FDIL 2.0

Introduction

The FDIL is a small library for 68K, PPC, and x86 applications that need to create, handle, modify, and query NewtonScript-compatible objects. Operations are provided via a C API. Because the objects the FDIL works on are NewtonScript-compatible, they can be exchanged with Newton OS devices using communications libraries such as the CDIL.

FDIL Concepts

When developing software for Newton OS devices, programmers use the NewtonScript programming language. NewtonScript is a dynamic, object-oriented language in that objects can be created and dynamically changed at runtime, and that operations can be performed on them by sending messages to them.

Software developers may find the need to create NewtonScript objects on the desktop instead of directly on the Newton OS device. For instance, developers creating any sort of desktop application that provides import/export functionality will need to receive NewtonScript objects from the Newton device and send NewtonScript objects to the Newton device. They will also need to extract data from the objects received from the Newton device, and create data that will be sent to the Newton device. Thus, they will need a Frames-like library on the desktop.

Newton, Inc., already provides such a library with the HLFDIL. HLFDIL stands for “High-level Frames Desktop Integration Library”. The HLFDIL works with the CDIL (“Communications Desktop Integration Library”) to perform the operations just described. However, there are several shortcomings of the HLFDIL:

- Because it is a “high-level” library, it lacks support for some low-level operations. The HLFDIL was designed to integrate with existing desktop products, and thus works with data models common in those products. But those data models are often at odds with the data models that NewtonScript object-savvy programs should adopt.

- The HLFDIL is memory intensive. It allocates a lot of memory for each object, reducing the total number of objects that an application can manipulate.

- The HLFDIL is not comprehensive in the kinds of objects it supports. For instance, it doesn’t support large-binary objects or object compression.

- The HLFDIL provides no support for nested aggregate objects (such as frames or arrays that contain other arrays or frames).

For all these reasons, the FDIL was developed as alternative to the HLFDIL.

Examples

NewtonScript developers use objects all the time. Even statements such as:

```
local myInteger := 5;
```

result in an object being created. Internally, an object is created and assigned the value of 5. A desktop developer using the FDIL would use the following code to perform the same operation:

```
FD_Handle   myInteger = FD_MakeInt(5);
```

In NewtonScript, a developer may then use “myInteger” as the value of a slot in a frame (frames and the other kinds of NewtonScript-supported objects are described in the next section). They would probably use something like the following:

```
local myFrame := { aNumber: myInteger, anotherNumber: 12 };
```

The equivalent using the FDIL API would be:

```
FD_Handle   myFrame = FD_MakeFrame();
FD_SetFrameSlot(myFrame, "aNumber", myInteger);
FD_SetFrameSlot(myFrame, "anotherNumber", FD_MakeInt(12));
```

In this example, we first create an empty frame. Next, we add the first slot. The name of the slot is specified by the string passed as the second parameter. The value of the slot is specified in the third parameter. Here, we pass the NewtonScript integer created earlier. The second slot is added in a similar fashion.

The desktop application could then save this object to disk for later, or send it to a companion application running on a Newton OS device. The desktop application could use the CDIL to send the object with something like the following:

```
DIL_Error   WriteCallback(const void* buf, long amt, void* userData)
{
      CD_Handle   pipe = (CD_Handle) userData;
      DIL_Error   err = CD_Write(pipe, buf, &amt, 0);
      return err;
}

DIL_Error   SendObjectToNewton(FD_Handle myObject)
{
      CD_Handle   myPipe = MyGetCDILHandle();
      FD_Flatten(myObject, WriteCallback, myPipe);
      return FD_GetError();
}
```

In this example, SendObjectToNewton calls FD_Flatten to stream out the object in a Frames-compatible bytestream format. This bytestream can then be written to some destination via a client-supplied callback function. In our example, the callback function writes the bytestream to a serial, TCP/IP or ADSP pipe managed by a CD_Handle. A NewtonScript application using the appropriate endpoint object can then read and resurrect the object on the Newton OS device.

The developer could have just as easily specified a callback that wrote the bytes to disk. Operations

for reading bytestreams and resurrecting objects from them are performed similarly.

The Objects

NewtonScript supports a number of different kinds of objects. These object types can be grouped into a simple taxonomy. At the top level, there are four object types:

- Integer objects
- Immediate objects
- Pointer objects
- Magic pointer objects

Integer, immediate, and magic pointer objects can be thought of as “value objects”, while pointer objects can be thought of as “reference objects”.

Value objects are objects that are right there in your hand. They are analogous to scalar types that you find in C/C++, such as “char”, “short”, “int”, and “long”, as well as “float” and “double”. Assigning one value object to another value object essentially makes a copy of that object. After make such an assignment, a change made to one object does not affect the other object.

Reference objects are actually references to a real object that resides elsewhere (in the case of the FDIL, these objects are allocated on the application’s heap). They are analogous to the pointers you’d find in C/C++, such as “void*” and “char*”. Copying one reference object to another doesn’t actually copy the object; it just copies the reference to the object. After making such an assignment, making a change to the object through one reference will be noticed when accessing the object through the other reference.

Integer Objects

Integer objects are just that: objects containing integral values. The integers are stored in a 30-bit field, allowing them a range of -536,870,912...536,870,911.

Immediate Objects

Immediate objects are other integral objects that aren’t exactly integers. Immediates are broken down into four sub-types:

- Special immediates
- Character immediates
- Boolean immediates
- Reserved immediates

Special immediates are generally used internally to the Frames library. The only kind of special immediate developers are likely to encounter is the NIL object. The NIL object is like a “nothing object” or a “different from everything else” object. It is often returned from functions that can’t performed the required task (like find and return a requested object) or as a parameter to a function when no other meaningful value can be passed (similar to passing zero or NULL as a parameter in

C/C++ programs). See the NewtonScript Programmer's Reference for more information on NewtonScript's "nil" keyword.

Character immediates contain single Unicode characters. In practice, these aren't used much. Developers manipulating text instead often use the routines that manipulate Unicode strings.

Boolean immediates comprise the small set of objects containing FALSE and TRUE. In practice, FALSE is never used. Instead, functions accepting or returning FALSE values instead use NIL. TRUE can be used wherever a non-NIL value is needed, though need for such a value is rare.

Support for reserved immediates is provided by the Frames and FDIL libraries, though their function has not yet been defined.

Pointer Objects

Pointer objects comprise the set of non-scalar type objects. These are the kinds of complex objects that would be allocated on a heap in programs developed with C or C++. They can range from simple binary objects containing unformatted or client-defined data to complex aggregate objects such as frames (an aggregate object is an object containing references to other objects).

Pointer objects can be broken down into four sub-types:

- Binaries
- Arrays
- Frames
- Indirect binaries

A binary object is analogous to a block of memory allocated returned by the Standard C Library function malloc. It is a raw, unformatted block of memory. The user can store any sort of data into it they like. Except for a few cases (described later), the FDIL will not look into that object and attempt to interpret its contents.

An array object is a variable-size object whose contents are divided into a series of other objects. Each division is called a "slot". Objects can be inserted into an array or appended to the end of an array. A single object or a range of objects can be removed from an array (with the remaining objects moving up in the array to take their place). Once an object has been added to an array, it can be replaced with another object. Arrays can be iterated to visit all their elements.

A frame is akin to a dictionary or associative array. It is a collection of objects where elements are stored as key/value pairs. Rather than using an index to retrieve a value that's been added to a frame (as you would with an array), you specify the key used when the value was added to the frame. Any number of values can be added to a frame, but only one value can be associated with any one key. This means that all keys added to a frame must be unique. As mentioned earlier, keys are symbol objects. After a key/value pair has been added to a frame, its key can be used to retrieve the value associated with it, remove the value from the frame, replace the value in the frame, or test whether or not a key/value pair exists in the frame. Frames can be iterated to visit all key/value pairs.

An indirect binary is one where the the contents of the binary are stored "elsewhere". Flexible

support is provided in Frames for indirect binaries where “elsewhere” can take on several different forms. The FDIL supports just one kind of indirect binary: the large binary.

A large binary object is similar to a binary object in that it is a raw, unformatted block of memory. It differs from a regular binary object in that it may not all fit into memory at once. To support such objects, the FDIL can optionally compress the object’s data when it is not being accessed, or swap parts of the object to and from disk.

The FDIL uses regular binary objects to implement the following additional kinds of objects:

- Floating point number
- String
- Symbol

A floating point number object is an 8-byte binary object containing an IEEE-754 floating point value. When using the FDIL library, it is important that you set any applicable compiler options for generating IEEE-754 floating point compatible code.

A string object is a variable-size object containing an array of Unicode characters. The array of characters is terminated with a NULL (zero) value.

A rich string is a variation of a string. A rich string is a string containing embedded “ink”, which is uninterpreted strokes created by the user of a Newton OS device. The FDIL does not support the extraction or interpretation of ink, and functions returning the characters of a string will contain either 0xF700 or 0x1A in the place of the embedded ink, depending on whether you are extracting 16-bit or 8-bit characters.

A symbol object is a variable-size object used as a token or identifier. Most often it is used as a slot name or object class (object classes are described later). It is composed of ASCII characters with values between 32 and 127 inclusive, excluding the vertical bar (‘|’) and backslash (‘\’). A symbol must also be shorter than 254 characters. When symbols are compared to each other, a case-insensitive comparison is performed.

All symbols created by the FDIL are remembered internally. If a client requests the creation of a symbol that’s already been created, a referenced to that previously created symbol is returned. If the requested symbol has not been previously created, it is created, added to an internal table, and returned to the client. Thus, only one instance of any unique symbol ever exists.

Magic Pointer Objects

Magic pointer objects are a mechanism used for late object binding. When a programmer develops their NewtonScript application on the desktop using a development environment such as the Newton Toolkit (NTK), they often need to create references to objects that exist only in the Newton device ROM. Because there can be no direct reference from an object on the desktop to an object in a Newton device ROM, a “magic pointer” is used as a placeholder when the developer’s package is created. Later, when the package is downloaded and executed, the magic pointer is used to lookup the real object in the ROM.

While the FDIL client is not likely to face the need of creating a magic pointer object, support for

them is provided for completeness.

Object Classes

All objects have a class associated with them. The class is used to help identify the type of object. For non-pointer objects, this class is implicit and cannot be changed. For most pointer objects, the class can be set (or changed from its default if that class is normally created with a default).

Often, a class is specified with a symbol object. However, there are cases where a class can be NIL, or where it is an integer object or even an immediate object. Generally, you should not worry about what kind of object a class is. Instead, you should use the predicate functions provided by the FDIL API, or compare an object's class to the standard set of classes exported by the FDIL. Only in rare cases should you need to dissect an object's class.

The following table lists the objects type just described and their associated classes.

Object type	Class
Integer	kFD_SymInteger
Character	kFD_SymChar
Boolean	kFD_SymBoolean
Other immediate	kFD_SymWeird_Immediate
Frame	kFD_SymFrame *
Array	kFD_SymArray *
String	kFD_SymString *
Symbol	kFD_SymSymbol
Binary	kFD_NIL *
Large binary	kFD_NIL *
MagicPointer	kFD_SymMagicPointer

* The classes of these object types can be changed by the FDIL client, so the values shown here are the default classes.

Library Reference

Data Types

`FD_Handle`

All objects created and managed by the FDIL are referenced via the FD_Handle type. When objects are created and returned to the user, the creating function returns an FD_Handle. In the non-debug version of the library, an FD_Handle is merely a long integer, making it a very lightweight data type. In the debug version of the library, FD_Handle is an 8-byte struct, with the additional 4 bytes being a pointer to internal debugging data structures.

`DIL_Error`

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

`DIL_WideChar`

There is currently no de facto standard for the Unicode character type. Some development environments define UniChar; some don't. Some development environments define wchar_t; some don't. Of those that define wchar_t, some define it as a 16-bit value, others as an 8-bit value. The FDIL supports Unicode characters as unsigned 16-bit values, and defines DIL_WideChar as such.

`FD_ImmediateType`

An enumerated type describing the four different types of immediate objects.

`FD_CompressionType`

An enumerated type describing the three different types of large binary compression.

Using The Library

```
DIL_Error   FD_Startup(void);
```

Initializes the FDIL. You must call this function before calling any other FDIL 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 FD_Shutdown are also made.

Example:

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

Error codes:

kDIL_OutOfMemory

```
DIL_Error   FD_Shutdown(void);
```

Closes the library. All memory allocated by the FDIL since FD_Startup was called is deallocated.

Example:

```
int CMyApp::ExitInstance()
```

```
{
      ...
      FD_Shutdown();
      return CWinApp::ExitInstance();
}
```

Error codes:
kFD_FDILNotInitialized

```
DIL_Error   FD_GetError(void);
```

Returns a value indicating the success or failure of the last operation performed by an FDIL function. Robust applications should check the result of FD_GetError after calling any FDIL function that can reasonably be expected to fail.

Example:

```
FD_Handle   myObject = FD_MakeBinary(10 * 1024 * 1024, NULL);
if (FD_GetError() != kDIL_NoError)
      /* an error occurred */;
```

Creating Objects

```
FD_Handle   FD_MakeInt(long);
```

Creates an integer object from the given value, returning the newly created object. The value of the integer can be between -536,870,912...536,870,911, inclusive.

Example:

```
FD_Handle   myInt = FD_MakeInt(100);
```

Error Codes:
kFD_FDILNotInitialized
kFD_ValueOutOfRange

```
FD_Handle   FD_MakeReal(double);
```

Creates a floating point object from the given value, returning the newly created object. The value can be any valid IEEE-754 floating point value.

NOTE: When using the FDIL library, it is important that you set any applicable compiler options for generating IEEE-754 floating point compatible code. For example, when compiling a 68K program with CodeWarrior, make sure the "8-byte Doubles" option is turned on.

Example:

```
FD_Handle   myReal = FD_MakeReal(10.0);
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory

```
FD_Handle   FD_MakeString(const char*);
FD_Handle   FD_MakeWideString(const DIL_WideChar*);
```

Creates a binary object containing a NULL-terminated Unicode string, returning the newly created object. FD_MakeString takes as input a NULL-terminated series of ASCII characters (in other words, a “C string”), converts them into Unicode characters, and stores them in an appropriately sized binary object. FD_MakeWideChar merely copies the NULL-terminates series of Unicode characters into an appropriately sized binary object. In both cases, the resulting object contains Unicode characters terminated with a NULL, and the object’s class is kFD_SymString.

Example:

```
FD_Handle   myString1 = FD_MakeString("Some text");
FD_Handle   myString2 = FD_MakeWideString(L"Some wide text");
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_NULLPointer

```
FD_Handle   FD_MakeSymbol(const char*);
```

Returns a symbol corresponding to the given string, creating it if necessary. Symbols are a pooled resource: once created, a symbol is added to an internal table. Subsequent requests to create a new symbol from the same text results in a reference to the previously created symbol to be returned.

In the examples shown below, only the first two calls to FD_MakeSymbol result in the creation of new symbols. Since the third call to FD_MakeSymbol specifies the same text as the first call, the object returned references the same object referenced by mySymbol1. In the fourth call to FD_MakeSymbol, text is specified that differs from that passed in the second call only in the capitalization of the characters. Because symbol text is treated in a case-insensitive fashion, no new symbol is created, and a reference to the symbol referenced by mySymbol2 is returned.

Example:

```
FD_Handle   mySymbol1 = FD_MakeSymbol("mySlotName1");
FD_Handle   mySymbol2 = FD_MakeSymbol("mySlotName2");
FD_Handle   mySymbol3 = FD_MakeSymbol("mySlotName1");
FD_Handle   mySymbol4 = FD_MakeSymbol("MySlOtNaMe2");
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_NULLPointer
kFD_SymbolTooLong
kFD_IllegalCharInSymbol

```
FD_Handle   FD_MakeArray(long size, const char* cls);
```

Creates an array large enough to hold the given number of elements, returning the newly created object. All of the initial elements are set to kFD_NIL. The length of the array is not fixed; it can be changed implicitly with calls to FD_InsertArraySlot, FD_AppendArraySlot, and FD_RemoveArraySlot, or explicitly with a call to FD_SetLength.

The class of the array is specified with the second parameter. Passing NULL results in the array being given a default class. Passing in anything else is passed internally to FD_MakeSymbol, and the result is used as the class.

Example:

```
FD_Handle   myArray1 = FD_MakeArray(100, "myArray");
FD_Handle   myArray2 = FD_MakeArray(0, NULL);   /* Zero's OK */
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ValueOutOfRange

```
FD_Handle   FD_MakeFrame(void);
```

Creates an empty frame, returning the newly created object. Initially, the frame contains no contents. They must be added with calls to FD_SetFrameSlot.

Example:

```
FD_Handle   myFrame = FD_MakeFrame();
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory

```
FD_Handle   FD_MakeBinary(long size, const char* cls);
```

Creates a raw, unformatted binary object of the given size, returning the newly created object. The contents of the object can later be accessed by calling FD_GetBinaryData.

The class of the binary object is specified with the second parameter. Passing NULL results in the binary object being given a default class. Passing in anything else is passed internally to FD_MakeSymbol, and the result is used as the class.

Example:

```
FD_Handle   myBinary1 = FD_MakeBinary(1000, "myObj");
FD_Handle   myBinary2 = FD_MakeBinary(0, NULL); /* Zero's OK */
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ValueOutOfRange

```
FD_Handle   FD_MakeChar(char);
FD_Handle   FD_MakeWideChar(DIL_WideChar);
```

Creates a character object from the given character, returning the newly created object. FD_MakeChar first converts the given character from ASCII to Unicode before creating the object. FD_MakeWideChar uses the given Unicode character when creating the object.

In the examples below, myChar1 and myChar2 end up containing equal character objects.

Example:

```
FD_Handle   myChar1 = FD_MakeChar('a');
FD_Handle   myChar2 = FD_MakeWideChar(L'a');
```

Error Codes:

kFD_FDILNotInitialized

```
FD_Handle   FD_MakeImmediate(long type, long value);
```

Creates the specified type of immediate object based on the given value, returning the newly created object. This is a low-level function that you should rarely, if ever, call. The kinds of immediate objects applications are likely to require are character objects (which can be created with the FD_MakeChar and FD_MakeWideChar function), NIL objects (which can be accessed through the kFD_NIL constant), or boolean objects (the sole type of which can be access through the kFD_True constant).

Note that in the following examples, the call to FD_MakeImmediate that creates a character object does not perform ASCII to Unicode conversion on the given character. That higher-level operation is performed only by FD_MakeChar.

Example:

```
FD_Handle   myNIL = FD_MakeImmediate(kFD_ImmedSpecial, 0);
FD_Handle   myTrue = FD_MakeImmediate(kFD_ImmedBoolean, 1);
FD_Handle   myChar = FD_MakeImmediate(kFD_ImmedChar, 'a');
```

Error Codes:

kFD_FDILNotInitialized
kFD_ValueOutOfRange

```
FD_Handle   FD_MakeMagicPointer(long value);
```

Creates a magic pointer object based on the given value, returning the newly created value. It is not likely that you should need to call this function; it is provided for completeness only. The only kind of applications that would need to create this kind of object would be a development environment, which needs to build application packages that reference ROM objects.

Each value that can be passed to FD_MakeMagicPointer corresponds to a different well-known object in ROM (such as the protoApp object). The list of values and objects they correspond to is not documented or provided here. You might want to try badgering DTS for them, but they might just laugh at you.

Example:

```
FD_Handle   myMP = FD_MakeMagicPointer(157);    /* protoApp */
```

Error Codes:

kFD_FDILNotInitialized
kFD_ValueOutOfRange

```
FD_Handle   FD_MakeLargeBinary(long size, const char* objClass,
                                    long compressionType);
```

Creates a large binary object of the given size, returning the newly created object. Whether or not the object's data is compressed internally is determined by the compressed parameter. Management for the large binary's data is performed by the functions specified in the FD_LargeBinaryProcs data structure established by the `FD_SetLargeBinaryProcs` function. For more information on these functions, see the section "Large Binary Object Functions".

Example:

```
FD_Handle   myLB = FD_MakeLargeBinary(500 * 1024L, "MyBlob",
                                                kFD_NoCompression);
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ErrorCreatingStore
kFD_ValueOutOfRange

Disposing of Objects

```
DIL_Error   FD_Dispose(FD_Handle);
DIL_Error   FD_DeepDispose(FD_Handle);
```

After you are through with an object, you will likely need to dispose of it. Passing an FD_Handle to FD_Dispose will dispose of that object's memory. FD_Dispose does a shallow dispose. In other words, if the object you pass to it is an aggregate object such as an array or a frame, the referenced sub-objects are not disposed. In order to dispose of the whole kit-and-kaboodle, call FD_DeepDispose.

FD_Dispose and FD_DeepDispose can be called on any type of object. The functions will query the type of object and act appropriately. For instance, passing an integer object to FD_Dispose won't do anything, as integer objects are not heap-based objects and don't need to be disposed. Calling FD_DeepDispose on a string will dispose of the string's memory, but no attempt will be made to dispose of any sub-objects, as strings don't have sub-objects. Calling either function on a symbol won't do anything, as symbols are pooled and shared; actually deleting a symbol could create dangling references elsewhere in the system.

Example:

```
FD_Handle   myInt = FD_MakeInt(5);
FD_Dispose(myInt);      /* Sets myInt to NIL */

FD_Handle   myReal = FD_MakeReal(5);
FD_Dispose(myReal);     /* Frees memory, sets myReal to NIL */

FD_Handle   myString = FD_MakeString("Hello");
FD_Handle   myArray = FD_MakeArray(1, NULL);
FD_SetArraySlot(myArray, 0, myString);
FD_Dispose(myArray);    /* myString still exists */

myArray = FD_MakeArray(1, NULL);
FD_SetArraySlot(myArray, 0, myString);
FD_DeepDispose(myArray);      /* Both myArray and myString are
                                 disposed of. However, only myArray
                                 is set to NIL; myString now ref-
                                 erences a disposed object, so
                                 watch out! */

FD_Handle   mySymbol = FD_MakeSymbol("mySlotName");
FD_Dispose(mySymbol);   /* mySymbol set to NIL, but no memory
                           is actually freed. */
```

Error Codes:

kFD_FDILNotInitialized
kFD_PointerObjectIsFree

```
long  FD_AllocatedMemory(void);
```

Returns the total amount of memory allocated by the FDIL, including that occupied by created objects and that used by internal data structures. This function can be useful to track how much memory is used by particular objects, or by the FDIL sub-system in general.

Example:

```
long        allocated1 = FD_AllocatedMemory();
FD_Handle   myObj = FD_MakeFrame();
long        allocated2 = FD_AllocatedMemory();
printf("An empty frame eats up %ld bytes.\n",
                                      allocated2 - allocated1);
```

Error Codes:

kFD_FDILNotInitialized

Testing Objects

```
int   FD_IsInt(FD_Handle);
```

Returns whether or not an object is an integer object. Only objects created with FD_MakeInt will cause this function to return true.

Example:

```
FD_Handle   myInt = FD_MakeInt(5);
int         result = FD_IsInt(myInt);     // returns TRUE

            result = FD_IsInt(kFD_NIL);   // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsPointerObject(FD_Handle);
```

Returns whether or not an object is a pointer object. Objects created with FD_MakeReal, FD_MakeString, FD_MakeWideString, FD_MakeSymbol, FD_MakeArray, FD_MakeFrame, FD_MakeBinary, and FD_MakeLargeBinary will cause this function to return true.

Example:

```
FD_Handle   myObj1 = FD_MakeBinary(5, NULL);
int   result = FD_IsPointerObject(myObj1);      // returns TRUE

FD_Handle   myObj2 = FD_MakeFrame();
      result = FD_IsPointerObject(myObj2);      // returns TRUE

      result = FD_IsPointerObject(kFD_NIL);     // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsReal(FD_Handle);
```

Returns whether or not an object contains a floating point number. Only objects created with FD_MakeReal will cause this function to return true.

Example:

```
FD_Handle   myReal = FD_MakeReal(5);
int         result = FD_IsReal(myReal);   // returns TRUE

            result = FD_IsReal(kFD_NIL);  // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsString(FD_Handle);
```

Returns whether or not an object is a string object. Objects created with FD_MakeString and FD_MakeWideString will cause this function to return true, as well as any objects for which FD_IsSubClass(object, "string") will return true.

Example:

```
FD_Handle   myString1 = FD_MakeString("Hello");
int         result = FD_IsString(myString1);    // returns TRUE

FD_Handle   myString2 = FD_MakeWideString(L"Hello");
            result = FD_IsString(myString2);    // returns TRUE

            result = FD_IsString(kFD_NIL);      // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsRichString(FD_Handle);
```

Returns whether or not an object is a rich string object. Rich string objects are string containing embedded ink. These object cannot be created by the FDIL, nor can the ink be extracted or interpreted. However, you may receive such objects from a Newton OS device via the PDIL, for example, and may need to detect strings that cannot be completely interpreted.

Example:

```
FD_Handle   myString1 = MyGetStringFromNewt();
if (FD_IsRichString())
      MyShowAlert("Warning: string can't be completely"
                  "translated. Some information may be lost");
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsSymbol(FD_Handle);
```

Returns whether or not an object is a symbol object. Only objects created with FD_MakeSymbol will cause this function to return true.

Example:

```
FD_Handle   mySymbol = FD_MakeSymbol("mySlotName");
int         result = FD_IsSymbol(mySymbol);     // returns TRUE

            result = FD_IsSymbol(kFD_NIL);      // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsArray(FD_Handle);
```

Returns whether or not an object is an array object. Only objects created with FD_MakeArray will cause this function to return true.

Example:

```
FD_Handle   myArray = FD_MakeArray(5, NULL);
int         result = FD_IsArray(myArray); // returns TRUE

            result = FD_IsArray(kFD_NIL); // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsFrame(FD_Handle);
```

Returns whether or not an object is a frame object. Only objects created with FD_MakeFrame will cause this function to return true.

Example:

```
FD_Handle   myFrame = FD_MakeFrame();
int         result = FD_IsFrame(myFrame); // returns TRUE

            result = FD_IsFrame(kFD_NIL); // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsBinary(FD_Handle);
```

Returns whether or not an object is a raw binary object. Objects created with FD_MakeBinary, FD_MakeReal, FD_MakeString, FD_MakeWideString, and FD_MakeSymbol will cause this function to return true.

Example:

```
FD_Handle   myBinary = FD_MakeBinary(5, NULL);
int         result = FD_IsBinary(myBinary);     // returns TRUE

FD_Handle   myReal = FD_MakeReal(5);
            result = FD_IsBinary(myReal);       // returns TRUE

FD_Handle   myArray = FD_MakeArray(5, NULL);
            result = FD_IsBinary(myArray);      // returns FALSE

            result = FD_IsBinary(kFD_NIL);      // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsChar(FD_Handle);
```

Returns whether or not an object is a character object. Objects created with FD_MakeChar, FD_MakeWideChar, or FD_MakeImmediate (with kImmedChar as the type) will cause this function to return true.

Example:

```
FD_Handle   myChar = FD_MakeChar('a');
int         result = FD_IsChar(myChar);   // returns TRUE

            result = FD_IsChar(kFD_NIL);  // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsFree(FD_Handle);
```

Returns whether or not an object reference refers to a deleted pointer object. FD_Objects containing non-pointer objects such as integers or NIL cause this function to return false.

NOTE: This function may return false, even if the object originally referenced by the given FD_Handle was deleted. This can occur, for example, if a new object was allocated in such a way that it the same space previously occupied by the deleted object. The FD_Handle will then effectively refer to the newly created object, causing FD_IsFree to return false. Thus, FD_IsFree is mostly useful in the tracking down of object allocation and deletion bugs, and should probably not be called in shipping code.

Example:

```
FD_Handle   myString = FD_MakeString("Hello");
FD_Handle   myArray = FD_MakeArray(1, NULL);
FD_SetArraySlot(myArray, 0, myString);

int         result = FD_IsFree(myString); // returns FALSE
FD_DeepDispose(myArray);
result = FD_IsFree(myString); // returns TRUE
result = FD_IsFree(myArray);  // returns FALSE (myArray was set
                              // to NIL, which is not a pointer
                              // object, and so can't be a
                              // deleted pointer object).
result = FD_IsFree(kFD_NIL);  // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsImmediate(FD_Handle);
```

Returns whether or not an object is an immediate object. Objects created with

FD_MakeChar, FD_MakeWideChar, and FD_MakeImmediate will cause this function to return true.

Example:

```
FD_Handle   myInt = FD_MakeInt(5);
int         result = FD_IsImmediate(myInt);     // returns FALSE
            result = FD_IsImmediate(kFD_NIL);   // returns TRUE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsMagicPointer(FD_Handle);
```

Returns whether or not an object is a magic pointer object. Only objects created with FD_MakeMagicPointer will cause this function to return true.

Example:

```
FD_Handle   myMP = FD_MakeMagicPointer(157);
int   result = FD_IsMagicPointer(myMP);         // returns TRUE
      result = FD_IsMagicPointer(kFD_NIL);      // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsBoolean(FD_Handle);
```

Returns whether or not an object is a boolean object. Only kFD_True or a boolean object created with FD_MakeImmediate will cause this function to return true.

Example:

```
int         result = FD_IsBoolean(kFD_True);    // returns TRUE
            result = FD_IsBoolean(kFD_NIL);     // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsLargeBinary(FD_Handle);
```

Returns whether or not an object is a large binary object. Only objects created with FD_MakeLargeBinary will cause this function to return true.

Example:

```
FD_Handle   myLB = FD_MakeLargeBinary(5, NULL,
                                      kFD_NoCompression);
int         result = FD_IsLargeBinary(myLB);    // returns TRUE
            result = FD_IsLargeBinary(kFD_NIL); // returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

```
int   FD_IsSubClass(FD_Handle, const char*);
```

Returns whether or not an object is an instance of the given object class. An object is an instance of a class if one of the following conditions is true:

1. The specified class is the same as the class returned by FD_GetClass for the given object.
2. The object's class describes a class hierarchy, and the specified class is a prefix of that hierarchy.
3. The object's class is a well-known subclass of the given class.
4. The specified class is the empty string.

In Newton 2.0 OS and later, an object's class can describe a class hierarchy, where each node of the hierarchy is a subclass name. The subclass names are catenated together, separated by periods. Thus, if 'phone is a subclass of 'string, and 'homeOfficePhone is a subclass of 'phone, then a object with the class 'string.phone.homeOfficePhone is considered to be a string object, a phone object or a home office phone object.

This mechanism did not exist before Newton OS 2.0. In earlier ROMs, there were lists of well-known class relationships. The FDIL supports the following:

'string
- 'address
- 'company
- 'name
- 'title
- 'phone
  - 'homePhone
  - 'workPhone
  - 'faxPhone
  - 'otherPhone
  - 'carPhone
  - 'beeperPhone
  - 'mobilePhone
  - 'homeFaxPhone

Example:

```
// The number in parentheses is the condition under which
// FD_IsSubClass returns TRUE.

int   result;

FD_Handle   none = FD_MakeBinary(0, NULL);
result = FD_IsSubClass(none, "");               // TRUE (4)
result = FD_IsSubClass(none, "string");         // FALSE

FD_Handle   string = FD_MakeString("Fred");
result = FD_IsSubClass(string, "");             // TRUE (4)
result = FD_IsSubClass(string, "string");       // TRUE (1)
result = FD_IsSubClass(string, "string.phone"); // FALSE
```

```
FD_Handle   phone1 = FD_MakeString("1-408-974-0701");
FD_SetClass(phone1, "workPhone");
result = FD_IsSubClass(phone1, "");             // TRUE (4)
result = FD_IsSubClass(phone1, "string");       // TRUE (3)
result = FD_IsSubClass(phone1, "phone");        // TRUE (3)
result = FD_IsSubClass(phone1, "workPhone");    // TRUE (1)
result = FD_IsSubClass(phone1, "string.phone"); // FALSE
result = FD_IsSubClass(phone1, "string.phone.workPhone");
                                                // FALSE

FD_Handle   phone2 = FD_MakeString("1-408-354-5000");
FD_SetClass(phone2, "string.phone.workPhone");
result = FD_IsSubClass(phone2, "");             // TRUE (4)
result = FD_IsSubClass(phone2, "string");       // TRUE (2)
result = FD_IsSubClass(phone2, "phone");        // FALSE
result = FD_IsSubClass(phone2, "workPhone");    // FALSE
result = FD_IsSubClass(phone2, "string.phone"); // TRUE (2)
result = FD_IsSubClass(phone2, "string.phone.workPhone");
                                                // TRUE (1)
```

Error Codes:

kFD_FDILNotInitialized
kFD_PointerObjectIsFree
kFD_NULLPointer

Extracting Data From Objects

```
long  FD_GetInt(FD_Handle);
```

Returns the long value stored in the object.

Example:

```
FD_Handle   myInt = FD_MakeInt(5);
long        result = FD_GetInt(myInt);    // result == 5
```

Error Codes:

kFD_FDILNotInitialized
kFD_ExpectedInteger

```
double      FD_GetReal(FD_Handle);
```

Returns the floating point value stored in the object.

Example:

```
FD_Handle   myReal = FD_MakeReal(5);
double      result = FD_GetInt(myReal);   // result == 5.0
```

Error Codes:

kFD_FDILNotInitialized

kFD_ExpectedReal

```
DIL_Error   FD_GetString(FD_Handle, char* buffer, long bufLen);
```

Gets the characters stored in the object. The characters are converted from Unicode to ASCII and stored in the location indicated by the buffer parameter. At most bufLen characters are copied. If there's enough room, a NULL terminator is added.

Example:

```
FD_Handle   myString = FD_MakeString("Hello");

char  buffer[10];
FD_GetString(myString, buffer, 10); // buffer == "Hello\0"
FD_GetString(myString, buffer, 3);  // buffer == "Hel"
```

Error Codes:

kFD_FDILNotInitialized
kFD_ExpectedString
kFD_NULLPointer
kFD_ExpectedNonNegativeValue

```
DIL_Error   FD_GetWideString(FD_Handle, DIL_WideChar*, long bufLen);
```

Gets the characters stored in the object. The characters are stored in the location indicated by the buffer parameter. At most bufLen characters are copied. If there's enough room, a NULL terminator is added.

Example:

```
FD_Handle   myString = FD_MakeString("Hello");

DIL_WideChar      buffer[10];
FD_GetWideString(myString, buffer, 10);   // buffer == L"Hello\0"
FD_GetWideString(myString, buffer, 3);    // buffer == L"Hel"
```

Error Codes:

kFD_FDILNotInitialized
kFD_ExpectedString
kFD_NULLPointer
kFD_ExpectedNonNegativeValue

```
const char* FD_GetSymbol(FD_Handle);
```

Returns a pointer to the NULL-terminated characters of the symbol. This pointer should be used for read-only purposes. It should not be used to change the characters of the symbol.

Example:

```
FD_Handle   mySymbol = FD_MakeSymbol("mySlotName");
```

```
const char* symbolText = FD_GetSymbol(mySymbol);
      // symbolText == "mySlotName"

printf("Slot name is: %s\n", symbolText);
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedSymbol

```
void* FD_GetBinaryData(FD_Handle);
```

Returns a pointer to the contents of a raw binary object. Before the pointer can be used to access (read or write) the contents of the object, it will need to be cast to the appropriate type.

Note: FD_GetBinaryData cannot be used to get a pointer to the contents of large binary objects. That's because there's no guarantee that the entire object can be read into memory at once. Instance, clients should use FD_ReadFromLargeBinary and FD_WriteToLargeBinary to access and modify a large binary's contents.

Example:

```
static const char kMyCRCTable[] = {... };
FD_Handle   myCRCTable = FD_MakeBinary(sizeof(kMyCRCTable),
                                       "CRCTable");
char*       dest = (char*) FD_GetBinaryData(myCRCTable);
for (int ii = 0; ii < sizeof(kMyCRCTable); ++ii)
      dest[ii] = kMyCRCTable[ii];
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedBinary

```
char              FD_GetChar(FD_Handle);
DIL_WideChar      FD_GetWideChar(FD_Handle);
```

Returns the characters stored in the given character object. FD_GetChar returns the character in ASCII form. FD_GetWideChar returns the character in Unicode form.

Example:

```
FD_Handle         myChar = FD_MakeChar('a');
char              asASCII = FD_GetChar(myChar);       // == 'a'
DIL_WideChar      asUnicode = FD_GetWideChar(myChar); // == L'a'
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedChar

```
int   FD_IsNIL(FD_Handle);
int   FD_NotNIL(FD_Handle);
```

Returns whether or not the given object is equal to NIL. The two functions are complementary. If FD_IsNIL returns true, FD_NotNIL returns false, and vice-versa.

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedChar

```
FD_Handle   FD_ASCIIString(FD_Handle);
```

Takes the given string, converts its contents into ASCII, and returns the result in a new binary object. You can call FD_GetBinaryData on this new object, cast the result to a char*, and treat the result as a normal C string pointer.

Example:

```
FD_Handle   myString = FD_MakeString("Hello");
FD_Handle   asASCII = FD_ASCIIString(myString);
const char* textPtr = (const char*) FD_GetBinaryData(asASCII);
printf("%s, world!\n", textPtr);
```

Error Codes:
kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedString

```
DIL_Error   FD_GetImmediate(FD_Handle, long* type, long* value);
```

Returns the components of an immediate object. The type parameter will receive one of kImmedSpecial, kImmedCharacter, kImmedBoolean, or kImmedReserved. The value parameter will receive the integral value associated with that type of immediate object.

Example:

```
long  type;
long  value;
FD_GetImmediate(kFD_NIL, &type, &value);
      // type == kImmedSpecial
      // value == 0
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedImmediate

```
long  FD_GetMagicPointer(FD_Handle);
```

Returns the value associated with the given magic pointer object.

Example:

```
FD_Handle   myMagicPtr = FD_MakeMagicPointer(157);
long  value = FD_GetMagicPointer(myMagicPtr);   // value == 157
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedMagicPointer

```
DIL_Error   FD_ReadFromLargeBinary(FD_Handle, long offset, void* buffer,
                                                              long count);
```

Reads “count” bytes starting from the given object, starting “offset” bytes from within the object’s contents. The bytes are written to the memory pointed to by “buffer”.

Example:

```
FD_Handle   myLB = FD_MakeLargeBinary(500, NULL,
                                                    kFD_LZCompression);

char  buffer[100];
FD_ReadFromLargeBinary(myLB, 200, buffer, sizeof(buffer));
      // Reads the middle 100 bytes from the large binary object
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedLargeBinary
kFD_ExpectedNonNegativeValue
kFD_NULLPointer
kFD_CouldNotDecompressData
kFD_ErrorReadingFromStore

```
DIL_Error   FD_WriteToLargeBinary(FD_Handle, long offset, const void* buffer,
                                                              long count);
```

Writes “count” bytes of data into the given object, starting at “offset” bytes into the object’s contents. The bytes are read from the memory pointed to by “buffer”.

Example:

```
static const char kMyCRCTable[] = {... };
FD_Handle   myCRCTable = FD_MakeLargeBinary(sizeof(kMyCRCTable),
                                          NULL, kFD_LZCompression);
FD_WriteToLargeBinary(myCRCTable, 0, kMyCRCTable,
                                          sizeof(kMyCRCTable));
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedLargeBinary
kFD_ExpectedNonNegativeValue
kFD_NULLPointer
kFD_CouldNotCompressData
kFD_ErrorWritingToStore

Array Functions

```
DIL_Error   FD_InsertArraySlot(FD_Handle array, long pos, FD_Handle item);
```

Inserts the given object into the array at the specified position. Any objects between that position and the end of the array are moved down in the array to make room.

Example:

```
FD_Handle   hello = FD_MakeString("Hello");
FD_Handle   comma = FD_MakeString(", ");
FD_Handle   world = FD_MakeString("world");
FD_Handle   array = FD_MakeArray(0, NULL);

FD_InsertArraySlot(array, 0, world);
      // array holds ["world"]

FD_InsertArraySlot(array, 0, hello);
      // array holds ["Hello", "world"]

FD_InsertArraySlot(array, 1, comma);
      // array holds ["Hello", ", ", "world"]

FD_InsertArraySlot(array, 9, kFD_NIL);
      // FD_GetError returns kFD_ValueOutOfRange
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedArray
kFD_ValueOutOfRange

```
DIL_Error   FD_AppendArraySlot(FD_Handle array, FD_Handle item);
```

Appends the given element to the end of the array.

Example:

```
FD_Handle   hello = FD_MakeString("Hello");
FD_Handle   comma = FD_MakeString(", ");
FD_Handle   world = FD_MakeString("world");
FD_Handle   array = FD_MakeArray(0, NULL);

FD_AppendArraySlot(array, hello);
      // array holds ["Hello"]

FD_AppendArraySlot(array, comma);
      // array holds ["Hello", ", "]

FD_AppendArraySlot(array, world);
      // array holds ["Hello", ", ", "world"]
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory

kFD_ExpectedArray
kFD_ValueOutOfRange

```
FD_Handle   FD_RemoveArraySlot(FD_Handle array, long pos);
```

Removes the object at the given position in the array. Any objects between that position and the end of the array are moved forward in the array to fill in the vacated slot. The removed object is returned to the caller so that the caller can, for example, dispose of the object.

Example:

```
FD_Handle   array = ...;
      // array holds ["Hello", ", ", "world"]

FD_Handle   item = FD_RemoveArraySlot(array, 1);
      // array holds ["Hello", "world"]
      // item holds ", "
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

```
DIL_Error   FD_RemoveArraySlotCount(FD_Handle array, long pos, long count);
```

Removes “count” slots from the array starting at the given position. Any objects between that position and the end of the array are moved forward in the array to fill in the vacated slots. The objects in the removed slots are not disposed of, so callers should address that before losing their last references to those objects.

Example:

```
FD_Handle   array = ...;
      // array holds ["Hello", ", ", "world"]

FD_RemoveArraySlotCount(array, 0, 2);
      // array holds ["world"]
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

```
FD_Handle   FD_SetArraySlot(FD_Handle array, long pos, FD_Handle item);
```

Sets the array slot at the given position to contain the specified new element. The object being replaced in the array is returned to the caller so that it can, for example, dispose of the object. No other array elements are affected, and the size of the array remains unchanged.

Example:

```
FD_Handle   array = ...;
```

```
            // array holds ["Hello", ", ", "world"]

        FD_Handle   oldHello = FD_SetArraySlot(array, 0,
                                        FD_MakeString("Wilkommen"));
            // array holds ["Wilkommen", ", ", "world"]
            // oldHello holds "Hello"

        FD_Handle   oldWorld = FD_SetArraySlot(array, 0,
                                        FD_MakeString("welt"));
            // array holds ["Wilkommen", ", ", "welt"]
            // hello holds "Hello"
            // oldWorld holds "world"
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

```
FD_Handle   FD_GetArraySlot(FD_Handle array, long pos);
```

Returns the object in the given slot of the array.

Example:

```
        FD_Handle   array = ...;
            // array holds ["Hello", ", ", "world"]

        FD_Handle   oldHello = FD_GetArraySlot(array, 0);
            // array holds ["Hello", ", ", "world"]
            // oldHello holds "Hello"
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

Frame Functions

```
FD_Handle   FD_SetFrameSlot(FD_Handle frame, const char* slotName,
                                                FD_Handle item);
```

Adds a key/value pair to the frame, where the key is specified by “slotName” and the value is specified by “item”. If a pair with the specified key already exists in the frame, its corresponding value object is replaced with “item”, and the old value is returned to the caller.

Example:

```
        FD_Handle   frame = FD_MakeFrame();
            // frame holds {}

        FD_Handle   hello = FD_MakeString("Hello");
```

```
FD_SetFrameSlot(frame, "mySlotName", hello);
      // frame holds {mySlotName: "Hello"}

FD_Handle   world = FD_MakeString("world");

FD_SetFrameSlot(frame, "mySlotName", world);
      // frame holds {mySlotName: "world"}
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedFrame
kFD_NULLPointer
kFD_ValueOutOfRange

```
FD_Handle   FD_GetFrameSlot(FD_Handle frame, const char* slotName);
```

Looks for the key/value pair identified by “slotName”. If the key is found in the frame, its associated value is returned to the caller. If the key cannot be found, kFD_NIL is returned.

Example:

```
FD_Handle   frame = FD_MakeFrame();
      // frame holds {}

FD_Handle   hello = FD_MakeString("Hello");

FD_SetFrameSlot(frame, "mySlotName", hello);
      // frame holds {mySlotName: "Hello"}

FD_Handle   result = FD_GetFrameSlot(frame, "mySlotName");
      // result now refers to the same string object as hello

result = FD_GetFrameSlot(frame, "fred");
      // No slot named fred, so result gets kFD_NIL
```

Error Codes:

kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_NULLPointer

```
int   FD_FrameHasSlot(FD_Handle frame, const char* slotName);
```

Returns whether or not a slot with the given name exists in the frame.

Example:

```
FD_Handle   frame = FD_MakeFrame();

FD_SetFrameSlot(frame, "mySlotName", kFD_NIL);
      // frame holds {mySlotName: NIL}
```

```
FD_Handle   result = FD_GetFrameSlot(frame, "mySlotName");
      // result gets kFD_NIL, because that's what's in the slot

result = FD_GetFrameSlot(frame, "fred");
      // result still gets kFD_NIL, because a slot named fred
      // doesn't exist

int   exists = FD_FrameHasSlot(frame, "mySlotName");
      // exists gets TRUE, because the slot exists

exists = FD_FrameHasSlot(frame, "fred");
      // exists gets FALSE, because the slot doesn't exist
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_NULLPointer

```
FD_Handle   FD_RemoveFrameSlot(FD_Handle frame, const char* slotName);
```

Removes the key/value pair identified by "slotName". If the entry existed in the frame, then the value object is returned to the caller. Otherwise, the function returns kFD_NIL.

Example:

```
FD_Handle   frame = ...;
      // frame holds {mySlotName: "Hello"}

FD_Handle   removed = FD_RemoveFrameSlot(frame, "mySlotName");
      // frame holds {}
      // removed holds "Hello"

removed = FD_RemoveFrameSlot(frame, "fred");
      // removed now holds kFD_NIL
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_NULLPointer

```
FD_Handle   FD_GetIndFrameSlot(FD_Handle frame, long pos);
```

Allows the client to traverse the list of slots in the frame. By calling FD_GetIndFrameSlot with values of "pos" ranging from zero to FD_GetLength(frame) - 1 (inclusive), the client can retrieve the contents of all the slots in the frame.

Note: The order in which the objects are returned is not defined. In particular, you should not expect to retrieve them in the order in which they were inserted.

Example:

```
FD_Handle   frame = ...;
      // frame holds {slot1: "Hello", slot2:", ", slot3:"world"}
```

```
FD_Handle   value1 = FD_GetIndFrameSlot(frame, 0);
      // value1 holds "Hello"

FD_Handle   value2 = FD_GetIndFrameSlot(frame, 1);
      // value2 holds ", "

FD_Handle   value3 = FD_GetIndFrameSlot(frame, 2);
      // value3 holds "world"
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_ValueOutOfRange

```
FD_Handle   FD_GetIndFrameSlotName(FD_Handle frame, long pos);
```

Allows the client to traverse the list of slots in the frame, getting the name for each one. By calling FD_GetIndFrameSlotName with values of “pos” ranging from zero to FD_GetLength(frame) - 1 (inclusive), the client can retrieve the names of all the slots in the frame.

Note: The order in which the slot names are returned is not defined. In particular, you should not expect to retrieve them in the order in which they were inserted.

Example:

```
FD_Handle   frame = ...;
      // frame holds {slot1: "Hello", slot2:", ", slot3:"world"}

FD_Handle   name1 = FD_GetIndFrameSlotName(frame, 0);
      // name1 holds "slot1"

FD_Handle   name2 = FD_GetIndFrameSlotName(frame, 1);
      // name2 holds "slot2"

FD_Handle   name3 = FD_GetIndFrameSlotName(frame, 2);
      // name3 holds "slot3"
```

Error Codes:
kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_ValueOutOfRange

Pointer Object Functions

```
DIL_Error   FD_SetClass(FD_Handle, FD_Handle oClass);
```

Sets the class of given object to the specified class. Only classes for non-symbol pointer objects can be set or changed. In general, classes should be specified as symbol objects.

However, you can also set a class to NIL. Additionally, for compatibility with Frames, a number of undocumented immediate values are supported.

Example:

```
FD_Handle   myBinary = FD_MakeBinary(1024, NULL);
FillWithCRCData(myBinary);
FD_SetClass(myBinary, FD_MakeSymbol("CRC"));
      // Identifies the binary object to the rest of the
      // application as one containing a CRC table.
```

Error Codes:

kFD_FDILNotInitialized
kFD_ExpectedPointerObject
kFD_InvalidClass

```
long  FD_GetLength(FD_Handle);
```

Returns the length of the given object. Only pointer objects have a length. For frames and arrays, the length is the number of elements they contain. For binary objects and large binary objects, the length is the number of bytes in the object.

Example:

```
FD_Handle   myArray = FD_MakeArray(0, NULL);
long        len = FD_GetLength(myArray);  // len == 0

FD_AppendArraySlot(myArray, kFD_NIL);
len = FD_GetLength(myArray);              // len == 1

FD_Handle   myString = FD_MakeString("Hello");
len = FD_GetLength(myString);             // len == 12
```

Error Codes:

kFD_FDILNotInitialized
kFD_ExpectedPointerObject

```
DIL_Error   FD_SetLength(FD_Handle, long newSize);
```

Sets the length of the object. Only non-frame pointer objects can have their lengths changed. For arrays, “newSize” specifies the number of slots that should be in the array. For binaries and large binaries, “newSize” specifies the number of bytes that should be allocated to the object.

If an array is grown as a result of settings its length, additional slots are appended to the end of the array and set to NIL. If the array is reduced, slots are removed from the end of the array. If those slots contained pointer objects, then it is up to the client to make sure that the objects are deleted or otherwise handled before the references to them in the array are lost.

Example:

```
FD_Handle   hello = FD_MakeString("Hello");
```

```
FD_Handle   world = FD_MakeString(", world");

long  size1 = FD_GetLength(hello);
long  size2 = FD_GetLength(world);

FD_SetLength(hello, size1 + size2 - 2);
memcpy((char*) FD_GetBinaryData(hello) + size1 - 2,
      FD_GetBinaryData(world), size2);

// hello ends up with the text "Hello, world"
```

Error Codes:
- kFD_FDILNotInitialized
- kDIL_OutOfMemory
- kFD_ExpectedPointerObject
- kFD_ValueOutOfRange

Any Object Functions

```
FD_Handle   FD_GetClass(FD_Handle);
```

Returns the class of the given object, according to the following table:

Object type	Class
Integer	kFD_SymInteger
Character	kFD_SymChar
Boolean	kFD_SymBoolean
Other immediate	kFD_SymWeird_Immediate
Frame	kFD_SymFrame *
Array	kFD_SymArray *
String	kFD_SymString *
Symbol	kFD_SymSymbol
Binary	kFD_NIL *
Large binary	kFD_NIL *
MagicPointer	kFD_SymMagicPointer

* The classes of these object types can be changed by the FDIL client, so the values shown here are the default classes.

Example:

Error Codes:
- kFD_FDILNotInitialized
- kFD_PointerObjectIsFree

Large Binary Object Functions

Large binary objects are binary objects whose contents may or may not fit entirely in memory. They are a solution to the problem of having a restricted NewtonScript object heap on Newton OS

devices.

Large binary objects relieve demands on the NewtonScript object heap in two ways. First, the memory to hold the object's contents is allocated from a store object, where it can be paged in from SRAM to DRAM on demand by the Newton OS VMM. Second, the object's data can be compressed while it is stored in SRAM, being decompressed when paged to DRAM.

On the desktop, large binary objects are handled as similarly as possible. Automatic paging of data from some backing store to RAM is not possible on all platforms, but behind-the-scenes compression is, as well as the manual paging of data from a backing store to RAM.

Data compression and decompression is handled automatically by the FDIL. The choice of a backing store is up to the FDIL client. The saving and restoring of data to and from a backing store is performed by a set of callback functions associated with the large binary object. The client can choose to use either a predefined set of callback functions provided by the FDIL, or they can roll their own. The FDIL provides a set of callback functions that store the large binary data in RAM (kFD_MemoryStoreProcs) or on disk (kFD_DiskStoreProcs). It also provides a set of functions that simply discard the data received from the external source (kFD_NullStoreProcs). The client can use any of these sets when creating a large binary, or they can define their own.

```
DIL_Error   FD_SetLargeBinaryProcs(const FD_LargeBinaryProcs*);
```

> Determines the set of callback functions used to handle the data of a large binary object. Only objects created subsequent to making the call to FD_SetLargeBinaryProcs (via either a call to FD_MakeLargeBinary or a call to FD_Unflatten) are affected. By default, kFD_MemoryStoreProcs are used.

FD_LargeBinaryProcs is defined as follows:

```
struct FD_LargeBinaryProcs
{
      DIL_Error   (*Create)         (void** cookie);
      DIL_Error   (*SetNumPages)    (void** cookie,
                                     long pageCount);
      DIL_Error   (*ReadPage)       (void** cookie,
                                     long pageNum,
                                     FD_PageBuff*);
      DIL_Error   (*WritePage)      (void** cookie,
                                     long pageNum,
                                     const FD_PageBuff*);
      DIL_Error   (*Destroy)        (void** cookie);
};
typedef struct FD_LargeBinaryProcs FD_LargeBinaryProcs;
```

Each of these functions is options, and can be set to NULL if no meaningful implementation needs to be provided.

Each function is called with a "cookie": a pointer to a "void*". This cookie is for the sole use of the set of callback functions. Typically, the Create function allocates some private storage and stores it in the cookie. The cookie is then passed to SetNumPages, ReadPage, and WritePage, which use it in whatever way is appropriate. Finally, the Destroy function deletes the cookie.

The Create function performs any necessary initialization of data private to the FD_LargeBinaryProcs. If it needs to, it can allocate a block of memory to hold this private data and store a pointer to that memory in fCookie.

The data in a large binary object is stored in an integral number of pages. The number of pages is determined by the length of the binary object. As a large binary object has its size changed, the SetNumPages function will be called to allow the private data structures to be resized as well.

When the FDIL needs to retrieve the partial contents of a large binary (perhaps in response to a FD_ReadFromLargeBinary or a FD_Flatten call), it calls the ReadPage function. ReadPage is given the number of the page to be returned, and the location of a buffer in which to put it. The buffer is described by the FD_PageBuff data structure:

```
#define kPageChunkSize        (1024L)
#define kCompressionExtra     (288L)

typedef struct FD_PageBuff
{
      long  fLength;
      char  fData[kPageChunkSize + kCompressionExtra];      /* Only fLength
                                                              bytes used */
}FD_PageBuff;
```

ReadPage should copy one page of data into the fData array, and store the number of bytes written in the fLength field. The page contents and their length should simply be the same as the ones specified when the page was stored with the WritePage call. If no WritePage call had ever been made for the requested page, ReadPage should return kFD_LBReadingFromUnwrittenPage. If any other error occurs while trying to retrieve the page, it should return kFD_ErrorReadingFromStore. Otherwise, it should return kDIL_NoError.

WritePage is called to store pages. Its function is merely to take the specified page's contents and size and store them away for later. If an error occurs while saving the data, WritePage should return kFD_ErrorWritingToStore. Otherwise, it should return kDIL_NoError. WritePage will never be called with a page number larger than that specified in a previous SetNumPages call.

Before the large binary is deleted, the FDIL will call the Destroy function. Destroy should deallocate any memory allocated during any previous function calls.

Example:

Following is a C++ example of a set of FD_LargeBinaryProcs that store pages using the Macintosh Resource Manager. Of course, you should never really use the Resource Manager as a database; this is an exmple of writing FD_LargeBinaryProcs, not how to properly use the Resource Manager.

```
struct      ResMgrLBData
{
      short       fRefNum;
      long        fNumPages;
      Str255      fFileName;
};
```

```
DIL_Error ResMgrCreate(void** cookie)
{
      ResMgrLBData*     myData = new ResMgrLBData;
      if (myData == NULL)
            return kDIL_OutOfMemory;

      tmpnam((char*) myData->fFileName);
      c2pstr((char*) myData->fFileName);
      short refNum = OpenResFile(myData->fFileName);
      if (refNum < 0)
      {
            delete myData;
            return kFD_ErrorCreatingStore;
      }

      myData->fRefNum = refNum;
      myData->fNumPages = 0;

      *cookie = myData;

      return kDIL_NoError;
}

DIL_Error ResMgrSetNumPages(void** cookie, long pageCount)
{
      ResMgrLBData*     myData = (ResMgrLBData*) *cookie;

      short oldRefNum = CurResFile();
      UseResFile(myData->fRefNum);
      SetResLoad(FALSE);
      for (long ii = pageCount; ii < myData->fNumPages; ++ii)
      {
            Handle      hdl = GetResource('page', ii);
            if (hdl)
                  RemoveResource(hdl);
      }
      SetResLoad(TRUE);
      UseResFile(oldRefNum);

      return kDIL_NoError;
}


DIL_Error ResMgrReadPage(void** cookie, long pageNum,
                         FD_PageBuff* page)
{
      ResMgrLBData*     myData = (ResMgrLBData*) *cookie;

      short       oldRefNum = CurResFile();
      UseResFile(myData->fRefNum);
      Handle      hdl = Get1Resource('page', pageNum);
      UseResFile(oldRefNum);
      if (!hdl)
            // Actually, we should look further; we might be out of memory.
            return kFD_LBReadingFromUnwrittenPage;
```

```
	page->fLength = GetHandleSize(hdl);
	BlockMove(*hdl, page->fData, page->fLength);

	return kDIL_NoError;
}


DIL_Error ResMgrWritePage(void** cookie, long pageNum,
                                                const FD_PageBuff* page)
{
	ResMgrLBData*	myData = (ResMgrLBData*) *cookie;

	short	oldRefNum = CurResFile();
	UseResFile(myData->fRefNum);
	Handle	hdl = Get1Resource('page', pageNum);
	UseResFile(oldRefNum);
	if (hdl)
	{
		SetHandleSize(hdl, page->fLength);
		if (MemError())
			return kDIL_OutOfMemory;
	}
	else
	{
		hdl = NewHandle(page->fLength);
		if (!hdl)
			return kDIL_OutOfMemory;
		UseResFile(myData->fRefNum);
		AddResource(hdl, 'page', pageNum, "\p");
		short error = ResError();
		UseResFile(oldRefNum);
		if (error)
			return kFD_ErrorWritingToStore;
	}

	BlockMove(page->fData, *hdl, page->fLength);

	ChangedResource(hdl);
	if (ResError())
		return kFD_ErrorWritingToStore;

	UpdateResFile(myData->fRefNum);
	if (ResError())
		return kFD_ErrorWritingToStore;

	return kDIL_NoError;
}


DIL_Error ResMgrDestroy(void** cookie)
{
	ResMgrLBData*	myData = (ResMgrLBData*) *cookie;

	CloseResFile(myData->fRefNum);
```

```
	FSDelete(myData->fFileName, 0);

	delete myData;
	*cookie= NULL;

	return kDIL_NoError;
}

const FD_LargeBinaryProcs	gResMgrStoreProcs = {
	ResMgrCreate,
	ResMgrSetNumPages,
	ResMgrReadPage,
	ResMgrWritePage,
	ResMgrDestroy
};
```

Object Comparison

```
int	FD_Equal(FD_Handle, FD_Handle);
```

Returns whether or not two objects are equal to each other. Objects of different types are never equal. Non-pointer objects are equal if their types and associated integral values are equal. Pointer objects are equal only if they refer to the same object.

Example:

```
	FD_Handle	myInt1 = FD_MakeInt(5);
	FD_Handle	myInt2 = FD_MakeInt(5);
	FD_Handle	myInt3 = FD_MakeInt(100);
	FD_Handle	myReal1 = FD_MakeReal(5);
	FD_Handle	myReal2 = FD_MakeReal(5);

	int	result1 = FD_Equal(myInt1, myInt2);	// returns TRUE
	int	result2 = FD_Equal(myInt1, myInt3);	// returns FALSE
	int	result3 = FD_Equal(myReal1, myReal2);	// returns FALSE
	int	result4 = FD_Equal(myInt1, myReal1);	// returns FALSE
```

Error Codes:

kFD_FDILNotInitialized

Object Duplication

```
FD_Handle	FD_Clone(FD_Handle);
```

Creates a copy of the given object, returning the newly created object. If the object is an aggregate object (i.e., an array or a frame), only the top-level object is cloned. None of the child objects are cloned.

Example:

```
	FD_Handle	myInt1 = FD_MakeInt(5);
	FD_Handle	myInt2 = FD_Clone(myInt1);
```

```
        // myInt2 now contains an integer object containing the
        // number 5.

        FD_Handle   myString = FD_MakeString("Hello");
        FD_Handle   myArray = FD_MakeArray(0, NULL);

        FD_AppendArraySlot(myArray, myString);

        FD_Handle   myArray2 = FD_Clone(myArray);

        FD_DeepDispose(myArray2);

        // myString is now disposed because a reference to it was in
        // myArray2 when it was deleted.
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_PointerObjectIsFree

```
FD_Handle   FD_DeepClone(FD_Handle);
```

Creates a copy of the given object, returning the newly created object. If the object is an aggregate object (i.e., an array or a frame), all child objects are cloned as well.

Example:

```
        FD_Handle   myInt1 = FD_MakeInt(5);
        FD_Handle   myInt2 = FD_DeepClone(myInt1);

        // myInt2 now contains an integer object containing the
        // number 5.

        FD_Handle   myString = FD_MakeString("Hello");
        FD_Handle   myArray = FD_MakeArray(0, NULL);

        FD_AppendArraySlot(myArray, myString);

        FD_Handle   myArray2 = FD_DeepClone(myArray);

        FD_DeepDispose(myArray2);

        // myString is still OK. It was a duplicate of myString that
        // was deleted when myArray2 when was deleted.
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_PointerObjectIsFree

Object Streaming

Objects (including object hierarchies) can be converted into a flat stream of bytes. This process is called “flattening”. Flattening is useful for converting your object data into something that can be written to disk, or sent over a serial line to a Newton device.

The opposite of flattening is “unflattening”. Unflattening is the process of creating an object or object hierarchy from the data found in a flattened object stream. Unflattening would be used to when reading object data from disk or receiving objects from a Newton device.

The client is responsible for providing low-level callback functions that ultimately read and write the streamed bytes. The callback functions have the following C prototypes:

```
typedef DIL_Error (*DIL_WriteProc)(const void* buf, long amt, void* userData);
typedef DIL_Error (*DIL_ReadProc)(void* buf, long amt, void* userData);
```

The client’s DIL_WriteProc is called when the FDIL needs to write some bytes. The bytes can be found at the memory location indicated by “buf”, and the number of bytes is passed in “amt”. The value in “userData” is whatever the user specified when calling FD_Flatten. If an error occurs, the DIL_WriteProc can return an error code as the function’s result, or it can return kDIL_NoError if no error occurred. The DIL_WriteProc can return a generic kDIL_ErrorWritingToPipe error code, or it can return any other non-kDIL_NoError value. Whatever it returns will be reported to the calling client via FD_GetError.

The client’s DIL_ReadProc is called when the FDIL needs bytes from the stream when resurrecting objects. DIL_ReadProc should provide “amt” bytes, storing them at the memory location indicated by “buf”. Again, the value in “userData” is whatever the user specified when calling FD_Unflatten. If an error occurs, the DIL_ReadProc can return an error code, or it can return kDIL_NoError if no error occurred. The DIL_ReadProc can return a generic kDIL_ErrorReadingFromPipe error code, or it can return any other non-kDIL_NoError value. Whatever it returns will be reported to the calling client via FD_GetError.

```
DIL_Error   FD_Flatten(FD_Handle, DIL_WriteProc, void* userData);
```

Converts the given object into a flat stream of bytes suitable for saving to disk or for transmission to a Newton OS device. FD_Flatten just performs the conversion of objects into bytes; the actual disposition of the bytes is determined by the DIL_WriteProc provided by the caller.

The userData parameter is passed to the DIL_WriteProc when it is called. It usually contains a pointer to data that the DIL_WriteProc needs to complete its job. For instance, it can contain a FILE* if the DIL_WriteProc writes data to disk, or a CD_Handle if the DIL_WriteProc sends data to a Newton OS device.

Example:

```
DIL_Error   WriteCallback(const void* buf, long amt,
                                                    void* userData)
{
      FILE*       file = (FILE*) userData;
      size_t      itemsWritten = fwrite(buf, 1, amt, file);
      if (itemsWritten != amt)
            return kDIL_ErrorWritingToPipe;
```

```
        return kDIL_NoError;
    }

    DIL_Error   WriteObjectToDisk(FILE* file, FD_Handle myObject)
    {
        FD_Flatten(myObject, WriteCallback, file);
        return FD_GetError();
    }
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kDIL_ErrorWritingToPipe
kFD_ErrorReadingFromStore

```
FD_Handle   FD_Unflatten(DIL_ReadProc, void* userData);
```

Creates an object from the bytestream data provided by the caller's DIL_ReadProc. FD_Unflatten doesn't care where the bytes come from. It is only responsible for using them to recreate the original objects from which they were formed.

The userData parameter is passed to the DIL_ReadProc when it is called. It usually contains a pointer to data that the DIL_ReadProc needs to complete its job. For instance, it can contain a FILE* if the DIL_ReadProc reads data from disk, or a CD_Handle if the DIL_ReadProc receives data from a Newton OS device.

Example:

```
    DIL_Error   ReadCallback(void* buf, long amt, void* userData)
    {
        FILE*       file = (FILE*) userData;
        size_t      itemsRead = fread(buf, 1, amt, file);
        if (itemsRead != amt)
            return kDIL_ErrorReadingFromPipe;
        return kDIL_NoError;
    }

    DIL_Error   ReadObjectFromDisk(FILE* file, FD_Handle* myObject)
    {
        *myObject = FD_Unflatten(ReadCallback, file);
        return FD_GetError();
    }
```

Error Codes:

kFD_FDILNotInitialized
kDIL_OutOfMemory
kDIL_ErrorReadingFromPipe
kFD_UnknownStreamVersion
kFD_StreamCorrupted
kFD_UnsupportedCompression
kFD_UnsupportedStoreVersion
kFD_ErrorCreatingStore

kFD_ErrorWritingToStore

Object Printing

There may be times when you would like a textual representation of your NewtonScript data. You may like this for debugging purposes (“Just what’s in this frame anyway? Why can’t I retrieve the slot I’m looking for?”) or to display to a (particularly sophisticated) user. Converting the FD_Handle into a text form is facilitated through the FD_PrintObject function:

```
DIL_Error   FD_PrintObject(FD_Handle obj, const char* EOLString,
DIL_WriteProc, void* userData);
```

FD_PrintObject converts an object into some user-readable text form. For instance, a frame will be turned into text looking something like the following:

```
{
   name:    "Fred",
   age:     15,
   phone:   [
               {
                  'home,
                  "555-1212"
               },
               {
                  'work,
                  "555-2121"
               }
            ]
   ...
}
```

As the text is generated, the DIL_WriteProc is called so that the application can deal with it as it deems fit. For instance, the app could buffer all the text into a Handle or CString object, or it could just turn around and call `printf("%s", str)` with it.

Because different platforms and different development environments use different end-of-line conventions, FD_PrintObject accepts a string containing the EOL sequence you want to use. Thus, it could be "\n" in MPW, "\r" in CodeWarrior, or "\n\r" in VC++.

Unicode Conversion

The FDIL handles text as Unicode characters. If the client application works directly with Unicode, then it can create strings with FD_MakeWideString and retrieve text with FD_GetWideString. If the client application works only with ASCII characters, it can create strings with FD_MakeString and retrieve text with FD_GetString or FD_ASCIIString.

For times when those functions aren’t flexible enough, FD_Handle provides the following Unicode conversion functions.

```
DIL_Error   FD_ConvertFromWideChar(char* dest, const DIL_WideChar* src,
                                                              long numChars);
```

Converts the characters in the buffer specified by “src” from Unicode to ASCII, storing the resulting characters in the buffer specified by “dest”. Only “numChars” characters are converted and transferred. No regard is given for NULL terminators.

Unicode characters which have no corresponding character in the destination character set are converted to 0x1A.

FD_ConvertFromWideChar is written in such a way that “dest” and “src” can refer to the start of the same buffer.

Example:

```
FD_Handle   text = FD_GetFrameSlot(frame, "text");

char  buffer[1024];
FD_ConvertFromWideChar(buffer, (const DIL_WideChar*)
                           FD_GetBinaryData(text), 1024);
```

Error Codes:

kFD_FDILNotInitialized
kFD_NULLPointer
kFD_ExpectedNonNegativeValue

```
DIL_Error   FD_ConvertToWideChar(DIL_WideChar* dest, const char* src,
                                                              long numChars);
```

Converts the characters in the buffer specified by “src” from ASCII to Unicode, storing the resulting characters in the buffer specified by “dest”. Only “numChars” characters are converted and transferred. No regard is given to NULL terminators.

FD_ConvertToWideChar is written in such a way that “dest” and “src” can refer to the start of the same buffer.

Example:

```
char  buffer[1024];
FillBuffer(buffer);

FD_Handle   text = FD_MakeBinary(1024 * sizeof(DIL_WideChar),
                                          NULL);
FD_ConvertToWideChar((DIL_WideChar*) FD_GetBinaryData(text),
                           buffer, 1024);
```

Error Codes:

kFD_FDILNotInitialized
kFD_NULLPointer
kFD_ExpectedNonNegativeValue

```
DIL_Error   FD_SetWideCharEncoding(long encoding);
```

Appropriate conversion between Unicode and an 8-bit character set requires the specification of the host character set. By default, the Macintosh version of the FDIL converts using the Macintosh character set, and the Windows version of the FDIL converts using the Windows character set. Currently, these are the only two character sets supported.

FD_SetWideCharEncoding changes the specified 8-bit character set. The “encoding” parameter can be one of kMacEncoding, kWindowsEncoding, or kDefaultEncoding (which is equal to kMacEncoding on Macintosh platforms, and kWindowsEncoding on Windows platforms).

Example:

```
char  buffer[100] = "åß∂ƒ©Δ¬";      // Extended Mac characters
size_t      numChars = strlen(buffer);

FD_SetWideCharEncoding(kFD_MacEncoding);
FD_ConvertToWideChar((DIL_WideChar*) buffer, buffer, numChars);

// buffer now contains the characters in Unicode format.

FD_SetWideCharEncoding(kFD_WindowsEncoding);
FD_ConvertFromWideChar(buffer, (DIL_WideChar*) buffer, numChars);

// buffer now contains the characters in Windows 8-bit format
// (assuming those characters are defined in the Windows
//  character set).
```

Error Codes:

kFD_FDILNotInitialized
kFD_ValueOutOfRange

Debugging Features

To aid in development, the FDIL comes in two versions: debug and non-debug. The non-debug version of the library is the one that should be included in your shipping application. The debug version can be used during application development to help identify bugs and their source. When using the debug version of the library, it is important that you #define DIL_ForDebug before including the FDIL.h header file.

NOTE: The definition of FD_Handle changes between the debug and non-debug versions of the FDIL library. To help ensure that your application links with the correct library, some macro-magic is performed in the FDIL.h header file. If you link with the incorrect library, your application will break into the debugger when it calls FD_Startup.

The debug version of the FDIL contains a debugging feature to aid in the tracking of memory leaks. Through the power of the C/C++ preprocessor, all calls to FDIL functions that result in new objects being created are instead routed to a parallel set of functions. This parallel set of functions take as additional parameters the file and line within the file from which the call is being made. This

information is recorded by the FDIL in the created object. At the end of your application, when all memory should be cleaned up and allocated objects deleted, you can call an FDIL function to print out all leaked objects, along with the locations in your program that allocated those objects.

The set of FDIL functions that have a corresponding memory-tracking version is:

```
FD_MakeReal
FD_MakeString
FD_MakeWideString
FD_MakeSymbol
FD_MakeArray
FD_MakeFrame
FD_MakeBinary
FD_Clone
FD_DeepClone
FD_Unflatten
```

```
long  FD_CheckForMemoryLeaks(const char* EOLString, DIL_WriteProc,
                             void* userData);
```

Reports any undeleted, user-allocated objects, along with the file and line within the file containing the function call that allocated that object. Returns as its result the number of user-allocated objects left undeleted.

Because different platforms and different development environments use different end-of-line conventions, FD_CheckForMemoryLeaks accepts a string containing the EOL sequence you want to use. Thus, it could be "\n" in MPW, "\r" in CodeWarrior, or "\n\r" in VC++.

Example:

```
FD_Startup();           // Line 20 of MyApp.c

FD_MakeFrame();         // Line 22 of MyApp.c

FD_CheckForMemoryLeaks("\n", MyPrintFn, myPrintData);
                        // Prints a message saying that a
                        // frame was allocated at line 22
                        // of MyApp.c.

FD_Shutdown();          // Deletes the memory anyway
```

Error Codes:

kFD_FDILNotInitialized

Error Codes

```
kDIL_NoError                        (0)
kDIL_ErrorBase                      (-98000)


/* --- General DIL errors --- */
```

```
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)


/* --- Base error numbers --- */

kFD_ErrorBase                       (kDIL_ErrorBase - 400)

/* --- FDIL error numbers --- */

     /* Hard errors -- you should always be looking for these. */

kFD_UnknownStreamVersion            (kFD_ErrorBase - 1)
kFD_StreamCorrupted                 (kFD_ErrorBase - 2)
kFD_UnsupportedCompression          (kFD_ErrorBase - 3)
kFD_CouldNotCompressData            (kFD_ErrorBase - 4)
kFD_CouldNotDecompressData          (kFD_ErrorBase - 5)
kFD_UnsupportedStoreVersion         (kFD_ErrorBase - 6)
kFD_ErrorCreatingStore              (kFD_ErrorBase - 7)
kFD_ErrorWritingToStore             (kFD_ErrorBase - 8)
kFD_ErrorReadingFromStore           (kFD_ErrorBase - 9)

    /* Soft errors -- you get these only if you feed in bad data. */

kFD_FDILNotInitialized              (kFD_ErrorBase - 19)

kFD_ExpectedInteger                 (kFD_ErrorBase - 20)
kFD_ExpectedPointerObject           (kFD_ErrorBase - 21)
kFD_ExpectedImmediate               (kFD_ErrorBase - 22)
kFD_ExpectedMagicPointer            (kFD_ErrorBase - 23)

kFD_ExpectedArray                   (kFD_ErrorBase - 24)
kFD_ExpectedFrame                   (kFD_ErrorBase - 25)
kFD_ExpectedBinary                  (kFD_ErrorBase - 26)
kFD_ExpectedLargeBinary             (kFD_ErrorBase - 27)

kFD_ExpectedReal                    (kFD_ErrorBase - 28)
kFD_ExpectedString                  (kFD_ErrorBase - 29)
kFD_ExpectedSymbol                  (kFD_ErrorBase - 30)
kFD_ExpectedChar                    (kFD_ErrorBase - 31)

kFD_NULLPointer                     (kFD_ErrorBase - 40)
kFD_ExpectedPositiveValue           (kFD_ErrorBase - 41)
kFD_ExpectedNonNegativeValue        (kFD_ErrorBase - 42)
kFD_ValueOutOfRange                 (kFD_ErrorBase - 43)
kFD_SymbolTooLong                   (kFD_ErrorBase - 44)
kFD_IllegalCharInSymbol             (kFD_ErrorBase - 45)
kFD_InvalidClass                    (kFD_ErrorBase - 46)
kFD_PointerObjectIsFree             (kFD_ErrorBase - 47)

    /* Internal errors -- you should never see these. */
```

```
kFD_LBReadingFromUnwrittenPage      (kFD_ErrorBase - 50)
```

FDIL Internals

This section describes the internal data structures used by the FDIL. It is provided for debugging purposes only. Do not write shipping code that relies on this information.

In non-debug builds, FD_Handle is defined as a long. In debug builds, it is defined as follows:

```
struct FD_ObjectHeader;
typedef struct FD_Handle
{
      long  ref;
      struct FD_ObjectHeader**      entry;
}FD_Handle;
```

In this structure, “ref” is the same long as FD_Handle is in non-debug mode. In the following discussions, this long will be referred to simply as “ref”, “the ref”, or “the object ref”.

In either version of the library, the lowest two bits of the ref contain flags determining the kind of object it represents, according to the following table:

```
00 = integer
01 = pointer object
10 = immediate
11 = magic pointer
```

If the object is an integer, then the upper 30 bits of the ref contain the integer’s value.

If the object is a pointer object, then the upper 30 bits contain an index into an object table. Each entry in the object able contains a pointer to the actual object located in the application heap.

If the object is an immediate, then the next two low-order bits determine its sub-type:

```
0010 = special immediate
0110 = character immediate
1010 = boolean immediate
1110 = reserved immediate
```

In all cases, the upper 28 bits contain an integer value associated with the immediate. For example, character immediates store the 16-bit Unicode character in the upper 28-bit field.

If the object is a magic pointer, then the magic pointer value is stored in the upper 30 bits of the ref.

As mentioned earlier, pointer objects contain an index into an object table. Each entry in the table is either a pointer to an object on the heap, or some fairly illegal value if the table entry is currently unused. This value is 0xCDCDCDCD on Windows platforms, and 0x50FF8001 on Macintosh platforms. As objects are created, empty entries in the table are used up. When all entries in the table are full, the table is automatically grown.

If the object table entry is valid, it points to an object on the heap. Every object is prefixed with the following data structure:

```
struct FD_ObjectHeader
{
      long        flags;
      long        size;       // size of user portion; does not
                              // include this header

#ifdef FD_TrackMemory
      const char* file;
      int         line;
#endif

      union
      {
            FD_Handle   oClass;
            FD_Handle   map;
      }u;

      // followed by object data: either bytes or an array of FD_Handle
};
```

The 2 lowest bits in the “flags” field are used to determine the type of pointer object, according to the following table:

```
00 = raw binary object
01 = array
10 = large binary object
11 = frame
```

The “size” field contains the sized1in bytesd1of the data that follows the standard object header. For binary objects, this is the number of user bytes in the object. For arrays and frames, this is the number of elements in the object times sizeof(FD_Handle). For large binary objects, this is sizeof(FD_LargeBinaryData).

When an object is allocated, it is tagged with the file and line within the file of the statement that caused the object to be allocated. That information is stored in the next two fields: “file” and “line”.

If the object is anything but a frame, the next field contains the class of that object. If the object is a frame, then the next field contains a reference to the frame’s map object. More on map objects later.

Following the standard header are the contents of the object, a variable length block of data as indicated by the “size” field. If the object is a large binary, the contents contain an FD_LargeBinaryData structure. If the object is a regular binary, the contents are whatever’s appropriate for that binary (for example, a string of Unicode characters, or 8 bytes of floating point number information). If the object is an array, the contents are formatted into an array of FD_Objects.

If the object is a frame, the contents are also formatted into an array of FD_Objects. This array contains the value objects in a key/value pair. The key objects are then stored in a “map”, which is

a parallel array stored in the “map” field in the FD_ObjectHeader. Any time a key/value pair is added to a frame, corresponding additions are made to both arrays. Similarly, any time a key/value pair is removed from a frame, corresponding deletions are made to both arrays.

Because frames don’t have any location reserved for storing the frames class, class information is stored as just another slot in the frame. The name of the slot is “class”, the the slot should contain a symbol identifying the class.

Because the map is a normal array, it contains all the attributes that a normal array would. This includes a field in its FD_ObjectHeader reserved for holding a class object. However, maps use this field as a series of bitflags instead. The bits are defined as follows:

```
all zero = plain map
1 = map is sorted
2 = map is shared
4 = map has _proto slot
```

None of these bit flags are currently supported in the FDIL; all maps created by the FDIL contain zero in this field.

Byteswapping Issues

The FDIL is a library that runs on big-endian (a lá Macintosh and Newton) and little-endian (a lá Intel and Apple][) architectures. Because the objects it creates and manages interoperate with Newton devices, there are some byteswapping issues clients on little-endian machines should watch out for.

The FDIL is pretty good about keeping most byteswapping issues hidden internally. For instance, when streaming out an integer object to the Newton, the FDIL byteswaps it before sending it. When creating larger objects for which endianness is an issue (such as Unicode strings an floating point objects), the FDIL performs the appropriate byteswapping at creation time. This means that clients calling FD_GetBinaryData on such objects will have to take into account the fact that the data is stored in big-endian format.

About the only time this might be an issue is when calling FD_ConvertToUnicode and FD_ConvertFromUnicode. These two functions are essentially exported versions used internally by FD_MakeString, FD_GetString, and FD_ASCIIString. As such, they expect to work on Unicode characters stored in big-endian format. Since that’s the way data is normally stored in objects, everything should be OK. But clients trying to interpret the data in little-endian format are doomed to failure. For instance, a client calling FD_GetBinaryData on an object containing Unicode characters and passing the resulting pointer to WideCharToMultiByte will recieve unintended results.