Newton Programming: Communications Overview

The Newton operating system is designed with communications as an integral part of the system. The pervasive approach is that whatever you can see in a Newton application, you can send. Part of this approach is the notion that, as much as possible, the user will have a very similar experience in sending data regardless of the medium used to send it.

From a programming point of view things are not quite so simple but the architecture is designed in a layered way so that little programming is required unless the situation is fairly unusual. In other words, there is a great deal of built-in communications software in the Newton which can be used to provide basic communications functionality for almost any program.

Figure 1 shows the various layers comprising the Newton communications system and the programming interfaces used to access these elements. The rest of this article gives brief descriptions of these APIs as well as providing references to more details about them.

2.0 Communications Overview

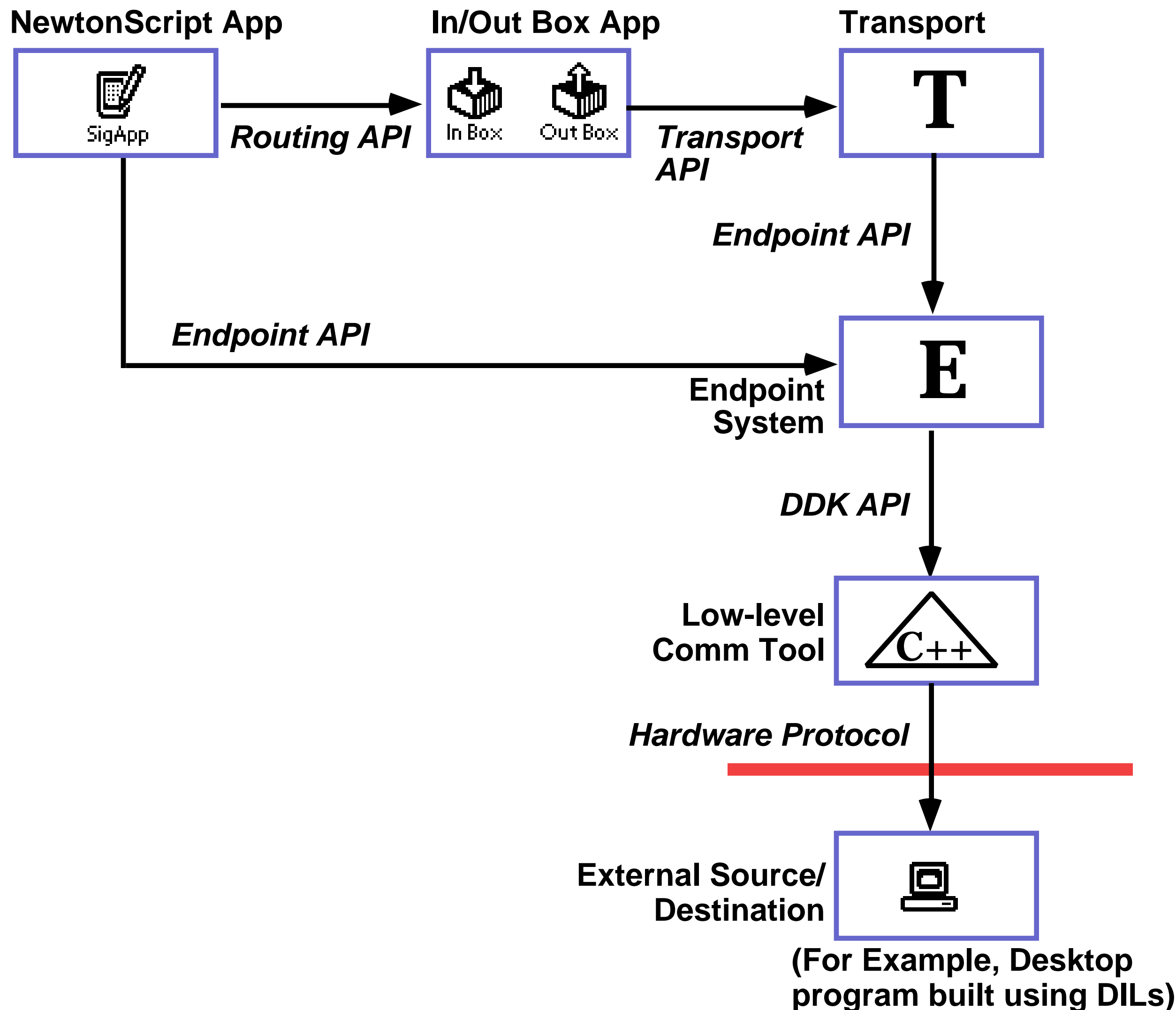


Figure 1: Newton Communications Layers

The following is a brief description of the items shown in Figure 1 and their APIs.

A NewtonScript application using the Routing API is the simplest way for application programmers to provide communications support from a Newton device. Any application written in NewtonScript can use communications modules which have been installed as Transports. In

Newton 2.0 OS, this includes the built-in transports for beaming, faxing, mailing, and printing. To use the available transports, the Routing application programming interface (API) is used to specify which data is being “routed”, what form it takes and how it should appear (for example, print format). The Newton 2.0 OS uses a store-and-forward model for this kind of communications and the In/Out boxes are where incoming or outgoing data is stored in the routing model.

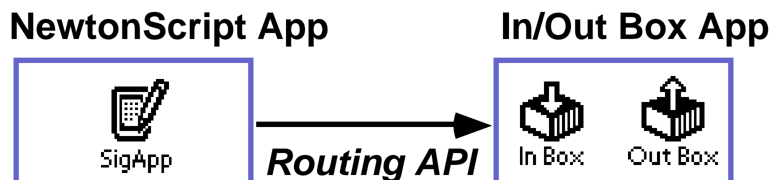
In/Out Box Application and Transport API. Built into the system is the In/Out Box application which manages the soups used to store incoming and outgoing data. This application communicates with one of several Transports, which consist of code provided to move the data to or from the appropriate destination or source. While there are several built-in transports in the system, NewtonScript programmers may write their own transport to provide system-wide data management.

Endpoint API and Endpoint System. The Endpoint API is a NewtonScript interface for performing direct communications with the outside world. Applications programmers may add endpoint code to their programs to communicate directly with an external source or destination. An example of this might be endpoint code which communicates directly with a GPS device on demand. Transports have endpoint code to move the data they receive out to an external destination or to receive data from an external source prior to passing it to the In/Out Box Application.

Low-level Communication Tools . These tools are implemented in C++ and actually communicate with the C++ interfaces to the hardware drivers. While these interfaces are not yet available, they will be published in the future.

Each of the APIs will be described in more detail in the remainder of this article.

Routing



2.0 Communications Overview

The Newton OS is provides a store-and-forward model for communications and uses the In/Out Boxes as the place where target data is stored. The term **store and forward** means that messages are routed (directed) to a distant communications device or from such a device to a Newton application through an intermediate holding area (the In/Out Boxes). Target data is any piece of information which is being routed in or out of the Newton.

The In/Out Boxes are actually a single application which provides the storage in the form of a soup, the functionality of sending and receiving messages, and the interface that lets the user look at pending message and dispose of them as he or she desires. Figure 2 shows what the In/Out Boxes might look like when there are messages pending. Note that at any time the user can switch from In Box to Out Box and vice versa via the radio buttons at the top of the view.

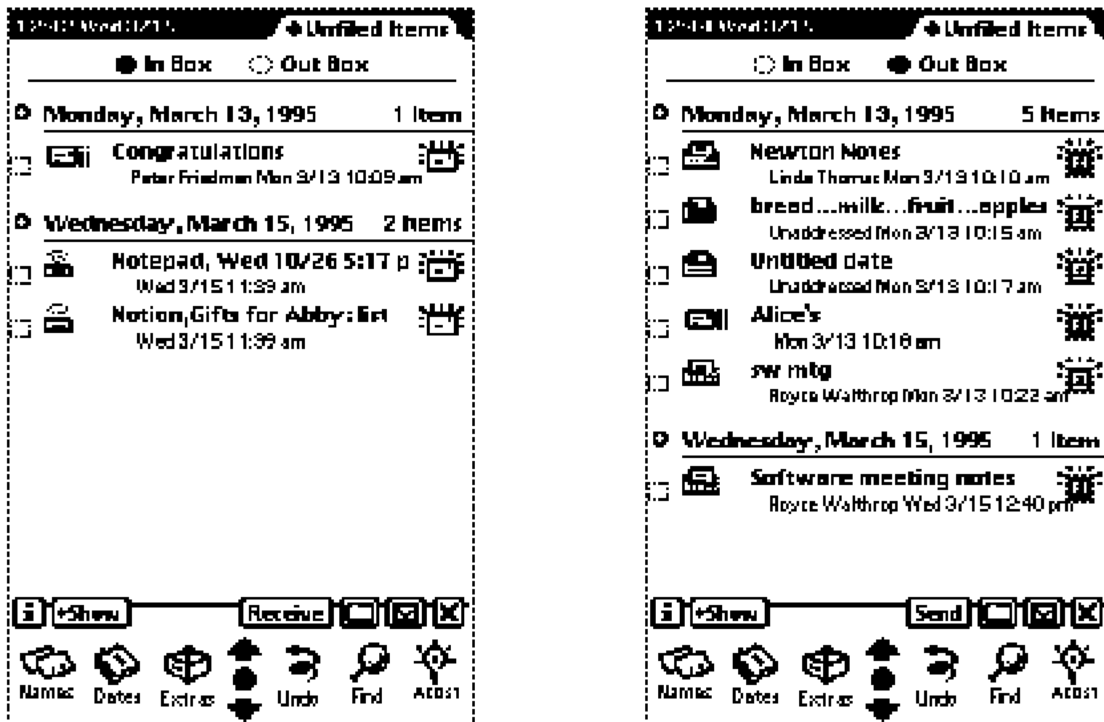


Figure 2. The In/Out Box User Interface

Routing is usually triggered by using the Action button that is displayed in the view from which something will be routed. The Action button is displayed in the view as a pop-up which shows available user actions as illustrated in Figure 3. Some applications will have one Action button in the status bar, others will have one in each of several views. The Names application is an example of a single Action button because normally only one name at a time is viewed. The Notes application has an Action button attached to each note since there may be many notes on the screen at any given time.

The Action button is created on screen by adding the prototype `protoActionButton` to the desired view.

Figure 3. The Action Button Popup

Each target object which is routed must have a meaningful class. For frames, this means that the frame must have a class slot which identifies the type of data associated with this kind of object. Normally, each application will supply its own class of data for routing, such as '|myData:MYSIG|. This class is used by the system to look up in the Data View Registry the list of routing frames which may be used to route data of a specific class. From these routing frames, a list of transports or communication methods (for example, faxing, printing, beaming) which can route the target data are supplied to the Action button. The net result of this is that when the user taps on the Action button a list of the destinations appear which are appropriate for the target data.

Figure 4 shows how this is all interconnected. An application, usually in its InstallScript, will put one or more frames named for the classes of data which it will route into the Data View Registry. These Frames will consist of one or more Routing Frames which describe what format the target data can

take when it is routed. The system uses this to search the list of installed transports and, when it finds a transport which supports one of the routingTypes, adds the transport name to the list to be displayed in the Action button.

So in the example shown in Figure 4, the application has installed a frame |forms:MYSIG| in the View Definition Registry which supports dataTypes of 'view, 'frame and 'text. These are used to choose the transports for printing, faxing, beaming and mailing so these appear in the Action button the user has pressed. Note that it doesn't pick up the compress transport whose dataType is 'binary.

Routed Object

```
targetData:={
      .
  class:'|forms:MYSIG|
      .
      .
      .
}
```

Routing Formats in Data View Registry for application forms:MYSIG

```
|forms:MYSIG|:{
twoColFormat:{dataTypes:['view]
...},
zapFormat:{dataTypes:['frame]
...},
mailFormat:{dataTypes:['text,'frame]
...},
};

format2:{...}

format3:{...}

                    <etc.>
```

Installed Transports

```
[printTransport:{dataTypes:['view]
...},
faxTransport:{dataTypes:['view,'text]
...},
beamTransport:{dataTypes:['frame]
...},
mailTransport:{dataTypes:['text]
...},
compressTransport:{dataType:['binary]
...},
]
```

Figure 4. The Routing System

When the user selects a transport from the Action button, an appropriate routing slip is displayed and all formats that in which the data can be displayed are displayed in the format picker as shown in Figure 5. Formats describe how the target data should be organized before sending it onward to the appropriate destination. For example, when printing, there might be several formats such as letter, memo, two-column, and so on, which describe how the target data will be printed.

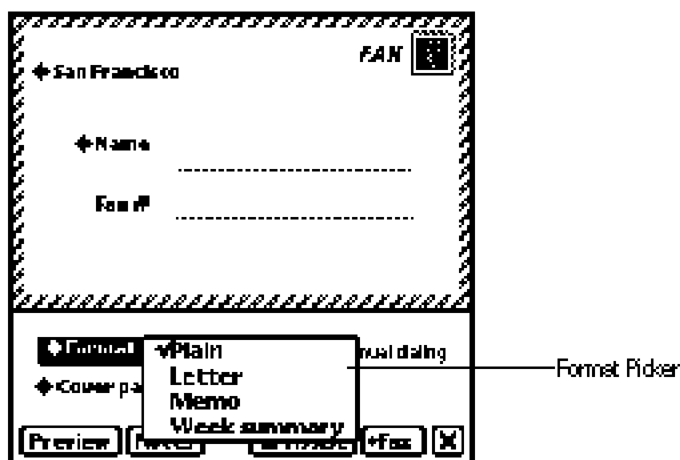
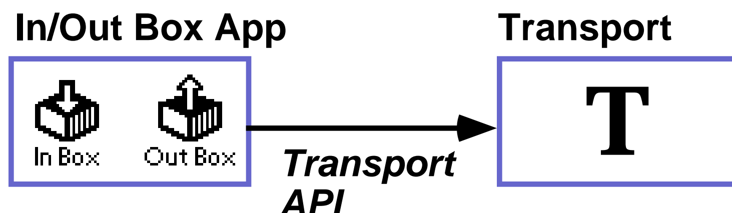


Figure 5. The Format Picker

When the user has selected a format for the target data and sent it off, the appropriate transport is then messaged with information about the target data and the data is placed in the Out Box for further disposition.

Transports

2.0 Communications Overview



The simplest definition of a transport is – something to which data can be routed. But a more useful definition is that a transport is a globally available service offered to applications for sending or receiving data. Because of the global nature of transports, it is not necessary, or even likely, for an individual application to define a transport.

The built-in transports include printing, faxing, mailing, and beaming, but one might imagine additional transports such as messaging, scanning, compressing, archiving, or encrypting. Thus, while transports are usually associated with hardware (printers, mail servers, and scanners, for example) this is not necessarily the case (e.g., compressing, archiving, encrypting), since a service may be offered that alters the data being routed without sending it to any outside hardware.

Transports are usually built as auto-load parts; they appear in the Extensions folder of the Extras Drawer. A transport's InstallScript registers it with the system by calling the global function `RegTransport()`. As described earlier in the routing discussion, if appropriate target data is routed, the transport's name will appear in the action list in an application when the user taps the Action button.

Because most transports have communications code that will be used to send or receive the target data, they will typically also include endpoint code that communicates with the destination.

Transports usually work with an application via the In/Out Box application. When an application routes data out to a transport, the transport provides a routing slip and is notified when the routed data reaches the Out Box. When items are sent, the transport will get the data from the Out Box and do whatever is necessary to send the data, setting status information at appropriate stages during the transfer.

In the case of a request to receive data, the sequence is just slightly more complicated. In the simplest case, when the user selects a transport from the

Receive button list in the In Box, the selected transport is sent a request to receive data. The transport will then connect to the remote source, get any pending data, and add it to the In Box list.

There is also an option for a transport to get information about the data being routed from the remote source and post this information into the In Box without actually getting the data. This is useful in a situation such as a mail transport where the user often wants to simply get the titles of pending messages so he or she may choose which messages they want to download to the Newton device.

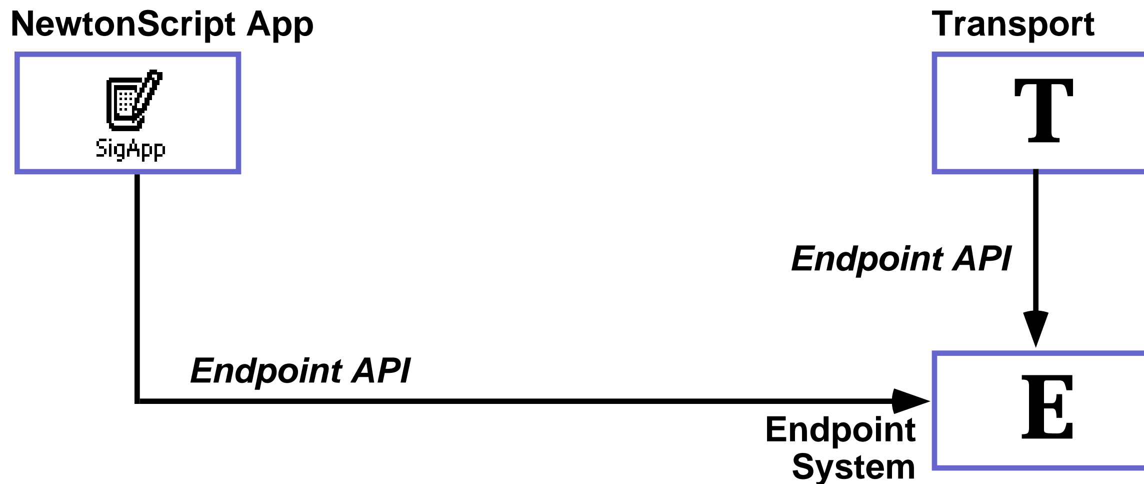
The main proto used to create a transport is `protoTransport`. The powerful thing about `protoTransport` is that in many cases, surprisingly little code other than the actual endpoint code must be written. This is because the transport defaults typically “do the right thing” to provide an interface and behavior for the transport. Only those features specific to the transport (for example, archive name for an archive transport) must be added to the standard interface.

Transports which can send data also have their own routing slips based on the prototype `protoFullRouteSlip`. This allows users to provide transport-specific options such as addresses in a particular format, and so on.

In particular, such things as displaying the status of a routing request, logging of routed items, error handling, power-off handling, and general user interfaces are handled well by the defaults if the transport simply sets or updates a few slots when appropriate. Only the actual service code (such as communications) will differ from one transport to another.

Endpoints

2.0 Communications Overview



Endpoints are the primary NewtonScript API for programming communications on the Newton device. They provide a “virtual pipeline” for all communications. They are designed to hide most of the specifics of a particular communications media and, once connected, endpoint input and output code is usually the same regardless of the media being used.

Endpoint code to receive data from an AppleTalk network can be identical to code to receive data through a modem, which can be identical to code to receive data over a serial line, and so on. Such things as packetization—which occurs in any network protocol—are hidden from the endpoint user during sending and receiving, as are operations such as flow control, error recovery, and so on.

The only exceptions to this rule occur when there are specific hardware limitations that push through the endpoint API. For example, IR beaming is a half-duplex protocol (it can only be in send mode or receive mode, not both at the same time) while serial, AppleTalk, or modem communications are all full-duplex (they can be in send and receive mode at the same time).

Of course, while sending and receiving are purposefully media-independent, the connection process is necessarily tied to the media being used. So, for example, with AppleTalk it is necessary to specify network addresses; for modem communications, a phone number; for serial communications, speed, parity, stop bits; and so on.

Figure 6 shows the life cycle of an endpoint. An endpoint is initially defined as a frame proto’ed from `protoBasicEndpoint`. The frame has several slots describing the settings of the endpoint and methods that may be called by

the system during the course of its existence. However, such a frame is not an endpoint. That is, it describes what an endpoint might look like, but it is not a NewtonScript object. To create such an object, it must first be instantiated. Note that since most objects in the Newton OS are views, and since the view system automatically instantiates a view object when it is opened, we usually don't see this step. But since an endpoint is independent of the view system, we must explicitly instantiate it to create an endpoint object.

Life Cycle of an Endpoint

EP:Instantiate()
EP:Open()
EP:Bind()
EP:Connect()/EP:Listen()
EP:Output()/SetInputSpec()
EP:Disconnect()
EP:Unbind()
EP:Dispose()

Note: EP is a fictitious reference to a NewtonScript endpoint frame

Figure 6. Life Cycle of an Endpoint

Once instantiated, an endpoint is opened by sending the `Open()` message to it. This ties the endpoint to a low-level communications tool in the system and spawns a new task at that lower level.

Once the endpoint is connected, it may need to be bound to a particular media-dependent address, node, and so on. An AppleTalk endpoint, for example, is bound to a node on the network. This is done by sending the endpoint the `Bind()` message. Note that some protocols (such as serial communications) do not have a required binding phase but it is still necessary to call `Bind()` (and later, `Unbind()`).

After binding the endpoint, the `Connect()` message is sent to connect to the particular media being used. For a remote service that is accessed through a modem endpoint, the endpoint would dial the service and establish the physical connection. Note that the endpoint does not handle protocol items

such as logging on, supplying passwords, and so on; these are part of an ongoing dialog that the application and the service must engage in once connection is established.

The endpoint method `Listen()` may be used to establish a connection instead of the `Connect()` method if the endpoint is instantiated and ready to listen to an “offer” by the remote source. Based on the particular situation with the communications media, an application may either reject the connection by sending the `Disconnect()` message to the endpoint, or accept it with the `Accept()` message. (Note that since infrared connections have one side sending and the other side receiving, in this case the passive side connects by calling `Listen()` instead of `Connect()`.)

After connecting, the endpoint is ready to send and receive data. Sending is fairly straightforward and is done by using the method `Output()`. When sending data, information about the form of the data (such as that it is a string, a NewtonScript frame, and so on) is usually sent. This gives the system a description of how the data should be formatted as it is being sent.

Output may be made either synchronously or asynchronously with asynchronous calls requiring that a callback method be specified.

Receiving data is a little more complex. Incoming data is buffered by the system below the application endpoint level. An application must set up a description of when it wants to get incoming data. This description is in the form of an `inputSpec`. For example, an `inputSpec` could be created which looked for the string “login:”, or it could be set to trigger when 200 characters were received. To some extent, it can be set to notify the endpoint of incoming data after a combination of these events (for example, after the string “login:” is seen or after 100 milliseconds, whichever comes first).

In any event, when an `inputSpec` input condition is met, an appropriate message is sent to the endpoint. This message will be different depending on the cause of the trigger; for example, a PartialScript message will be sent if the condition causing the event is a 100-millisecond wait, while an InputScript message will be sent if a specified string has been received or a character limit reached.

At any given time, the endpoint will have only one `inputSpec` active. By default, the active inputSpec will remain active until told otherwise.

By chaining inputSpecs together, a communications state machine of arbitrary complexity can be created. For example, before connecting to a

service, an application might set an `inputSpec` that looks for the string "login:". Once that string is seen, a new `inputSpec` might be activated which looks for the string "password:". When this string arrives, an `inputSpec` that triggers every 100 characters or when a carriage return character is received might be activated. In this way, endpoints can be used to build a communications protocol based on expected behavior of the service.

As mentioned before, the form of the data being sent or received may be described to the endpoint. At a slightly lower level, there is a way to set the translation tables to be used for all incoming or outgoing character data. The default translation table converts ASCII-to-Unicode (16-bit character descriptions used by the Newton OS) for incoming characters and Unicode-to-ASCII for outgoing characters.

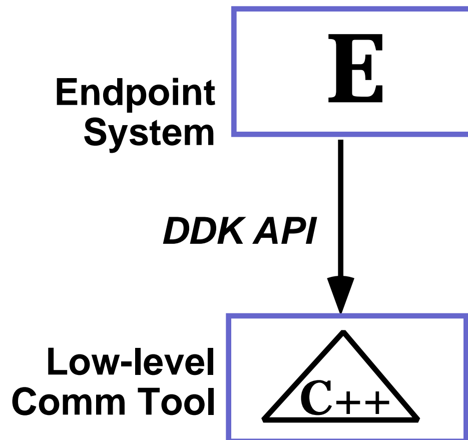
When an application is done sending and receiving data and wishes to tear down the endpoint, there are several messages which must be sent to an endpoint to reset the endpoint state and then disconnect and dispose of it. The first step is to terminate any outstanding inputSpecs by sending a Cancel message. This terminates the current inputSpec and establishes the fact that no more input will be accepted.

The connection is broken by calling the `Disconnect()` method.

Next the `Unbind()` message is sent to break the address association between the endpoint and the media.

Finally a `Dispose()` message is sent to destroy the endpoint object. Note that the endpoint may be left instantiated but disconnected if it is anticipated that it may be reconnected later in the life of the application.

Low-Level Communication Tools

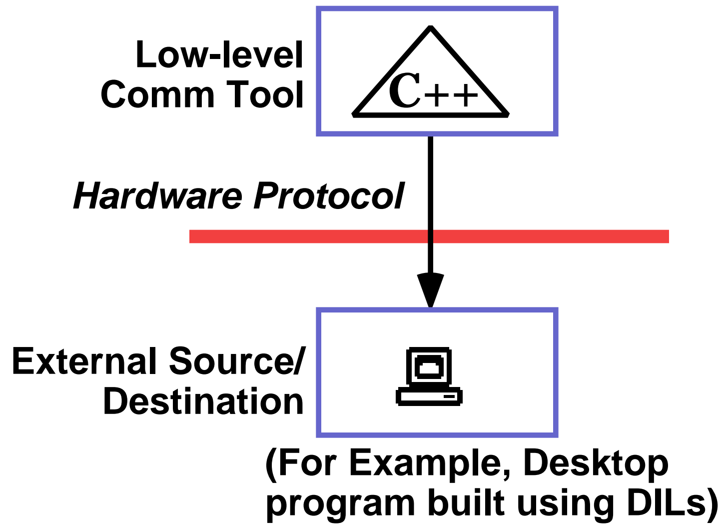


Below the NewtonScript level, there are built-in system tools which provide basic communications functionality. For example, when an endpoint is instantiated, one of the things which must be defined is what type of endpoint it is. This definition causes the endpoint to be connected to one of the existing low-level tools which are written in C++ and which actually run in a separate task thread.

While the details of these tools are beyond the scope of this article, at some point in the future Apple will release the necessary programming interfaces and tools to support the development of third party communications tools.

DILs (Desktop Integration Libraries)

2.0 Communications Overview



An important addition to Newton communications programming is a set of libraries called the Desktop Integration Libraries or DILs. Please see the companion article Desktop Integration Libraries: Overview for information about DILs.