Newton Application Development

Newton Desktop Integration Libraries

Version 1.0

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

About This Book

This book, *Newton Desktop Integration Libraries*, describes programmer software used to give Windows and Mac OS applications the ability to exchange data with a Newton device. It has the following chapters:

- Chapter 1: "Introduction" discusses the Desktop Integration Libraries (DILs) and describes some non-standard types and values that are used in the DIL code.
- Chapter 2: "The Communications Desktop Integration Library" describes the basic communications facility that the DILs provide.
- Chapter 3: "The High-Level Frames Desktop Integration Library" describes the facility the DILs provide for communicating with a Newton using high-level frame structures.
- Chapter 4: "The Newton Side of the DIL Connection" gives general directions on writing Newton applications that can communicate with the DILs.

Conventions Used in This Book

This book uses the following conventions to present various kinds of information.

Special Fonts

This book uses the following special fonts:

- **Boldface**. Key terms and concepts appear in boldface on first use.

- `Courier typeface`. Code listings, code snippets, and special identifiers in the text such as slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text. If you are programming, items that appear in Courier should be typed exactly as shown.
- *Italic typeface*. Italic typeface is used in code to indicate replaceable items, such as the names of function parameters, which you must replace with your own names. The names of other books are also shown in italic type, and *rarely*, this style is used for emphasis.

Overview

You can use the Newton Desktop Integration Libraries (DILs) to create desktop computer applications that can exchange data with a Newton. The DILs:

- provide layered, fully functional C libraries to enable desktop applications to communicate with Newton devices
- provide flexible, high-performance common communications APIs (applications programming interfaces) for use in Mac OS and Windows-based applications

DILs are delivered for computers running the Mac OS computers as static linkable libraries and for computers running the Windows operating system as Dynamic Linked Libraries.

The function prototypes for the DILs are designed to be used by multiple compilation environments. You should consult the README file for instructions on building in your environment.

This book documents two DILs, the Communication DIL (CDIL) and the High-Level Frames DIL (HLFDIL).

The CDIL has two purposes:

- It creates the basic connection with the Newton. It gives you communications with a Newton through a modem, serial port, or AppleTalk ADSP. (Windows applications currently can only use MNP modem connections.)
- It allows low-level communication, where you can send data that is treated by the Newton as a stream of bits. This allows you to use non-Newton types and also to encrypt your data.

The HLFDIL lets you send and receive data in the form of Newton data structures such as arrays and frames. When you use the HLFDIL, you still use the CDIL to create the basic connection with the Newton.

Data Types and Values

The DILs use a few non-standard data types and values, which are described in Table 1-1. Many of these are defined in the header files. Some types and values described in this section are standard to the Mac OS, and may not be in Mac OS header files.

Table 1-1 Data Types and Values

Identifier	Meaning
`Boolean`	Defined as an enumerated type that can have the value `true` or `false`.
`CDILCompletionProcPtr`	A pointer to the type of callback function used by `CDPipeListen` and `CDPipeWrite`.
`CDILDecryptionProcPtr`	A pointer to the type of callback function used by `CDDecryptFunction`.

Table 1-1 Data Types and Values (continued)

Identifier	Meaning
`CDILEncryptionProcPtr`	A pointer to the type of callback function used by `CDEncryptFunction.`
`CDILPipe`	A pipe object. Defined as `void`. Used in the form of a pointer, `CDILPipe*`, which is actually a `void*`.
`CDILPipeCompletionProcPtr`	A pointer to the type of callback function used by `CDPipeRead`.
`CommErr`	Defined as `long`.
`DILObj`	An HLFDIL object. Defined as `void`. Used in the form of a pointer, `DILObj*`, which is actually a `void*`.
`objErr`	Defined as `long`.
`false`	Defined as zero.
`nil`	A value that indicates nothing, none, no, or anything negative or empty. In particular, it indicates a null or undefined pointer. It is similar to `(void*)0` in C. Defined as `NULL`.
`Size`	Defined as `long`.
`true`	Defined as 1.

The Communication Desktop Integration Library

This document describes the Communications Desktop Integration Library (CDIL).

Although the CDIL is written in C++ and is based on C++ objects, you use it by making calls to C functions. The `CDCreateCDILObject` function creates a C++ object, and returns a pointer to that object. You use that pointer in calling other functions. (Those functions are actually "wrappers" for C++ methods.)

General Concepts

The CDIL is modeled on a **virtual pipe**. That is, the CDIL is based on a pipe metaphor, where data is put into a pipe that is flushed at appropriate times. You can also view the CDIL model as being a stream in the C++ sense.

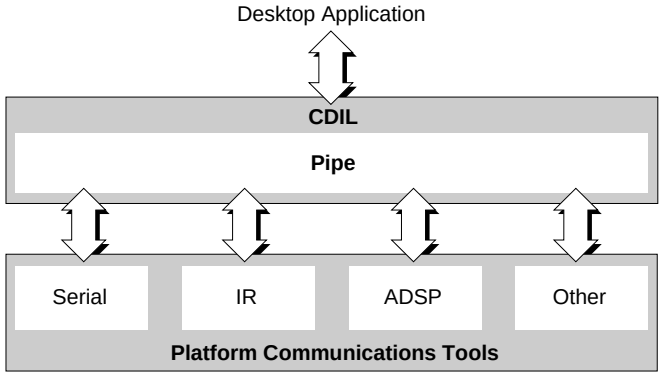
The CDIL supports the following features.

- The CDIL is transport-independent. The function you use to configure the pipe takes as parameters:
  - the type of transport that is to be used
  - the configuration options for the selected transport (such as baud rate)
  - the appropriate port to use

  The function calls (other than as required for configuration) are the same no matter what the transport type is.
- The CDIL supports the **pull model**. The desktop application sets up a pipe, and then listens to see if there is a connection. Thus, the desktop application is a passive listener. Once the connection is open, both the desktop application and the Newton can read from and write to the pipe.
- You can tell the pipe to perform automatic data conversions such as byte-swapping and ASCII-to-Unicode conversions.

Figure 2-1 shows the general architecture of the CDIL.

Figure 2-1 CDIL High Level Components



Using the CDIL

You go through the following steps to use the CDIL.

Note
The code in this chapter does not show any error handling. You should be sure to check for errors whenever you call DIL functions; most of the functions return an error code or `kCommErrNoErr`, which is 0. The possible errors are shown in "Error Codes" beginning on page 2-34. ◆

1. Initialize the CDIL. For example:

```
fErr = CDInitCDIL();
```

2. Create a pipe object. For example:

```
thePipe = CDCreateCDILObject();
```

3. Initialize the pipe. For example:

```
fErr = CDPipeInit(thePipe,
                  "Modem",
                  "",
                  "ModemType \"Newton Serial Connection\"\
dataBits 8 Parity None Baud 38400 Port ",
                  "Modem",
                  kDefaultBufferSize, kDefaultBufferSize);
```

4. Tell the pipe to listen for a connection. The connection is always initiated by a Newton, so, from the desktop side, you need to put the CDIL in a state where it is listening for the connection:

```
fErr = CDPipeListen(thePipe, 30000, NULL, 0);
```

The second parameter is a timeout. The third parameter can give a callback function; in this case, there is no callback function, so you give

NULL. The fourth parameter is a long value that you can use to pass information to your callback function, if you have one.

If you supply a callback function, the CDPipeListen function returns immediately and, when the listening operation is finished, the CDIL calls the callback function. One reason you might want to use a callback function is so that you can put up a status dialog that allows your user to cancel the connection attempt. If the user chooses to cancel the attempt, you can call CDPipeAbort to cancel it.

5. Watch for the state to become something other than kCDIL_Listening. (In general, CDPipeListen does not return until that happens, but it is possible for it to return early.) For example:

```
while (kCDIL_Listening == CDGetPipeState(thePipe)){}
```

6. Check what the state now is, and take appropriate action. In particular, when the pipe state is kCDIL_ConnectPending, accept the connection. For example:

```
if (kCDIL_ConnectPending == CDGetPipeState(thePipe))
   {fErr = CDPipeAccept(thePipe);}
   else HandleError();
```

   In this example, HandleError is a function you'd write that would figure out what to do, since the pipe is not in its expected state.

7. You can now send data by calling CDPipeWrite and receive data by calling CDPipeRead. If you keep the pipe open for any length of time, call CDIdle often.

8. When you are done, disconnect the pipe.

```
fErr = CDPipeDisconnect(thePipe);
```

9. Depending on your application, you might open the connection again. In that case, you need to call go back to step 3.

10. Before your application ends, destroy the pipe object, and clean up the CDIL.

```
fErr = CDDisposeCDILObject(thePipe);
fErr = CDDisposeCDIL();
```

CDIL Reference

This section documents the pipe states, which are important for any program that uses the CDIL, and then describes the CDIL functions, divided into related areas. Finally, it describes the error codes you might receive.

CDIL Pipe States

During your CDIL session, the CDIL pipe goes through a series of states that tell your application what is going on with the connection. Table 2-1 lists the states. Table 2-2 on page 2-6 diagrams the state transitions.

Table 2-1 Pipe States

State Constants	Meaning
`kCDIL_Uninitialized`	The pipe is uninitialized. This is the initial state of the pipe after `CDCreateCDILObject` and before `CDPipeInit`.
`kCDIL_InvalidConnection`	The pipe tried to bring up a connection, but it failed. This can occur after calling `CDPipeInit`.
`kCDIL_Startup`	The pipe has been initialized. This is the state after `CDPipeInit` and before `CDPipeListen`.
`kCDIL_Listening`	The CDIL is listening for a connection on the pipe. This is the state after `CDPipeListen` and before a connection is made.

Table 2-1 Pipe States (continued)

State Constants	Meaning
`kCDIL_ConnectPending`	A connection is pending. This state occurs when an application on the Newton is attempting to connect to this pipe.
`kCDIL_Connected`	The pipe is connected. This is the state after `CDPipeAccept`.
`kCDIL_Busy`	The pipe is either reading or writing.
`kCDIL_Aborting`	The pipe is currently aborting. You can get this after calling `CDPipeAbort`, but before the aborting process is complete.
`kCDIL_Disconnected`	The pipe has been disconnected. This is the state of the pipe after `CDPipeDisconnect`.
`kCDIL_Userstate`	You can have pipe states that have meaning to your program by adding integers to this constant (see "CDSetPipeState" on page 2-33).

Table 2-2 Pipe State Transitions

Time	State Constant	Notes
Initial state, before calling anything, except possibly `CDInitCDIL`	Undefined	You need to have created a CDIL object before you can get a state.
After calling `CDCreateCDILObject`	`kCDIL_Uninitialized`	The pipe needs to be initialized before it can be used.
After calling `CDPipeInit`	`kCDIL_InvalidConnection`	The pipe tried to initialize, but it failed.
After calling `CDPipeInit`	`kCDIL_Startup`	The pipe has been initialized.

Table 2-2 Pipe State Transitions (continued)

Time	State Constant	Notes
After calling `CDPipeListen`	`kCDIL_Listening`	The CDIL is listening for a connection on the pipe. If your connection type is `Serial`, the state will go to `kCDIL_ConnectPending` almost immediately.
	`kCDIL_ConnectPending`	A connection is pending. This state occurs when an application on the Newton has requested a connection to the running DIL application.
After calling `CDPipeAccept`	`kCDIL_Connected`	The pipe is connected.
After calling `CDPipeWrite` or `CDPipeRead`	`kCDIL_Busy`	The pipe is either reading or writing.
After calling `CDPipeAbort`	`kCDIL_Aborting`	The pipe is currently aborting. You can get this after calling `CDPipeAbort`, but before the aborting process is complete. If the connection type is `Serial`, the pipe will go to `kCDIL_Disconnected` immediately.
After calling `CDPipeDisconnect` or `CDPipeAbort`	`kCDIL_Disconnected`	The pipe has been disconnected.

Setting Up the Pipe

You use the functions in this section to set up the pipe.

You need to initialize the CDIL using `CDInitCDIL` first. Then you need to create a pipe object using `CDCreateCDILObject`. If you are writing a Windows application, you need to call `CDSetApplication` to set the application instance.

CDCreateCDILObject

```
CDILPipe *CDCreateCDILObject ( void ) ;
```

This function creates a CDIL pipe object and returns a pointer to it. If there is an error, it returns `NULL`. You use the returned pointer in calling other functions. After this call is successful, the new pipe has the state `kCDIL_Uninitialized`.

CDInitCDIL

```
CommErr CDInitCDIL ( void ) ;
```

This function initializes the underlying communications mechanism and prepares the environment for the CDIL. It is intended to be called as part of application initialization.

This function can return errors on Mac OS-based computers if initialization of the Communications Toolbox (CTB) fails. On Windows-based computers you can get device driver or memory errors. It returns zero when successful.

CDSetApplication

```
void CDSetApplication ( CDILPipe *pipe, HINSTANCE appInst) ;
```

This function only applies to Windows applications, and is required in those applications. It sets the application instance so that driver timers can work.

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
appInst	Gives an application instance.

Destroying the Pipe and Cleaning up the CDIL

When your application is finished, you need to use `CDDisposeCDILObject` to free the space used by a pipe and then use `CDDisposeCDIL` to clean up the CDIL.

CDDisposeCDILObject

```
CommErr CDDisposeCDILObject ( CDILPipe *pipe );
```

This function destroys a CDIL pipe. You should call this function when you are done with the pipe so that the memory used by the pipe is available to other programs.

This function returns zero or an error code. It returns `kPipeNotInitialized` if *pipe* is `NULL`.

The parameter to this function is:

pipe A pointer to the pipe you want to destroy.

CDDisposeCDIL

```
CommErr CDDisposeCDIL ( void ) ;
```

This function closes the underlying communications mechanism and cleans up the CDIL environment. This function is intended to be called as part of application shutdown. You should call this function after you call `CDDisposeCDILObject`.

This function returns an error on Windows-based computers if there is a problem with driver cleanup. No Mac OS errors are defined.

Making the Connection

You use these functions to get the connection with the Newton. Your application may connect and disconnect many times during the course of a session.

You need to call these functions in this order:

1. `CDPipeInit`. This function defines the connection.
2. `CDPipeListen`. This function tells the CDIL to wait for the Newton to initiate a connection. You need to wait until the pipe state becomes `kCDIL_ConnectPending`, which means that the connection has been initiated by the Newton.
3. `CDPipeAccept`. This function tells the CDIL to accept the pending connection.

CDPipeAccept

```
CommErr CDPipeAccept (CDILPipe *pipe) ;
```

If a connection is pending, the connection will be accepted. A connection is pending when an application on the Newton is attempting to connect to your application, which is indicated by a pipe state of `kCDIL_ConnectPending`. After this call is successful, the pipe has the state `kCDIL_Connected`.

This function returns an error code or zero on no error. On Mac OS-based computers, it can return low-level Communications Toolbox (CTB) errors. On Windows-based computers, this can return driver errors.

The parameter to this function is:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDPipeInit

```
CommErr CDPipeInit(  CDILPipe *pipe,
                     const char* connectionType,
                     const char* connectionName,
                     const char* configInfo,
                     const char* pipePort,
                     Size  readSize,
                     Size  writeSize) ;
```

This function opens an endpoint connection with the appropriate definitions, initializing the specific communications environment. After this call is successful, the pipe has the state `kCDIL_Startup`.

You need to call `CDPipeInit` again if you want to re-open the connection after calling `CDPipeDisconnect` or `CDPipeAbort`. You can check the pipe state (see "CDGetPipeState" on page 2-32) to see if you need to call `CDPipeInit`. You must call it when the pipe state is `kCDIL_Uninitialized` or `kCDIL_Disconnected`.

Note
Each time you call `CDPipeInit`, some memory is allocated that is not freed by `CDPipeDisconnect` or `CDPipeAbort`. Therefore, you should be careful not to close and re-open the connection too many times. ◆

The parameters to the function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
connectionType	A C string that describes the connection. Currently available types are shown in Table 2-3.
connectionName	A C string that you can use to provide a logical name for this connection. AppleTalk requires a logical name, and the other connection types currently available do not. You can get this string later so that if you have several pipes you can tell them apart. See "CDConnectionName" on page 2-31.
configInfo	A C string that describes the connection for this pipe. On Mac OS-based computers, this parameter is a Communications Toolbox configuration string (see the Communications Toolbox documentation for information). On a Windows-based computer, this parameter is a DOS MODE string (See the DOS documentation for information). You can get this string later (see "CDGetConfigStr" on page 2-32). If you give a `NULL`, this value defaults to `"COM1:38400,n,8,1"` for

	Windows-based computers, and `"Baud 38400 dataBits 8 Parity None Port"` on Mac OS-based computers, which is a string for the Apple Serial tool
pipePort	A platform-specific C string describing the port to be used by this pipe.The only current possibilities on Windows-based computers are `"COM1"`, `"COM2"`, `"COM3"`, and so on. Examples of Mac OS strings are `"Modem"`, `"Printer"`, and `"Printer-Modem"`. The Mac OS strings are localized for different versions of the Mac OS, and can vary. See the Communications Toolbox documentation for information. You can get this string later so that you can identify which port you're using. See "CDGetPortStr" on page 2-32.
readSize	Maximum size of the read buffer in bytes. The `Size` type is equivalent to `long`. You can use the constant `kDefaultBufferSize`, which is defined as 1024 bytes.
writeSize	Maximum size of the write buffer in bytes. The `Size` type is equivalent to `long`. You can use the constant `kDefaultBufferSize`, which is defined as 1024 bytes.

Table 2-3 Connection Types

Connection Type Description	Mac OS C String	Windows C String
Modem using MNP (Microcom Networking Protocol)	`Modem`	`MNP`
Serial	`Serial`	
AppleTalk ADSP	`ADSP`	

This function returns an error code or zero on no error. Since it declares memory in the heap, it can return memory errors. On Mac OS-based computers, it can return Communications Toolbox (CTB) errors associated

with validating configuration information and with the specific communications tool specified. On Windows-based computers, this can return errors on driver setup and on validation of configuration information.

Here is a sample call to this function for Mac OS-based computers, which includes a configuration string for the Apple Modem Tool.

```
fErr = CDPipeInit(pipe,
                  "Modem",
                  "Modem connection",
                  "ModemType \"Newton Serial\ Connection\" dataBits
8 Parity None Baud 38400 Port ",
                  "Serial-Modem",
                  kDefaultBufferSize, kDefaultBufferSize);
```

Here is a sample call to this function for Windows-based computers:

```
fErr = CDPipeInit(pipe,
                  "MNP",
                  "Modem connection",
                  "COM1:38400,n,8,1",
                  "COM1",
                  kDefaultBufferSize, kDefaultBufferSize);
```

CDPipeListen

```
CommErr CDPipeListen (CDILPipe *pipe,
                      long timeout,
                      CDILCompletionProcPtr completionHook,
                      long refCon ) ;
```

This function starts a listening connection on the pipe, so the CDIL listens for a Newton to initiate a connection. While the CDIL is listening, the pipe has the state `kCDIL_Listening`. If the Newton does not connect before the *timeout,* the state becomes `kCDIL_Disconnected`.

What the state is when this call completes successfully depends on a few factors:

- If you call this function asynchronously, the state will usually be `kCDIL_Listening`. The *completionHook* function is called when the state becomes `kCDIL_ConnectPending`.
- If you call this function synchronously, the state on return depends on the connection type. With some types, `CDPipeListen` does not return until `CDPipeListen` times out or the state is `kCDIL_ConnectPending`. With other types, `CDPipeListen` returns while the state is still `kCDIL_Listening`. Because of this, you should always wait in a loop until the state changes from `kCDIL_Listening` to some other state.

The parameters to the function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
timeout	The timeout to be used in this listening operation. Timeouts are measured in milliseconds on Windows-based computers and ticks (sixtieths of a second) on Mac OS-based computers. The constant `kDefaultTimeout` is defined as 30 seconds. 0 indicates that you want the default timeout. -1 indicates no timeout.
completionHook	Supply this parameter if you want to call this function asynchronously. (See "Getting and Sending Data" on page 2-16 for a description of asynchronous vs. synchronous calls.) This is a pointer to a callback function that is called upon receipt of a connection request or on timeout or other failure. See the description of the callback function that follows this parameter list.
refCon	This is a reference constant to be passed to the callback function. It has no meaning to the CDIL, and is provided so that you can give information to your callback function, if you have one.

This function returns an error code or zero on no error. On Mac OS-based computers, it can return low-level Communications Toolbox (CTB) errors. On Windows-based computers, it can return driver errors. If you call this function asynchronously, it always returns 0 and any error code is returned to the *completionhook* function.

If you provide a *completionhook* parameter, it must be a procedure pointer to a function that follows this definition format:

```
void CompletionHook( CommErr errorValue, long refCon ) ;
```

The parameters to this function are:

errorValue	An error code, if there was an error, or 0, if there was no error.
refCon	A reference constant you supplied to the original function, for your own tracking purposes.

Breaking the Connection

You use `CDPipeDisconnect` to break the connection with the Newton. If you want to break a pending connection, rather than one that has been accepted, call `CDPipeAbort` (page 2-29).

CDPipeDisconnect

```
CommErr CDPipeDisconnect(CDILPipe *pipe) ;
```

This function closes an open pipe. After this call is successful, the pipe has the state `kCDIL_Disconnected`.

Note

Some older 16-bit Windows-based computers may lock up if the Newton device is disconnected after the desktop computer has been disconnected. For best performance in machines such as this, always disconnect the Newton first. ◆

This function returns an error code or zero on no error. On Mac OS-based computers, it can return low-level Communications Toolbox (CTB) errors. On Windows-based computers, this can return driver errors.

The parameter to this function is:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.

Getting and Sending Data

Some functions may be called in either synchronous or asynchronous mode. In **synchronous** mode, the function waits and returns when it has completed its task. In **asynchronous** mode, the function returns immediately, and when the action of the function is completed, CDIL calls a function you've supplied. The function you supply is called a **callback** function. The functions that can be called asynchronously have a function pointer parameter that you use to specify the callback function. If the parameter is `NULL`, the function is called synchronously.

If the operation is performed asynchronously, errors are not returned directly from the call—any error condition is reported to the specified callback routine.

Each pipe can have 25 outstanding asynchronous calls.

Two optional functions, `CDDecryptFunction` and `CDEncryptFunction`, can be used to set up encryption and decryption callback functions to encrypt your data before writing it to the pipe and decrypt your data after reading it from the pipe.

CDDecryptFunction

```
void CDDecryptFunction(CDILPipe *pipe,
                       CDILDecryptionProcPtr decryptFunction,
                       long refCon) ;
```

After you call this function, the CDIL calls *decryptFunction* whenever data is read with `CDPipeRead` or the HLFDIL function `FDget`.

This is an optional function; you do not need to call it unless you want to decrypt your data in this way. This way of decrypting data may not work for all programs; it is possible that you will not get a complete decryption unit from a given pipe read operation. If that is the case with your application, you should accumulate the decryption unit in your program and decrypt it yourself.

Note

Using current software, you cannot encrypt frames on the Newton, so you should not set up a decryption function if you are using the HLFDIL. (You could, though, use the DILs to write an application that communicates with another desktop computer. In that case, you can use decryption even with the HLFDIL.) If you have turned on decryption for a particular pipe, you can turn it off by calling `CDDecryptFunction` giving `NULL` for *decryptFunction*. ◆

The parameters to the function are:

pipe This is a pointer to the internal pipe object returned by `CDCreateCDILObject`.

decryptFunction This is a pointer to a decryption function that the CDIL calls to decrypt the data stream prior to returning data from the read.

refCon This is a reference constant to be passed to the decryption function. It has no meaning to the CDIL, and is provided so that you can give information to your decryption function.

The decryption function definition must follow this format:

```
CommErr DecryptFunc( void *pData, Size Count, long refCon );
```

The parameters to the function are:

pData This is a pointer to the data buffer which you passed in when you called `CDPipeRead`. The data there should be decrypted and written back out to the buffer, taking care not to exceed the size of the buffer.

Count	This is the number of bytes of data in the buffer.
refCon	This is a reference constant you supplied to `CDDecryptFunction` for your own tracking purposes.

You should return a non-zero error code if something goes wrong, such as a buffer overflow. Note that you must return zero (0) if there is no error; if you do not supply a return value, a random non-zero value is returned and the CDIL assumes that means there has been an error and aborts processing of the current read operation.

CDEncryptFunction

```
void CDEncryptFunction(CDILPipe *pipe,
                       CDILEncryptionProcPtr encryptFunction,
                       long refCon) ;
```

After you call this function, the CDIL calls *encryptFunction* whenever data is written to the pipe with `CDPipeWrite` or the HLFDIL function `FDput`.

This is an optional function; you do not need to call it unless you want to encrypt your data.

Note

Using current software, you cannot decrypt frames on the Newton, so you should not set up an encryption function if you are using the HLFDIL. (You could, though, use the DILs to write an application that communicates with another desktop computer. In that case, you can use encryption even with the HLFDIL.) If you have turned on encryption for a particular pipe, you can turn it off by calling `CDEncryptFunction` giving `NULL` for *encryptFunction*. ◆

The parameters to the function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
encryptFunction	This is a pointer to the encryption function that the CDIL calls to encrypt the data stream prior to writing it to the pipe.

refCon This is a reference constant to be passed to the encryption function. It has no meaning to the CDIL, and is provided so that you can give information to your encryption function.

The encryption function definition follows this format:

```
CommErr EncryptFunction(void *pData, Size Count, long
refCon );
```

The parameters to the function are:

pData This is a pointer to the data buffer which you passed in when you called `CDPipeWrite`. The data there should be encrypted and written back out to the buffer, taking care not to exceed the size of the buffer.

Count This is the number of bytes in the buffer.

refCon This is a reference constant you supplied to `CDEncryptFunction` for your own tracking purposes.

You should return a non-zero error code if something goes wrong, such as a buffer overflow. Note that you must return zero (0) if there is no error; if you do not supply a return value, a random non-zero value is returned and the CDIL assumes that means there has been an error and aborts processing of the current write operation.

CDPipeRead

```
CommErr CDPipeRead(CDILPipe *pipe,
                   void* p,
                   long* count,
                   Boolean* eom,
                   long  swapSize,
                   long  destEncoding,
                   long  timeOut,
                   CDILPipeCompletionProcPtr completionHook,
                   long refCon) ;
```

This function reads data from the Newton into the specified buffer *p*. All byte swapping and data conversions are performed automatically. As implied by the *completionHook* parameter, this function can complete asynchronously.

The parameters to the function are:

pipe The pointer to the internal pipe object returned by `CDCreateCDILObject`.

p Data is read into the buffer pointed to by this address. If you've installed a decryption function (see "CDDecryptFunction" on page 2-16) you should make sure this buffer is large enough to hold the decrypted data.

count The number of bytes to read. The number of bytes actually read is returned in this parameter.

eom This value is set by the underlying transport code and currently is always returned as `false`. Some future transport types may set this value to `true` if the code determines there is no more data. You can pass in `NULL` instead of a pointer to a `Boolean` if you don't care about this value.

swapSize This is usually useful only on Windows-based computers. The `swapSize` value deals with switching

Newton values from big-endian to little-endian. This is the size of byte chunks to swap, if byte-swapping is to be used. This value should be zero to avoid byte-swapping. A computer can be described as being either big-endian or little-endian depending on how it arranges bytes within a word. In a big-endian system, byte 0 is always the most significant (leftmost) byte. In a little-endian system, byte 0 is always the least significant (rightmost) byte. A Mac OS-based computer is a big-endian system; an Intel x86 machine is a little-endian system. The ARM processor used in Newton can operate in either mode; Newton uses it in big-endian mode. Thus, Mac OS-based computers and the Newton are both big-endian, while Windows-based computers generally run on little-endian computers. Only buffers 128 bytes or smaller can be byte-swapped. What the *swapSize* value should be depends on the type of data. For example, if you want to swap Unicode strings, the *swapSize* should be 2 (or `sizeof(short)`); if you want to swap `long` values, the *swapSize* should be 4 (or `sizeof(long)`.

destEncoding Selects the encoding tables for Unicode conversion. Encoding is used to convert Unicode characters to ASCII and to convert ASCII characters to Unicode. Mac OS-based computers and Windows-based computers use different encoding tables for ASCII character codes above 127. If you do not want Unicode conversion, set

this value to `kUnicode`. You currently can use the enumerated values shown in Table 2-4.

Table 2-4 Encoding Table Values

Enumerated Value	Meaning
`kUnicode`	Do not do Unicode conversion
`kMacRomanEncoding`	Mac OS Roman
`kPCRomanEncoding`	PC Roman

timeout The timeout to be used in this read operation. Timeouts are measured in milliseconds on Windows-based computers and ticks (sixtieths of a second) on Mac OS-based computers. 0 indicates that you want the default timeout, which is 30 seconds. -1 indicates no timeout.

completionHook Supply this parameter if you want to call this function asynchronously. This is a pointer to a callback function that is called when the data is received or the operation times out or otherwise fails (see description of the callback function following this parameter list). Set to `NULL` if you want the function called synchronously.

refCon This is a reference constant to be passed to the callback function. It has no meaning to the CDIL, and is provided so that you can give information to your callback function, if you have one.

This function returns an error code or zero on no error. If you call this function asynchronously, it always returns 0 and any error code is returned to the *completionhook* function. On Mac OS-based computers, this function can return low-level Communications Toolbox (CTB) errors. On Windows-based computers, it can return driver errors. This function can also return an unknown exception if the internal pipe object has reached a bad

state. (An **exception** is an error or other exceptional condition.) In addition, since this function declares memory on the heap, it can return memory errors. In certain conditions, this function will return `kPipeNotInitialized`, `kPipeNotReady`, or `kInvalidParameter`.

If you want to decrypt your data, set up your decryption function by calling `CDDecryptfunction` (page 2-16).

If you provide a *completionhook* parameter, it must be a procedure pointer to a function that follows this definition format:

```
void CompletionHook( CommErr errorValue,
                     void *pData, Size Count,
                     long refCon, long flags ) ;
```

Any data returned is in the buffer pointed to by the *pData* parameter.

The parameters to the function are:

errorValue	An error code or zero. This is the same value that would be the return value of `CDPipeRead` in a synchronous call.
pData	Pointer to the data buffer returned to the function.
Count	Number of bytes successfully transferred in the read operation.
refCon	A reference constant you supplied to the original function, for your own tracking purposes.
flags	This may be zero or 1, with a *flags* value of 1 indicating that an end-of-message indicator was reached.

CDPipeWrite

```
CommErr CDPipeWrite(  CDILPipe *pipe,
                      void* p,
                      long* count,
                      Boolean eom,
                      long  swapSize,
                      long  srcEncoding,
                      long  timeOut,
                      CDILCompletionProcPtr completionHook,
                      long refCon ) ;
```

This function writes data to the Newton from the specified buffer *p*. As implied by the parameters, this function can complete asynchronously.

The parameters to the function are:

pipe The pointer to the internal pipe object returned by `CDCreateCDILObject`.

p Data is written from the buffer pointed to by this address. If you have installed an encryption function (see "CDEncryptFunction" on page 2-18) you should be certain that this buffer is big enough to hold the encrypted data.

count The number of bytes to write. The number of bytes actually written is returned in this parameter.

eom If you set this to `true`, the CDIL flushes the buffer after this write. If you don't want the buffer flushed, set this to `false`.

swapSize This is usually only useful on Windows-based computers. The `swapSize` value deals with switching Newton values from big-endian to little-endian. This is the size of byte chunks to swap, if byte-swapping is to be used. This value should be zero to avoid byte-swapping. A computer can be described as being either big-endian or little-endian depending on how it

arranges bytes within a word. In a big-endian system, byte 0 is always the most significant (leftmost) byte. In a little-endian system, byte 0 is always the least significant (rightmost) byte. A Mac OS-based computer is a big-endian system; an Intel x86 machine is a little-endian system. The ARM processor used in Newton can operate in either mode; Newton uses it in big-endian mode. Thus, Mac OS-based computers and the Newton are both big-endian, while Windows-based computers generally run on little-endian computers. Only buffers 128 bytes or smaller can be byte-swapped. What the *swapSize* value should be depends on the type of data. For example, if you want to swap Unicode strings, the *swapSize* should be 2 (or `sizeof(short)`); if you want to swap `long` values, the *swapSize* should be 4 (or `sizeof(long)`.

srcEncoding To select the encoding tables for Unicode conversion. Encoding is used to convert Unicode characters to ASCII and to convert ASCII characters to Unicode. Mac OS-based computers and Windows-based computers use different encoding tables for ASCII character codes above 127. If you do not want Unicode conversion, set this value to `kUnicode`. You currently can use the enumerated values shown in Table 2-5.

Table 2-5 Encoding Table Values

Enumerated Value	Meaning
`kUnicode`	Do not do Unicode conversion
`kMacRomanEncoding`	Mac OS Roman
`kPCRomanEncoding`	PC Roman

timeout	The timeout to be used in this write operation. Timeouts are measured in milliseconds on Windows-based computers and ticks (sixtieths of a second) on Mac OS-based computers. 0 indicates that you want the default timeout, which is 30 seconds. -1 indicates no timeout.
completionHook	Supply this parameter if you want to call this function asynchronously. This is a pointer to a callback function that is called when the data is written or the operation times out or otherwise fails (see the description of the callback function following this parameter list). Set to `NULL` if you want the function called synchronously.
refCon	This is a reference constant to be passed to the callback function. It has no meaning to the CDIL, and is provided so that you can give information to your callback function, if you have one.

This function returns an error code or zero on no error. If you call this function asynchronously, it always returns 0 and any error code is returned in the *completionhook* function. On Mac OS-based computers, it can return low-level Communications Toolbox (CTB) errors. On Windows-based computers, this can return driver errors. This function can also return an unknown exception if the internal pipe object has reached a bad state. In addition, since this function declares memory on the heap, it can return memory errors. In certain conditions, this function will return `kPipeNotInitialized`, `kPipeNotReady`, or `kInvalidParameter`.

Note
You should be careful not to send more data than the Newton communication buffers can handle. If you do, data will be lost. One way to make sure is to have your Newton application send an "all clear" message to the desktop application once it has received one batch of data. Your desktop application can then wait for that message before calling `CDPipeWrite` again. Note that this is not a built-in capability; you need to write the code for the "all-clear" protocol yourself. ◆

If you want to encrypt your data, set up the encryption function using `CDEncryptFunction` (page 2-18).

If you provide a *completionhook* parameter, it must be a procedure pointer to a function that follows this definition format:

```
void CompletionHook( CommErr errorValue, long refCon ) ;
```

The parameters to the function are:

errorValue	An error code or zero. This is the same value that would be the return value of `CDPipeWrite` in a synchronous call.
refCon	A reference constant you supplied to the original function, for your own tracking purposes.

Pipe Maintenance

This group has some functions you can use to abort operations, to flush the pipe to make certain that data has been written out, and to perform idle processing. Most applications need to call the idle function, `CDIdle`, periodically in their main event loop. The other functions are optional.

CDFlush

```
CommErr CDFlush ( CDILPipe *pipe, CDILPipeDirection dir) ;
```

This function flushes the specified pipe, and ensures that all data is written out to the appropriate driver. The pipe is cleared of all data. Read calls that are waiting for data that is not yet in the pipe continue to wait.

If the input is bad, this function does nothing and returns an error code. Otherwise, it returns zero.

pipe The pointer to the internal pipe object returned by `CDCreateCDILObject`.

dir This indicates the direction of the pipe to flush. The pipe direction constants are shown in Table 2-7.

Table 2-6 Pipe Direction Constants

Constant	Meaning
`kUndefinedDirection`	No defined direction; if you give this, the function does nothing
`kReadPipe`	Read pipe direction; if you give this, and there are no read calls waiting for data, any data coming from the Newton is discarded
`kWritePipe`	Write pipe direction
`kAllPipes`	Apply to read and write direction

CDIdle

```
void CDIdle (CDILPipe *pipe) ;
```

This function performs required idle processing. You should call this function periodically, generally from your program's main event loop, whenever the pipe is open.

`CDIdle` manages asynchronous calls and maintains the CDIL's internal buffer. There is no need to call it if you are certain that the Newton is not

sending any data. (You should be sure that no data is going to be received because the CDIL's internal buffer can easily overflow if data is received and `CDIdle` is not called.)

How frequently you should call this depends on the amount of data to be transferred and how often it is transferred, and needs to be determined by experimentation. If you're losing data, you can try calling `CDIdle` more frequently. On the other hand, if you call `CDIdle` more than necessary, it can slow your application.

The parameter to this function is:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDPipeAbort

```
CommErr CDPipeAbort ( CDILPipe *pipe,
                      CDILPipeDirection dir ) ;
```

This function cancels any pending pipe operations on the selected pipe direction. If a connection is pending, the connection is rejected. While the aborting process is going on, the pipe state is `kCDIL_Aborting`. When it is finished, the state is `kCDIL_Disconnected`.

The parameters to the function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.

dir This indicates the direction of the pipe to abort. The pipe direction is defined by the constants shown in Table 2-7.

Table 2-7 Pipe Direction Constants

Constant	Meaning
`kUndefinedDirection`	No defined direction; if you give this, the function does nothing
`kReadPipe`	Read pipe direction
`kWritePipe`	Write pipe direction
`kAllPipes`	Apply to read and write direction

This function returns an error code or zero on no error. On Mac OS-based computers, it can return low-level Communications Toolbox (CTB) errors. On Windows-based computers, this can return driver errors. This function can also return an unknown exception if the internal pipe object has reached a bad state.

Information Functions

You use the following functions to get and set information about the pipe.

CDBytesInPipe

```
CommErr CDBytesInPipe(CDILPipe *pipe,
                      CDILPipeDirection dir,
                      long *count) ;
```

This function obtains the current number of bytes in the pipe.

The return value is zero if everything is fine. If there is bad input or another problem, this function returns an error code.

The parameters of this function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
dir	Gives the pipe direction. The pipe directions you can use are shown in Table 2-8. Only one pipe direction may be specified, so the `kAllPipes` value is not allowed in calls to this function.
count	The current number of bytes in the pipe.

Table 2-8 Pipe Direction Constants

Constant	Meaning
`kUndefinedDirection`	No defined direction; if you give this, you always get a count of zero
`kReadPipe`	Read pipe direction
`kWritePipe`	Write pipe direction

CDConnectionName

```
char * CDConnectionName ( CDILPipe *pipe ) ;
```

This function returns the current connection name, which you set using `CDPipeInit` (see page 2-10).

It does not return any error condition, but returns NULL if there is a bad pipe state and the request can not be answered for some reason.

The parameter to this function is:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDGetConfigStr

```
char * CDGetConfigStr ( CDILPipe *pipe ) ;
```

This returns the configuration string that you set using `CDPipeInit` (see page 2-10).

It does not return any error condition, but returns NULL if there is a bad pipe state and the request can not be answered for some reason.

The parameter to this function is:

pipe The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDGetPipeState

```
CDIL_State CDGetPipeState (CDILPipe *pipe) ;
```

This function returns the current state of the pipe. The state values are shown in Table 2-1 on page 2-5. Table 2-2 on page 2-6 shows the state transitions.

The parameter to this function is:

pipe The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDGetPortStr

```
char * CDGetPortStr ( CDILPipe *pipe ) ;
```

This function returns the current port name, which you set using `CDPipeInit` (see page 2-10). For example, for a Windows-based computer, this might be `"COM2".`

It does not return any error condition, but returns `NULL` if there is a bad port state and the request can not be answered for some reason.

The parameter to this function is:

pipe The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDGetTimeout

```
long CDGetTimeout ( CDILPipe *pipe ) ;
```

This function returns the current timeout setting. The timeout is set by individual `CDPipeRead` and `CDPipeWrite` calls.

A return value of -1 indicates there is no timeout.

It does not return any error condition, but returns zero if the connection has a bad state and the request can not be answered for some reason.

The parameter to this function is:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.

CDSetPipeState

```
CommErr CDSetPipeState ( CDILPipe *pipe,
                         CDIL_State your_state) ;
```

This function allows you to set the state for a pipe to any value, so you can set it to a value that has meaning to your program. This value should be a positive number greater than the `kCDIL_Userstate` value. (See "CDGetPipeState" on page 2-32.) If you have called this function to set your own value, and then want to get communications-based state information from `CDGetPipeState`, call this function and pass in a zero.

This function returns an error code or zero on no error.

The parameters to this function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
your_state	The new state value.

Advanced Functions

The functions in this section perform advanced CDIL functions.

CDPad

```
void CDPad ( CDILPipe *pipe, long length ) ;
```

This function pads the write buffer to the boundary indicated by the parameter. (That is, it fills the write buffer with leading 0 bytes until the length of the data in the buffer is a whole-number multiple of *length*.) It will do nothing if the input is bad, such as a negative number.

The parameters to this function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
length	A number of bytes.

CDSetPadState

```
void CDSetPadState ( CDILPipe *pipe, Boolean  paddingOn ) ;
```

This function turns automatic padding on or off. It is initially off. If you haven't called `CDPad`, padding is to four-byte boundaries.

Automatic padding is done only for write operations where the `eom` parameter is set to `true`.

The parameters to this function are:

pipe	The pointer to the internal pipe object returned by `CDCreateCDILObject`.
paddingOn	Indicates whether padding should be turned on (`true`) or turned off (`false` or any other value).

Error Codes

Table 2-9 shows the error codes currently defined for the CDIL. You can also get some Newton Connection errors, because the Newton uses some

Connection-related code to communicate with the DILs. Those are shown in Table 2-10.

Table 2-9 CDIL Error Codes

Error Code	Numerical Value	Meaning
`kCommErrNoErr`	`0`	No error
`kOutOfMemory`	`-28701`	Error on memory allocation
`kBadPipeState`	`-28702`	DIL pipe was set to a bad state
`kExceptionErr`	`-28703`	An unknown exception has occurred
`kQueueFullError`	`-28704`	The queue of asynchronous calls is full; there can be no more than 25 outstanding asynchronous calls for a given pipe
`kPipeNotInitialized`	`-28705`	Pipe has not been initialized
`kInvalidParameter`	`-28706`	Parameter passed in was invalid
`kPipeNotReady`	`-28707`	Pipe is not ready for operation

Table 2-10 Newton Connection Error Codes

Error Code	Numerical Value	Meaning
`kDAborted`	`-28003`	The communication operation was aborted.
`kDBadConnection`	`-28009`	Connection-related code on the Newton encountered a serious error while establishing or in the course of using a connection.
`kDOutOfMemory`	`-28017`	Connection-related code ran out of memory.
`kDCantConnectToModem`	`-28029`	The modem isn't responding. It may not be plugged in.
`kDDisconnected`	`-28030`	The connection has been torn down. Maybe there was a line failure, maybe the Newton timed out.
`kDDisconnectInRead`	`-28100`	While reading, Connection-related code detected a disconnect.
`kDReadFailed`	`-28101`	While reading, Connection-related code encountered an error.
`kDCommToolNotFound`	`-28102`	The communications tool you requested is not found in the extensions folder.
`kDBadModemToolVersion`	`-28103`	On Mac OS-based computers, the DILs only support Apple Modem Tool 1.5.1, currently.

The High-Level Frames Desktop Integration Library

The High-Level Frames Desktop Integration Library (HLFDIL) helps desktop applications:

- Create C data structures that mimic Newton object formats
- Receive data that a Newton sends in Newton object formats
- Send data that appears to a Newton to be in Newton object formats

You need to use the CDIL, described in the previous chapter, to make the connection with the Newton.

Although the HLFDIL is written in C++ and is based on C++ objects, you use it by making calls to C functions. The `FDCreateObject` function creates a C++ object, and returns a pointer to that object. You use that pointer in calling other functions. (Those functions are actually "wrappers" for C++ methods.)

General Concepts

HLFDIL supports the following features:

- The HLFDIL takes C data types and converts them to Newton objects.
- The HLFDIL produces **flattened** frames that can be sent over a communications link and "inflates" flattened frames that are received over a communication link. The frame is a structured type similar to a C `struct`. Flattening is a process where a frame is reduced to a series of bytes that can be sent over an output stream. You never see flattened frames; the frame data is inflated before your application gets it.

Objects Handled by the HLFDIL

This section discusses the Newton object model and the corresponding HLFDIL object model. See the *NewtonScript Programming Language* for more information on Newton types.

The Newton Object Model

To use the HLFDIL, you create data structures in the desktop program that correspond to Newton data structures. The individual pieces of Newton data end up in variables or other reserved space in the desktop program. To understand how you should set up these variables, you need to understand how Newton data is structured.

The Newton object model uses two kinds of 32-bit values to represent objects:

- **immediate**, in which the 32 bits contain data
- **reference**, in which the 32 bits refer indirectly to an object; the object referenced can be an immediate or another reference

You cannot directly transfer an immediate value to a desktop computer; all immediate data that is transferred over a communications link is accessed through a reference.

References refer to four basic types of objects:

- immediate, which is a piece of data. Examples of immediates are characters and integers. Although 32 bits are allocated for immediate objects, two bits are used for the class, so the data can be up to 30 bits long.
- **binary**, which is a reference to a string of bytes. Examples of binary objects are strings and symbols. In a desktop program, you would ignore the fact that this is a reference, and represent it by a variable of the appropriate type. A binary is also considered immutable, because when you assign a new value to a binary, the original string of bytes is destroyed rather than modified.
- **array**, which is an array of values. The values can be references or immediates. The HLFDIL defines an array object that represents the array; you attach a C array or other reserved space to the object to hold the array's data.
- **frame**, which is a set of tag/value pairs (called "slots"). The values can be references or immediates. A frame is used in the same ways as a record is used in other data representations. The HLFDIL defines a frame object that represents the base object; you attach variables to that object to hold the slot data.

When you send data between a Newton and a desktop computer, you generally send frames, although you can also send arrays. The data is contained in the slots of the frame or the elements of the array. Those slots or elements can be references or immediates. If they are references, they can be binaries, arrays, or frames. Those second-level arrays and frames can also contain immediates or references. Ultimately, when you send a frame or array, you send a tree structure, where the data is contained in the leaves of the tree.

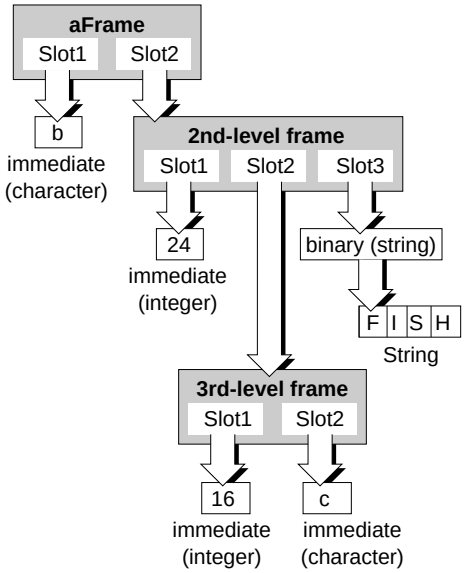
For example, consider a frame that is defined in NewtonScript this way:

```
aFrame := { slot1: $b,
            slot2: { slot1 : 24,
                     slot2 : {   slot1 : 16,
                                 slot2 : $c},
                     slot3 : "FISH"} }
```

This defines the frame shown in Figure 3-1. It has a top level frame, `aFrame`, that has two slots:

- `slot1`, which contains the character b
- `slot2`, which contains an unnamed frame.

Figure 3-1 Sample Frame on the Newton



The unnamed frame has three slots:

- `slot1`, which contains the integer 24
- `slot2`, which contains another unnamed frame
- `slot3`, which contains a binary object that is the string `"FISH"`

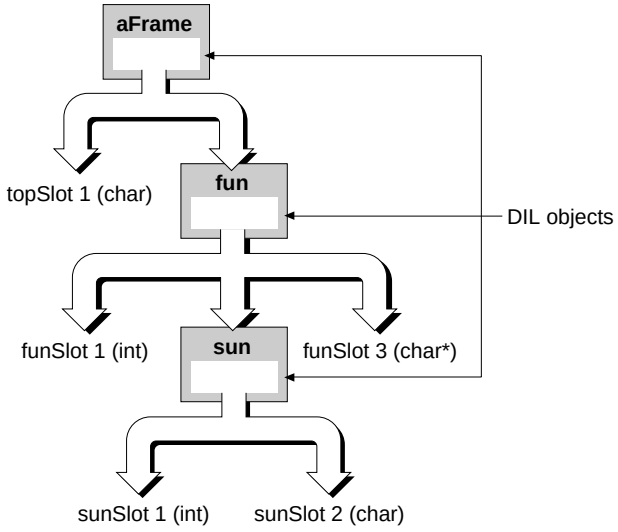
The third-level unnamed frame has two slots:

- `slot1`, which contains the integer 16

- `slot2`, which contains the character c

To get this frame from HLFDIL, you would create a set of objects that "shadow" the NewtonScript object, as illustrated in Figure 3-2. You need to create variables or reserve other space for the data that is contained in the leaves of the tree. You create HLFDIL objects for the structures that make up the branches of the tree.

Figure 3-2 HLFDIL Representation on the Desktop



Here is some C code that would create the structure that you need. You can't have unnamed objects in C, so the object that represents the first unnamed frame is called `fun` (for first unnamed), and the object that represents the second unnamed frame is called `sun` (for second unnamed).

Note

The code in this chapter does not show any error handling. You should be sure to check for errors whenever you call DIL functions; most of the functions return an error code or `kobjErrNoErr`, which is 0. The possible HLFDIL errors are shown in "Error Codes" beginning on page 3-25. ◆

```
//These are the variables for the "leaf" data
char topSlot1;
int funSlot1;
char* funSlot3;
int sunSlot1;
char sunSlot2;

//These are used to define the data lengths
int intlength = 4;
int charlength = 1;
int strlength = 4;

/*These are the objects for the branches of the tree.
The HLFDIL returns pointers to the objects, which you
store in DILObj* variables.*/
DILObj *aFrame, *fun, *sun ;
aFrame= FDCreateObject ( kDILFrame, NULL);
fun= FDCreateObject ( kDILFrame, NULL);
sun= FDCreateObject ( kDILFrame, NULL);

/*The following calls "bind" the leaves and objects
together into the necessary structure.
First, put together the top-level frame with its slots.
Here is slot1, which is an integer.*/
fErr= FDbindSlot( aFrame,
                  "slot1",
                  (void *)&topSlot1,
```

```
                kDILCharacter,
                charlength, -1, NULL) ;
/* Here is slot2, which is the first unnamed frame,
represented by fun */
fErr = FDbindSlot(   aFrame, "slot2",
                     fun,kDILFrame,0, -1, NULL);
/* The following binds the slots of fun */
fErr = FDbindSlot(   fun,
                     "slot1",
                     (void*)&funSlot1,
                     kDILInteger,
                     intlength, -1, NULL);
fErr = FDbindSlot(   fun, "slot2",
                     sun, kDILFrame,0, -1, NULL);
fErr = FDbindSlot(   fun,
                     "slot3",
                     (void*)&funSlot3,
                     kDILCharacter,
                     charlength, -1, NULL);
/* The following binds the slots of sun */
fErr = FDbindSlot(   sun,
                     "slot1",
                     (void*)&sunSlot1,
                     kDILInteger,
                     intlength, -1, NULL);
fErr = FDbindSlot(   sun,
                     "slot2",
                     (void*)&sunSlot2,
                     kDILCharacter,
                     charlength, -1, NULL);
```

As you can see from this example, you associate a Newton frame with an HLFDIL object by telling the HLFDIL to create an object of type `kDILFrame`. You then make a series of function calls that describe the slots of the frame or

the elements of the array. This is called **binding** the Newton slots to the DIL object; thus, the function you use to make the association is `FDbindSlot`.

You handle an array in a similar way. For example:

```
/*First reserve space for the array data. This assumes
the Newton array consists of 100 Newton integers*/
long CArray[100];
/*Then create a kDILArray object*/
DILObj* anArray;
anArray = FDCreateObject ( kDILArray, NULL);
/*Finally, bind the array space to the array object*/
fErr = FDbindSlot(   anArray, "",
                     (void *)&CArray,
                     kDILArray, 0, -1, NULL);
```

Arrays on the Newton can be named or anonymous. This array is anonymous. If you wanted to use a named array, you would give a name instead of an empty string as the second parameter for the `FDbindSlot` call.

As you can see, when you use the HLFDIL you bind your Newton data types to HLFDIL objects. These HLFDIL objects are then used to:

- receive information from the Newton
- send information to the Newton
- allow access to Newton data in the desktop application

Because you are responsible for binding the data, two types of data can be received from the Newton: bound and unbound data.

Bound data is data that has a structure matching an HFDIL object that you have created and defined.

Unbound data is data that was received from the Newton that does not correspond to any known data binding. You can ignore this data or you can retrieve it using the `FDGetUnboundList` function. See “Unbound Data” on page 3-19 for more information.

Slots, array elements, and immediates can have names associated with them. There can only be one symbol with any particular name at a given level within a given object. Thus, when you bind a slot with a particular name to a DIL object, you cannot bind a slot with the same name to the same DIL object.

Every DIL object is internally assigned a class that is related to the corresponding Newton class. That is the purpose of the symbols you can see in the sample `FDbindSlot` calls; `kDILInteger`, for example, tells the HLFDIL that the Newton class of that slot is `Integer`.

Using the HLFDIL

The HLFDIL is designed to be a flexible system that you can use in different ways. This section describes the most common use of the HLFDIL.

The following describes creating HLFDIL objects and using the objects. The sample object has a frame that contains two strings and a frame. The frame that is contained in the top-level frame itself has two slots.

1. Define space for the individual pieces of Newton data. You can do this in any way you wish. For example:

```
char fName[256] = "Mickey";
char lName[256] = "Duck";
char addr[256]  = "123 Chenery Street";
char city[256] = "Anytown";
```

   You could also use `new` to allocate space; it does not matter how the space is reserved. There must be space reserved for each Newton slot, immediate, or array element, but not for slots or array elements that contain frames or arrays.

2. Initialize the HLFDIL. For example:

```
fErr = FDInitFDIL ( );
```

   This assumes you've already declared `fErr` as an `objErr` type value.

3. Create one or more HLFDIL objects for your data. For example:

```
entry = FDCreateObject (kDILFrame, "") ;
name = FDCreateObject (kDILFrame, "") ;
```

These assume that you've declared `entry` and `name` as `DILObj*` variables.

Notice that you create an object for the top-level frame, `entry`, and also for the frame, `name`, that is contained in a slot of `entry`. You can see the binding that places the `name` object into the `entry` object in the next step.

4. Bind the slots in your Newton structures to the new DIL objects. For example:

```
fErr = FDbindSlot(   name,"First",(void *)&fName, kDILString,
                     strlen(fName), 0, NULL);
fErr = FDbindSlot(   name,"Last", (void *)&lName, kDILString,
                     strlen(lName),0,NULL);
fErr = FDbindSlot(entry, "Name", name, kDILFrame, 0,0,NULL);
fErr = FDbindSlot(   entry, "Address", (void*)&addr, kDILString,
                     strlen(addr), 0, NULL) ;
fErr = FDbindSlot(   entry, "City", (void *)&city, kDILString,
                      strlen(city), 0, NULL);
```

Notice that the third binding call is different from the others; it binds the `name` object (which represents a frame) in as a slot in the `entry` object.

These calls create a two-level structure, where there is a `name` frame that is contained within an `entry` frame.

5. Open a pipe to the Newton. See the CDIL chapter for information on initializing the CDIL and opening the pipe. Note that an application on the Newton must be running initiate the connection. See Chapter 4 for information on writing a Newton application for that purpose.

6. Once the pipe is open, you can send or receive frames. For example, this sends a frame:

```
fErr = FDput(entry, kDILFrame, thePipe ) ;
```

This call takes the data that is in the variables bound to `entry` (`addr` and `city` and in the subframe name, `fName` and `lName`) and puts that data in `thePipe`.

7. To get some data from the Newton call:

```
fErr = FDget(entry, kDILFrame, thePipe, 15, NULL, 0);
```

(See "FDget" beginning on page 3-16 for details of the parameters.) The HLFDIL waits for data to come down the pipe. If the format of the data matches that defined for `entry`, the data is placed in the variables bound to `entry`. Any data whose format does not match `entry` or anything bound to `entry` is placed on the unbound data list. You can get the unbound data list by calling `FDgetUnboundList`.

8. When you are done, you need to use HLFDIL calls to destroy the objects you have created and shut down the HLFDIL.

```
fErr = FDDisposeObject(name);
fErr = FDDisposeObject(entry);
fErr = FDDisposeFDIL();
```

9. You also need to shut down the CDIL. See the CDIL chapter for information and examples for doing that.

HLFDIL Reference

The functions in this section are divided into related areas.

Setting Up and Shutting Down

Before beginning, you must call the `FDInitFDIL` function. Before shutting down your application, you should call `FDDisposeFDIL`.

FDInitFDIL

```
objErr FDInitFDIL ( void );
```

This function initializes the underlying object system mechanism and prepares the environment for using DILs.

FDDisposeFDIL

```
objErr FDDisposeFDIL ( void ) ;
```

This function shuts down the underlying object system mechanism and cleans up the environment. You generally call it as part of application shutdown.

Creating, Destroying, and Defining Objects

You need to create objects that represent Newton objects, and define the structures of the objects to match the Newton structures that you need. When you are done with the objects, you need to free the memory they use by disposing of the objects.

FDCreateObject

```
DILObj *FDCreateObject ( short objectType, char *objectClass );
```

This function tells the HLFDIL to create an object.

The return value is a pointer to the new object.

The parameters to this function are:

objectType The type of object that you want created. Give either `kDILFrame` or `kDILArray`.

objectClass You supply this parameter if you have an array with a class. It can be a string giving the Newton class of the object you want created, which you use if you want to create a class derived from one of the DIL object classes.

FDDisposeObject

```
objErr FDDisposeObject  ( DILObj *Dobj );
```

This function tells the HLFDIL to dispose of an object. You should call this when you are done with the object in order to free up the memory used by the object.

The parameter to this function is:

Dobj	A pointer to the object that you want to destroy.

FDbindSlot

```
objErr FDbindSlot ( DILObj *Dobj, char *slotName, void
*bindVar, short varType, long maxLen, long curLen, char
*objClass ) ;
```

This function binds something to a slot of a DIL object, so that it represents a slot of a corresponding Newton object. The thing you bind is always either a variable, a piece of allocated memory, or another DIL object.

The parameters to this function are:

Dobj	The DIL object returned by `FDCreateObject`.
slotName	This is the name of the slot to be bound. This is required for frames and is required for arrays.
bindVar	This is a pointer to a variable or allocated memory that will hold the information for this slot. That memory is bound to *Dobj*. If this slot contains a frame or array, you must have already created a `DILFrame` or `DILArray` object for that frame or array; you give a pointer to that `DILFrame` or `DILArray` object and that `DILFrame` or `DILArray` object is bound to this DIL object. Unlike the other parameters of this function, the object you give for *bindVar* is referenced during the `FDbindSlot` call but is not used until you try to send data from the slot to a Newton using `FDput` or get data from a Newton and place it in the slot using `FDget`.

When you use `FDget` or `FDput`, you must ensure that the memory blocks bound to the DIL object are allocated and fixed until the completion of `FDget` or `FDput`. For instance, if you plan to use an asynchronous `FDput`, the space allocated for the slots of the DIL object must not be memory locations on the stack, in memory locations that might move (such as unlocked Mac OS memory handles), or in locations of memory used by an `FDget` or an `FDput` before the first `FDput` is completed.

varType This is the data type of this slot; see Table 3-1 on page 3-22.

maxLen This is the maximum length of this data slot. If you're binding a frame or array, give zero for this.

curLen This is the current length of the data in the slot. -1 means that the current length should be considered to be the same as *maxLen*. If you're binding a frame or array, this should be zero or -1. If you're binding another type, other than a binary, this should be -1. For a binary, you can use this value if the bound space is essentially a buffer that is not currently full, and you can give the current size. The HLFDIL changes the current length value dynamically when data is received from the Newton.

objClass This is the class of the object used in the Newton object system. For example, the class of a binary object could be `"Real"`. Use `NULL` to specify no class.

This function returns an error code or a zero if there is no error.

Here is an example binding call:

```
fErr = FDbindSlot(   obj, "TheSlot", (void *)&myData, kDILString, 7,
                    -1, NULL) ;
```

This call creates a slot named `TheSlot` in the DIL object pointed to by `obj`. The data of the slot is in a variable called `myData`. The data is a string with a

maximum length of 7 bytes. The `curLen` value is set to -1, which specifies that the current length is the same as the maximum length. The object has a `NULL` class value.

Getting Data To and From the Newton

The functions in this section allow you to get objects from an input stream and put objects into an output stream. Before you call these functions, you need to set up a pipe. You use the CDIL to set up a pipe; see the CDIL chapter for more information.

Objects are transmitted through a stream in a special format called a flattened frame. HLFDIL automatically converts objects to and from flattened frame format, so you do not have to deal with that format.

FDput

`objErr FDput(DILObj *`*Dobj*`, short` *objectType*`, CDILPipe*`*pipe*`);`

This function sends the given Newton object through the pipe specified.

The parameters to this function are:

Dobj	The DIL object that has the data you want to send to the Newton. This is the pointer returned by `FDCreateObject`, to which you have bound the variables containing the data by using `FDbindSlot` calls.
objectType	The type of the object. Give either `kDILFrame` or `kDILArray`. This is almost always `kDILFrame`, because that is the basic type used for transport to the Newton.
pipe	This is the CDIL pipe to use in communications with the Newton.

This function returns an error code upon encountering an error and a zero on no error.

Note

You should be careful not to send more data than the Newton communication buffers can handle. If you do, data will be lost. One way to make sure is to have your Newton application send an "all clear" message to the desktop application once it has received one batch of data. Your desktop application can then wait for that message before calling `FDput` again. Note that this is not a built-in capability; you need to write the code for the "all-clear" protocol yourself. ◆

Here is an example `FDput` call:

```
fErr = FDput(obj, kDILFrame, thePipe ) ;
```

This puts the data that is in the variables bound to `obj` into the pipe. You must have opened and initialized the pipe first.

FDget

`objErr FDget ( DILObj *`*Dobj*`, short` *objectType*`, CDILPipe *`*pipe*`, long` *timeOut*`, CDILPipeCompletionProcPtr` *completionHook*`, long` *refCon* `);`

This function reads data from the pipe specified. The data is delivered in the variables bound to *Dobj*. Data that does not match the format of *Dobj* is placed on the unbound data list. (See "Unbound Data" on page 3-19.)

`FDget` can be called **synchronously** or **asynchronously**. When you call it synchronously, the function does not return until it gets data or the operation times out or otherwise fails. When you call `FDget` asynchronously, the function returns immediately. In your function call, you supply a **callback function** that the HLFDIL calls when the operation is completed.

The parameters to this function are:

Dobj	The DIL object pointer that you have used in `FDbindSlot` calls to describe the kind of data you want.
objectType	The type of the object. Give either `kDILFrame` or `kDILArray`.

pipe	This is the CDIL pipe to use in communications with the Newton.
timeout	The timeout to be used in this read. Timeouts are measured in milliseconds on Windows-based computers and ticks (sixtieths of a second) on Mac OS-based computers. 0 indicates that you want the default timeout, which is 30 seconds. -1 indicates no timeout.
completionHook	This function can be called synchronously or asynchronously. If you supply a *completionHook* value, the function is called asynchronously, and *completionHook* is a pointer to a callback function that is called upon receipt of data. (See information following the parameter list.) Supply `NULL` if you want this function to be called synchronously.
refCon	This is a reference constant to be passed to the callback function. It has no meaning to the HLFDIL, and is provided so that you can give information to your callback function, if you have one.

When called synchronously, this function returns an error code upon encountering an error and a zero on no error. When called asynchronously, it always returns zero, and any error code is returned to the *completionHook* callback function.

If you provide a *completionHook* parameter, it must be a procedure pointer to a function that follows this definition format:

```
void CompletionHook( CommErr errorValue,
                     void *pData, Size Count,
                     long refCon, long lFlags ) ;
```

Any data returned is in the data variables you supplied to the DIL object used to call `FDget`.

The parameters to the function are:

errorValue	This is an error code, if there was an error, or zero, if there was no error.
pData	This is `NULL`, since the data is returned in the data variables you supplied. (`FDget` actually calls the function `CDPipeRead`, so the callback function follows the format used by that call. That call normally returns the data in space pointed to by *pData*.)
Count	This is the number of bytes successfully transferred in the function.
refCon	This is a reference constant you supplied to the original function, for your own tracking purposes.
lFlags	This is 0 or 1, with an *lFlags* value of 1 indicating that an end-of-message indicator was reached.

When receiving data from the Newton, if data is encountered that is not in the set of bound data, this data is added to the unbound list. See "Unbound Data" on page 3-19 for information.

Here is an example `FDget` call that assumes you've opened the pipe pointed to by `thePipe`:

```
fErr = FDget(  obj, /*  Points to the object that
                        describes data you want */
               kDILFrame, /*  Says that obj is a frame*/
               thePipe, /* Points to the pipe that
                           connects to the Newton*/
               15000, /*   Indicates a 15 second
                           timeout*/
               NULL, /*Specifies there is no callback
                      function, so this call is
                      synchronous*/
               0 /*Since no callback function was
                   provided, this value is not used.*/
               );
```

This call tries for 15 seconds to get data from `thePipe` and, if it gets some, puts it in the variables bound to `obj` or on `obj`'s unbound data list. No callback function is given, so this is a synchronous call.

You can also make an asynchronous call:

```
fErr = FDget(obj, kDILFrame, thePipe, 15000, proc, 0);
```

This call returns immediately, and, when data is available from `thePipe` or the operation times out or otherwise fails, the HLFDIL calls the procedure `proc`.

Cyclical Frames

When you are getting unbound data (see the next section), you can get a cyclical frame; that is, a frame that contains itself. (There is currently no way to define a frame like that using `FDbindSlot`, so the only way you can get a cyclical frame is in the unbound data list.)

When the HLFDIL finds a cyclical frame, it sets bit 8 of the `internalFlags` field of the `slotDefinition` record. You can, therefore, recognize that you have a cyclical frame by checking that bit.

Unbound Data

When you call `FDget` and the pipe has data that have definitions that do not match any definitions bound to the DIL object used in the `FDget` call, the HLFDIL places the data on the unbound data list. There can be one unbound data list for each DIL object.

The unbound data list is actually a `slotDefinition` struct. The `slotDefinition` struct has the following fields:

`varType` This is a `short` value that gives the data type of this variable in the desktop application; see Table 3-1 on page 3-22.

`var` This is a `void*` value that gives the actual variable pointer.

`length`	This is an `unsigned long` value that gives the length of this string, symbol, or binary variable (not used for other types).
`maxLength`	This is an `unsigned long` value that gives the maximum length of this string, symbol, or binary variable (not used for other types).
`slotName`	This is a `char*` value that gives the slot name for this variable.
`oClass`	This is a `char*` value that gives the class of this object, if it has one.
`slotType`	This is a `short` value that gives the data type of this slot on the Newton; currently, this is the same as the `varType`.
`truncSize`	This is an `unsigned long` value that gives current size of a truncated object.
`childCnt`	This is a `long` value that gives the number of child nodes. Child nodes contain the data from slots of the Newton frame.
`peerCnt`	This is a `long` value that gives the number of peer nodes (that is, nodes at the same level as this node).
`children`	This is a `slotDefinition*` that gives the child nodes.
`next`	This is a `slotDefinition*` that gives the peer nodes.
`internalFlags`	This is a set of flags for internal use. The one flag that you may be interested in is bit 8; if that bit is set, then this is a cyclical frame. (See "Cyclical Frames" on page 3-19.)

The following functions are used to get and free the unbound data list.

To see how this works, suppose that you got the frame shown in Figure 3-1 on page 3-4. Here is the NewtonScript definition of that frame.

```
aFrame := { slot1: $b,
            slot2: { slot1 : 24,
                     slot2 : {   slot1 : 16,
                                 slot2 : $c},
                     slot3 : "FISH"} }
```

Suppose you hadn't defined the DIL object structure shown in the text in that section, but you defined a DIL object `obj` that you used to call `FDget`. You could get the unbound data list and use that to get the values `b` and `"FISH"` like this:

```
slotDefinition* unBound = FDGetUnboundList(obj);
// Get the character from slot1
char bee = *unBound.children.var;
// Get the string from slot3 of the second unnamed frame
char* fish = unBound.children.next.children.next.next.var;
```

If you don't know the type and location of the data before hand, you can use the `varType` field to determine what kind of data you are dealing with.

FDGetUnboundList

```
slotDefinition *FDGetUnboundList ( DILObj* Dobj ) ;
```

The return value of this function is the structure that contains the unbound data for *Dobj*.

The parameter to this function is:

Dobj The DIL object that you passed to `FDGet`.

FDFreeUnboundList

```
objErr FDFreeUnboundList ( DILObj* Dobj,
                           slotDefinition *list ) ;
```

This function frees the memory held by the list of unbound data.

The parameters to this function are:

Dobj	The DIL object that you passed to `FDGetUnboundList`.
list	The unbound list returned by `FDGetUnboundList`.

DIL Variable Types

The HLFDIL uses the type symbols shown in Table 3-1. The table shows the C types that you can use to represent the various Newton types. The Newton types you actually have may be subclasses of types listed here; if you have a subclass, you can treat it in the same way as the parent type.

Table 3-1 HLFDIL Variable Type Symbols

Newton Type	Symbol	Desktop Type
`Array` (Anonymous Array)	`kDILPlainArray`	`DILArray` object that is bound to a C array or other space with sufficient memory
`Array` (Named Array)	`kDILArray`	`DILArray` object that is bound to a C array or other space with sufficient memory
`Boolean`	`kDILBoolean`	Boolean (see Table 1-1 on page 1-2)
`Char` (The standard Newton character is a two-byte Unicode character)	`kDILUnicodeCharacter`	`short`

Table 3-1 HLFDIL Variable Type Symbols (continued)

Newton Type	Symbol	Desktop Type
`Char` (Although the Newton always uses two-byte characters, if you know that you have one of the 128 ASCII characters (character code values 0-127), you can represent them with 1-byte C char values; you may be able to do that with character code values over 127, but the platform-specific translation tables determine what Unicode character results from values over 127, so you may not get consistent results)	`kDILCharacter`	`char`
`Frame`	`kDILFrame`	`DILFrame` object
Frame that defines a small rectangle, (such as `viewBounds`) with four slots, one for each side, using one unsigned byte for each slot, in this order: top, left, bottom, right.	`kDILSmallRect`	`long` (each byte of the long has a value from one slot of the frame)
`immediate` (Indeterminate Immediate type)	`kDILImmediate`	`long`

Table 3-1 HLFDIL Variable Type Symbols (continued)

Newton Type	Symbol	Desktop Type
`Integer`	`kDILInteger`	`long` (note that Newton `Integers` are 30 bits long, so you should be careful you do not use all 32 bits of the `long`)
Large binary object more than 32 kilobytes (this is a currently treated the same as a `kDILBinaryObject`, so you cannot currently have objects larger than 32 kilobytes)	`kDILBLOB`	Not currently available
`nil` (a special immediate)	`kDILNIL`	`nil` (see Table 1-1 on page 1-2) or `NULL`
`Real` (8 bytes) or other small binary object (less than 32 kilobytes)	`kDILBinaryObject`	`double` or other corresponding type with sufficient space for the data
`String` (a kind of binary object)	`kDILString`	`char*`
`Symbol` (a kind of binary object used as an identifier)	`kDILSymbol`	`char*`

Error Codes

HLFDIL functions return error codes, using the type `objErr`, which is actually a `long` value. Table 3-2 has the error codes.

Table 3-2 HLFDIL Error Codes

Error Code	Numerical Value	Meaning
`kobjErrNoErr`	0	No error
`kObjectHeapNoMemory`	`-28801`	Out of heap memory
`kTempNoMemory`	`-28802`	Out of other memory
`kUnknownSlot`	`-28803`	Slot not known
`kSlotSizeExceeded`	`-28804`	Slot defined size exceeded
`kSlotSizeRequired`	`-28805`	A required slot size is missing

The Newton Side of the DIL Connection

In order to use the DILs you need to have an application on the Newton that initiates a connection with your desktop application and sends and receives data over the connection. You will probably need to do some Newton programming in order to have an application that is specialized for your purpose. You write Newton applications in NewtonScript using the Newton Toolkit (NTK).

This chapter deals with the most common case, where you want to add the ability to send and receive frames and data between a Newton application and a DIL application.

NewtonScript Facilities

There are two parts to the Newton side of a DIL connection: internal code and user interface code.

Internal Code

To connect to the CDIL, you create a `protoEndpoint` object and send it messages. The `protoEndpoint` object handles the Newton side of the connection.

The Communications chapter of the *Newton Programmer's Guide:* documents the `protoEndpoint` proto and has the information you need to connect to a DIL application and send and receive data. Here are the methods you will use from that chapter:

- `Connect`. You use this method to begin the connection. The connection with a DIL is always initiated from the Newton side. When the CDIL pipe has the state `kCDIL_Listening` and the Newton calls `Connect`, the state of the CDIL pipe changes to `kCDIL_ConnectPending`. When the DIL calls `CDPipeAccept`, the state changes to `kCDIL_Connected`. (See Table 2-1 on page 2-5 for details of all the pipe states; Table 2-2 on page 2-6 details the state transitions.) The `protoEndpoint` also has a `Listen` method; you do not use that with the CDIL, because the Newton always initiates the connection.
- `Output`. You use this method to send data to the CDIL.
- `OutputFrame`. You use this method to send data to the HLFDIL.
- `FlushOutput`. You use this to clear the pipe after calling `Output` or `OutputFrame`.
- `SetInputSpec`. You use this method to specify the format of input data.
- `Input`. You use this method to receive data.
- `Disconnect` and `Dispose`. These close a connection.

There are additional methods that allow you to flush the buffer, to get incomplete information, to make AppleTalk connections, and to do other tasks.

User Interface Code

You can use the `NetChooser` function or use other protos to create a user interface for your connection. `NetChooser` displays a view that allows the

user to choose various connection options. You can then get those options and use them in the `Connect` method.

The Newton Side of the DIL Connection

Index

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W