPMnative service from NewtonScript, you specify the communication service by providing a `'service` option, in the endpoint frame. Every endpoint must have one `'service` option. The RPMnative service is specified as follows:

Constant	**Value**	**Service**
`kCMSRPMnative`	`"rpmn"`	RPMnative

Note
If any other `'service` options are specified in the RPMnative endpoint frame, this is an error and an exception will be thrown. •

Note [re: ongoing development]
The constant, `kCMSRPMnative`, or any of the other constant symbols, are not yet in a Newton Tool Kit (NTK) platform file; thus, the appropriate values must be used directly in NewtonScript programs that use the RPMnative endpoint [26jul94]•

Thus, to create an RPMnative endpoint, you would do as follows:

```
myRPMnativeEndpoint :=
{       _proto:protoEndpoint,
        configOptions:
        [       {       label:  kCMSRPMnative,
                        type:           'service,
                        opCode:         opSetRequired },
        ],
};
```

After specifying the `'service` option, you then may desire to set certain `'option` options, which depending on your particular usage of the RPMnative endpoint. The RPMnative endpoint does not support any `'address` options.

Note
If an `'address` option is specified in the RPMnative endpoint frame, it is ignored. There may be some sort of addressing-related option required in future releases; perhaps addressing in the sense of which wireless network to join. •

Let's look at the RPMnative service in more detail.

RPMnative Endpoints

RPMnative endpoints can involve a range of operating parameters: radio battery saving settings, event notification settings, and so on.

Here's an example of a typical usage:

```
myRPMnativeEndpoint :=
{       _proto:protoEndpoint,
        configOptions:
        [
        {       label:  kCMSRPMnative,
                type:   'service,
                opCode:  opSetRequired       },

        {       label:  kCMOrpmnPowerPrefParms,
                type:   'option,
                opCode:         opSetNegotiate,
                data:   [ 0,0,0,4                   //batterySaveLevel set to 4
                        ]},

        {       label:  kCMOrpmnEventParms,
                type:   'option,
                opCode:         opSetRequired,
                data:   [       1,          //rcvdMsgsEvtEnb = true
                                1,          //batteryEvtEnb = true
                                0,          //radioEvtEnb = false
                                0,          //onOffEvtEnb = false
                                1,          //svcEvtEnb = true
                                1,          //setOptCmplEvtEnb = true
                                1,          //outCmplEvtEnb = true
                                0           //DON'T CARE must be 0
                                        ]},
        ],
};
```

Note that one `'option` specification uses an `opCode` of `opSetNegotiate`, which means that if the requested values are not available, a reasonable substitute will be accepted.

Additionally, note that the data slot of each option contains an array of integers. There must always be a multiple of four integers. This is very important. Communications scripting under 1.x versions of the Newton Operating System does not support the passage of complex option `data` slots to the RPMnative endpoint. Options are passed as "generic", with the `data` slot containing an `'array` object whose elements are exclusively integers. This documentation specifies the exact order and interpretation of each of these integers as option data significant to the RPMnative endpoint. The upper three bytes of each integer are ignored; only the low byte of each integer is passed by Communications scripting down to the actual routines that interpret the options.

The RPMnative `'option` labels are as follows:

Label	**Value**
kCMOrpmnRadioParms	"rdio"
kCMOrpmnMessageSizeMaxParms	"mgsz"
kCMOrpmnEventParms	"evnt"
kCMOrpmnPowerMgmtParms	"pwrp"

kCMOrpmnModemIDParms	“mdid”
kCMOrpmnDisconnectParms	“dsco”
kCMOrpmnFrameCvtCtlParms	“fcvt”

General Parameters

The `kCMOrpmnRadioParms` option specifies certain wireless radio-related parameters. The `data` array of the option has several significant elements, which may be set to `0` or `1`, to disable or enable the corresponding features, respectively. The processing of this option may take significant time and the `kCMErpmnsetOptCmpl` event may be useful for the Newton Script programmer to determine when the `SetOption` call may be used again.

This option has the following form:

```
{       label:  kCMOrpmnRadioParms,
        type:   ‘option,
        opCode:         opSetNegotiate,         // or
        data:   [       1,                              //transmitterEnable = true
                        1,                              //receiverEnable = true
                        0,0,                            //DON’T CARE must be 0
                ]},
```

The following elements are present:

Element name	**Default Value**
`transmitterEnable`	`1`
`receiverEnable`	`1`

Note
It is not a valid Option setting to specify that the transmitter remain enabled and that the receiver be disabled. •

The `kCMOrpmnMessageSizeMax` option is a read-only option which has one effective item in its `data` array in which the maximum size of a message in bytes is returned. The processing of this option is immediate. This option has the following form:

```
{       label:  kCMOrpmnMessageSizeMax ,
        type:   ‘option,
        opCode:         opGetCurrent,
        data:   [       0,0,0,0                 //messageSizeMaxBytes..
                                                //..returned here
                                ]},
```

Element name	**Value**
`messageSizeMaxBytes`	*`integer`* `extracted as data[0] << 24 + data[1] << 16 + data[2] << 8 + data[3]`

Event Parameters

The `kCMOrpmnEventParms` lets you specify which endpoint events should be enabled, resulting in calls to the method in your `EventHandler` slot, as described below. The processing of this option is immediate. This option has the following form:

```
{       label:  kCMOrpmnEventParms ,
        type:   'option,
        opCode:         opSetRequired,
        data:   [                       1,      //rcvdMsgsEvtEnb = true
                                        1,      //batteryEvtEnb = true
                                        0,      //radioEvtEnb = false
                                        1,      //onOffEvtEnb = true
                                        1,      //svcEvtEnb = true
                                        1,      //setOptCmplEvtEnb = true
                                        1,      //outCmplEvtEnb = true
                                        0       //DON'T CARE must be 0
                                ]},
```

Use the following elements in your `kCMOrpmnEventParms` Option frame's `data` array to access these parameters:

Element name	**Default Value**
rcvdMsgEvtEnb	1
batteryEvtEnb	0
radioEvtEnb	0
onOffEvtEnb	1
svcEvtEnb	1
setOptCmplEvtEnb	1
outCmplEvtEnb	1

Power Management Parameters

The `kCMOrpmnPowerMgmtParms` lets you specify the power conservation/management setting in the underlying radio packet modem. The processing of this option may take significant time and the `kCMErpmnSetOptCmpl` event may be useful in determining when the `SetOption` call may be used again. This option has the following form:

```
{       label:  kCMOrpmnPowerMgmtParms ,
        type:   'option,
        opCode:         opSetNegotiate,
        data:   [       0,0,0,3                 //batterySaveLevel = 3
                ]},
```

Use the following elements in your `kCMOrpmnPowerMgmtParms` Option frame's `data` array to access these parameters:

Element name	**Default Value**
batterySaveLevel	4

The `batterySaveLevel` element specifies "how hard" the endpoint, together with the radio packet modem, will try to save modem batteries. This can vary between 0, for no battery saving, but highest network responsiveness, to 5, for maximum battery savings, at a possible cost of delayed receipt and transmission of messages.

Note
The current Motorola radio packet modems only implement a binary choice for power-saving modes. Thus, the RPMnative endpoint activates their power-save features for any non-zero value of `batterySaveLevel`.•

Modem Identification Parameters

The `kCMOrpmnModemIDParms` option is an option which cannot be set. The `data` array of the option has several informational elements. The processing of this option is immediate. This option has the following form:

```
{       label:  kCMOrpmnModemIDParms ,
        type:   'option,
        opCode:         opGetCurrent,
        data:   [       0,0,0,0,                //modemHexIDStr returned
                        0,0,0,0,                //modemVersion
                ]},
```

The following elements are present:

Element name	**Value**
`modemHexIDStr`	`*integer* representing 8 packed hexidecimal digits extracted as data[0] << 24 + data[1] << 16 + data[2] << 8 + data[3]`
modemVersion	4 ASCII digits

Disconnect Control Parameters

The `kCMOrpmnDisconnectParms` option allows for the control of the actions the RPMnative endpoint takes in its execution of its `Disconnect` method. The processing of this option is immediate.

This option has the following form:

```
{       label:  kCMOrpmnDisconnectParms ,
        type:   'option,
        opCode:     opSetRequired,
        data:   [   0,0,0,           // DON'T CARE must be 0
                    1     // disconnectLeavingRPMEnabled = true
                ]},
```

The `data` array of the option has one element:

Element name	**Default Value**
`disconnectLeavingRPMEnabled`	1

This parameter provides you with a way to control whether you would like to keep the Radio Packet Modem "on the air" or not after the endpoint has been disconnected. Certain model RPMs can be kept "on the air" and function independently of the Newton, as "pager-like" devices.

Frame Conversion Control Parameters

The `kCMOrpmnFrameCvtCtlParms` lets you control the NewtonScript Frame Conversion Facility in the RPMnative endpoint. This Facility has three primary modes: Idle, the default mode, Flattening, and Unflattening. Each mode has substates that you both explicitly control and implicitly control by using the `GetOptions` method. These substate settings must be coordinated with your usage of the endpoint's `Output`, `OutputFrame`, and `SetInputSpec` methods. The processing of this option is immediate. This option has the following form:

```
     {    label:  kCMOrpmnFrameCvtCtlParms,
          type:   'option,
          opCode:     opSetRequired,
          data:   [   0,0,0,0,    //conversionMode = kcvtModeNoChg
                      0,0,0,2,    //substate = kcvtSubstIn
                      0,0,0,0,    //byteCount
                  ]},
```

All three elements are integers which must be extracted from the 4 NewtonScript integers, e.g.

```
     conversionMode = data[0] << 24 + data[1] <<16
                    + data[2] << 8 + data[3]; .
```

Use the following element in your `kCMOrpmnFrameCvtCtlParms` Option frame's `data` frame to access the parameters which must be controlled and inspected during the Frame Conversion process:

Element name	**Default Value**
conversionMode	kcvtModeNoChg (0)
substate	kcvtSubstUndef (0)
byteCount	0

The `conversionMode` element specifies what mode you wish to set for the Conversion Facility. You can use the following values in this slot:

Constant	Value	Meaning
kcvtModeNoChg	0	do not change mode
kcvtModeIdle	1	no conversion in progress or allowed
kcvtModeFlatten	2	flattening operation in progress
kcvtModeUnflatten	3	unflattening operation in progress

The `kcvtModeNoChg` value, which is only used with the `SetOptions` method, specifies that the current mode should not be changed. The `kcvtModeIdle` value, specifies the current mode to be the Idle mode; in this mode the Conversion facility is not active. The `kcvtModeFlatten` value, specifies the current mode to be the "flatten" mode, in which the Conversion facility is active in the conversion of an object of class `'frame` into bytes. The `kcvtModeUnflatten` value, specifies the current mode to be the "unflatten" mode, in which the Conversion facility is active in the conversion of bytes into an object of class `'frame`.

The `substate` element specifies what substate you wish to set for the Conversion Facility. You can use the following values in this element:

Constant	Value	Meaning
kcvtSubstUndef	0	substate is undefined
kcvtSubstOut	1	accepting data for conversion
kcvtSubstIn	2	providing converted data

The `kcvtSubstUndef` value, which is only seen with the `GetOptions` method, specifies that the current substate is undefined. The `kcvtSubstOut` value, specifies the current substate to be the Output mode; in this mode the Conversion facility is ready to accept data for conversion. The `kcvtSubstOut` value, specifies the current substate to be the Output mode; in this mode the Conversion facility is ready to accept data for conversion.

You inspect the `byteCount` element while making certain substate transitions during the Conversion procedure to determine the number of bytes in a flattened representation of a NewtonScript `'frame` object.

You should implement the following procedure to convert a `'frame` object into a flattened array of bytes:

a) Make sure you have no active input specification. [If you do, either wait for it to complete, i.e. your method in its `InputScript` slot is called, or cancel it by setting the endpoint's `nextInputSpec` slot to `nil` and then calling your endpoint's `SetInputSpec` method with a `nil` argument.]

b) Use your endpoint's `SetOptions` method to set `conversionMode` to `kcvtModeFlatten`. [This implicitly sets the `substate` to `kcvtSubstOut`.]

c) Use your endpoint's `OutputFrame` method with the frame to be flattened as the first *`data`* argument, and `nil` as the value of the *`flags`* argument.

c1) Use your endpoint's `Output` method to output a single integer, with `nil` as the value of the *`flags`* argument. The value of this integer is a "don't care".

c2) Use your endpoint's Flush`Output` method to allow the endpoint to finish processing..

d) Use your endpoint's `GetOptions` method to get the current value of the `kCMOrpmnFrameCvtCtlParms` options; the `byteCount` element of the `data` array returned will have the number of bytes produced in the conversion of the frame;. [This implicitly sets the `substate` to `kcvtSubstIn`.]

e) Form an input specification without a `recvFlags` slot, with a `inputForm` slot set to `'raw`, and a `byteCount` slot equal to the number of bytes obtained in the `GetOptions` method call. Use the endpoint's `SetInputSpec` method to activate this input spec.

f) Your input specification's `inputScript` will be invoked in very short order. The *`data`* argument will be the flattened frame data, which will have been formatted as `'raw`. Now, you may do what is necessary to continue the data communication of this converted frame (e.g. packetization, sequencing, checksum formation, etc.).

g) Finally, you must use your endpoint's `SetOptions` method to set `conversionMode` to `kcvtModeIdle`; this completes your usage of the Conversion Facility for the flattening of the frame.

On the receiving side, once you have received an array of bytes, which you know to be a complete NewtonScript `'frame` object, and which was flattened and then transmitted previously (i.e. via the protocols you have defined and implemented), you should implement the following procedure to convert this array of bytes back into a `'frame` object:

a) Make sure you have no active input specification. [If you do either wait for it to complete, i.e. your method in its `InputScript` slot is called, or cancel it my setting the endpoint's `nextInputSpec` slot to `nil` and then calling your endpoint's `SetInputSpec` method with a `nil` argument.]

b) Use your endpoint's `SetOptions` method to set `conversionMode` to `kcvtModeUnflatten`. [This implicitly sets the `substate` to `kcvtSubstOut`.].

c) Use a single invocation your endpoint's `Output` method with your received array of bytes that is a flattened `'frame` object as the first *`data`* argument, and `nil` as the value of the *`flags`* argument.

c1) Use your endpoint's `Output` method to output a single integer, with `nil` as the value of the *`flags`* argument. The value of this integer is a "don't care".

c2) Use your endpoint's Flush`Output` method to allow the endpoint to finish processing.

d) Use your endpoint's `SetOptions` method to set the `substate` to `kcvtSubstIn`.

e) Form an input specification without a `recvFlags` slot, with a `inputForm` slot set to `'frame`. Use the endpoint's `SetInputSpec` method to activate this input spec.

f) Your input specification's `inputScript` will be invoked in very short order. The *`data`* argument will be the `'frame` object. Now, you may do what is necessary to process this now-normal NewtonScript object

g) Finally, you must use your endpoint's `SetOptions` method to set `conversionMode` to `kcvtModeIdle`; this completes your usage of the Conversion Facility for the unflattening of the frame; wait for `kCMErpmnSetOptCmpl` event.

Instantiating the RPMnative Endpoint

After you've defined your endpoint and supplied the necessary options in the `configOptions` slot, you then instantiate the RPMnative endpoint using the same procedures as for any other endpoint.

Establishing a Connection

The RPMnative endpoint defines connection in a way that is different than other endpoints, such as the modem or MNP serial endpoints. There is a "connection" involved, but it is not a connection to another peer endpoint. Rather, in this context, "connection" denotes the connection of the RPMnative endpoint to the "ether" of the wireless radio network

Establishing a connection thus represents getting the endpoint attached to the wireless DataTAC network.

To initiate this connection process, call `Connect`, using standard endpoint procedures.

Note
If `Listen` is called in an attempt to establish a connection to the wireless network, this is an error and an exception will be thrown. •

Sending Data

Output

endpoint : Output (*data* , *flags*)

As is the case for all endpoints, two methods are provided for sending data using the RPMnative endpoint: `Output` and `OutputFrame`. However to send a frame, keeping the structure intact and without any conversion, `OutputFrame` can be used but such usage is problematic, in that the conversion inherent in this method of a NewtonScript frame into transmittable bytes may produce more data than that amount which can be sent in a RPMnative radio packet. The correct method to send a NewtonScript frame is to use the RPMnative endpoint's Frame Conversion Facility to flatten it into "bytes" and then sent these bytes using the `Output` method.

Both `Output` and `OutputFrame` have a second parameter, `flags`, which must not be set to `nil`. The RPMnative endpoint only supports the end-of-message (EOM) flag communications; the communications services this endpoint provides are packet in nature. Thus, to send data with the RPMnative endpoint, you must either specify `kFrame + kMore`, if additional output is to come (for the message currently being "composed"), or just `kFrame`.

It is important to note that the first byte of every message output will be sent to the RPM radio as the NCL-defined "Send Options Bit Field byte", which is defined in the NCL spec under the discussion of the Command Class SDU "Send Message(SEND)". The layout of this byte is as follows:

MSB							LSB
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	1	0	0	0	- priority-		resend

where Bit7->Bit4 are reserved, Bit2->Bit1 are priority [00 is "normal", 01 is "very low", 10 is "low", and "11" is "high"], and Bit 0 is the "RESEND" setting. Please see the NCL specification for more information.

If `nil` is specified as the value of the `flags` parameter, this is not a data communications operation; depending on the state of the NewtonScript Frame Conversion Facility, which was detailed in an above section of this chapter supplement, this may or may not be an error.

For example:

```
sendOptionsByte := 0x42;      //1st byte of the message is..
                              //..the "Send Options byte" with.
                              //..resend bit clear and the..
                              //.."priority" set to "very low"

endpoint:Output(sendOptionsByte, kFrame + kMore);         //Don't..
                                                          //..forget to output this byte
endpoint:Output("MG", kFrame + kMore);
endpoint:Output("2832531", kFrame + kMore);
```

endpoint:Output("sometext", kFrame + kMore);
endpoint:Output("endofArdismessageMark", kFrame);

Note
It is important that Newton Script programmers use, and write their software to be adaptive to, the maximum size of a message, as is obtained from a `GetOptions` call with label `kCMOrpmnMessageSizeMax`.•

IMPORTANT
For the RPMnative endpoint `OutputFrame` should not be used for data communications; it should only be used in conjunction with the RPMnative endpoint's NewtonScript Frame Conversion Facility, as discussed above. It is important to note, that if you were to try to use `OutputFrame` with flags including `T_FRAME`, that `OutputFrame` may produce more data bytes than the maximum size of the message supported by the RPMnative endpoint. This would result in an exception being thrown. Use `GetOptions`, with label `kCMOrpmnMessageSizeMax`, to get the maximum for the currently instantiated endpoint. •

Receiving Data

As with all endpoints, to receive data, you need to provide an input specification. The RPMnative endpoint supports input specification termination on detection of EOM from the radio packet modem, attached to the wireless network. This is the intended usage mode of the RPMnative endpoint; an entire wireless network message is to be received using one input specification. This effectively places limitations on the values of the slots of this input specification, as is discussed in the next paragraph.

The slots of an input spec, as it is called, defines an input block, and can have the following slots:

`recvFlags`
: A control field. For RPMnative, this must be set to `kFrame`. This is slot is omitted during the usage of NewtonScript Frame Conversion Facility.

`byteCount`
: A byte count. For RPMnative, this should be larger than the maximum network message size, i.e. at least 2049.

`endCharacter`
: A termination character. For RPMnative, this should be `nil`.

`discardAfter`
: For RPMnative, this must be set to the same value as the `byteCount` slot.

`inputScript`
: A method to be called when a block has been received.

`inputForm`
: A format in which to return data. You can specify:

`'string` to return a character string, converting ASCII into Unicode as required.

`'raw` to return a raw byte stream. These bytes are not converted into Unicode. This is (probably) the preferred method using RPMnative.

`'frame` should only be used during the usage of NewtonScript Frame Conversion Facility.

`'array` to return an array of integers, each corresponding to one byte of input. This format is the default format. This is also a useful method using RPMnative.

`partialScript`

For RPMnative, this should be `nil`.

`partialFrequency`

For RPMnative, this should be `nil`.

After defining your input spec, you set it in action by calling `SetInputSpec`, as is the standard procedure with all endpoints. Since your input spec frame had `recvFlags: kFrame` in it, your `inputScript` is called with the complete received message passed to it, i.e. your `inputScript` is terminated by the internal endpoint interpreter when it has received an end-of-message (EOM) indication. You may obtain the size of this message either from the data itself using protocols you have defined, or by using a NewtonScript array function, such as `Length()`.

<u>Receiving Partial Input</u>

The RPMnative endpoint does not support the reception of partial input; the `PartialScript` slot of your input spec should always be nil. Additionally, you should not call `Input()`, `Partial()`, `FlushInput()`, or `FlushPartial()`. If you would like to discard input data, you must do this in your program.

Handling Events

The RPMnative endpoint provides a mechanism to give you notification of certain communications related events. These events include those related to the arrival of messages in the radio packet modem, changes in wireless network coverage, and also changes in the battery level in the modem, to mention a few.

By using either the `configOptions` slot in your endpoint frame or the `SetOptions` method, you may enable the generation of event information by the RPMnative endpoint.

To process these status events, you should define a `EventHandler` slot in your endpoint frame, which contains a method to be called when an event occurs.

EventHandler

endpoint : EventHandler (*eventKind* , *eventData*)

If provided in your endpoint frame, this method is called whenever a event, whose generation has been enabled via Option control, occurs.

`eventKind`

An integer code which identifies that a tool-specific event which has occurred. This is always, T_SPECEVT, which has a value 1.

`eventData`

An integer which provides information concerning the RPMnative event; it both identifies the event and supplies 16 bits of event-type-specific information in the low half word.

The list of *`eventData`* values is as follows, where XX in the literal or XXXX in the hex value represent a half-word (16 bits) of additional event-type-specific information:

Label	**Value**
kCMErpmnRcvdMsgs	'\00mXX' (0x006DXXXX)
kCMErpmnBattery	'\00pXX' (0x0070XXXX)
kCMErpmnRadio	'\00rXX' (0x0071XXXX)
kCMErpmnOnOff	'\00oXX' (0x006FXXXX)
kCMErpmnSvc	'\00sXX' (0x0073XXXX)
kCMErpmnOutCpml	'\00xXX' (0x0078XXXX)
kCMErpmnSetOptCmpl	'\00cXX' (0x0063XXXX)

`For kCMErpmnRcvdMsgs` events, the *`eventData`* low half-word is the number of messages available, locally in the Newton and/or radio packet modem.

For `kCMErpmnBattery`, the *`eventData`* low half-word represents the relative charge condition of the radio packet modem's battery, where 0 is dead and 100 is fully changed.

For `kCMErpmnRadio`, the *`eventData`* low half-word represents the relative quality of the radio packet modem's current assessment of signal strength, where 0 is no signal and 50 is best.

For `kCMErpmnOnOff`, the *`eventData`* low half-word is 0 if the radio has been turned off.

For `kCMErpmnSvc`, the *`eventData`* low half-word is 1 if there is network service available, a 0 if there is not.

For `kCMErpmnSetOptCmpl` and `kCMErpmnOutCpml`, the *`eventData`* low half-word is a Newton Script error code, which reports the outcome of the processing of a `SetOptions` or `Out-put` call, respectively. This half-word should be OR-ed with 0xFFFF0000 to produce the negative integer Newton error code.

Changing or Adding Options

As is standard practice with endpoints, you’ll want to set most options in the `configOptions` slot of the endpoint. If you want to change or set certain options after the endpoint has been instantiated, use the `SetOptions` method. You can determine current or default values of one or more options by calling `GetOptions`.

Handling Exceptions

The RPMnative endpoint’s exception handling is no different than any other endpoint. You can also provide an `ExceptionHandler` method in your endpoint frame.

Closing a Connection

The RPMnative endpoint operates no differently than any other endpoint.

DIS-

TRIBUTION/CHANGE HISTORY:

v0.2 26 July 94 1316
Copies to RM and Apple Program Management (Steve).
v0.201 26 July 94
typo corrections
v0.202 26 July 94 1556
typo corrections; added r/o option `kCMOrpmnMessageSizeMax`
v0.203 02 Aug 94 1424
typo corrections; moved `recvFlags:kFrame` specification from endpoint frame to input specification frame; changed values of events to be 30bit due to sign extension in NewtonScript; changed range of `eventInfo` for `kCMErpmnBattery` to 0..100; changed range of `eventInfo` for `kCMErpmnRadio` to 0..50; changed default value of `radioEvtEnb` for `kCMOrpmnEventParms` to `true`; added event `kCMErpmnOnOff` and `kCMErpmnSvc` ; Copies to RM, Apple Prgm.Mgmt. (Steve), & internal.
v0.204 08 Aug 94 1040
typo corrections; added r/o options `kCMOrpmnDisconnectParms` and `kCMOrpmnFrameCvtCtlParms`, extensive discussion of Frame Conversion Facility and an example of its usage. Copies to RM , Apple Prgm.Mgmt. (Steve), & internal Apple.
v0.205 16 Aug 94 1633
typo corrections; reworks the details of the `data` slot of the options to pass all information as integers as is necessary in 1.x versions of the OS. Copies to internal Apple.
v0.206 18 Aug 94 1633
typo corrections; changed value of event codes to avoid sign extension problems in NewtonScript interpreter. Changed specification of d`iscardAfter` slot of input specs. Copies to RM, Apple Prgm.Mgmt. (Steve), & internal Apple.
v1.0a1 13 Sept 94 1055
typo corrections and misc. cleanup; added concept of “asynchronous” `Output` and `SetOptions` and the events related to them; made `Listen` call an error. Copies to Apple Prgm.Mgmt. (Steve, Leo), & internal Apple.
v1.0a2 1 Nov 94 1055
Changed the definition of the arguments to EventHandler: *`eventData`* now contains all the useful information. Added new steps “c1)” and “c2)” to the discussion of frame conversion both to and from a flattened array of bytes.

Copies to RM, PCOE, Apple Prgm.Mgmt. (Steve, Leo), & internal Apple.

v1.0a3   7 Mar 95 955

Changed to reflect that NewtonScript user must send the “Send Options Byte” as the 1st byte Output in every message. This correction reflects the sematic change made to “balance” the message content of sent and received messages.

v1.1b3   20 Apr 95 1627

Changed to have document rev, i.e. “1.1b3” match the software release in Beta test.