Newton Binary Communications

by Ryan Robertson, Apple Computer, Inc.

Sending and receiving binary data using the Newton OS has always been something of a voodoo art. Unlike other computer systems, the Newton's communication architecture does not provide a GetByte or PutByte equivalent. The Newton OS instead implements a very flexible endpoint architecture. Sending data is fairly straight-forward; you create your data, then pass it to the endpoint's `Output` method. You can specify to send data either synchronously or asynchronously. Receiving data is much different than with a traditional computer operating system. To receive data, you must post an input specification which specifies the type of data you want to receive. Input specifications are -- by nature -- asynchronous.

There are advantages and disadvantages to the flexibility of this system.

Advantages include:

• You can post an input specification, then go do other processing until the termination conditions of the specification are met.
• The abstraction provided by the Newton's communication architecture makes it very easy to switch between different transport types. Its just as easy to open a serial connection as it is to open an ADSP connection.

Disadvantages include:

• You must do extra work to send and receive just a few bytes of data.
• The architecture doesn't lend itself well to stream based programming.
• It is very difficult to simulate synchronous input.

In this article I will discuss how to send and receive binary data using the Newton's communication architecture. I will also discuss some of the stumbling blocks to watch for as you write your endpoint code. This article assumes that you are familiar with the basics of setting up an endpoint, sending data, and using input specifications. For more information on any of these, check out the Newton Programmer's Guide chapter titled "Endpoint Interface".

Sending Binary Data

After you have setup and connected an endpoint, you send data by using the endpoint's `Output` method. The `Output` method takes three arguments:

• The data to output. For our purposes, this data will be your binary object. Your binary object can either be allocated from the heap using the global function `MakeBinary`, or it can be allocated from a store using either the `NewVBO` or `NewCompressedVBO` store method. If you are outputting data that ranges in size from 1 byte to 2 KB, you are better off allocating the binary object from the NewtonScript heap. If you are outputting data that ranges from 2 KB on up, you should use a VBO. For more information about using VBOs, check out the Newton Programmer's Guide chapter titled "Data Storage and Retrieval".
• An array of output options. This argument is currently only used by the Newton Internet Enabler transport. If you are using UDP, you would specify the address and port of the machine to send the packet to.
• An output specification. An output specification is a frame that encapsulates information about how to send the data. The output specification tells the endpoint what type of data you are sending and whether you want to send asynchronously or synchronously. For sending binary data, you

also specify a `target` slot. The `target` slot holds a frame with an `offset` and a `length` slot. The `offset` slot specifies the offset into the binary object at which to start sending data. The `length` slot specifies how many bytes to send from the offset. If you are outputting packetized data, you will need to specify packet flags using the `sendFlags` slot.

There are two different ways you might choose to output your data. You might send the entire binary object with one `Output` call or you may choose to send the data using consecutive `Output` calls.

Sending all of the binary data at once is very straight-forward. You pass the binary object as the first parameter to the endpoint's `Output` method, then setup the `target` slot of the output specification so that the offset is zero and the length is the length of the binary data. Note that you do not need a `target` slot if you want to send all the binary data. If you output data synchronously, the output call will return when the entire binary object has been sent. If you output asynchronously, the output specification's `CompletionScript` will be called when the entire binary object has been sent. If you do output asynchronously, be sure that you do not modify the binary object until the `CompletionScript` has been called. Also, note that the calling context of the `CompletionScript` is the output specification itself, not the endpoint. This can make accessing your application's methods and data structures a little more difficult. For some tips on making this easier, see the "Tidbits" section below.

A drawback of sending the entire binary object at once is that you cannot use a deterministic progress indicator to let the user know how much of the data has been sent. You can, however, use a non-deterministic indicator -- a barber pole for instance -- to let the user know that some action is taking place. Note that you will not be able to use a barber pole if you output synchronously. It is highly recommended that you use the asynchronous form of all endpoint methods. It is much easier on the Newton OS, and will increase the performance of your data transfer. Below is a code example showing how to send all the binary data at once.

```
local myData := MakeBinary( 1024, 'binary );
local theCompletionScript := func( ep, options, result )
                        begin
                              // Handle completion here...
                        end;

// A synchronous output example
fEndpoint:Output( myData, nil, {async: nil,
                                form: 'binary} );

// An asynchronous output example
fEndpoint:Output( myData, nil,
                          {async: true,
                           form: 'binary,
                           CompletionScript: theCompletionScript} );
```

The second way to output data is to use consecutive `Output` calls. I call this "chunking" the data. If you are chunking the data, you will set a different offset in the `target` slot of the output specification with each call to `Output`. Depending on your data, you may also set a different length in the `target` slot. An advantage of outputting your data in chunks is that you can use a deterministic progress indicator to let the user know exactly how much of the data has been sent. Note that you should limit the size of each chunk to under 2 KB.

Outputting data in chunks is a place where you can potentially run into problems using synchronous outputs. Each time you call `Output` synchronously, the Newt task (the task that all NewtonScript runs in) will be forked. Each fork will require more system memory. If you output data in chunks synchronously inside of a loop, you run a risk of running out of system memory

with repeated calls to `Output.` Forks are not "cleaned up" until execution returns to the main NewtonScript event loop. Below are some code examples showing the difference between outputting chunks synchronously and asynchronously.

```
// A code example showing synchronous output
local myData := GetDefaultStore():NewVBO( 'binary, 40960 ); // 40 KB

// Output in 1 KB chunks.
for i := 0 to 40960 -1024 by 1024 do
   begin
      fEndpoint:Output( myData, nil, {async: nil,
                           form: 'binary,
                           target: {offset: i, length: 1024} } );

      // Update progress indicator here...
   end;
```

At first glance, the above code example looks very simple, however the Newton will very likely reset halfway through the data transfer. There is just not enough system memory to fork the Newt task 40 times. Here is an example using asynchronous outputs.

```
// A code example showing asynchronous output.
fEndpoint.data := GetDefaultStore():NewVBO( 'binary, 4096 ); // 4 KB

// The offset slot is used in the output specification's CompletionScript
fEndpoint.offset := 0;

// The amountSent slot is only necessary for updating a progress bar
fEndpoint.amountSent := 0;

fEndpoint.completionFunc := func( ep, options, result )
                        begin
                            if NOT result then
                                begin
                                    // Update the amount of data that has been sent
                                    ep.amountSent := ep.amountSent + 1024;

                                    // Update the progress bar here...

                                    // Make sure we have not sent all the data before
                                    // trying to send another chunk
                                    if ep.offset + 1024 < Length( ep.data ) then
                                        begin
                                            ep.offset := ep.offset + 1024;

                                            ep:Output( ep.data, nil,
                                                       {async: true,
                                                       completionScript: ep.completionFunc,
                                                       form: 'binary,
                                                       target: {length: 1024, offset:
ep.offset}} );
                                        end;
                                end;
                        end;

// Setup a progress bar here

fEndpoint:Output( fEndpoint.data, nil, {async: true,
                                   form: 'binary,
                                   completionScript: fEndpoint.completionFunc,
                                   target: {length: 1024, offset: fEndpoint.offset} } );
```

This code is definitely more complex than the synchronous case. To output asynchronously, we must keep a few different variables around so that we know how much we have output, what our current offset is, what the data is, and what the completion function is. Notice that almost all of the work now takes place inside of the output specification's `CompletionScript` method. We first check to make sure that the `result` parameter of the `CompletionScript` was nil. If it was non-nil, then there was an error outputting data so we should probably abort the output and notify the user. Next, we update the amount sent variable. The amount sent variable is only used for updating a progress bar. We then check to make sure that we have not yet sent all the data. This code example is assuming that we are outputting in multiples of the length of the binary data. Finally, we output the next chunk of data. All of this adds a little more code over the synchronous case, however its is well worth it.

When do you want to send data in chunks versus sending it all at once? It depends on the size of the data. If you are outputting a small amount of data that requires one or two seconds to transmit, I would recommend sending all the data at once. If your data would require more than a couple of seconds to send, I would recommend sending it in chunks. That way you will be able to let the user know exactly how much of the data has been sent. Users are much happier when they have an idea of how long the data transfer will take.

Receiving Binary Data

You must first post an input specification to receive any type of data through an endpoint. An input specification is a frame that describes the characteristics of the data that you want to receive. An input specification tells the endpoint what kind of data you want to receive, and what the terminating conditions are for the data. Input specifications are, by definition, asynchronous. If necessary, you can emulate synchronous input by opening a modal view, handling input inside of that view, then closing the view. Doing this is cumbersome and not recommended, and suffers from the same forking issues as synchronous communications.

When you post an input specification, you provide a callback routine called the `InputScript`. The `InputScript` is called when a terminating condition of the input specification has been met. Terminating conditions can include:

• Byte counts. A byte count tells the endpoint how much data you want to receive for this particular input specification.
• End sequences. This is typically used when receiving strings. Examples include characters such as unicodeCR, $., etc. You can also specify a string such as "Login:" or "Password:".
• End of packet. This is used for packetized transports such as ADSP and Sharp IR.

When receiving binary data, the only two terminating conditions you may use are byte count and end of packet. However, you may not even need these. When you post an input specification to receive binary data, you must add a target slot to that specification. The target slot is a frame with two slots: `data` and `offset`. Unlike any other input form, you must preallocate the binary object you want incoming data munged into. This means that you have to know the length of data you will be receiving ahead of time. The `data` slot in the target frame holds the binary object to write the incoming data to. The `offset` slot in the target frame tells the endpoint where to start placing data. For instance, if you have an offset of 10, all incoming data will be added to the binary object starting at byte 10.

A binary input specification is normally terminated when the target binary object is filled up. So normally, you don't need to specify any type of terminating conditions. You can, however, specify a byte count if you want the input specification to terminate before the target binary object is filled.

As with outputting data, there are two different ways to input your data: You may receive an entire binary object with one input specification or you may choose to receive a "chunk" of the

binary data with one input specification.

Receiving all of the binary data at once is very straight-forward. You will post an input specification with a target binary object and no byte count. When enough binary data has been received to fill up the target binary object, the `InputScript` of the input specification will be called, and will be passed the target binary object as its second parameter. As with outputting the entire binary object, you cannot use a deterministic progress indicator when receiving all of the data at once. Below is a code example of how to specify an input specification to receive a large binary object.

```
// Prepare the binary object to receive data into
local myData := GetDefaultStore():NewVBO( 'binary, 4096 ); // 4 KB

// Setup the input specification
local inputSpec :=   {
                     form: 'binary,
                     target: {data: myData, offset: 0},
                     InputScript: func( ep, data, terminator, options)
                        begin
                           // Process the binary data here...
                        end,
                     CompletionScript: func( ep, options, result )
                        begin
                           // Handle errors here
                        end,
                     };

// Finally, post the input specification
fEndpoint:SetInputSpec( inputSpec );
```

When 4 kilobytes of data have been received, the input specification's `InputScript` will be called. A common mistake is to assume that the calling context of the `InputScript` is within your application's view hierarchy. The calling context of an `InputScript` is the input specification frame. See the "Tidbits" section below for information on how to access your application's methods and data structures from within the `InputScript` .

Receiving the data in chunks requires re-posting an input specification with a different offset in the `target` slot. You will use the byte count terminator to determine the size of the chunk. Here is an example of what your input specification might look like:

```
// Prepare the binary object to receive data into
local myData := GetDefaultStore():NewVBO( 'binary, 4096 ); // 4 KB

// Setup the input specification
fEndpoint.offset := 0;
fEndpoint.amountReceived := 0;
fEndpoint.inputSpec :=   {
                         form: 'binary,
                         termination: {byteCount: 1024},
                         InputScript: func( ep, data, terminator, options)
                            begin
                                ep.amountReceived := ep.amountReceived + 1024;

                                // Update progress indicator here...

                                if ep.offset + 1024 < Length( data ) then
                                    begin
                                        ep.offset := ep.offset + 1024;
```

```
                                        ep:SetInputSpec( {_proto: ep.inputSpec, target:
{data: data, offset: ep.offset} } );
                                    end;
                                end,
                            CompletionScript: func( ep, options, result )
                                begin
                                    // Handle errors here
                                end,
                            };

   fEndpoint:SetInputSpec( {_proto: fEndpoint.inputSpec, target: {data: myData, offset:
fEndpoint.offset} } );
```

The InputScript will be called after 1024 bytes of data has been received. There are a couple of important things to point out in the above code example. First, we are explicitly setting the target slot of the next input specification inside of the `InputScript`. A common misconception is that you only need to post one input specification, and that the `offset` slot gets updated automatically for you as each chunk of data is received. In reality, you must update the `offset` slot, then re-post the input specification each time you want to receive a chunk of data. You will notice that I use proto inheritance for each input specification. This is done to reduce the RAM footprint of the code. The third important thing to notice is that in the `InputScript`, I am setting up the next input specification using the data parameter of the `InputScript`. The data parameter is simply a reference to the original binary object. Instead of storing a reference to the data as we did in the output case, we can just use that parameter when we post the next input specification.

Again, the same rules apply towards receiving the data in chunks and towards receiving all the data at once.

Tidbits

The calling context of the `Output` method's `CompletionScript` is the output specification, and the calling context of the input specification's `InputScript` is the input specification. Because of this, accessing your application's methods and data structures requires a little more work.
Here are three possible ways to accomplish this:

• Create your endpoint as a child of your application by adding an `_parent` slot to the endpoint frame which refers to the application base view or some other view within your application. You will then be able to access your application through the first argument to the `InputScript` or the `CompletionScript`.
• Add an `_parent` slot to your input or output specification frame which refers to the application base view or some other view within your application.
• Reference your application's base view directly by using GetRoot().(kAppSymbol).

Binarily Challenged

There are two bugs in binary communications that could cause you some trouble. The first deals with switching input forms.

There are two major forms of input types; stream-based which include the `binary` form and `frame` form, and byte-based which include the forms of `bytes`, `string`, and `numbers`. The difference between these forms is in how incoming data is buffered. The stream-based form writes directly into a destination object, whereas the byte-based form writes data into an intermediate NewtonScript buffer for `endSequence` and `filter` processing.

Buffered data can be lost when you switch between the two types: The data is not correctly copied between the different buffers.

The work around to this problem is to turn your communications protocol into a request-respond protocol. Do not send data to the Newton device until it signals it is ready to receive new data.

The second bug deals with allocating a binary object.

When you allocate your binary object, a temptation is to allocate it inline. By inline, I mean that you call `MakeBinary`, `NewVBO` or `NewCompressedVBO` inside of the `target` slot of the input specification. Here is an example:

```
local inputSpec := {
      form: 'binary,
      target: {data: MakeBinary( 1024, 'binary), offset: 0},
      InputScript: func( ep, data, terminator, options )
         begin
            // Handle input here
         end,
      CompletionScript: func( ep, options, result )
         begin
            // Handle errors here
         end,
   };
```

Because of a bug in the Newton OS, if you allocate a binary object inline, the unit will reset when it receives the first byte of data. To work around this bug, allocate the binary object as a local variable, then use that variable in the `data` slot of the `target` frame.

Virtually Yours

Virtual binary objects (VBOs) can be very useful, although there are some important memory issues to keep in mind. A nice feature of VBOs is that you do not have to allocate the entire object before you start adding data to it. As you add data via `BinaryMunger` or `StrMunger`, the virtual binary object will grow, if necessary, to accommodate the data. Note that you do have to preallocate the entire object if you are using it in an input specification.

There is a downside to the flexibility of VBOs. Resizing a VBO can add a performance hit to your code. If possible, try to allocate the entire binary object ahead of time.

In Conclusion

Here are some important things to remember:

- It is recommended that you use the asynchronous form of all the endpoint methods.
- If you receive data in chunks, the offset slot of the input specification's `target` frame is not updated automatically. You must update this slot yourself before you post another input specification.
- If you send or receive more than 2 KB of data, you should send it in chunks in order to give the user a better indication of the progress, and to reduce memory overhead.

Sending and receiving binary data on a Newton device can seem complex. Once you understand the quirks you will also realize the flexibility that is offered in the Newton's communication architecture. And always remember that the facts, although interesting, are irrelevant.

Ryan Robertson would like to thank Jim Schram for his help with this article.