PDIL 2.0

Introduction

The PDIL is a library designed to make it easy (and possible) for desktop developers to write applications to exchange data with Newton devices using the built-in Dock application. The initial implementation will not allow a developer to easily write a full–featured backup and restore program like NBU. The requirements for such a program are complex and the ROM protocol is somewhat fragile.

The ideal usage of the PDIL will be for applications that want to provide built in synchronization to their products without writing the currently required Newton application. Custom apps, such as those that may be written by VAR's can use the PDIL to exchange data between desktop applications or databases and Newton applications.

Design Assumptions

The PDIL 2.0 will be based on the FDIL 2.0.

The PDIL will know nothing about CDIL. The developer must implement all communications functions including listen, accept, read, write and disconnect. The developer provides a read and write data procptr which is used by the PDIL for getting and sending all data.

The PDIL assumes nothing about the underlying communications implementation. The read/write procs must be synchronous - operation is completed on return. However, the read/write procs may internally do anything including blocking the current thread, calling WaitNextEvent, or calling a status proc. All details are up to the developer.

All operands and results will be FDIL entities. The developer is responsible for disposing all results and operands using the FDIL API. Developers must use the FDIL API to access the details of results and operands.

All PDIL calls have the ability to return communication errors. A preliminary list is in PDIL.h (prefaced by the comment string "PDIL last result error numbers"), but the format of these errors and how the are returned and the values are most likely going to change before the final release.

Basic Flow Using PDIL

Because PDIL talks to the Dock application in the Newton OS, you need to follow a certain set of steps in using PDIL. The basic flow is as follows:

- Get the store information for the Newton device by using PD_GetAllStores or PD_GetCurrentStore.
- Set the current store you wish to work with by using PD_SetCurrentStore.
- Get the soup information by using PD_GetAllSoups or PD_GetCurrentSoup.
- Set the current soup you wish to work with by using PD_SetCurrentSoup
- At this point you can now use the entry functions or the query functions to manipulate soup entries.

The most likely result if this order isn't followed is a kPD_ProtocolError, or errors like kPD_BadCurrentSoup or kPD_StoreNotFound or kPD_SoupNotFound. (These are currently defined in PDIL.h)

If you have other issues with the appropriate order, the best place to turn at this point is the dil-talk mailing list. Go here:

http://www.newton.apple.com/dev/newdevs.html#help

for more information on how to subscribe to this list. The entire DIL development team is actively on the list, helping developers like yourselves.

Why PDIL?

Before getting into all the features of the PDIL, here's a short example showing the ways in which it might be used.

(Example to be supplied later)

Library Reference

Data Types

`typedef void* PD_Handle;`

All sessions created and managed by the PDIL are referenced via the PD_Handle type. When sessions are created and returned to the user, the creating function returns a PD_Handle. Most PDIL functions take a PD_Handle as their first parameter.

`typedef long DIL_Error;`

A signed long used to return error codes generated by the PDIL.

`typedef long PD_Status;`

A signed long used to return status about the current PDIL session.

`typedef long PD_Extension;`

A signed long used to identify a loaded protocol extension. This typically expressed as a 4-character identifier, and Newton, Inc. reserves the set of all lower-case identifiers. Since protocol extensions are only loaded and active for the duration of a particular communication session, developers do not need to worry about id conflicts outside the scope of their application.

`typedef void* PD_Cursor;`

All cursors created and managed by the PDIL are referenced via the PD_Cursor type. When cursors are created and returned to the user, the creating function returns a PD_Cursor. Most PDIL cursor functions take a PD_Cursor as their first parameter.

Error Codes

```
kPD_NotInitialized      (kPD_ErrorBase - 1)
kPD_InvalidSession      (kPD_ErrorBase - 2)
kPD_InvalidStore        (kPD_ErrorBase - 3)
kPD_InvalidSoup         (kPD_ErrorBase - 4)
kPD_InvalidCursor       (kPD_ErrorBase - 5)
kPD_InvalidResult       (kPD_ErrorBase - 6)
kPD_InvalidROMVersion   (kPD_ErrorBase - 7)
```

Status codes

```
kPD_Okay                0
kPD_AutoDock            1
kPD_Cancel              2
kPD_Disconnect          3
kPD_Hello               4
```

Callbacks

```
typedef DIL_Error (*DIL_ReadProc)(void* buffer,
                                  long* count,
                                  void* userData);
```

Read the specified number of bytes into the buffer.

```
DIL_Error ReadBytes(void* buf, long amt, void* userData)
{
   CD_Handle pipe = (CD_Handle)userData;
   DIL_Error err = CD_Read(pipe, buf, amt);
   return err;
}
```

```
typedef DIL_Error (*DIL_WriteProc)(const void* buffer,
                                   long  count,
                                   void* userData);
```

Write the specified number of bytes from the buffer.

```
DIL_Error WriteBytes(const void* buf, long amt, void* userData)
{
   CD_Handle pipe = (CD_Handle)userData;
   DIL_Error err;

   if (amt == -1)
      err = CD_FlushOutput(pipe);
   else
      err = CD_Write(pipe, buf, amt);

   return err;
}
```

IMPORTANT NOTE: Your write procedure will be called with a count of -1 when it is time to flush the output buffer. You must check the count or else you'll get an error from the CDIL.

```
typedef DIL_Error (*DIL_StatusProc)(long* bytesAvailable,
                                    void* userData);
```

Return the number of bytes waiting to be read.

```
DIL_Error StatusBytes(long *bytesAvailable, void* userData)
{
   CD_Handle pipe = (CD_Handle)userData;
   DIL_Error err = CD_BytesAvailable(pipe, bytesAvailable);
   return err;
}
```

Setting up and Shutting down the PDIL

The following calls are made to start and stop the PDIL. The startup call allocates some common data structures and makes all the rest of the calls work and should be called during program initialization. The shutdown call releases all allocated memory and should be called prior to program termination.

DIL_Error PD_Startup(void);

Initializes the PDIL. 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.

Example:

```
BOOL CMyApp::InitInstance()
{
      ...
      DIL_Error   err = PD_Startup();
      ...
}
```

Error codes:
kDIL_OutOfMemory

DIL_Error PD_Shutdown(void);

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

Example:

```
int CMyApp::ExitInstance()
{
      ...
      PD_Shutdown();
      return CWinApp::ExitInstance();
}
```

Error codes:
kPD_NotInitialized

Session control

The next sequence of calls control a PDIL session. A session is simply defined as the current connection to a Newton device. A session is associated with a PD_Handle. The PDIL will support multiple, simultaneous sessions to different Newton devices.

The session calls mimic their associated CDIL calls. Typically, PD_CreateSession is called after a connection is accepted by the CDIL. PD_Idle must be called periodically to process unexpected data coming from the Newton. Unexpected data includes disconnects, cancels, and other commands. After the PDIL session is complete, PD_Dispose is called to disconnect from the Newton.

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

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

inReadProc and inWriteProc are developer supplied functions to read and write data. The functions must not return until the specified number of bytes has been read or written. Typically, these will be CDIL–based functions, but a developer can choose to implement them differently.

inStatusProc is a developer supplied function that will be called by PD_Idle to determine whether any bytes are waiting to be read from the Newton.

inUserData will be passed as a parameter to each of the callback procs.

Example:

Error codes:

```
      kDIL_OutOfMemory        if the session cannot be created
      kDIL_InvalidParameter   if any of the callback procs
                              are not specified
      kPD_NotInitialized      if PD_Startup has not been called
      kPD_InvalidROMVersion   if connected to a 1.x device
```

```
DIL_Error PD_Dispose(PD_Handle inSession);
```

Close the specified session by sending a disconnect command (if the Newton is still connected). Upon return, inSession will no longer be valid.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
?? comm errors ??	as a result of the disconnect call

PD_Status PD_Idle(PD_Handle inSession);

Idle the specified session and return the status of the connection. This function must be called periodically to give the PDIL's time to handle unexected data arriving from the Newton.

This function need not be called if you are actively communicating with the Newton. For example, if your UI 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 statusProc supplied to PD_CreateSession.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session

Status codes:

kPD_Okay	0	everything is okay, nothing to do
kPD_AutoDock	1	an AutoDock command has been received
kPD_Cancel	2	the user pressed the Stop button
kPD_Disconnect	3	the Newton disconnected
kPD_Hello	4	informational, shouldn't get these

Information functions

This section describes a few useful utility functions. PD_GetNewtonName and PD_GetNewtonInfo are fairly obvious. PD_GetNewtonError should be called when any other PDIL function returns a kPD_InvalidResult error code.
PD_SetStatusText controls the text shown in the spinning barber pole slip on the Newton, but only works on 2.1 devices.

```
DIL_Error PD_GetNewtonError(PD_Handle inSession);
```

Return the last result code sent by the Newton. This function should only be called in response to a kPD_NewtonError error code. Calling at any other time will return an unreliable result.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inSession is not a real session
      ?? result errors ??     list of errors returned by kDResult
```

```
DIL_Error PD_GetNewtonInfo(PD_Handle  inSession,
                           SNewtonSysInfo* outVersionInfo);
```

Return information about the connected Newton device. The developer owns the pointer returned in outVersionInfo and should call free() on it when finished.

The version information is an array of longs, containing the following:

```
      newtonUniqueID
      manufacturer id
      machine type
      rom version
      rom stage
      ram size
      screen height
      screen width
      system update version
      Newton object system version
      signature of internal store
      vertical screen resolution
      horizontal screen resolution
      screen depth
            // the following information is only on 2.1 devices
      systemFlags
      serialNumber[2]
      targetProtocol
```

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if outVersionInfo is null

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

Return the owner name of the connected Newton device. The developer owns the returned string, and should call FD_Dispose() on it when finished.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if outNewtonName is null

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

Sets the text of the message displayed in the "spinning barber pole" slip. Note that this function only exists on 2.1 devices, but will fail silently on earlier devices.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if inText is not a string

Store functions

These functions control which store the rest of the PDIL commands operate on. There are no PDIL calls that operate on union soups. If you have (or could have) a soup which spans multiple stores, then you must iterate over all the stores yourself. PD_GetAllStores returns an array of store frames that can be used to perform this iteration.

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 other calls. (See the Cursor section at the end for exceptions to this rule.)

PD_GetCurrentStore is primarily a convenience function in case you forgot which store you set to be current, and will simply return a clone of the store frame you passed in. Specifically, the Newton will not be asked for the current store.

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

Return an array of store frames. A subset of each element of the array may be used as a parameter to the PD_SetCurrentStore function.

Example:

Result:
Each array slot contains the following information about a store:

```
[{name: "Internal",
  signature: 22315107,
  TotalSize: 3767328,
  UsedSize: 1490936,
  kind: "Internal",
  info: {lastrestorefromcard: -487836541,
         defaultStore: TRUE},
  readOnly: NIL,
  storepassword: NIL,
  storeversion: 4},
 {name: "Card",
  signature: -246638930,
  TotalSize: 969488,
  UsedSize: 756068,
  kind: "Storage card",
  info: {defaultStore: TRUE},
  readOnly: NIL,
  storepassword: NIL,
  defaultStore: TRUE,
  storeversion: 4}]
```

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if outStores is null
```

DIL_Error PD_GetDefaultStore(PD_Handle inSession,
FD_Handle* outStore);

Return a store frame describing the default store as set by the Newton user. This frame contains the same information returned for _GetAllStores.

Example:

Result:

```
{name: "Card",
 signature: -246638930,
 TotalSize: 969488,
 UsedSize: 756068,
 kind: "Storage card",
 info: {defaultStore: TRUE},
 readOnly: NIL,
 storepassword: NIL,
 defaultStore: TRUE,
 storeversion: 4}
```

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if outStores is null
```

DIL_Error PD_GetCurrentStore(PD_Handle inSession,
FD_Handle* outStore);

Return the current store frame as last set by the application. This function is a PDIL convenience function and the Newton device is not asked for the current store. If PD_SetCurrentStore has not been called, this will return kFD_NIL.

Example:

Result:

```
{name: "Internal",
 kind: "Internal",
 signature: 22315107,
 info: {lastrestorefromcard: -487836541}}
```

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session

DIL_Error PD_SetCurrentStore(PD_Handle inSession,
FD_Handle inStore);

Set the current store. 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 other calls. If inStore is kFD_NIL, the current store will be set to the default store as defined on the Newton, and a subsequent call to PD_GetCurrentStore will get the default store frame.

Example:

inStore is a frame containing (at least) the following slots:

```
{name: "Internal",
 kind: "Internal",
 signature: 0,
 info: {soup info frame },
}
```

The info slot is optional. If it is included, then the soup info on the Newton will be updated. Other slots (such as those returned in the _GetAllStores frame) will be ignored for this command.

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session

Soup functions

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

Create the specified soup on the current store using inSoupIndex as the array of index frames. Note that even if you have only one index, it must be placed into an array. If inSoupName already exists, this function is the same as PD_SetCurrentSoup (and the soup index does NOT get changed!)

Example:

Show an example of creating a valid array of soupIndex frames

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if inSoupIndex is not an array (or nil)
                        or doesn't have the required slots ??
                        or inSoupName is 0 or > 39 characters
```

```
DIL_Error PD_DeleteSoup(PD_Handle inSession,
                        FD_Handle inSoupName);
```

Delete the specified soup on the current store. inSoup is the name of the soup to delete.

Example:

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if inSoup is not a string
```

```
DIL_Error PD_EmptySoup(PD_Handle inSession,
                       FD_Handle inSoupName);
```

Remove all the entries from the specified soup on the current store. inSoup is the name of the soup to empty.

Example:

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if inSoupName is not a string
```

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

Return an array of soup names and signatures from the current store. The array is actually an array of arrays. FD_GetLength on the array will give you the number of soups on the store, and FD_GetArraySlot will allow you to extract the inner array which has the name and signature of the soup.

Result:

```
[["Calendar", -241498083],
 ["Calendar Notes", -242377639],
 ["Directory", -309224836],
 ["InBox", -314656770],
 ["Library", -231384509],
 ["Names", 213450357],
 ["NewtWorks", 88726189],
 ["Notes", 384199549],
 ["OutBox", -257835846],
 ["Packages", -100387713],
 ["Repeat Meetings", -114179748],
 ["Repeat Notes", 135516493],
 ["System", 494944721],
 ["SystemAlarmSoup", 258019192],
 ["To do", 219290207],
 ["To Do List", 528579200]]
```

Example:

```
FD_Handle   soupList;
FD_Handle   soupName;
FD_Handle   soupSignature;
long        nSoups;
...
PD_GetAllSoups(gSession, &soupList);

      // iterate through all the soups
for (int ii=0; ii < FD_GetLength(soupList); ++i)
{
      FD_Handle element = FD_GetArraySlot(soupList, ii);
      soupName = FD_GetArraySlot(element, 0);
      soupSignature = FD_GetArraySlot(element, 1);
}
```

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if outSoupNames or outSoupSignatures
                        is NULL
```

```
DIL_Error PD_GetCurrentSoup(PD_Handle inSession,
                            FD_Handle* outSoup);
```

Return the current soup as last set by the application. This function is a PDIL convenience function and the Newton device is not asked for the current soup. If PD_SetCurrentSoup has not been called, this will return kFD_NIL.

Example:

```
FD_Handle   gCurrentSoup;
PD_Handle   gSession;
...
PD_GetCurrentSoup(gSession, &gCurrentSoup);
```

Result:

The name of the current soup.

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
```

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

Set the soup on the current store for subsequent entry functions. inSoup is the name of the soup to use. This function must be called before any of the entry functions.

Example:

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if inSoupName is not a string
```

Entry functions

The following functions are used only after a current store and current soup have successfully been set. See the section called "Basic Flow Using PDIL" at the begining of this document on what routines to call when.

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

Add the specified entry, and return the new unique ID.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if inEntry is not a frame
	if outID is not a pointer

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

Change the specified entry.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if inEntry is not a frame

```
DIL_Error PD_DeleteEntries(PD_Handle inSession,
                           FD_Handle entryIDs);
```

Remove the entries specified by the array of entryIDs from the current soup.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if entryIDs is not an array

```
DIL_Error PD_GetEntry(PD_Handle inSession,
                      long entryID);
```

Get the entry with the specified uniqueID from the current soup.

Example:

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
```

```
DIL_Error PD_GetSoupIDs(PD_Handle  inSession,
                        FD_Handle* outSoupIDs);
```

Return an array of entry ID's from the current soup. The resulting entryID can then be used as a parameter to the _GetEntry or _DeleteEntries functions.

Example:

```
[0,
 5,
 6,
 7]
```

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if outSoupIDs is NULL
```

Cursor functions

These cursor functions are an alternative to the "Store, Soup, Entry" set of functions, and are generally easier to use if all you want to do is read entries from the Newton. It is very important to note that you cannot mix and match these functions with the others. For example, you can not generate a query and then make a PD_DeleteEntries or PD_AddEntry call. The PDIL does nothing to prevent that, but the ROM will generate errors if the wrong calls are made (typically a kDBadCurrentSoup error)

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

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

```
      soup  string      the name of the soup
      query kFD_NIL     to simply iterate through using
                        the default index
      frame             a query spec
                        Details to be provided
```

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inSession is not a real session
      kDIL_InvalidParameter   if inSoupName is not a string
                              if inSoupQuery is not a frame
```

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

Return the number of entries in the specified cursor.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
      kDIL_InvalidParameter   if outCount is not a pointer
```

```
DIL_Error PD_DisposeCursor(PD_Handle inSession,
                           PD_Cursor cursor);
```

Dispose of the specified cursor.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
```

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

Return the current entry from the specified cursor.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
      kDIL_InvalidParameter   if outEntry is not a pointer
```

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

The entry at the specified key location is returned. Nil is returned if there is no entry with the specified key.

Example:

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inCursor is not a real cursor
kDIL_InvalidParameter   if outEntry is not a pointer
                        if inKey is not a string or an integer
```

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

Move the specified cursor the number of entries specified by offset from the current position, and return the resulting entry. Offset can be positive or

negative.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
      kDIL_InvalidParameter   if outEntry is not a pointer
```

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

Advance the cursor to the next entry and return the entry.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
      kDIL_InvalidParameter   if outEntry is not a pointer
```

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

Backup the cursor to the previous entry and return the entry.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
      kDIL_InvalidParameter   if outEntry is not a pointer
```

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

Position the cursor to the beginning and return the first entry.

Example:

Error codes:

```
      kPD_NotInitialized      if PD_Startup has not been called
      kDIL_InvalidHandle      if inCursor is not a real cursor
      kDIL_InvalidParameter   if outEntry is not a pointer
```

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

Position the cursor to the end and return the last entry.

Example:

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inCursor is not a real cursor
kDIL_InvalidParameter	if outEntry is not a pointer

Package Loading

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

Load a package that is lenPackage bytes long. 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.

Note: We recommend a chunkSize of 1k (1024 bytes) to allow for responsiveness on the desktop side,.

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
```

A simple example of package loading:

```
void loadPackage(char* filename, PD_Handle session)
{
   FILE*       package;
   fpos_t      filesize;

   if ((package = fopen(filename, "r")) == NULL)
      return;

/* get the size of the package file */
   fseek(package, 0, SEEK_END);
   fgetpos(package, &filesize);
   fseek(package, 0, SEEK_SET);

   PD_LoadPackage(session, filesize,
               kLoadPackageDefaultChunkSize,
               ReadPackage, package));
   fclose(package);
}

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

A relatively complex example of a callback for package loading:

```
struct CallbackData
{
   CFile*      fFile;
   DWORD       fFileSize;
   DWORD       fAmtRead;
   CDialog*    fDialog;
};

DIL_Error      DownloadCallback(void* buffer, long* amtRead,
                           void* userData)
{
   CallbackData*     data = (CallbackData*) userData;

   *amtRead = data->fFile->Read(buffer, *amtRead);
// Actually, CFile::Read will throw an exception on error; we
// should catch and handle it, possibly returning the result
// of GetLastError as this function's result, or perhaps
// CFileException::m_cause or CFileException::m_lOsError.

   data->fAmtRead += *amtRead;

   CProgressCtrl*    bar = (CProgressCtrl*)
               data->fDialog->GetDlgItem(IDC_PROGRESS_BAR);
   bar->SetPos(data->fAmtRead / 1024);

   // Could also check for a click on Cancel here.
   return (*amtRead == 0);
}

DIL_Error      DoDownload(PD_Handle session, CFile& pkg,
                     CDialog& progress)
{
   DWORD packageSize = pkg.GetLength();

   CallbackData      data;
   data.fFile        = &pkg;
   data.fFileSize    = pkg.GetLength();
   data.fAmtRead     = 0;
   data.fDialog      = &progress;
   CProgressCtrl*    bar = (CProgressCtrl*)
                     progress.GetDlgItem(IDC_PROGRESS_BAR);
   bar->SetRange(0, data.fFileSize / 1024);

   return PD_DownloadPackage(session,
                     data.fFileSize,
                     kLoadPackageDefaultChunkSize,
                     DownloadCallback,
                     &data);
}
```

Protocol Extensions

Protocol extensions can be used to add functionality beyond that provided by the PDIL. The extension is a Newton Script closure that must be compiled on the desktop by NTK. Typically, an NTK project is set up to create a stream file, and the contents of the resulting stream file must be read by the application and passed to this function.

The protocol extension can be called and the result will be returned by PD_CallExtension.

If necessary, the extension can be removed by PD_RemoveExtension, although all protocol extensions are automatically removed when the connection terminates.

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

Load a protocol extension from inExtension and assign it the specified inExtensionID.

Note: Protocol extension id's are usually represented by 4 characters. Newton, Inc. reserves all lower-case identifiers.

Error codes:

kPD_NotInitialized	if PD_Startup has not been called
kDIL_InvalidHandle	if inSession is not a real session
kDIL_InvalidParameter	if inExtension is not a frame
	if outExtensionID is not a pointer

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

Call the specified protocol extension, passing params as a parameter array, and receiving results from the extension.

Note: The protocol extension MUST return a result and can be either a simple integer or a NewtonScript object.

Example: This very simple protocol extension simply beeps the specified number of times. This source can be pasted into a .f file and used by NTK to create a stream file. The extension calls :ReadCommandData() to read the passed in value (which in this case is a simple integer) and calls :WriteCommand() to return the result of 0 (which is the simplest return value and usually indicates the command completed successfully)

```
output := func(ep)
begin
      local nTimes := ep:ReadCommandData();
      for i := 1 to nTimes do
            GetRoot():SysBeep();
      ep:WriteCommand("BEEP", 0, true);
      return nil;
end;
```

This is the method for calling the extension from the PDIL to make the Newton device beep 3 times:

```
FD_Handle result;
err = PD_CallExtension(gSession, 'BEEP',
                                 FD_MakeInt(3),
                                 &result);
```

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
kDIL_InvalidParameter   if inParams is not a frame
                        or kFD_NIL
                        if outResults is NULL
```

```
DIL_Error PD_RemoveExtension(PD_Handle inSession,
                             long extensionID);
```

Remove the specified protocol extension. All protocol extensions are automatically removed when the Newton Dock application terminates.

Example:

Error codes:

```
kPD_NotInitialized      if PD_Startup has not been called
kDIL_InvalidHandle      if inSession is not a real session
```