C H A P T E R 4

PDIL Interface

The Protocol Desktop Integration Library (PDIL) is a library designed to make it easy (and possible) for desktop developers to write applications to exchange data with Newton devices by communicating with the built-in Dock (Connection) application.

About the PDIL

The PDIL allows a desktop application to communicate with the Dock application on a Newton 2.x device. The PDIL supports multiple sessions to different Newton devices. The PDIL also supports password protection to your application, where a password needs to be entered on the Newton device before a PDIL session is created.

The PDIL requires the use of the FDIL. The PDIL and the Dock application trade NewtonScript objects, which are FDIL objects on the desktop. The PDIL also requires the use of some communication scheme. You can use the CDIL, but this is not required. You may implement this link in some other manner.

The PDIL provides functions to:

- get a list of stores and soups

- add, empty, and delete soups
- read, edit, and create new entries in a soup
- perform soup queries and navigate soup with cursors
- call global functions and root view methods
- download packages
- extend the protocol to execute an arbitrary NewtonScript function object

Using the PDIL

Creating a PDIL Session

You must initialize the PDIL by calling `PD_Startup` before calling any other PDIL functions. When you are finished using the PDIL, you should call `PD_Shutdown` to free up resources.

To create a session you will need to provide three callback functions to read and write bytes to the Newton device, and to report the number of bytes available for reading. You would normally simply turn around and call the CDIL functions `CD_Read`, `CD_Write`, and `CD_BytesAvailable`, but you may also choose to implement these callbacks in any way you choose.

You create a PDIL session with `PD_CreateSession`, passing it your three callback functions. At that point you can perform any of the actions allowed by the PDIL / Dock protocol, such as performing a soup query, or calling a global function.

When the session is in progress, and you are not actively communicating with the Dock application, you should call `PD_Idle` to allow the PDIL to attend to any unexpected request from the Dock application. When you are finished using your PDIL session, call `PD_Dispose` to terminate the connection.

You can optionally provide password protection to your desktop application. You must simply supply `PD_CreateSession` with a string for the correct password. The Dock application puts up a password slip for the user, and deals with incorrect passwords. Up to three attempts at the proper password are allowed.

Obtaining Information About the Newton Device

There are two functions available to obtain information about the Newton device. `PD_GetNewtonName` retrieves the name on one of the owner cards, the first card entered. `PD_GetNewtonInfo` returns a `PD_NewtonInfo struct` with the information about system parameters; see "PD_NewtonInfo" (page 4-15). This information is similar to what is returned by the NewtonScript function `Gestalt` using the `kGestalt_SystemInfo` selector.

Setting the Current Store

To manipulate any soup-based data in a Newton device, you must first set the current store. There is no support for union soups in the PDIL. You can operate on soups on only one store at a time. If you have, or might possibly have, soups that span more than one store, you must iterate over these stores yourself.

The PDIL allows you to retrieve a list of all stores on a Newton device with `PD_GetAllStores`, or the user-selected default store with `PD_GetDefaultStore`. You set the current store with `PD_SetCurrentStore`. Once you have set the current store you may perform the following operations:

- Retrieve a list of all soups on the store with `PD_GetAllSoups`. This list contains the name and signature of all soups on the current store.
- Create a new soup with `PD_CreateSoup`, and delete or empty an existing soup with `PD_DeleteSoup` and `PD_EmptySoup`.
- Set a soup to be the current soup with `PD_SetCurrentSoup`, allowing you to use the soup functions; see "Using the Soup Functions" (page 4-4).
- Query a soup with `PD_Query`, creating a cursor that iterates over the entries in the soup; see "Soup Queries" (page 4-5). Querying a soup also sets the current soup, allowing you to use the soup functions.

Using the Soup Functions

Once you have set the current soup with PD_SetCurrentSoup (or by performing a soup query), you can:

- Get a list of all entries on the current soup, with PD_GetEntryIDs. This returns a list of the unique integer ID of all entries in the soup. With an entry's ID number you can call PD_GetEntry to retrieve the soup entry, PD_DeleteEntryID or PD_DeleteEntryIDList to delete one or more entries from the current soup, and PD_ChangeEntry to store an edited soup entry back on the Newton device.
- Add entries to the current soup with PD_AddEntry.
- Get a list of the current soup's indexes with PD_GetSoupIndexes. For information on soup indexes, see "Indexes" (page 11-8) in *Newton Programmer's Guide.*
- Retrieve and set the soup info frame. Each soup contains an information frame. You retrieve the information frame for the current soup with PD_GetSoupInfo. You can set the information with PD_SetSoupInfo. You must be very careful that you do not erase important information when setting the soup information frame. In general, you should read in the information frame with PD_GetSoupInfo, alter a limited number of slots, and use this same frame when calling PD_SetSoupInfo. If you add any slots to this frame, you should append your developer signature to the slot name, to guarantee uniqueness.

Listing 4-1 Iterating through every entry on a Newton device

```
long i, j, k;
FD_Handle allStores, allSoups, allEntryIDs, curEntry;


PD_GetAllStores(gSession, &allStores);

for (i = 0; i < FD_GetLength(allStores); i++)
{
    PD_SetCurrentStore(gSession, FD_GetArraySlot(allStores, i));
    PD_GetAllSoups(gSession, &allSoups);
    for (j = 0; j < FD_GetLength(allSoups); j++)
    {
        //we need the soup name which is the first element in the
        //array that represents a soup
        PD_SetCurrentSoup(gSession,
                    FD_GetArraySlot(FD_GetArraySlot(allSoups, j),0));
        PD_GetEntryIDs(gSession, &allEntryIDs);
        for (k = 0; k < FD_GetLength(allEntryIDs); k++)
        {
            PD_GetEntry(gSession, &curEntry,
                        FD_GetInt( FD_GetArraySlot(allEntryIDs,k)) );
            // do something with each entry
            // ....
            FD_DeepDispose(curEntry);
        }
        FD_DeepDispose(allEntryIDs);
    }
    FD_DeepDispose(allSoups);
}
FD_DeepDispose(allStores);
```

Soup Queries

You perform a query on a soup with PD_Query. PD_Query accepts as input the soup's name to query and a query spec, and creates a cursor that iterates the soup's entries matching the query spec. For information on query specs, see "Queries" (page 11-10) in *Newton Programmer's Guide.*

Once you have a soup cursor, you can use it to retrieve entries with PD_Entry. The PD_CountEntries function calculates the number of entries a cursor iterates over. If you make a change in a soup entry, you can write this change back to the Newton device with PD_ChangeEntry. Entries are added to the soup you have performed a query on with PD_AddEntry. Entries are deleted from the soup with PD_DeleteEntry and PD_DeleteEntryList.

The following navigation functions are provided:

PD_Next	Moves cursor forward one entry.
PD_Prev	Moves cursor backward one entry.
PD_Reset	Moves cursor to the first entry.
PD_ResetToEnd	Moves cursor to the last entry.
PD_Move	Moves cursor to the n entries over.
PD_GotoKey	Moves cursor to the entry that contains a particular value in the slot that is the basis of this query.

Note

The functions that move a cursor around retrieve the entry that the cursor now points to. You are responsible for calling `FD_DeepDispose` on the soup entries retrieved. ◆

Listing 4-2 Performing a soup query

```
FD_Handle myQuerySpec, curEntry, thinCrustPizzas, allStores, soupName;
PD_Cursor myCursor;

thinCrustPizzas = FD_MakeArray(0,NULL);

soupName = FD_MakeString("pizzaSoup");

myQuerySpec = FD_MakeFrame ();
FD_SetFrameSlot (myQuerySpec, "indexPath", FD_MakeSymbol("crust"));
FD_SetFrameSlot (myQuerySpec, "beginKey", FD_MakeSymbol("thin"));
FD_SetFrameSlot (myQuerySpec, "endKey", FD_MakeSymbol("thin"));

//we search only the internal store - the 0th element
PD_GetAllStores(gSession, &allStores);
PD_SetCurrentStore(gSession, FD_GetArraySlot(allStores, 0));

PD_Query (gSession,  &myCursor, soupName , myQuerySpec);
PD_Entry (myCursor, &curEntry);

while ( FD_NotNIL(curEntry) )
{
    FD_AppendArraySlot (thinCrustPizzas, curEntry);
    PD_Next (myCursor, &curEntry);
}

PD_DisposeCursor (myCursor);

FD_DeepDispose (thinCrustPizzas);
FD_DeepDispose (allStores);
FD_DeepDispose (myQuerySpec);
FD_Dispose (soupName);
```

Calling Global Functions and Root View Methods

The PD_CallGlobalFunction and PD_CallRootMethod functions allow you to execute global functions and root view methods on a Newton device.

Listing 4-3 Calling global functions and root view methods on a Newton device

```
FD_Handle  result; //result returned by function calls
FD_Handle  params; //parameters sent to these functions

// turn on the Newton device’s backlight
params = FD_MakeArray(0, NULL);
FD_AppendArraySlot(params, FD_MakeInt(1));
PD_CallGlobalFunction(gSession, "Backlight", params, &result);
FD_DeepDispose(result);

// make the Newton device beep
FD_RemoveArraySlot(params,0);
err = PD_CallRootMethod(gSession, "SysBeep", params, &result);
FD_DeepDispose(result);
FD_DeepDispose(params);
```

Using Protocol Extensions

The Dock application can service PDIL requests for a set number of actions. You can extend this set by installing a protocol extension, which is a NewtonScript function executed at the request of a desktop application. The protocol extension is passed in an arbitrary set of parameters and must return a NewtonScript object.

The function object that is the protocol extension is created in NTK as a stream file. Create a project containing a text file that assigns a function object to a variable. Then set the project output to stream file, and the Result field to that variable that contains the function object.

This function is passed in an endpoint object as it’s sole argument. You call this endpoint’s ReadCommandData method to retrieve the “parameters” sent by the PDIL. Your protocol extension should perform a small set of operations, since the lower level protocols need to communicate every few seconds or they time out. Your code must catch any exception thrown, since an uncaught exception could crash the Dock application. It must also call the endpoint's `WriteCommand` method to return a value to the PDIL. It should also not write, nor read in, a large amount of data. Furthermore, you should minimize the use of the NewtonScript heap; the Dock application uses quite a bit, so there is not much left for your protocol extension.

You read in the function object from the stream file with `FD_Unflatten`. You can then load in the protocol extension with `PD_LoadExtension`, passing it the FDIL object retrieved with `FD_Unflatten` and a `long` value used as the ID of this protocol extension. These IDs are usually specified as four characters, such as `'MyID'`; Apple reserves the all-lowercase IDs.

You then call the protocol extension with `PD_CallExtension`. This function accepts as arguments the protocol extension's ID, an FDIL array with the parameters, and a pointer to an FDIL object that is set to what the protocol extension returns.

You can call `PD_RemoveExtension` to remove the protocol extension, but need not, since it is removed automatically when the PDIL session ends. You may want to call it to free up heap space, however.

Listing 4-4 An example protocol extension, calling an application's method

```
// The protocol extension; this code should be compiled by NTK to
// produce a stream file. It calls an application's method, and
// returns the result.
setResultFieldToThisVariable := func (ep)
begin
    try
        local params := ep:ReadCommandData();
        local result := if GetRoot().(myAppSym) exists and params then
                       Perform(GetRoot().(myAppSym),'MethodName,params);
    onexception |evt.ex| do
        result := nil;
    ep:WriteCommand("MyID", result, true);
end;


// This C code loads and calls the protocol extension
FILE * streamFile;
FD_Handle ext, params, result;
```

```
streamFile = fopen(gStreamFileName, "rb");
ext = FD_Unflatten(ReadFromDiskCallBack, streamFile);
fclose(streamFile);
PD_LoadExtension(gSession, 'MyId', ext);
PD_CallExtension(gSession, 'MyId', params, &result);
```

Your protocol extension may return more than one value, that is call the `WriteCommand` method more than once. The first time it is called, the value returned is passed out through the *outResults* parameter to `PD_CallExtension`. You are informed of subsequent values returned by your protocol extension by `PD_Idle`. When your protocol extension returns subsequent values, `PD_Idle` returns the extension ID instead of a status or error code. You can then call `PD_GetNewtonData` to retrieve that value. This process is exemplified in Listing 4-5.

Listing 4-5 Returning more than one value from a protocol extension

```
PD_CallExtension(gSession, myID, myParams, &myResult);
// myResult gets the first value returned.

while (true)
{
    status = PD_Idle(gSession);

    //check for expected return command
    if (status == myID)
    {
        PD_GetNewtonData(gSession, &myResult2);
        break;
    }
}
```

Loading Packages

The `PD_LoadPackage` function loads a package to a Newton device from a desktop package file. You must provide a function to read the package file. This function is in addition to the read, write, and status functions you provide to create a PDIL session.

Listing 4-6 Downloading a package

```
/* This is the callback */
DIL_Error ReadPackage(void* buf, long amt, void* userData)
{
    fread(buf, 1, amt, (FILE*)userData);
    return kDIL_NoError;
}

void loadPackage(const char* filename)
{
    FILE*      package;
    fpos_t     filesize;

    if ((package = fopen(filename, "rb")) == NULL)
    {
        printf("File not found: %s\n", filename);
        return;
    }

    fseek(package, 0, SEEK_END);  // position to the end of the file
    fgetpos(package, &filesize);  // get the size of the package file
    fseek(package, 0, SEEK_SET);  // go back to the beginning

    PD_LoadPackage(gSession, filesize, 1024L, ReadPackage, package);
    fclose(package);
}
```

Setting the Message in the Status Slip

When the Dock application is communicating with you desktop application, it displays a status slip. You can set the message displayed in this status slip with `PD_SetStatusText`. This function only works when communicating with Newton 2.1 devices, however.

Error Handling

Most PDIL functions return an error code indicating their success. There are two error values that the PDIL generates: `kPD_NotInitialized` and `kPD_NewtonError`. A `kPD_NotInitialized` error is returned by a function if `PD_Startup` had not been called. A `kPD_NewtonError` is returned if a

NewtonScript error occurred. If a kPD_NewtonError is returned, you can call PD_GetNewtonError to retrieve the value of that error. This value will be either one of the values listed in "Newton Error Codes" (page 4-13) or any NewtonScript error code from those listed in *Newton Programmer's Reference* .

In addition functions that communicate with a Newton device, return any error returned by the call back functions you provide.

Note

NewtonScript exceptions presently cause the Dock application to disconnect. ◆

Memory Management

The PDIL returns a number of objects, and accepts a number of objects as parameters. You are responsible for disposing of both objects that the PDIL functions return, and objects that you pass into these functions. If a PDIL function requires that a particular object exist after the function completes, it will create a copy of that object.

PDIL Reference

Type Definitions

PD_Handle	A PDIL session object.
PD_Status	The status of the session.
PD_Cursor	A cursor object.

Data Structures

Protocol Extension Endpoint Parameter

Protocol extensions are passed in an endpoint object, this endpoint has two methods you need to use, `ReadCommandData` and `WriteCommand`.

ReadCommandData

endpointArg`:ReadCommandData()`

Reads in the parameters passed to the protocol extension in the call to `PD_CallExtension`.

return value	The the parameters passed to the protocol extension in the *inParams* parameter to `PD_CallExtension`.

WriteCommand

endpointArg`:WriteCommand(`*extensionID*`,` *returnValue*`, true)`

Writes the return value of the protocol extension to the desktop application.

extensionID	A four character string containing the protocol extension's ID.
returnValue	The object to return as the *outResults* parameter to `PD_CallExtension`.
`true`	Always pass in `true` for the third parameter.
return value	Unspecified; do not rely on what `WriteCommand` returns.

DISCUSSION

You must call this function from within your protocol extension at least one time. Return the value `nil`, if you have no data to send; never call this method twice. The first time you call this method, it is returned through `PD_CallExtension`, subsequent calls must have their values returned through `PD_GetNewtonData`.

Constants

Status Constants

The following positive values are returned by PD_Idle:

kPD_Okay	Everything is okay, nothing to do. This equal zero, which equals kDIL_NoError.
kPD_AutoDock	An AutoDock command has been received.
kPD_Cancel	The user tapped the Stop button.
kPD_Disconnect	The Newton device disconnected.

Error Codes

```
kDIL_NoError                    (0)
kDIL_ErrorBase                  (-98000)
kDIL_OutOfMemory                (kDIL_ErrorBase - 1)
kDIL_InvalidParameter           (kDIL_ErrorBase - 2)
kDIL_InternalError              (kDIL_ErrorBase - 3)
kDIL_ErrorReadingFromPipe       (kDIL_ErrorBase - 4)
kDIL_ErrorWritingToPipe         (kDIL_ErrorBase - 5)
kDIL_InvalidHandle              (kDIL_ErrorBase - 6)

kPD_ErrorBase                   (kDIL_ErrorBase - 600)

kPD_NotInitialized              (kPD_ErrorBase - 1)
kPD_NewtonError                 (kPD_ErrorBase - 6)
```

Newton Error Codes

```
kPD_BadStoreSignature           (-28001)
kPD_BadEntry                    (-28002)
kPD_Aborted                     (-28003)
kPD_BadQuery                    (-28004)
kPD_ReadEntryError              (-28005)
kPD_BadCurrentSoup              (-28006)
kPD_BadCommandLength            (-28007)
kPD_EntryNotFound               (-28008)
kPD_BadConnection               (-28009)
kPD_FileNotFound                (-28010)
kPD_IncompatableProtocol        (-28011)
kPD_ProtocolError               (-28012)
```

kPD_DockingCanceled	(-28013)
kPD_StoreNotFound	(-28014)
kPD_SoupNotFound	(-28015)
kPD_BadHeader	(-28016)
kPD_OutOfMemory	(-28017)
kPD_NewtonVersionTooNew	(-28018)
kPD_PackageCantLoad	(-28019)
kPD_ProtocolExtAlreadyRegistered	(-28020)
kPD_RemoteImportError	(-28021)
kPD_BadPasswordError	(-28022)
kPD_RetryPW	(-28023)
kPD_IdleTooLong	(-28024)
kPD_OutOfPower	(-28025)
kPD_BadCursor	(-28026)
kPD_AlreadyBusy	(-28027)
kPD_DesktopError	(-28028)
kPD_CantConnectToModem	(-28029)
kPD_Disconnected	(-28030)
kPD_AccessDenied	(-28031)

Store Frames

A store frame contains the following slots:

Slot description

name	A string for the user-visible name of the store.
signature	An integer for the unique ID of the store.
totalSize	An integer for the number of bytes in the store.
usedSize	An integer for the number of bytes that are used.
kind	Either the string “Internal” or “Card.”
readOnly	Nil or non-nil indicating if the store is read only.
storeVersion	The version of the store format.
defaultStore	True if this is the user specified, default store, nil or absent otherwise.
info	A frame with information about the store. If you add any slots to this frame, make sure your slot name includes your developer signature.

PD_NewtonInfo

A `struct` with the following fields:

Field descriptions

fNewtonID	An almost unique ID which represents a particular Newton. It is a random number from a very large domain, so very close to unique. This number is
fManufacturer	An integer indicating the manufacturer of the Newton device.
fMachineType	An integer indicating the hardware type this ROM was built for.
fROMVersion	An integer indicating the ROM version number.
fROMStage	An integer indicating the language (English, German, French) and the stage of the ROM (alpha, beta, final).
fRAMSize	The amount of RAM on the Newton device.
fScreenHeight	An integer representing the height of the screen in pixels. The height takes into account the current screen orientation.
fScreenWidth	An integer representing the width of the screen in pixels. The width takes into account the current screen orientation.
fPatchVersion	This value is 0 on an unpatched Newton device, and non-zero otherwise.
fNOSVersion	The version of the NewtonScript interpreter.
fInternalStoreSig	The signature of the internal store. Note that this value is changed with a hard reset.
fScreenResolutionV	The number of horizontal pixels per inch.
fScreenResolutionH	The number of vertical pixels per inch.
fScreenDepth	The number of bits per pixel.
fSystemFlags	A bit field. The following two bits are defined
	1 = has serial number
	2 = has target protocol
fSerialNumber	An 8-byte object containing the unique hardware serial number of the Newton device on those devices that contain this hardware.
fTargetProtocol	The version of the protocol used by the Dock application. On Newton 2.1 devices this is 11, Newton

2.0 devices use 9 and 10.

Note
The manufacturer, machineType, ROMVersion, and ROMStage fields provide internal configuration information and should not be relied on. ◆

Functions

PD_Startup

`DIL_Error PD_Startup()`

Initializes the PDIL.

return value	An error code.

DISCUSSION

You must call this function before calling any other PDIL function. It is generally called just once at the beginning of your application, but can be called more than once as long as an equal number of calls to PD_Shutdown are also made.

PD_Shutdown

`DIL_Error PD_Shutdown()`

Closes the library.

return value	An error code.

DISCUSSION

If this is the last call to PD_Shutdown, then all memory allocated by the PDIL since PD_Startup was called is deallocated.

ERROR CODES

kPD_NotInitialized

PD_CreateSession

`DIL_Error PD_CreateSession(PD_Handle*` *outSession*`, DIL_ReadProc` *inReadProc*`, DIL_StatusProc` *inStatusProc*`, DIL_WriteProc` *inWriteProc*`, void *` *inUserData*`, const char*` *inPassword*`)`

Creates a new PDIL session.

outSession	The new PDIL session.
inReadProc	A function you supply to read bytes, see "DIL_ReadProc" (page 3-31). This functions must not return until the specified number of bytes has been read.
inStatusProc	A function you supply to determine the number of bytes that are waiting to be read, see "DIL_StatusProc" (page 3-32).
inWriteProc	A function you supply to write bytes, see "DIL_WriteProc" (page 3-30). This function must not return until the specified number of bytes has been written.
inUserData	This pointer is passed as a parameter to each of the callback procedures.
inPassword	A string representing an optional password which can be used to protect access to your program and desktop data. If you don't want to use the password protection, pass an empty string ("") or NULL as the password.
return value	An error code.

DISCUSSION

This function should be called after a connection from the Newton has been accepted. The function connects to the Newton using the defined 2.0 connection protocol, and does not return until it completes.

Typically, the procedures to read and write bytes are CDIL–based functions, but you may choose to implement them differently.

ERROR CODES

`kPD_NotInitialized`

PD_Dispose

`DIL_Error PD_Dispose(PD_Handle inSession)`

Closes the specified session by sending a disconnect command (if the Newton is still connected).

inSession	A PDIL session.
return value	An error code.

DISCUSSION

Upon return, *inSession* is no longer valid.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Idle

`PD_Status PD_Idle(PD_Handle inSession)`

Idles the specified session and returns the status of the connection.

inSession	A PDIL session.
return value	The current status of the session; see “Status Constants” (page 4-13), or an error code if `PD_Idle` fails, or the ID of a protocol extension that has returned a value accessible with `PD_GetNewtonData`. Note that error values are negative, and status values are positive.

DISCUSSION

This function must be called periodically to give the PDIL time to handle unexpected data arriving from the Newton.

This function need not be called if you are actively communicating with the Newton. For example, if your user interface puts up a dialog waiting for user input, you should call `PD_Idle` while the dialog is displayed. However, once the choice is made and you are issuing commands and reading responses, `PD_Idle` need not be called.

`PD_Idle` calls the status procedures supplied to `PD_CreateSession` in the *inStatusProc* parameter.

SPECIAL CONSIDERATIONS

When this function is being called, `CD_Idle` should not be called.

ERROR CODES

```
kPD_NotInitialized
```

PD_GetNewtonError

`DIL_Error PD_GetNewtonError(PD_Handle` *inSession*`)`

Returns the last result code sent by the Newton.

inSession	A PDIL session.
return value	An error code.

DISCUSSION

This function should only be called in response to receiving a `kPD_NewtonError` error code. Calling at any other time returns an unreliable result.

ERROR CODES

```
NewtonScript error
kPD_NotInitialized
```

PD_GetNewtonInfo

`const PD_NewtonInfoPtr PD_GetNewtonInfo(PD_Handle` *inSession*`)`

Returns information about the connected Newton device.

inSession	A PDIL session.
return value	An internal PDIL structure with information about a Newton device, see “PD_NewtonInfo” (page 4-15).

DISCUSSION

The pointer returned is to the PDIL's internal copy of the information block. You must not alter the data in this data structure in any way. If you have not connected to a Newton device, every field in the information block contains all zeros.

PD_GetNewtonName

`DIL_Error PD_GetNewtonName(PD_Handle` *inSession*`, FD_Handle*` *outNewtonName*`)`

Returns the owner name of the connected Newton device.

inSession	A PDIL session.
outNewtonName	An FDIL string.
	You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

You own the returned string, and should call `FD_Dispose` on it when you no longer need it. Note that it is possible that the Newton device has more than one owner card. In this case there is no guarantee about whose name is returned.

ERROR CODES

`kPD_NotInitialized`

PD_SetStatusText

`DIL_Error PD_SetStatusText(PD_Handle` *inSession*`, const char*` *inText*`)`

Sets the text of the message displayed in the "spinning barber pole" slip.

inSession	A PDIL session.
inText	A string with the text to set.
return value	An error code.

DISCUSSION

This function only works on Newton 2.1 OS devices, but fails silently on earlier devices.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetAllStores

`DIL_Error PD_GetAllStores(PD_Handle` *inSession*`, FD_Handle*` *outStores*`)`

Returns an array of store frames.

inSession	A PDIL session.
outStores	An FDIL array containing store frames, see “Store Frames” (page 4-14). You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetDefaultStore

`DIL_Error PD_GetDefaultStore(PD_Handle` *inSession*`, FD_Handle*` *outStore*`)`

Returns a store frame describing the default store as set by the Newton user.

inSession	A PDIL session.
outStore	A store frames, see “Store Frames” (page 4-14). You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_SetCurrentStore

`DIL_Error PD_SetCurrentStore(PD_Handle` *inSession*`, FD_Handle` *inStore*`, short` *inSetStoreInfo*`)`

Sets the current store for the session.

inSession	A PDIL session.
inStore	An store frame containing at least the following store frame slots: `name`, `kind`, `info` and `signature`; see “Store Frames” (page 4-14). You may pass in `kFD_NIL` to set the session to the default store as defined on the Newton device. You are responsible for disposing of this object.
inSetStoreInfo	Pass in zero if you do not want the store’s information frame to be set to the value of the `info` slot of *inStore*. Pass in anything else to set the store information frame. Only true backup / restore type programs should pass in anything but zero, and then only when performing a restore operation. You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

The current store is used by subsequent soup and entry functions. You must call `PD_SetCurrentStore` to set the store you want to operate on before making any soup, entry, or cursor calls.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_CreateSoup

`DIL_Error PD_CreateSoup(PD_Handle` *inSession*`, FD_Handle` *inSoupName*`, FD_Handle` *inSoupIndex*`)`

Creates the specified soup on the current store using *inSoupIndex* as the array of index frames.

inSession	A PDIL session.
inSoupName	An FDIL string for the name of the soup. You are responsible for disposing of this object.
inSoupIndex	An FDIL array of index spec frames; see “Single-Slot Index Specification Frame” (page 9-5) and “Multiple-Slot Index Specification Frame” (page 9-7) in *Newton Programmer’s Reference.* Note that even if you have only one index spec frame, it must be placed into an array. You may pass in `kFD_NIL` to create a soup without indexes. You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

If *inSoupName* already exists, this function is the same as `PD_SetCurrentSoup` and the soup index does not change.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_DeleteSoup

`DIL_Error PD_DeleteSoup(PD_Handle` *inSession*`)`

Deletes the current soup.

inSession	A PDIL session.
return value	An error code.

DISCUSSION

The current soup is undefined after this call.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_EmptySoup

`DIL_Error PD_EmptySoup(PD_Handle` *inSession*`)`

Removes all the entries from the current soup.

inSession	A PDIL session.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetAllSoups

`DIL_Error PD_GetAllSoups(PD_Handle` *inSession*`, FD_Handle*` *outSoups*`)`

Returns an array of soup names and signatures from the current store.

inSession	A PDIL session.
outSoups	An FDIL array of arrays. There is one element in the top level array for each soup on the store. Each of the inner arrays contain two elements. The first element contains an string with the soup name, and the second element contains an integer with the soup's signature. You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

Calling `FD_GetLength` on the *outSoups* array gives you the number of soups on the store. `FD_GetArraySlot` allows you to extract the inner array which has the name and signature of the soup.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_SetCurrentSoup

`DIL_Error PD_SetCurrentSoup(PD_Handle` *inSession*`, FD_Handle` *inSoupName*`)`

Sets the current soup on the current store.

inSession	A PDIL session.
inSoupName	An FDIL string for the soup name.
	You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

This function must be called before any of the entry functions.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetSoupIndexes

`DIL_Error PD_GetSoupIndexes(PD_Handle` *inSession*`, FD_Handle*` *outSoupIndexes*`)`

Returns an array of index spec frames from the current soup.

inSession	A PDIL session.
outSoupIndexes	An array of index spec frames. For more information about index spec frames, see Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetSoupInfo

`DIL_Error PD_GetSoupInfo(PD_Handle` *inSession*`, FD_Handle*` *outSoupInfo*`)`

inSession	A PDIL session.
outSoupInfo	The current soup's information frame. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_SetSoupInfo

`DIL_Error PD_SetSoupInfo(PD_Handle` *inSession*`, FD_Handle` *inSoupInfo*`)`

inSession	A PDIL session.
inSoupInfo	A frame to be made into the current soup's information frame.
	You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

You must be very careful when using this function. You should read in the soup information frame with `PD_GetSoupInfo`, access a limited number of slots, and use this same frame when calling `PD_SetSoupInfo`. If you add any slots to the soup information frame, append your developer signature to the slot name.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_AddEntry

`DIL_Error PD_AddEntry(PD_Handle` *inSession*`, FD_Handle` *inEntry*`, long*` *outID*`)`

Adds the specified entry, and returns the new unique ID.

inSession	A PDIL session.
inEntry	An FDIL frame to be made into a soup entry.
	You are responsible for disposing of this object.
outID	The new entry's unique ID.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_ChangeEntry

`DIL_Error PD_ChangeEntry(PD_Handle` *inSession*`, FD_Handle` *inEntry*`)`

Stores a changed entry back in the soup.

inSession	A PDIL session.
inEntry	A soup entry retrieved with `PD_GetEntry`.
	You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_DeleteEntry

`DIL_Error PD_DeleteEntry(PD_Handle` *inSession*`, FD_Handle` *inEntry*`)`

Removes the entry from the current soup.

inSession	A PDIL session.
inEntry	A soup entry retrieved with `PD_GetEntry` or `PD_Entry`.
	You are responsible for disposing of this object.
return value	An error code.

SPECIAL CONSIDERATIONS

Entries are not removed instantaneously. It is possible to delete an entry, then call `PD_Next` and `PD_Prev`, and retrieve the supposedly deleted entry.

ERROR CODES

```
error returned by communication callback function
```

```
kPD_NotInitialized
kPD_NewtonError
```

PD_DeleteEntryID

`DIL_Error PD_DeleteEntryID(PD_Handle` *inSession*`, FD_Handle` *inEntryID*`)`

Removes the entry specified by the entry ID from the current soup.

inSession	A PDIL session.
inEntryID	A soup entry's ID number, see Special Considerations.
return value	An error code.

SPECIAL CONSIDERATIONS

The *inEntryID* parameter must be a valid ID number. If an incorrect ID is supplied, then the next soup entry is deleted!

Entries are not removed instantaneously. It is possible to delete an entry, then call `PD_Next` and `PD_Prev`, and retrieve the supposedly deleted entry.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_DeleteEntryIDList

`DIL_Error PD_DeleteEntryIDList(PD_Handle` *inSession*`, FD_Handle` *inEntryIDList*`)`

Removes the entries specified by the array of entry IDs from the current soup.

inSession	A PDIL session.
inEntryIDList	An FDIL array of entry IDs from the current soup, see Special Considerations. You are responsible for disposing of this object.
return value	An error code.

SPECIAL CONSIDERATIONS

The *inEntryIDList* parameter must contain valid ID numbers. If an incorrect ID is supplied, then the next soup entry is deleted!

Entries are not removed instantaneously. It is possible to delete an entry, then call `PD_Next` and `PD_Prev`, and retrieve the supposedly deleted entry.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_DeleteEntryList

`DIL_Error PD_DeleteEntryList(PD_Handle` *inSession*`, FD_Handle` *inEntryList*`)`

Removes the entries from the current soup.

inSession	A PDIL session.
inEntryList	An FDIL array of soup entries from the current soup.
	You are responsible for disposing of this object.
return value	An error code.

SPECIAL CONSIDERATIONS

Entries are not removed instantaneously. It is possible to delete an entry, then call `PD_Next` and `PD_Prev`, and retrieve the supposedly deleted entry.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetEntry

`DIL_Error PD_GetEntry(PD_Handle` *inSession*`, FD_Handle*` *outEntry*`,`
`long` *entryID*`)`

Retrieves the entry with the specified unique ID from the current soup.

inSession	A PDIL session.
outEntry	An FDIL frame for the soup entry.
	You are responsible for disposing of this object.
entryID	The ID of the entry to retrieve; see `PD_GetEntryIDs`.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetEntryIDs

`DIL_Error PD_GetEntryIDs(PD_Handle` *inSession*`, FD_Handle*`
outEntryIDs`)`

Returns an array of entry ID's from the current soup.

inSession	A PDIL session.
outEntryIDs	An FDIL array of entry IDs in the current soup.
	You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

The resulting entry IDs can be used as a parameter to the `PD_GetEntry` and `PD_DeleteEntryID` and `PD_DeleteEntryIDList` functions.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Query

`DIL_Error PD_Query(PD_Handle` *inSession*`, PD_Cursor*` *outCursor*`, FD_Handle` *inSoupName*`, FD_Handle` *inQuerySpec*`)`

Performs a query on the specified soup on the current store.

inSession	A PDIL session.
outCursor	The cursor object created.
inSoupName	An FDIL string for the soup to query, or `kFD_NIL` to use the current soup. You are responsible for disposing of this object.
inQuerySpec	A query spec. You can pass `kFD_NIL` to create a cursor that iterates over every entry in the soup, or a query spec frame as specified in "Query Specification Frame" (page 9-10) in *Newton Programmer's Reference.* You can also create complex queries that include NewtonScript function objects as a stream file in NTK. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_CountEntries

`DIL_Error PD_CountEntries(PD_Cursor` *inCursor*`, long*` *outCount*`)`

Returns the number of entries the cursor iterates over.

inCursor	A cursor object.
outCount	The number of entries the cursor iterates over.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
```

```
kPD_NotInitialized
kPD_NewtonError
```

PD_DisposeCursor

`DIL_Error PD_DisposeCursor(PD_Cursor` *inCursor*`)`

Disposes of the specified cursor.

cursor	A cursor object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Entry

`DIL_Error PD_Entry(PD_Cursor` *inCursor*`, FD_Handle*` *outEntry*`)`

Retrieves the current entry from the specified cursor.

inCursor	A cursor object.
outEntry	An FDIL frame for the entry. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GotoKey

DIL_Error PD_GotoKey(PD_Cursor *inCursor*, FD_Handle *inKey*, FD_Handle* *outEntry*)

Returns the entry at the specified key location.

inCursor	A cursor object.
inKey	The key of the entry to advance to. An entry's key is the value in the slot that was designated the index of the soup. For example, if a soup is indexed on the 'firstName slot, "Elizabeth" is a possible key. If the soup has a multi-slot index, this should be an array of values. You are responsible for disposing of this object.
outEntry	An FDIL frame for the entry, or kFD_NIL if there is no entry with the specified key. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Move

DIL_Error PD_Move(PD_Cursor *inCursor*, long *inOffset*, FD_Handle* *outEntry*)

Moves the specified cursor the specified number of entries.

inCursor	A cursor object.
inOffset	How many entries to move over, this can be a positive or negative integer.
outEntry	An FDIL frame for the entry the cursor points to in its new position, or kFD_NIL if moving over this many places causes the cursor to run of the end of the list. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Next

DIL_Error PD_Next(PD_Cursor *inCursor*, FD_Handle* *outEntry*)

Advances the cursor to the next entry and returns this entry.

inCursor	A cursor object.
outEntry	An FDIL frame for the entry the cursor points to in its new position, or `kFD_NIL` if at the end of the list. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Prev

DIL_Error PD_Prev(PD_Cursor *inCursor*, FD_Handle* *outEntry*)

Backs up the cursor to the previous entry and returns this entry.

inCursor	A cursor object.
outEntry	An FDIL frame for the entry the cursor points to in its new position, or `kFD_NIL` if at the beginning of the list. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_Reset

`DIL_Error PD_Reset(PD_Cursor` *inCursor*`, FD_Handle*` *outEntry*`)`

Positions the cursor to the beginning and returns the first entry.

inCursor	A cursor object.
outEntry	An FDIL frame for the entry the cursor points to in its new position. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_ResetToEnd

`DIL_Error PD_ResetToEnd(PD_Cursor` *inCursor*`, FD_Handle*` *outEntry*`)`

Positions the cursor to the end and returns the last entry.

inCursor	A cursor object.
outEntry	An FDIL frame for the entry the cursor points to in its new position. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_LoadPackage

```
DIL_Error PD_LoadPackage(PD_Handle inSession, long lenPackage,
long chunkSize, DIL_ReadProc readProc, void* userData)
```

Loads a package.

inSession	A PDIL session.
lenPackage	The number of bytes in the package.
chunkSize	The number of bytes to read at a time. It is recommended that you use a 1K, 1024, chunk size.
readProc	A function you supply to read bytes, see "DIL_ReadProc" (page 3-31).
userData	A pointer passed to your *readProc*.
return value	An error code.

DISCUSSION

The *readProc* is called to read *chunkSize* bytes of data at a time (until the last call which may be less). If the *readProc* returns an error (either a disk error or the user cancels) the package load is terminated and the connection is broken. The *userData* parameter is passed to the *readProc*, and is typically the platform representation of the package file.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_LoadExtension

`DIL_Error PD_LoadExtension(PD_Handle` *inSession*`, long` *inExtensionID*`, FD_Handle` *inExtension*`)`

Loads a protocol extension.

inSession	A PDIL session.
inExtensionID	An ID that identifies this protocol extension. These IDs are usually specified by a set of four characters. The all-lowercase IDs are reserved by Apple.
inExtension	A function object to be executed when the protocol extension is called, see DISCUSSION. You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

The *inExtension* function object is created in NTK and saved as a stream file. The function object can then be retrieved from the stream file with the `FD_Unflatten` function. When this function object is eventually called, with `PD_CallExtension`, it is passed in an endpoint object. There are two methods of this endpoint object you need to use, `ReadCommandData` and `WriteCommand`, to read in a set of parameters and write out a return value. These endpoint object methods are described in “Protocol Extension Endpoint Parameter” (page 4-12).

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_CallExtension

`DIL_Error PD_CallExtension(PD_Handle` *inSession*`, long` *inExtensionID*`, FD_Handle` *inParams*`, FD_Handle*` *outResults*`)`

Calls a protocol extension added with `PD_LoadExtension`.

inSession	A PDIL session.
inExtensionID	The extension ID used in the call to `PD_LoadExtension`.
inParams	The parameters to pass to the protocol extension.
	You are responsible for disposing of this object.
outResults	The result returned by the protocol extension.
	You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_GetNewtonData

`DIL_Error PD_GetNewtonData (FD_Handle` *inSession*`, FD_Handle*` *outNewtonData*`)`

Retrieves data from a second, or subsequent, call to the endpoint `WriteCommand` method from a protocol extension.

inSession	A PDIL session.
outNewtonData	The result returned by the protocol extension, or `kFD_NIL` if there is no pending value.
	You are responsible for disposing of this object.
return value	An error code.

DISCUSSION

You are notified of when to call this function with `PD_Idle`.

ERROR CODES

`kPD_NotInitialized`

PD_RemoveExtension

`DIL_Error PD_RemoveExtension(PD_Handle` *inSession*`, long` *inExtensionID*`)`

Removes the specified protocol extension.

inSession	A PDIL session.
inExtensionID	The extension ID used in the call to `PD_LoadExtension`.
return value	An error code.

DISCUSSION

You need not call this function. The protocol extension is automatically removed when the PDIL session terminates. You may want to call it to free up heap space, however.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_CallGlobalFunction

`DIL_Error PD_CallGlobalFunction(PD_Handle` *inSession*`, const char*` *inFunctionName*`, FD_Handle` *inParamsArray*`, FD_Handle*` *outResult*`)`

Calls a global function, returning the function’s result.

inSession	A PDIL session.
inFunctionName	The name of the function to call.
inParamsArray	An FDIL array with the parameters to pass to *inFunctionName*. If the function takes no parameters, pass in an empty array. You are responsible for disposing of this object.
outResult	The return value of *inFunctionName*. You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PD_CallRootMethod

DIL_Error PD_CallRootMethod(PD_Handle *inSession*, const char* *inMethodName*, FD_Handle *inParamsArray*, FD_Handle* *outResult*)

Calls a root view method, returning the function's result.

inSession	A PDIL session.
inMethodName	The name of the root method to call.
inParamsArray	An FDIL array with the parameters to pass to *inMethodName*. If the function takes no parameters, pass in an empty array.
	You are responsible for disposing of this object.
outResult	The return value of *inMethodName*.
	You are responsible for disposing of this object.
return value	An error code.

ERROR CODES

```
error returned by communication callback function
kPD_NotInitialized
kPD_NewtonError
```

PDIL Summary

Type Definitions

```
PD_Handle
PD_Status
PD_Cursor
```

Data Structures

Protocol Extension Endpoint Parameter

endpointArg:`ReadCommandData()`
endpointArg:`WriteCommand(`*extensionID*`,` *returnValue*`, true)`

Constants

Status Codes

```
kPD_Okay
kPD_AutoDock
kPD_Cancel
kPD_Disconnect
```

Error Codes

```
kDIL_NoError
kDIL_ErrorBase
kDIL_OutOfMemory
kDIL_InvalidParameter
kDIL_InternalError
kDIL_ErrorReadingFromPipe
kDIL_ErrorWritingToPipe
kDIL_InvalidHandle
```

```
kPD_ErrorBase
kPD_NotInitialized
kPD_NewtonError
```

Newton Error Codes

```
kPD_BadStoreSignature
kPD_BadEntry
kPD_Aborted
kPD_BadQuery
kPD_ReadEntryError
kPD_BadCurrentSoup
kPD_BadCommandLength
kPD_EntryNotFound
kPD_BadConnection
kPD_FileNotFound
kPD_IncompatableProtocol
kPD_ProtocolError
kPD_DockingCanceled
kPD_StoreNotFound
kPD_SoupNotFound
kPD_BadHeader
kPD_OutOfMemory
kPD_NewtonVersionTooNew
kPD_PackageCantLoad
kPD_ProtocolExtAlreadyRegistered
kPD_RemoteImportError
kPD_BadPasswordError
kPD_RetryPW
kPD_IdleTooLong
kPD_OutOfPower
kPD_BadCursor
kPD_AlreadyBusy
kPD_DesktopError
kPD_CantConnectToModem
kPD_Disconnected
kPD_AccessDenied
```

Store Frames

{name: *string*,
signature: *integer*,
totalSize: *integer*,
usedSize: *integer*,
kind: *string*,

```
readOnly: Boolean,
storeVersion: integer,
defaultStore: Boolean,
info: frame}
```

PD_NewtonInfo

```
typedef struct PD_NewtonSystemInfo
{
    long    fNewtonID;
    long    fManufacturer;
    long    fMachineType;
    long    fROMVersion;
    long    fROMStage;
    long    fRAMSize;
    long    fScreenHeight;
    long    fScreenWidth;
    long    fPatchVersion;
    long    fNOSVersion;
    long    fInternalStoreSig;
    long    fScreenResolutionV;
    long    fScreenResolutionH;
    long    fScreenDepth;
    long    fSystemFlags;
    long    fSerialNumber[2];
    long    fTargetProtocol;
} PD_NewtonSystemInfo;
```

Functions

```
DIL_Error PD_Startup()
DIL_Error PD_Shutdown()
DIL_Error PD_CreateSession(PD_Handle* outSession,
    DIL_ReadProc inReadProc, DIL_StatusProc inStatusProc,
    DIL_WriteProc inWriteProc, void * inUserData,
    const char* inPassword)
DIL_Error PD_Dispose(PD_Handle inSession)
PD_Status PD_Idle(PD_Handle inSession)
DIL_Error PD_GetNewtonError(PD_Handle inSession)
const PD_NewtonInfoPtr PD_GetNewtonInfo(PD_Handle inSession)
DIL_Error PD_GetNewtonName(PD_Handle inSession,
    FD_Handle* outNewtonName)
DIL_Error PD_SetStatusText(PD_Handle inSession, const char* inText)
DIL_Error PD_GetAllStores(PD_Handle inSession, FD_Handle* outStores)
DIL_Error PD_GetDefaultStore(PD_Handle inSession, FD_Handle* outStore)
DIL_Error PD_GetCurrentStore(PD_Handle inSession, FD_Handle* outStore)
DIL_Error PD_SetCurrentStore(PD_Handle inSession, FD_Handle inStore,
    short inSetStoreInfo)
DIL_Error PD_CreateSoup(PD_Handle inSession, const char* inSoupName,
    FD_Handle inSoupIndex)
DIL_Error PD_DeleteSoup(PD_Handle inSession)
DIL_Error PD_EmptySoup(PD_Handle inSession)
DIL_Error PD_GetAllSoups(PD_Handle inSession, FD_Handle* outSoups)
DIL_Error PD_GetCurrentSoup(PD_Handle inSession, FD_Handle* outSoup)
DIL_Error PD_SetCurrentSoup(PD_Handle inSession, FD_Handle inSoupName)
DIL_Error PD_GetSoupIndexes(PD_Handle inSession,
    FD_Handle* outSoupIndexes)
DIL_Error PD_GetSoupInfo(PD_Handle inSession, FD_Handle* outSoupInfo)
DIL_Error PD_SetSoupInfo(PD_Handle inSession, FD_Handle  inSoupInfo)
DIL_Error PD_AddEntry(PD_Handle inSession, FD_Handle inEntry,
    long* outID)
DIL_Error PD_ChangeEntry(PD_Handle inSession, FD_Handle inEntry)
DIL_Error PD_DeleteEntry(PD_Handle inSession, FD_Handle inEntry)
DIL_Error PD_DeleteEntryID(PD_Handle inSession, FD_Handle inEntryID)
DIL_Error PD_DeleteEntryIDList(PD_Handle inSession,
    FD_Handle inEntryIDList)
DIL_Error PD_DeleteEntryList(PD_Handle inSession, FD_Handle inEntryList)
DIL_Error PD_GetEntry(PD_Handle inSession, FD_Handle * outEntry,
    long entryID)
DIL_Error PD_GetEntryIDs(PD_Handle inSession, FD_Handle* outEntryIDs)
DIL_Error PD_Query(PD_Handle inSession, PD_Cursor* outCursor,
    FD_Handle inSoupName, FD_Handle inQuerySpec)
DIL_Error PD_CountEntries(PD_Cursor inCursor, long* outCount)
DIL_Error PD_DisposeCursor(PD_Cursor inCursor)
DIL_Error PD_Entry(PD_Cursor inCursor, FD_Handle* outEntry)
DIL_Error PD_GotoKey(PD_Cursor inCursor, FD_Handle inKey,
    FD_Handle* outEntry)
DIL_Error PD_Move(PD_Cursor inCursor, long inOffset,
    FD_Handle* outEntry)
DIL_Error PD_Next(PD_Cursor inCursor, FD_Handle* outEntry)
DIL_Error PD_Prev(PD_Cursor inCursor, FD_Handle* outEntry)
DIL_Error PD_Reset(PD_Cursor inCursor, FD_Handle* outEntry)
DIL_Error PD_ResetToEnd(PD_Cursor inCursor, FD_Handle* outEntry)
DIL_Error PD_LoadPackage(PD_Handle inSession, long lenPackage,
    long chunkSize, DIL_ReadProc readProc, void* userData)
DIL_Error PD_LoadExtension(PD_Handle inSession, long inExtensionID,
    FD_Handle inExtension)
DIL_Error PD_CallExtension(PD_Handle inSession, long inExtensionID,
    FD_Handle inParams, FD_Handle* outResults)
DIL_Error PD_GetNewtonData (FD_Handle inSession,
    FD_Handle* outNewtonData)
DIL_Error PD_RemoveExtension(PD_Handle inSession, long inExtensionID)
DIL_Error PD_CallGlobalFunction(PD_Handle inSession,
    const char* inFunctionName, FD_Handle inParamsArray,
    FD_Handle* outResult)
DIL_Error PD_CallRootMethod(PD_Handle inSession,
    const char* inMethodName, FD_Handle inParamsArray,
    FD_Handle* outResult)
```