Mini-Meta Data: Another way to get information to your PC

Ryan Robertson, Apple Computer, Inc.

This article describes the development of a pair of applications that export data from a Newton device to a desktop computer. There are two applications: one that runs on a Newton device and one that runs on a desktop computer.[1] They are designed to allow a developer to register data definitions so that soup information can be transferred between the Newton device and the desktop computer.

You Are the Connection

With all the additions to the Newton 2.0 OS, it may seem like exporting data to your desktop computer has been overlooked: it has not: the intention is for application developers to incorporate the Desktop Integration Libraries (DILs) into their existing applications. This approach will allow a user to directly connect to their Newton device using their favorite desktop application. Before the DILs became available, the user was required to use a second application called Newton Connection Kit to transfer their Newton device's data to a more generic format that could then be used by desktop applications.

The Desktop Integration Libraries are a set of platform-independent C libraries and APIs that can be easily incorporated into an existing application. They provide all the necessary support to connect to your Newton and to transfer data between a Newton device and a desktop computer.

There are currently two types of DILs: the communication DILs (CDILs), and the frame DILs (FDILs). The communication DILs are used to open a connection with the Newton and to read and write bytes of data. The frame DILs let you read and write other Newton data types, such as frames and arrays.

The two largest advantages to using the DILs are:

1) The DILs abstract all underlying transport details into an easy to use API.
2) Your code will run on MacOS™ and Windows™ with very little modification.

The implementation of the Newton application described in this article is intended to be as generic and extensible as possible and allows a developer to register information about how to format data before it is sent to the desktop computer. By doing this, the Newton application can export data from many different soups using many different formats (this format will be explained below in further detail). For instance, you will be able to export your Names file to a tab-delimited format that could be read into a database or a spreadsheet.

The desktop application will take the incoming data and dump it into a text file. It should also be designed so that it will easily port to other platforms. This means that the user interface code will be separated from the implementation code. This implementation only deals with text data, so the FDILs are not needed.

I'll begin this discussion by describing the protocol used for transferring data between the Newton device and the desktop computer. After that, I'll go into more detail of some of the major design decisions for both the Newton application and the desktop application.

---

[1] Mini-MetaData is a Newton DTS sample that should be available by press time. You can find the Mini-MetaData source code on AppleLink and the Newton WWW Site (http://dev.info.apple.com/newton/newtondev.html). The next Newton Developer CD will also contain this sample.

Yakity Yak, Do Talk Back

The protocol used for sending and receiving data is fairly simple. First I will discuss the Newton side of the protocol.

At various times during the connection, the Newton will send one of three things: a command code, a string length, or a string. The command codes have been purposefully selected as large numbers so as to avoid conflict with the string length. Table 1 summarizes the command codes sent from the Newton during the protocol.

Table 1. Command Codes Used by the Newton Protocol

Command code name	Command code value	Description
kNewtonCancelled	0x0FFF	Sent when the user presses the “Cancel” button on the Newton.
kNewtonFinished	0x0FFE	Sent when all of the data has been transferred to the desktop application.

Command codes are only sent when the user is canceling the operation or the export has completed. The rest of the protocol on the Newton side consists of sending strings to the desktop computer. In our protocol, the length of the string is sent first to let the desktop application know how large the receiving buffer needs to be. By using this technique, we guarantee that there will be no ambiguity as to whether the received data is a command code or a string length.

Here is the C function that reads data from the CDIL pipe on the desktop. It returns a value indicating whether the read was a success, a failure, a cancel, or whether the Newton is ready to disconnect.

```
long ReadBuffer( LPSTR bufferPtr, long* length )
{
    Boolean    eom;
    CommErr        anErr;
    long       command;

    // read the first four bytes, this will either be a command code or a string length
    *length = 4;
    anErr = CDPipeRead( gOurPipe, &command, length, &eom, 0, 0, kPipeTimeout, 0, 0 );
    if (anErr) {
        return kReadError;
    }

    // interpret the command code and act on it.  If the data was not a command code,
    // then it is a string length, so read in the string
    if (command == kNewtonCancelled)
        return kNewtonCancelled;
    else if (command == kNewtonFinished)
        return kNewtonFinished;
    else if (command) {
        *length = command;

        // resize the buffer to the size of the string plus one for the null character.
        if ( realloc( bufferPtr, command+1 ) ) {
            anErr = CDPipeRead( gOurPipe, bufferPtr, length, &eom, 0, 0,
                        kPipeTimeout, 0, 0 );
            if (anErr)
                return kReadError;

            bufferPtr[(*length)] = (char)0;                    // Null terminate the string
```

```
            return kReadSuccess;
        }
    }

    return kReadError;
}   // ReadBuffer
```

The desktop PC side of the protocol consists of four command codes which are summarized in Table 2.

Table 2. Command Codes Used by the Desktop PC Protocol

Command code name	Command code value	Description
kHelloCommand	0x0FFD	Sent when the connection has been established. This tells the Newton application to start the protocol.
kGoCommand	0x0FFC	Sent when the desktop application is ready to start receiving the export data.
kAckCommand	0x0FFB	Sent after the desktop has successfully received a line of data.
kErrorCommand	0x0FF9	Sent if there is an error during the connection.

Because most of the data transfer consists of the Newton device sending data to the desktop machine, the Newton application uses only one input specification for the entire protocol.

```
{form:            'number,

 InputScript: func( ep, data, termination, options ) begin
    if data = kHelloCommand then begin          // Hello command was received,
                                                // start the protocol.
        ep:DoEvent( 'StartProtocol, nil );
    end else if data = kGoCommand then begin    // go command was received.
                                                // Initialize and output the first line
        ep._parent.fStatusView:StopBarber();

        local numEntries := ep.fCursor:CountEntries();
        ep:Parent().fStatusView:GoGoGadgetGauge( numEntries, kSendingDataString );

        ep:DoEvent( 'OutputData, nil );
    end else if data = kAckCommand then         // ack command was received,
                                                //output the next line
        ep:DoEvent( 'OutputData, nil );
    else begin                                  // There was an error, so disconnect
        GetRoot():Notify( kNotifyAlert, kAppName, kProtocolErrorString );
        ep:DoEvent( 'Cancel, nil );
        ep:DoEvent( 'Disconnect, nil );
    end;

    nil;
 end,

 CompletionScript: func( ep, options, result ) begin
    ep:DoEvent( 'Disconnect, nil );
 end,
}
```

If an unknown command code is received on the Newton device, the Newton application signals a cancel and disconnects. An unknown command code is likely to be caused by a communications error. If the protocol were more robust, the Newton could try to resync with the desktop machine and start sending data again

Protocol of the Wild

Once the connection has been established, `kHelloCommand` is sent from the desktop PC to the Newton device. Seeing the `kHelloCommand`, the Newton application will send the name of the application for verification purposes. This name is checked on the desktop PC to make sure the connection is with the Mini-MetaData application and not with the Newton's built-in Connection application or with the Toolkit App.

The next step is for the Newton application to send the name of the file that data will be exported to. Once the desktop PC receives this name, the standard save dialog will be opened with that file name as the default.

When the user finishes selecting the target file, the desktop application will send `kGoCommand` indicating it is ready to begin receiving data.

At this point, the Newton application begins sending data in the following pairs: a string length followed by the string. When the desktop successfully receives and writes this string to the file, it will send an `kAckCommand` to the Newton to signal that it is ready for more data.

Finally, the Newton application sends `kNewtonFinished` when it has finished transferring data. It then disconnects.

If the desktop encountered an error during the protocol, it will send `kErrorCommand` to the Newton and disconnect.

Here is an example of the protocol in action:

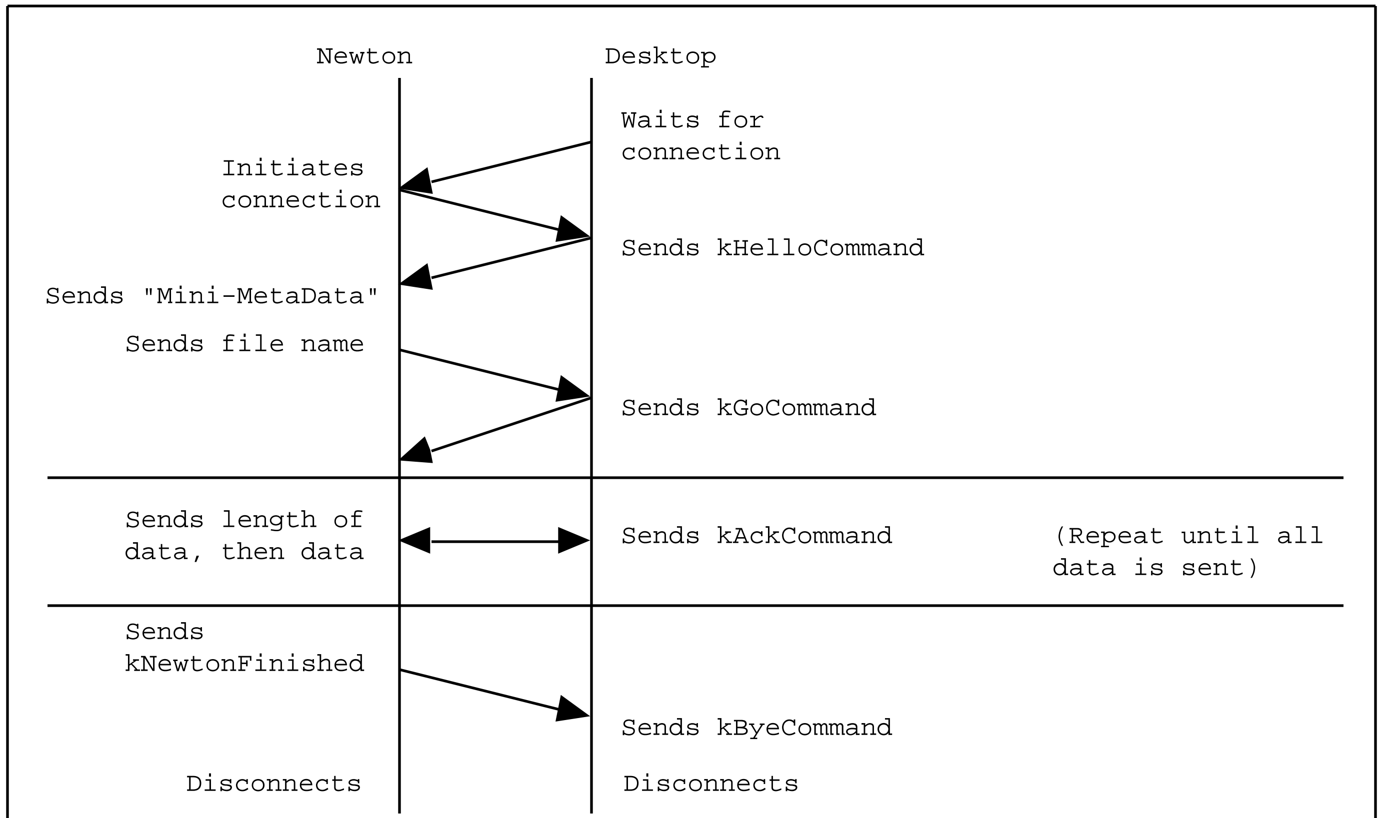

Figure 1. Newton-Desktop Communication Protocol

Now that you understand the protocol, lets dive into the code on the Newton.

Newton Side Up

To extend the mini-meta data application, you will add a format frame to a global registry. The format frame includes such information as which soup to send data from, what the query specification is, and how to create a formatted string from a soup entry. This registry will be discussed in more detail below.

The Newton application handles the format information and provides a simple interface for selecting which format to use. To keep the implementation as generic as possible, a form of meta data was created. Using this meta data, a developer can have a maximum amount of control over the format of outgoing information without explicitly having to know much information about the Newton storage or communications systems.

Here is a screen shot of what the Newton interface looks like.

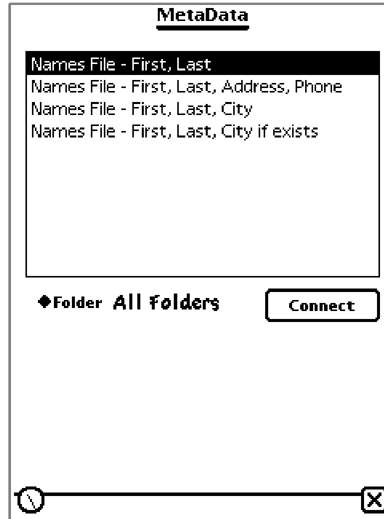

Figure 2. The Mini-MetaData User Interface

The NTK Project for the Newton application consists of 10 files, 4 of which are layout files.

Mini-MetaData

Seq.	Name	Type	Size	Mod. Date	Path Name
1	Mini-MetaData.rsrc	Resource	491	4/5/96-3:00 PM	HD:Desktop Folder:Mini-MetaData-1 …
2	Project Constants	Text	2364	5/7/96-3:40 PM	HD:Desktop Folder:Mini-MetaData-1 …
3	protoEvent	Proto	2444	3/18/96-6:57 PM	HD:Desktop Folder:Mini-MetaData-1 …
4	protoState	Proto	2444	3/18/96-6:56 PM	HD:Desktop Folder:Mini-MetaData-1 …
5	protoFSM	Proto	34808	3/20/96-12:47 PM	HD:Desktop Folder:Mini-MetaData-1 …
6	Endpoint.t	Layout	7183	5/7/96-2:13 PM	HD:Desktop Folder:Mini-MetaData-1 …
7	StatusView.t	Layout	5546	5/7/96-2:12 PM	HD:Desktop Folder:Mini-MetaData-1 …
8	ProtocolFSM	Layout	30443	5/7/96-6:10 PM	HD:Desktop Folder:Mini-MetaData-1 …
9	•Main.t	Layout	13111	5/7/96-3:46 PM	HD:Desktop Folder:Mini-MetaData-1 …

Figure 3. The NTK Project Window for the Mini-MetaData Application

The important files to look at are: Endpoint.t, StatusView.t, ProtocolFSM, and Main.t.

The hierarchy of the Newton application is illustrated in Figure 4.

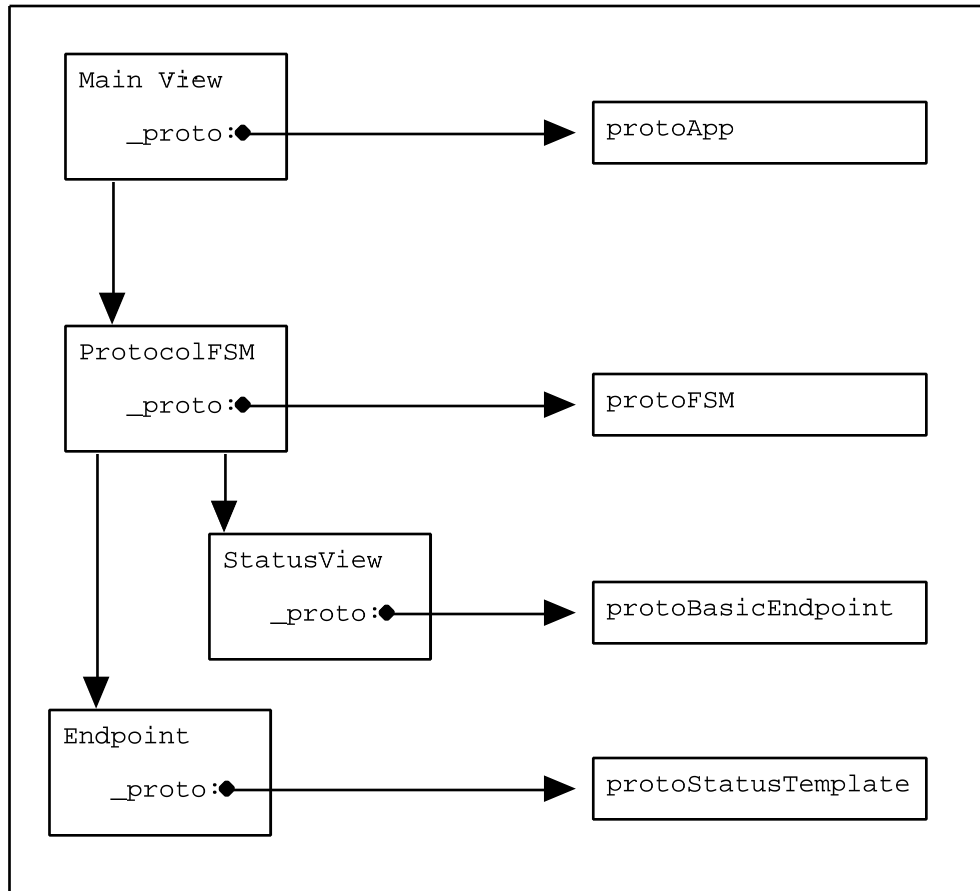

Figure 4. Hierarchy of the MiniMetaData Application

GoGoGadgetStatusView

It is very important to give the user feedback during the connection. Newton 2.0 OS provides a terrific proto, `protoStatusTemplate`, for conveying status information to a user. StatusView.t contains the template for the status view that is used during the connection. One of the beauties of using `protoStatusView` is that it has multiple personalities. Among other things, a view based on protoStatusView can be a single line of text, a barber pole, or a gauge. During our connection we will use all three of these.

The barber pole element is used during the connection phase. The barber pole was chosen because the time it takes to connect is not a known value, and a simple line of text doesn't necessarily give the user the impression that a lengthy operation is taking place. During the connection phase, the user may forget to signal a "wait for connection" event on the desktop which leaves the Newton waiting until the connect request times out.

The gauge element is used while data is being sent. Because we know the number of items that will be sent, a deterministic interface element is a more appropriate choice here.

The simple status view is used for disconnecting. A barber pole was not used because the disconnect operation is usually very fast. The disconnect operation will also complete successfully regardless of whether the desktop computer is disconnecting.

The status view template has three main methods of interest. They are: `GoGoGadgetBarberPole`, `GoGoGadgetGauge`, and `GoGoGadgetSimpleStatus`. Each of these methods will set up the status template with the correct information and open it if necessary.

There are also some additional methods for updating the text, the gauge, and the barber pole once the view has already been opened.

Back to the Basics

The mini-meta data application uses `protoBasicEndpoint` as the prototype for the connection endpoint. Using `protoEndpoint` is not recommended, and is actually impossible to use in a "2.0 only" application. This new endpoint proto is much more reliable and functional than `protoEndpoint`.

Endpoint.t contains the template for our endpoint. In addition to the standard endpoint methods, there is one other method of interest: `OutputLine`. `OutputLine` calls a helper function to format a soup entry into an output string (this method will be discussed in more detail later). It then outputs that string and updates the status view.

Here is the definition of OutputLine:

```
func() begin
    local entry := fCursor:Entry();

    // if there is an entry, then output the next line of data.  Otherwise,
    // output kNewtonFinished and disconnect.
    if entry then begin
        fData := :CreateStringFromEntry( entry, fMetaDataFrame );
        fCursor:Next();

        // Output the length of the data then output the data.  If either
        // output fails then post a 'cancel event.
        :Output( StrLen(fData), nil,
                  {async:  true,
                   form:      'number,
                   CompletionScript:  func( ep, options, result)
                   begin
                        if NOT result then
                            ep:Output( ep.fData, nil,
                                        {async: true,
                                        form: 'string,
                                        CompletionScript: func( ep, options, result)
                                        begin
                                            if NOT result then begin
                                                ep._parent.fStatusView:UpdateGauge();
                                                ep.fData := "";
                                            end else
                                                ep:DoEvent( 'Cancel, nil );
                                        end,
                            } );
                        else
                            ep:DoEvent( 'Cancel, nil );
                    end,
        } );
    end else begin
        // Output kNewtonFinished command and disconnect when the Output completes.
        :Output( kNewtonFinished, nil, {async: true,
                                            form: 'number,
                                            CompletionScript: func( ep, options, result )
                                            begin
                                                ep._parent.fStatusView:FinishGauge();
                                                ep:DoEvent( 'Disconnect, nil );
                                            end;
                                            } );
    end;
```

```
end
```

Holy Finite State Machines Batman!

Using a deterministic finite-state machine for communications was covered in depth in the April 1996 issue of NTJ (volume II, issue 2). This application leverages off of the sample code produced for that article. The file of interest is ProtocolFSM which has the layout of all the states and events needed for our application.

There are three events worth pointing out. The first event is the 'Create' event in the Genesis state. This event sets up the endpoint, the status view, and registers a power off function. Any initializations needed for the connection should be done here.

Next we have the 'Connect Success' event in the 'Connect' state. This event sets the input specification for our protocol, and also has the definition of our input specification in the fInputSpecification instance variable. This event is performed once there has been a successful connection with the desktop computer.

Finally, we have the 'OutputData' event in the 'Connected' state. This event simply calls the endpoint's `OutputLine` method described above. So why is this event of interest to us? Another possible implementation for outputting data would have been to call the `OutputLine` method directly from the input specification. Doing this would remove an event from the state machine, and make the code more centralized. However, by placing the `OutputLine` method in an event, canceling functionality is provided for free. When the finite state machine receives a cancel event, all posted communications requests will be canceled, including the input specification.

By using the finite state machine sample, the code is more understandable, and more modular. This type of modularity provides an almost complete separation between the interface code and the communications code. Having this separation will make future revisions easier.

All communications code on the Newton side is asynchronous. This decision was made because synchronous comms are generally evil. When you post a synchronous comms request on the Newton, an additional task is created - that's Newton lingo for a new thread. This adds needless overhead to the system, and can potentially reveal some interesting problems. For instance, you may be outputting lots of data in a loop using synchronous output requests. Each time through the loop a new task will be created, which is a rather expensive operation. The new task will take up system memory, and will not release control until it returns to the main event loop (which does not happen until you are finished with your output loop). As a consequence, the Newton will eventually run out of system memory and come crashing to its knees. Another drawback of using synchronous comms is that the user loses control of their Newton while the comms request is waiting to complete.

Grand Central

Our main layout file is main.t. This file contains the code for selecting a format, and creating an output string from a soup entry.

The important function to look at is `CreateStringFromEntry`. This method is called repeatedly during the protocol. It is passed a soup entry and will return a string representation of that entry by using the format frame. It iterates over the field array in the format frame, building a string from the elements of that array.

```
func( entry, metaFrame )
begin
    local line, lineItem, result;
```

```
    line :=    foreach lineItem in metaFrame.fields collect begin
               // build the item string from the meta data frame.

               // if lineItem is a path expression, the resolve it and return the value
               if ClassOf( lineItem ) = 'pathExpr OR ClassOf( lineItem ) = 'symbol then begin
                   if entry.(lineItem) then
                       entry.(lineItem) & metaFrame.itemSeparator;
                   else
                       metaFrame.emptySpace & metaFrame.itemSeparator;
               end else if IsFunction(lineItem.format) AND
                               HasSlot( lineItem, 'pathExpr ) then begin
                   // if we have a format function then pass in the value found using
                   // the pathExpr slot to the function.
                   result := call lineItem.format with ( entry.(lineItem.pathExpr) );
                   if result then
                       result & metaFrame.itemSeparator;
                   else
                       metaFrame.emptySpace & metaFrame.itemSeparator;
               end else if lineItem.format = 'quotedString AND
                               HasSlot( lineItem, 'pathExpr ) then begin
                   // if format is 'quotedString, then quote the value found using
                   // the pathExpr slot.
                   result := result;
                   if result then
                       $" & result & $" & metaFrame.itemSeparator;
                   else
                       metaFrame.emptySpace & metaFrame.itemSeparator;
               end else if lineItem.format = 'quoteIfExists AND
                               HasSlot( lineItem, 'pathExpr ) AND
                               entry.(lineItem.pathExpr) then begin
                   // if format is 'quoteIfExists then quote if the value found using
                   // the pathExpr slot exists
                   $" & entry.(lineItem.pathExpr) & $" & metaFrame.itemSeparator;
               end else
                   metaFrame.emptySpace & metaFrame.itemSeparator;
           end;

    // return a string with the proper line separator
    return Stringer( line ) & metaFrame.lineSeparator;
end
```

Don’t Forget the Desktop

As discussed earlier, the desktop application uses the DILs to transfer data between the Newton device and the output file. The requirements of this application were simple enough that only the CDILs were needed.

To help in the effort to create cross platform code, the project is broken into two C files. There is a file for the main OS event handling code and a file for the protocol code. They are Interface.c and Protocol.c. The event code and the dialog code is not cross platform because much of that code is specific to either platform. The protocol code is cross platform and consists of the code to open the connection with the Newton, handle the protocol, and close the connection.

There are four functions in Interface.c that are not used for handling OS events. They are `CreateNOpenFile`, `WriteToFile`, `UpdateNCloseFile`, and `InitializePipe`. The first three are not in Protocol.c because they contain file access routines that are specific to one platform. Why `InitializePipe` is not in Protocol.c is not as obvious: the underlying transport options are specified slightly differently depending on whether you are running on MacOS or on Windows.

OS Event Handling

The interface code is in the Interface.c file if you are using MacOS and is in the INTERFAC.C file if you are using Windows. These files contain all the standard event handling code and should probaby look pretty familiar. In addition to the above mentioned functions (`CreateNOpenFile`, `WriteToFile`, `UpdateNCloseFile`, and `InitializePipe`), the MacOS code contains one other function of interest: `SetupPortMenu`. This function correctly creates a list of the ports available on the given machine. For instance, most Macintosh's have a printer and a modem port. However, if the user is running on a Duo there is one printer/modem port.

Protocol.c

This file contains all the code necessary to handle the protocol and the various states of the connection. It also contains the code to handle error reporting to the user. Most of the functions and procedures in this file are easy to understand. However, there are a couple of areas that warrant further discussion.

The procedure that handles most of the protocol is `DoProtocol()`, and is defined as follows:

```
void DoProtocol()
{
    StandardFileReply     fileReply;
    short             fileRef = 0;
    long              length;
    char              *bufferPtr = NULL;
    long              fBufferResult;
    long              anErr;

    // preallocate a buffer that we think will be large enough for most data.
    // This buffer will be resized as data is received.
    if ( !(bufferPtr = malloc( 256 )) ) {
        ConductErrorDialog( kNoMemoryString );
        return;
    }

    // Send kHelloCommand to the Newton
    fBufferResult = WriteCommand( kHelloCommand );
    if (fBufferResult == kWriteError)
        Fail(FailWrite);

    // Make sure the we have connected to the Mini-MetaData app on the Newton
    fBufferResult = ReadBuffer( bufferPtr, &length );
    switch (fBufferResult) {
        case kReadSuccess:
            if ( strcmp( kHeloResponse, bufferPtr ) ) {
                Fail(FailWrongApp);
            }
            break;
        case kNewtonCancelled:
            Fail(NewtonCancelled);
        case kReadError:
            Fail(FailRead);
    }

    // Read the filename to save the incoming data to
    fBufferResult = ReadBuffer( bufferPtr, &length );
    switch (fBufferResult) {
        case kNewtonCancelled:
            Fail(NewtonCancelled)
        case kReadError:
            Fail(FailRead);
    }
```

```
// create and open the file, then start dumping data into it.
anErr = CreateNOpenFile( bufferPtr, &fileReply, &fileRef );
if ( anErr == noErr ) {
    fBufferResult = WriteCommand( kGoCommand );

    if (fBufferResult == kWriteError) {
        UpdateNCloseFile( fileRef, &fileReply );
        Fail(FailWrite);
    }

    // Loop until there is either an error, or until the Newton sends a cancel
    // command or a finished command
    while( true ) {
        CDIdle( gOurPipe );
        fBufferResult = ReadBuffer( bufferPtr, &length );

            switch (fBufferResult) {
            case kReadSuccess:
                anErr = WriteToFile( fileRef, &length, bufferPtr );

                // if there was an error writing to the file, close the file, display
                // an error and return.
                if (anErr) {
                     Fail(FailWriteFile);
                }

                // send an kAckCommand, if there was an error then handle it.
                fBufferResult = WriteCommand( kAckCommand );
                if (fBufferResult == kWriteError) {
                    UpdateNCloseFile( fileRef, &fileReply );
                    Fail(FailWrite);
                }
                break;
            case kNewtonCancelled:
                UpdateNCloseFile( fileRef, &fileReply );
                Fail(NewtonCancelled);
            case kNewtonFinished:
                ConductErrorDialog( kDownloadWasSuccessful );
                UpdateNCloseFile( fileRef, &fileReply );
                free( bufferPtr );
                bufferPtr = NULL;
                return;
            case kReadError:
                UpdateNCloseFile( fileRef, &fileReply );
                Fail(FailRead);
        }
    }
}

WriteCommand( kErrorCommand );
free( bufferPtr );
bufferPtr = NULL;
return;

// These are the Goto locations that are jumped to using the Fail() macro.
FailWrite:
    WriteCommand( kErrorCommand );
    ConductErrorDialog( kBufferWriteErrorString );
    free( bufferPtr );
    bufferPtr = NULL;
    return;

FailRead:
```

```
        WriteCommand( kErrorCommand );
        ConductErrorDialog( kBufferReadErrorString );
        free( bufferPtr );
        bufferPtr = NULL;
        return;

    NewtonCancelled:
        ConductErrorDialog( kNewtonCancelledString );
        free( bufferPtr );
        bufferPtr = NULL;
        return;

    FailWriteFile:
        ConductErrorDialog( kFileWriteErrorString );
        WriteCommand( kErrorCommand );
        UpdateNCloseFile( fileRef, &fileReply );
        free( bufferPtr );
        bufferPtr = NULL;
        return;

    FailWrongApp:
        ConductErrorDialog( kWrongAppString     );
        free( bufferPtr );
        bufferPtr = NULL;
        return;

}   // HandleProtocol
```

`ReadBuffer` and `WriteCommand` are helper functions used to read to and write from the CDIL pipe.

The return value is checked to make sure there were no errors in the protocol. If an error occurred, it is assumed that there is a problem with the connection, and the connection is aborted. In an effort to retain some amount of synchronicity, `kErrorCommand` is sent after an error has occurred. There is a good chance that the `kErrorCommand` may not be sent because the original error was a communications error. A more robust protocol would examine the error value and take appropriate action based on that value. It may be possible to recover from the error and continue receiving data.

Registering the Meta Data

To extend the mini-meta data application, you will add a format frame to a global registry. The symbol for this registry is '|MiniMetaDataRegistry:DTS|. Here is an example of how you add a format frame:

```
local registry;
if GlobalVarExists( '|MiniMetaDataRegistry:DTS| ) then
    registry = GetGlobalVar( '|MiniMetaDataRegistry:DTS| );
else
    registry := DefGlobalVar( EnsureInternal('|MiniMetaDatRegistry:DTS|),
                              EnsureInternal([]) );

AddArraySlot( registry, myFormatFrame );
```

Here is how you would remove your format from the registry:

```
if GlobalVarExists( '|MiniMetaDataRegistry:DTS| ) then begin
    local registry = GetGlobalVar( '|MiniMetaDataRegistry:DTS| );
    local pos := LSearch( registry, myFormatSym, 0, '|=|, 'symbol );
```

```
    if pos then
            RemoveSlot( registry, pos );
end;
```

Here are some examples of what a format frame might look like:

```
{title: "Names File - First, Last",
 symbol: '|Format1:DTS|,
 soupName: "Names",
 lineSeparator: unicodeCR,
 itemSeparator: ",",
 emptySpace: " ",
 fields: ['name.first, 'name.last]}

{title: "Names File - First, Last, Address, Phone",
 symbol: '|Format2:DTS|,
 soupName: "Names",
 fileName: "Names Export",
 fields: ['name.first, 'name.last, {format: func(s) if s then CapitalizeWords(s) else nil,
                                    pathexpr: 'address},
          [pathexpr: 'phones, 0]]}

{title: "Names File - First, Last, City",
 symbol: '|Format3:DTS|,
 soupName: "Names",
 itemSeparator: ",",
 lineSeparator: unicodeCR,
 emptySpace: " ",
 fields: ['name.first, 'name.last, {format: 'quotedString, pathexpr: 'city}]}
```

Each MetaData frame must have the following slots: title, symbol, fields, either GetSoupName or soupName, and either GetQuerySpec or querySpec. It may optionally have a lineSeparator slot, emptySpace slot, and an itemSeparator slot.

Here is a more in-depth description of each slot:

`title`	The name of this particular meta data frame. This is the name that will appear to the user in the list of installed meta data frames.
`Symbol`	This is a unique symbol used to identify this particular meta data frame. If you register two meta data frames with the same symbol, the second one that is installed will overwrite the first one. Append your developer signature to the symbol.
`soupName`	The name of the soup to export data from, or nil.
`GetSoupName`	If you cannot know the name of the soup at compile time, specify this slot instead of the soupName slot. GetSoupName will hold a function that will return the name of the soup. You will need to either specify a `soupName` slot or a `GetSoupName` slot. If both are specified the `GetSoupName` slot will take precedence.
`querySpec`	The query specification to be used when exporting data. If you do not provide a query specification, the default `nil` will be used.
`GetQuerySpec`	Optional. If you do not provide a querySpec in the querySpec slot, then you must provide a function that returns a query specification in this slot. You will need to either specify a `querySpec` slot or a `GetQuerySpec` slot. If both are specified the `GetQuerySpec` slot will take precedence.

`fileName`	The name of the file to export data to. The default is Untitled for Macintosh and UNTITLED.TXT for Windows.
`lineSeparator`	This slot holds either a character or a string that will be appended to the end of each line formatted using the fields slot. The default value is a carriage return.
`emptySpace`	This slot holds either a character or a string that will be used if a path in the fields slot did not exist in the soup entry. The default value is a space.
`itemSeparator`	This slot holds either a character or a string that will be used to separate each item in the fields slot. The default value is the comma.
`fields`	This slot holds an array that defines how to export your data. The fields slot holds all the information necessary to export data from one entry in the soup. See table 3 for information on filling in this array.

Table 3. The fields Slot Array

Path expression	The path expression used to find data in the soup entry.
Format frame	A frame with a format slot and a pathExpr slot. The object obtained from the path expression will be formatted using the format slot. The format slot can either be ‘quotedString, ‘quoteIfExists, or a function. If it is ‘quotedString, then the data found in pathExpr will have quotation marks put around it. If ‘quoteIfExists is specified then the data will be quoted if it exists in the soup entry. If format is a function, then the data will be passed to the function for formatting. This function should return the formatted string. Be sure to check that the argument passed to the function is non-nil before manipulating it.

Ryan Robertson would like to thank David Fedor and Maurice Sharp for their help with this article.