CHAPTER 3

FDIL Interface

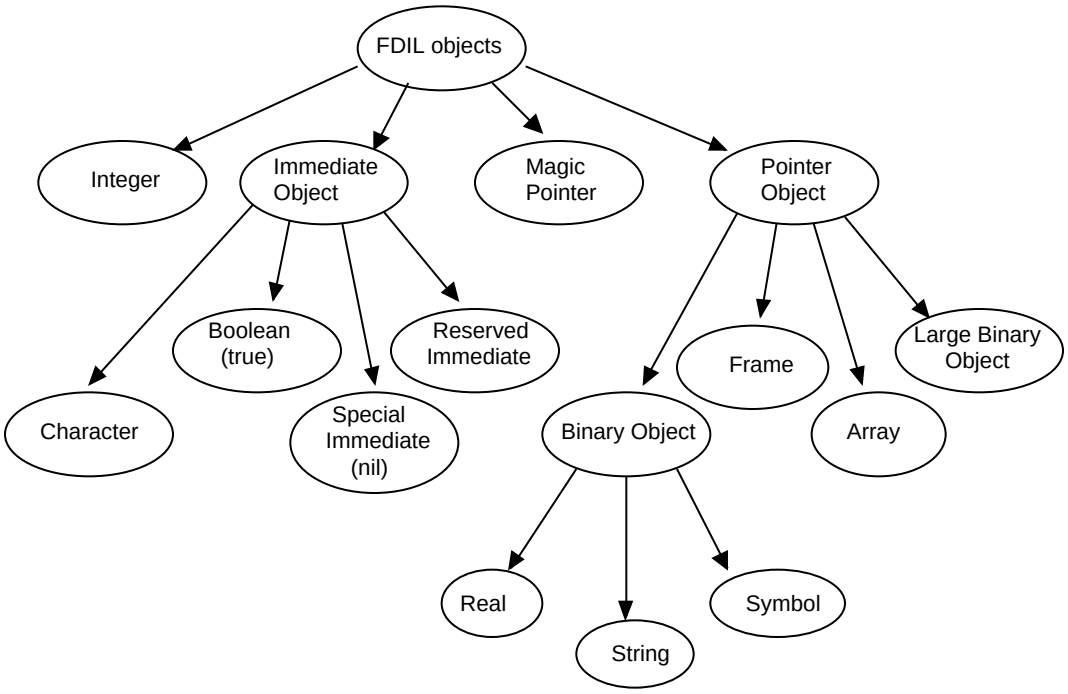
The Frames Desktop Integration Library (FDIL) is a small library allowing the creation and manipulation of NewtonScript objects on the Windows and Macintosh platforms through a C language API. Because the objects the FDIL manipulates on are NewtonScript-compatible, they can be exchanged with Newton devices using communications libraries such as the CDIL.

The FDIL can create any type of NewtonScript object, including virtual binary objects, and frames and arrays with circular references. The FDIL, unlike NewtonScript, does not provide automatic garbage collection.

About the FDIL Objects

The objects the FDIL manipulate mimic the NewtonScript objects. There is a one-to-one correspondence between NewtonScript and FDIL objects. There are a number of minor implementation details that differ, however. The object hierarchy is diagramed in Figure 3-1.

Figure 3-1 The FDIL object hierarchy



Each object is represented by an `FD_Handle`. An `FD_Handle` is a lightweight object; in non-debug builds an `FD_Handle` is a `long`. Two bits of this value contain type information, describing the four top-level types of FDIL objects:

- integers
- immediates
- pointer objects
- magic pointers

Immediate, integer, and magic pointer objects are stored entirely within the `FD_Handle`. Pointer objects consists of a chunk of data in addition to the `FD_Handle`; the `FD_Handle` of a pointer object contains a reference to this data.

The FDIL functions that create pointer objects allocate this additional memory for you, but unlike NewtonScript, you are responsible for freeing this memory. Magic pointer objects contain a reference to an object in the Newton ROM. Only applications providing a Newton development environment should need to create magic pointer objects.

Integers Objects

Integer objects are just that: objects containing integral values within their `FD_Handle`. The integers are stored in a 30-bit field, allowing them a range of -536,870,912...536,870,911. Integers are created with the `FD_MakeInt` function. The value stored in an integer object can be retrieved with the `FD_GetInt` function. The `FD_IsInt` function determines if an FDIL object is an integer object.

Listing 3-1 Using integer objects

```
FD_Handle  myInt   = FD_MakeInt(5);
long       result  = FD_GetInt(myInt);    // result == 5
int        isInt   = FD_IsInt(myInt);     // isInt != 0
```

Note

Since an `FD_Handle` is a `long` in the non-debug version of the library, confusing an FDIL integer with the value it contains does not trigger a compiler error. You can catch these errors by compiling your program with the debug library. In the debug library an `FD_Handle` is a `struct`, and confusing an FDIL integer with its value triggers a compile-time type error. ◆

Immediate Objects

Immediate objects contain their values within their `FD_Handle`. There are four types of immediate objects

- characters
- special immediates

- Booleans
- reserved immediates

Other than characters, the only immediate objects that you need are the true object and the nil object, which are specified by the kFD_True and kFD_NIL constants. While you should not need to use them, the following functions can be used to create, extract data from, and test immediate objects: FD_MakeImmediate, FD_IsImmediate, and FD_GetImmediate. There are a separate set of functions that manipulate character objects, and functions to test if an object is the nil object.

Characters

Character objects are immediate objects which contain a 16 bit Unicode character. Since there is no standard amongst C/C++ development environments on how to accommodate 16-bit characters, the DIL library defines the DIL_WideChar type to enforce a standard. A DIL_WideChar is an unsigned 16-bit value. There are functions to create character objects from both ASCII (8-bit) characters and from DIL_WideChar characters: FD_MakeChar and FD_MakeWideChar. And similarly two functions to retrieve 8 and 16 bit characters from an FDIL character object: FD_GetChar and FD_GetWideChar. You can test if an object is a character object with FD_IsChar.

Listing 3-2 Using character objects

```
FD_Handle       myChar = FD_MakeChar('a');
FD_Handle       myOtherChar = FD_MakeWideChar(L'a');

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

int             isChar = FD_IsChar(myChar); // isChar != 0
                isChar = FD_IsChar(kFD_True); // isChar == 0
```

Unicode-ASCII Translation

The FDIL performs automatic translation from Unicode to Macintosh and Windows character sets, and the other way around. The FDIL uses the Macintosh character set when performing character translations on a

Macintosh, and the Windows character set when on a Windows machine. You can however set this programmatically with the FD_SetWideCharEncoding.

Every character in the Macintosh and Windows character sets has a Unicode equivalent, however the inverse is not true. Unicode characters that do not exist in the Macintosh or Windows character set, are mapped to 0x1A.

There are a number of functions that perform Unicode to ASCII conversion at the string level, see “Strings” (page 3-8).

Booleans

In theory there are two Boolean objects: the true object, and the false object. In practice, only the true object is used; the nil object is used to signify falsity. The true object can be specified with the constant kFD_True. You can test if an object is a Boolean with the FD_IsBoolean function.

Special Immediates

There is only one special immediate object that you encounter, the nil object. This object, which you can refer to with the constant kFD_NIL, is used to signify the lack of information or Boolean falsehood. There are two functions which test if an object is the nil object: FD_IsNIL and FD_NotNIL.

Reserved Immediates

Reserved immediates are only used internally by the FDIL. You should never see such an object.

Pointer Objects

A pointer object is an object whose FD_Handle contains a reference to the data comprising the object. The pointer objects consist of the aggregate types: arrays and frames, and two types of raw binary objects: simple binary objects and large binary objects.

Binaries

A binary object consist of a series of raw bytes. You may store any data you wish in a binary object. The object may also contain a class symbol identifying the data. There are three types of binary objects for with there is special support: reals, strings, and symbols. There are special functions for creating these objects, and accessing their data.

You can create an empty binary object with the `FD_MakeBinary` function. This function allocates a specified number of bytes, which you are responsible for disposing when the binary object is no longer needed. `FD_MakeBinary` returns an `FD_Handle`. To access the block of data that has been allocated for this binary object, use the `FD_GetBinaryData` function. `FD_GetBinaryData` returns a pointer to that block of data, a `void *`. You need to cast this pointer to the appropriate type before using it. The `FD_IsBinary` function tests if an FDIL object is a binary object.

You can change the size of a binary object with the `FD_SetLength` function. However, any pointers to a binary object's contents are invalidated by calling `FD_SetLength`, since the data might have been moved.

Binary objects are limited to a size of 16 MB.

Listing 3-3 Using a binary object

```
// this is some data we want to send to a Newton device
static const char  kMyCRCTable[] = {... };

// create a binary object to hold the data
FD_Handle  myCRCTable = FD_MakeBinary(
                                  sizeof(kMyCRCTable),
                                  "CRCTable"); //the obj.'s class

// get a pointer to the data
char*  dest = (char*) FD_GetBinaryData(myCRCTable);

// copy over the bytes
memcpy(dest, kMyCRCTable, sizeof(kMyCRCTable));

//send the data..

//... and later reclaim the memory
```

```
FD_Dispose(myCRCTable);
```

Reals

A real is a binary object that contains a double precision floating point number. It is an 8-byte binary object containing an IEEE-754 floating point value.

IMPORTANT

When using the FDIL library, it is important that you set any applicable compiler options for generating IEEE-754 floating point compatible code. ◆

You can create a real from a `double` with the `FD_MakeReal` function, and retrieve a `double` with the `FD_GetReal` function. The `FD_IsReal` function tests if an FDIL object is a real. The following example demonstrates how real objects are manipulated.

Listing 3-4 Using real number objects

```
FD_Handle  myReal = FD_MakeReal(5);
double     result = FD_GetReal(myReal); // result == 5.0
int        isReal = FD_IsReal(myReal); // isReal != 0
           isReal = FD_IsReal(kFD_NIL); // isReal == 0
FD_Dispose (myReal); // remember that reals are pointer objects
```

Symbols

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

Symbols are a pooled resource. When a symbol is created, it stored in an internal table. If a new symbol is subsequently created with the same string, a reference to the first symbol is returned; therefore only one version of the symbol exists. Note that this comparison of strings is case-insensitive. A symbol is not disposed of when passed to `FD_Dispose`. It is not removed from

that internal table, because other references to this symbol may exist elsewhere in your program.

Symbols are created with the FD_MakeSymbol function, you can test if an object is a symbol with FD_IsSymbol, and get a pointer to the string in the internal table with FD_GetSymbol. The string accessed through an FD_GetSymbol call must be treated as read-only.

Listing 3-5 Using symbol objects

```
// these two calls create new symbols internally
FD_Handle  mySymbol1 = FD_MakeSymbol("mySlotName1");
FD_Handle  mySymbol2 = FD_MakeSymbol("mySlotName2");

// these two calls return refernces to the already-existant symbols
FD_Handle  mySymbol3 = FD_MakeSymbol("mySlotName1");
FD_Handle  mySymbol4 = FD_MakeSymbol("MySlOtNaMe2");

int        result = FD_IsSymbol(mySymbol4); // result != 0
           result = FD_IsSymbol(kFD_NIL); // result == 0

const char*    symbolText = FD_GetSymbol(mySymbol4);
// Note that symbolText is set to "mySlotName2" not "MySlOtNaMe2"

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

//this does nothing
FD_Dispose (mySymbol1);
```

Strings

An FDIL string object is a binary object consisting of a NULL-terminated series of Unicode characters. There are functions for creating string objects from both a NULL-terminated array of ASCII characters (a C string), and from a NULL-terminated array of DIL_WideChars: FD_MakeString and FD_MakeWideString. And similarly, there are two functions to retrieve the contents of a string object, copying the characters to an 8 or 16 bit character buffer: FD_GetString and FD_GetWideString.

There is another function, FD_ASCIIString, to simply convert a string's data to 8-bit strings. After passing a string object to FD_ASCIIString, 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.

You may receive a rich string from a Newton device. A rich string is a string with imbedded ink data. You cannot create a rich string, nor interpret the data in the ink portion of a rich string. When translating rich strings, a 0xF700 or 0x1A character is inserted in the place of the embedded ink, depending on whether you are extracting 16-bit or 8-bit characters.

You can test if an object is a string with FD_IsString and if an object is a rich string with FD_IsRichString.

There are two further functions that convert character arrays to and from Unicode: FD_ConvertFromWideChar and FD_ConvertToWideChar. For more information about Unicode to ASCII translation, see “Unicode-ASCII Translation” (page 3-4).

Listing 3-6 Using string objects

```
// Create two string objects. These two objects are basically
// equivalent. The first is more convinient, and the second
// allows for a wider range of input.
FD_Handle  myString1 = FD_MakeString("Some text");
FD_Handle  myString2 = FD_MakeWideString(L"Some wide text");

//test the identity of some objects
int     result = FD_IsString(myString1);      // result != 0
        result = FD_IsString(myString2);      // result != 0
        result = FD_IsString(kFD_NIL);        // result == 0

FD_Handle      myString3 = MyGetStringFromNewt();
if (FD_IsRichString(myString3))
    MyShowAlert("Warning: string can't be completely translated. Some
information may be lost");

// get C strings from FDIL string objects in 2 ways:
// this first way copies over exactly the number of chars. requested
// to a separate buffer
FD_Handle  myString4 = FD_MakeString("Hello");

char    buffer[10];
FD_GetString(myString4, buffer, 10);  // buffer == "Hello\0"
FD_GetString(myString4, buffer, 3);   // buffer == "Hel" (no NULL
```

```
                                        //                terminator!)

// this second way converts from 16 to 8 bit in place
FD_Handle      myString5 = FD_MakeString("Hello");
FD_Handle      asASCII = FD_ASCIIString(myString5);
const char*    textPtr = (const char*) FD_GetBinaryData(asASCII);
printf("%s, world!\n", textPtr);

FD_Dispose(myString1);
FD_Dispose(myString2);
FD_Dispose(myString3);
FD_Dispose(myString4);
FD_Dispose(myString5);
FD_Dispose(asASCII);
```

Large Binaries

A large binary object mimics the functionality of a virtual binary object (VBO). It contains a large amount of unformatted binary data, that is paged in from a backing store, and optionally compressed. Each large binary object has an associated set of storage procedures that actually page the data in and out from the backing store. The FDIL provides functions to store the large binary object in main memory and on disk, and a set of functions that simply discard the data. You may write your own functions to store the data; see “Creating Your Own Large Binary Storage Procedures” (page 3-11).

You set which functions are used with `FD_SetLargeBinaryProcs`. By default, the main memory storage procedures are used. When you call `FD_SetLargeBinaryProcs`, all large binary objects created from that point on use the storage procedures you specify. This includes large binaries created implicitly by reading from a byte stream using `FD_Unflatten`.

The `FD_MakeLargeBinary` function creates a large binary object. You write to this object with `FD_WriteToLargeBinary`, and read from one with `FD_ReadFromLargeBinary`. These functions read or write a block of data to a buffer from a specified offset. The `FD_IsLargeBinary` function tests if an object is a large binary.

Listing 3-7 Using a large binary object

```
// a large table
```

```
static const char  kMyCRCTable[] = {... };

//store this, and all future binary objects on disk
FD_SetLargeBinaryProcs (&kFD_DiskStoreProcs);

//make a large binary object
FD_Handle  myFDILCRCTable = FD_MakeLargeBinary( sizeof(kMyCRCTable),
                            "theObjsClass", kFD_LZCompression);

//copy the whole table into the large binary object
FD_WriteToLargeBinary( myFDILCRCTable, 0, kMyCRCTable,
                       sizeof(kMyCRCTable));

// ... read the bytes 200 through 300
char buffer[100];
FD_ReadFromLargeBinary(myFDILCRCTable, 200, buffer, 100);

FD_Dispose(myCRCTable);
```

Creating Your Own Large Binary Storage Procedures

The FDIL provides a set of procedures that you can use to store large binary objects in main memory (`kFD_MemoryStoreProcs`) and on disk (`kFD_DiskStoreProcs`), and a set of functions that discard the data (`kFD_NullStoreProcs`). You may also create your own. A set of large binary procedures is a structure of the following format:

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

The `Create` function is called when a large binary object is being created, in response to a call to `FD_MakeLargeBianry` or `FD_Unflatten`. The `Destroy` function is called when the object is no longer needed.

The *cookie* argument is intended to allow your various functions to communicate. It is a pointer to a `void *`. Typically, your `Create` function allocates some memory for use by all your storage functions, and sets the `void *` that cookie points to this memory block. This memory is then usually freed by your `Destroy` function.

A large binary object is stored in an integral number of pages. If the number of pages changes due to the object growing or shrinking, `SetNumPages` is called with the new number of pages to allow you to modify your own data structures.

`ReadPage` and `WritePage` are called to copy over a page of data to and from a buffer. These functions are passed in the page number, as well as a pointer to `FD_PageBuff` object that describes the buffer to copy data to and from. An `FD_PageBuff` object has the following form:

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

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

Your `ReadPage` function should copy over the required page to the buffer located at *pageBufPtr* `->fData` and set the *pageBufPtr* `->fLength` field to the number of bytes copied over. The contents of the page, and its length, should be the same as those specified in the call to your `WritePage` function when it stored this page. 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 a non-zero value not equal to `kFD_LBReadingFromUnwrittenPage`. Otherwise, it should return `kDIL_NoError`. `ReadPage` is never called with a page number larger than, or equal to, that specified in a previous call to `SetNumPages`.

Conversely, your `WritePage` function is passed the page number to write and pointer to an `FD_PageBuff` object describing the buffered data to store. It should store *pageBufPtr* `->fLength` bytes starting at *pageBufPtr* `->fData`. If an error occurs while saving the data, `WritePage` should return a non-zero value. Otherwise, it should return `kDIL_NoError`. `WritePage` is never called with a

page number larger than, or equal to, that specified in a previous call to SetNumPages.

Remember to call FD_SetLargeBinaryProcs if you want to use these procedures to store large binaries streamed in with FD_Unflatten, as well as in those large binaries you create with FD_MakeLargeBinary.

You may use NULL for fields in your FD_LargeBinaryProcs, if there is no need to implement that particular function.

Listing 3-8 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 example of writing FD_LargeBinaryProcs, not how to properly use the Resource Manager.

Listing 3-8 Custom large binary storage procedures

```
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);
    for (long ii = pageCount; ii < myData->fNumPages; ++ii)
    {
        SetResLoad(FALSE);
        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 ResError();
    }

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

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

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

    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
};
```

Arrays

An array object is a variable-size object whose contents are divided into a series of other objects. Each division is called a “slot”. Each slot consists of an FDIL object, that is, an FD_Handle. Objects can be inserted into an array or appended to the end of an array.

You create an array with FD_MakeArray. This call sets the array’s initial length and class. The array’s slots are initialized to kFD_NIL. You can access the value of a particular slot with FD_GetArraySlot and FD_SetArraySlot. New slots are added to the end of an array with FD_AppendArraySlot. You can also insert a slot in an arbitrary position with FD_InsertArraySlot; any objects between that position and the end of the array are moved down one spot in order to make room. A single object or a range of objects can be removed from an array with FD_RemoveArraySlot and FD_RemoveArraySlotCount; the remaining objects moving up in the array to take their place. The length of the array can be directly manipulated with FD_SetLength; this function adds slots at the end of an array initialized to kFD_NIL, or removes slots from the end of the array.

The FD_IsArray function tests if an object is an array. You can dispose of an array with FD_Dispose, as you would any other FDIL object. You can also call FD_DeepDispose to recursively dispose of all the objects in the array.

Array objects are limited to a size of 16 MB / sizeof (FD_Handle).

Listing 3-9 Using array objects

```
FD_Handle  myArray1 = FD_MakeArray(10, "myArraysClass");
FD_Handle  myArray2 = FD_MakeArray(0, NULL); //Zero's OK,
                                             //NULL for default class

FD_Handle hello = FD_MakeString("Hello");
FD_Handle comma = FD_MakeString(", ");
FD_Handle world = FD_MakeString("world");
FD_Handle period = FD_MakeString(".");

FD_InsertArraySlot(myArray2, 0, world);
// myArray2 holds ["world"]

FD_InsertArraySlot(myArray2, 0, hello); // moves "world" over
// myArray2 holds ["Hello", "world"]

FD_InsertArraySlot(myArray2, 1, comma);
// myArray2 holds ["Hello", ", ", "world"]

FD_AppendArraySlot(myArray2, period);
// myArray2 holds ["Hello", ", ", "world", "."]

FD_InsertArraySlot(myArray2, 9, kFD_NIL); /* FD_GetError returns
                                             kFD_ValueOutOfRange */

FD_Handle theComma = FD_RemoveArraySlot(myArray2, 1);
FD_Dispose(theComma);
// myArray2 holds ["Hello", "world", "."]

//get an array element
FD_Handle theHello = FD_GetArraySlot(myArray2, 0);

// print every element
for (long i = 0; i < FD_GetLength(myArray2); i++)
{
    printf("%s ", (char *) FD_GetBinaryData( FD_ASCIIString(
                  FD_GetArraySlot(myArray2, i)) ) );
};

//get rid of the array and everything in it
FD_DeepDispose(myArray1);
FD_DeepDispose(myArray2);
```

Frames

A frame is an aggregate object where each element, called a “slot,” contains any FDIL object, and is indexed by name. The slot name itself is a symbol. Rather than using an integer index to retrieve a value that’s been added to a frame (as you would with an array), you specify the slot name to get the slot value.

FD_MakeFrame creates a new, empty frame, and FD_IsFrame tests if an object is a frame. A slot is added with FD_SetFrameSlot; if the slot already exists the value of that slot changed. You access slots with FD_GetFrameSlot; if the slot does not exist, kFD_NIL is returned. You can remove slots with FD_RemoveFrameSlot. You can test if a frame has a particular slot with FD_FrameHasSlot.

You can iterate through a frame’s slots with the FD_GetIndFrameSlot and FD_GetIndFrameSlotName functions. These functions allow you to access a slots value and name, respectively, by an integer index.

Frame objects are limited to a size of 16 MB / sizeof (FD_Handle).

Listing 3-10 Using frame objects

```
/* Create a frame with the following format:
{   name: { first: "Bob", last: "Anderson"},
    address : "51 Partlow Road",
    address2 : "Fine, NY 13639",
    phones : ["555-1234","555-4321"],
} */

FD_Handle myFrame = FD_MakeFrame();

FD_Handle nameFrame = FD_MakeFrame();
FD_SetFrameSlot(nameFrame, "first", FD_MakeString("Bob"));
FD_SetFrameSlot(nameFrame, "last", FD_MakeString("Anderson"));
FD_SetFrameSlot(myFrame, "name", nameFrame);

FD_SetFrameSlot(myFrame, "address", FD_MakeString("51 Partlow Road"));
FD_SetFrameSlot(myFrame, "address2", FD_MakeString("Fine, NY 13639"));

FD_Handle phones = FD_MakeArray(0,NULL);
FD_AppendArraySlot(phones, FD_MakeString("555-1234"));
FD_AppendArraySlot(phones, FD_MakeString("555-4321"));
```

```
FD_SetFrameSlot(myFrame, "phones", phones);

// get the last name
FD_Handle lastName = FD_GetFrameSlot( FD_GetFrameSlot ( myFrame,
                                                        "name"),
                                      "last");

// get rid of the first phone number
// Note that here we get a reference to the phones slot via the frame.
// We could have done the same by using our local variable “phones”.
// They refer to the same object.
FD_Handle firstPhone = FD_RemoveArraySlot( FD_GetFrameSlot( myFrame,
                                                            "phones"),
                                           0);
FD_Dispose (firstPhone);

// iterate over all frame slots
FD_Handle slotName, slotValue;
for (int i = 0; i < FD_GetLength(myFrame); i++)
{
    slotName = FD_GetIndFrameSlotName (myFrame, i);
    slotValue = FD_GetIndFrameSlot (myFrame, i);
};

//get rid of the frame and all imbedded objects
FD_DeepDispose (myFrame);
```

Magic Pointer Objects

A magic pointer object contains a pointer to objects in a Newton devices ROM. You should only need to create magic pointer objects if you are writing a Newton development environment. The only likely way to run into a magic pointer object in your code is reading an NTK stream file with the `FD_Unflatten` function. You should never see a magic pointer object from data sent from a Newton device, through a CDIL pipe. Magic pointers are resolved before being sent from a Newton device.

Magic pointer objects are created with `FD_MakeMagicPointer`. You can access this value with `FD_GetMagicPointer`. And you can test if an object is a magic pointer with `FD_IsMagicPointer`.

Using the FDIL

Initializing the Library

Before calling any FDIL function, you should initialize the library by calling FD_Startup. When you are done using the library, call FD_Shutdown; this function deallocates all memory used by the FDIL. Usually you just call FD_Startup once, but you can call it multiple times as long as an equal number of calls to FD_Shutdown are made.

Object Comparison

The FD_Equal function compares two FDIL objects. Objects of different types are never equal. Note that this is unlike NewtonScript, where the integer 3 and the real 3.0 are considered equal. All pointer objects: binaries, arrays, frames, and large binaries, are equal only if they refer to the same object.

Object Duplication

The FD_Clone and FD_DeepClone create duplicates of an FDIL object. If the object is an aggregate object, that is an array or frame, FD_Clone only copies the top level objects. FD_DeepClone also makes copies of any nested objects, recursively.

Object Printing

The FD_PrintObject function prints formatted FDIL objects. FD_PrintObject actually just converts the object into formatted text. You must supply a function to actually print the formatted text.

Error Handling

All functions set an internal error code indicating the success of that operation. A few functions also return that error code directly. You can access the internal error code value with the FD_GetError function. You should call FD_GetError after every FDIL function code that does not return an error code. The functions listed in “FDIL Reference” (page 3-29) list the possible error codes that each particular function might create.

Object Streaming

The FD_Flatten function converts any FDIL object, including aggregate objects such as frames and arrays, to a flat stream of bytes in Newton Stream Object Format (NSOF). FD_Flatten then calls a callback function you provide to actually write the data. You could, for instance, send the data to a Newton device over a CDIL pipe with the CD_Write function, or store it to disk. The FD_UnFlatten function conversely converts from an NSOF byte stream to an FDIL object, calling a callback function you provide to get the NSOF byte stream. For a description of NSOF, see Chapter 4, “Newton Streamed Object Format,” in *Newton Formats*.

Writing an FDIL Object to a Newton Device or to Disk

The FD_Flatten function is passed an FDIL object, a callback function to actually deal with the byte stream, and an extra parameter it passes on to the callback function. The FD_Flatten function converts your FDIL object to a byte stream, which your callback then either stores, or sends.

Listing 3-11 shows two calls to FD_Flatten, each with a corresponding write callback function. One writes the byte stream to disk, and the other sends the byte stream to a Newton device through a CDIL pipe.

Listing 3-11 Call to FD_Flatten and callback functions to write a streamed FDIL object to disk and to CDIL pipe.

```
// Write to file
DIL_Error err = FD_Flatten(myFDILObject,WriteToDiskCallback,myFilePtr);
DIL_Error WriteToDiskCallback(const void* buf,long amt,void* userData)
```

```
{
    FILE*   fp = (FILE*) userData;
    size_t  itemsWritten = fwrite(buf, 1, amt, fp);
    if (itemsWritten != amt)
        return kDIL_ErrorWritingToPipe;
    return kDIL_NoError;
}


// Write to Newton device through a pipe
DIL_Error err = FD_Flatten(myFDILObject,WriteToPipeCallback,myPipePtr);
DIL_Error WriteToPipeCallback(const void* buf,long amt,void* userData)
{
    CD_Handle* pipePtr = (CD_Handle *) userData;
    return CD_Write(*pipePtr, buf, amt);
}
```

Reading FDIL Objects from a Newton Device or from Disk

The `FD_Unflatten` function takes a read callback function and an extra argument that it passes to this callback function and returns an FDIL object. The read callback function is responsible for copying over a specified number of bytes of an NSOF byte-stream to a buffer. `FD_Unflatten` converts the contents of that buffer to an FDIL object.

Listing 3-12 shows two calls to `FD_Unflatten`. Each with a corresponding call back function. One set of calls reads the byte stream from a disk, the other reads the byte stream from a Newton device through a CDIL pipe.

Listing 3-12 Call to `FD_Unflatten` and two callback functions to read a streamed FDIL object both from disk and from CDIL pipe.

```
FD_Handle  obj;
DIL_Error  err;

// Read an object from a disk file
obj = FD_Unflatten(ReadFromDiskCallback, myFilePtr);
err = FD_GetError();
DIL_Error ReadFromDiskCallback(void* buf, long amt, void* userData)
{
    FILE*   fp = (FILE*) userData;
    size_t  itemsRead = fread(buf, 1, amt, fp);
    if (itemsRead != amt)
```

```
        return kDIL_ErrorReadingFromPipe;
    return kDIL_NoError;
}

// Read an object from a Newton device through a pipe
obj = FD_Unflatten(ReadFromPipeCallback, myPipePtr);
err = FD_GetError();
DIL_Error ReadFromPipeCallback(void* buf, long amt, void* userData)
{
    CD_Handle* pipePtr = (CD_Handle *) userData;
    return CD_Read(*pipePtr, buf, amt);
}
```

Object Classes

All objects have a class. An object's class is primarily for your use as a programmer in giving a meaning to your data. The class of integer, immediate, and magic pointer objects is immutable. Pointer objects have default classes, but you can change them with the `FD_SetClass` function.

Table 3-1 Default object 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`
Magic pointer	`kFD_SymMagicPointer`

The FD_IsSubClass function determines if an object's class is a subclass of a given class. This function uses the same algorithm used in the NewtonScript language, namely:

- Every class is a subclass of the empty class "".
- Every class is a subclass of itself.
- A class x is a subclass of y, if y is a prefix of x at a period (.) boundary. For example, "foo.bar" is a subclass of "foo".
- For compatibility with the version of NewtonScript found on Newton 1.x OS devices, the following classes are considered subclasses of "string":

  "address"
  "company"
  "name"
  "title"
  "phone"

  Furthermore the following classes are considered subclasses of "phone":

  "homePhone"
  "workPhone"
  "faxPhone"
  "otherPhone"
  "carPhone"
  "beeperPhone"
  "mobilePhone"
  "homeFaxPhone"

Memory Management

You are responsible for calling the FD_Dispose function to free any memory allocated to a pointer object, when that object is no longer needed. This memory can be allocated in one of three ways: the FD_MakeXXX functions that create pointer objects, the FD_Clone and FD_DeepClone functions, or from a byte stream via FD_Unflatten.

You have to be careful not to lose the last reference to a pointer object. A reference to a pointer object can exist either as variable or within an array or frame. When you set the value of an array or frame slot, you might be losing the last reference to the object that previously occupied that slot. The

FD_RemoveArraySlot, FD_SetArraySlot, FD_SetFrameSlot, and FD_RemoveFrameSlot return the object being replaced or removed, for you to dispose of. The FD_RemoveArraySlotCount function, however, cannot return all the objects removed, since it potentially removes multiple objects.

The FD_DeepDispose function recursively deallocates the memory in arrays and frames, and their component objects.

You can use the FD_IsFree function to determine if an object has been disposed of. It returns non-zero if passed an object that is a pointer object whose memory has been freed. However, it is possible that the memory returned to the system by calling FD_Dispose on a pointer object is later reused. If this occurs, calling FD_IsFree inaccurately returns zero, indicating that the memory has not been freed. For this reason you should not call FD_IsFree in a shipping version of your application.

The FD_AllocatedMemory function returns the number of bytes used by the FDIL. You can use this function to track the memory consumption of a particular object, as demonstrated in Listing 3-13, or of the FDIL component in general.

Listing 3-13 Checking memory consumption of a particular object

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

The Internal Representation of an FDIL Object

An FDIL object is represented as a long value called a ref. The lowest two bits determine the object's basic type, as follows:

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

If the ref is an integer, the value is contained in the upper 30 bits. If the object is an immediate, the next two low order bits represent the object's type:

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

In an immediate object, the upper 28 bits contain the object's value. For example, these upper 28 bits hold the 16-bit Unicode character in a character object.

A pointer's upper 30 bits contain an index into an internal object table. In the debug version of the library, an FDIL object is a struct containing both the ref and a pointer to the heap object; see "The Debug Version of the FDIL" (page 3-26).

A magic pointer object's upper 30 bits contain the magic pointer value.

The Debug Version of the FDIL

In the debug version of the library, all functions that create a pointer object also note the file and line number where the object was created in an internal table, and contain a pointer to the actual heap object.

The Debug Version of FDIL Objects

In the normal version of the library, an FDIL object is a `long`. In the debug version an FDIL object is a `struct` of the following format:

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

The `ref` field is the same `long` as in the non-debug `FD_Handle`. For a description of this `ref` object, see "The Internal Representation of an FDIL Object" (page 3-25). If the object is a pointer object, the `entry` field points, indirectly, to an `FD_ObjectHeader struct`. This `struct` has the following format:

```
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 `flags` field contains a bit field describing the object. The lowest 2 bits of this field specifies the object's type, as follows:

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

The `size` field specifies the object's size. 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 created, the file name and line number of the function call that created it are stored in the `file` and `line` fields. The `FD_CheckForMemoryLeaks` function uses these fields to report to you where all currently existing objects were allocated; see "Finding Memory Leaks" (page 3-28).

The next field is the object's class, `oClass`, if the object is anything but a frame, or the frame map, `map`, if the object is a frame. If the object is a frame it's class is stored in a slot named `"class"` containing a symbol. A frame's map is simply an array of symbols containing the slot names used in the frame. There is one difference between a frame map and a regular array. A frame map contains the value zero in its `oClass` field.

Following the `oClass` or `map` field, is the data in the pointer object. For a binary object, this is the actual raw binary data. For an array or frame, this is an array of `FD_Handle`s for the constituent objects. For a large binary object, this is an `FD_LargeBinaryData struct`. The format of a `FD_LargeBinaryData` is not described here.

Finding Memory Leaks

In the debug version, all functions that create a pointer object also note the file and line number where the object was created in an internal table. You can then call the FD_CheckForMemoryLeaks function at a point where all memory should have been freed, such as when your program exists, before calling FD_Shutdown. FD_CheckForMemoryLeaks reports the file name and line number, of the function call that created any unfreed pointer objects.

The functions that can cause the creation of a pointer object are:

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

Listing 3-14 Checking for memory leaks

```
FD_Startup();       // Line 21 of MyApp.c
FD_MakeFrame();     // Line 22 of MyApp.c

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

FDIL Reference

Type Definitions

FD_Handle

An FDIL object. In non-debug builds, an `FD_Handle` is a `long`. In debug builds this is a larger object, containing information about where the object was created; see “The Debug Version of the FDIL” (page 3-26).

DIL_Error

A `long` integer containing an error code, as listed in “Error codes” (page 3-34).

DIL_WideChar

A 2-byte object suitable for holding a Unicode character; see “Characters” (page 3-4).

FD_LargeBinaryProcs

A set of large binary procedures; it is a structure of the following format:

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

For a description of these functions see “Creating Your Own Large Binary Storage Procedures” (page 3-11).

DIL_WriteProc

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

A function called to write data.

buf	A pointer to the data to be written.
amt	How many bytes to write. Note that the PDIL calls your DIL_WriteProc with a value of -1 for this parameter, to signal that no more data is to be sent, and you should flush the buffer.
userData	A pointer to data you provided to the function that calls your writing procedure. 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 device, or NULL if no extra data is needed.
return value	An error code.

DISCUSSION

Your DIL_WriteProc is called when the FDIL needs to write some bytes. Return kDIL_NoError if no error occurred. Otherwise return a kDIL_ErrorWritingToPipe or kFD_ErrorWritingToStore error code, or any other non-kDIL_NoError value. Whatever your DIL_WriteProc returns is reported to the calling client via FD_GetError.

This interface is used by FD_Flatten to write bytes from flattening an object, by a few debugging functions to report information to you, and by the PDIL.

SPECIAL CONSIDERATIONS

If you write the object to a file, you must open the file in binary mode. Note that fopen defaults to text mode.

DIL_ReadProc

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

A function called to read data.

buf	A pointer to the buffer for data that you have read.
amt	How many bytes to read.
userData	A pointer to data you provided to the function that calls your reading procedure. For instance, it can contain a FILE* if the DIL_ReadProc reads data from disk, or a CD_Handle if the DIL_ReadProc gets data from a Newton device, or NULL if no extra data is needed.
return value	An error code.

DISCUSSION

Your DIL_WriteProc is called when the FDIL needs to read some bytes. Return kDIL_NoError if no error occurred. Otherwise return a kDIL_ErrorReadingFromPipe or kFD_ErrorReadingFromStore error code, or any other non-kDIL_NoError value. Whatever your DIL_ReadProc returns is reported to the calling client via FD_GetError.

This interface is used by FD_Unlatten to read bytes of a flattened object, and by the PDIL.

SPECIAL CONSIDERATIONS

If you read the object from a file, you must open the file in binary mode. Note that fopen defaults to text mode.

DIL_StatusProc

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

A function called to retrieve the number of bytes available to be read.

bytesAvailable	Store the number of bytes available here.
userData	A pointer to data you provided to the function that calls your reading procedure. For instance, it can contain a `FILE*` if the `DIL_ReadProc` reads data from disk, or a `CD_Handle` if the `DIL_ReadProc` gets data from a Newton device, or `NULL` if no extra data is needed.
return value	An error code.

DISCUSSION

This interface is only used by the PDIL.

Constants

FDIL objects

constant	meaning
kFD_NIL	The nil object; see “Special Immediates” (page 3-5)
kFD_True	The true object; see “Booleans” (page 3-5).

Large Binary Storage Procedures

constant	meaning
kFD_MemoryStoreProcs	Store data in RAM.
kFD_DiskStoreProcs	Store data on disk.
kFD_NullStoreProcs	Discards data.

Large Binary Compression Options

constant	meaning
kFD_NoCompression	Don’t compress data.
kFD_LZCompression	Use LZ compression. This is the only type of compression you should use when calling FD_MakeLargeBinary.
kFD_ZippyCompression	Use Zippy compression. You should never use this value.

Immediate Types

constant	meaning
kImmedSpecial	A special immediate.
kImmedCharacter	A character.
kImmedBoolean	A Boolean.
kImmedReserved	A reserved immediate.

Error codes

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

kFD_ErrorBase                       (kDIL_ErrorBase - 400)

/* 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)
```

Functions

Integer Object Functions

FD_MakeInt

```
FD_Handle FD_MakeInt(long val)
```

Creates an integer object.

val	An integer between -536,870,912...536,870,911, inclusive.
return value	An integer FDIL object.

ERROR CODES

```
kFD_FDILNotInitialized
kFD_ValueOutOfRange
```

SEE ALSO

For an example call to this function, see Listing 3-1 (page 3-3).

FD_IsInt

```
int FD_IsInt(FD_Handle obj)
```

Determines whether or not an FDIL object is an interger object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-1 (page 3-3).

ERROR CODES

kFD_FDILNotInitialized

FD_GetInt

```
long FD_GetInt(FD_Handle obj)
```

Returns the `long` value stored in the object.

obj	An FDIL integer object.
return value	A long.

SEE ALSO

For an example call to this function, see Listing 3-1 (page 3-3).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedInteger

Immediate Object Functions

FD_MakeImmediate

FD_Handle FD_MakeImmediate(long *type*, long *value*)

Creates the specified type of immediate object.

type	One of the following constants: kImmedSpecial, kImmedCharacter, kImmedBoolean, or kImmedReserved.
value	The value of the immediate object.
return value	An immediate FDIL object.

DISCUSSION

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 functions), 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 FD_MakeImmediate does not perform ASCII to Unicode conversion when creating a character object. That higher-level operation is performed only by FD_MakeChar.

ERROR CODES

kFD_FDILNotInitialized
kFD_ValueOutOfRange

FD_IsImmediate

int FD_IsImmediate(FD_Handle *obj*)

Determines whether or not an FDIL object is an immediate object.

obj	The object to test.
return value	Zero or non-zero.

SPECIAL CONSIDERATIONS

In NewtonScript the term “immediate” includes integers. Therefore, the NewtonScript function `IsImmediate` differs from `FD_IsImmediate`.

ERROR CODES

`kFD_FDILNotInitialized`

FD_GetImmediate

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

Returns the components of an immediate object.

obj	An FDIL immediate object.
type	A pointer to where the type should be stored. This value will be set to `kImmedSpecial`, `kImmedCharacter`, `kImmedBoolean`, or `kImmedReserved`.
value	A pointer to where the value should be stored. If this value is `NULL`, the immediate value is simply not returned, no error is signaled.
return value	An error code.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_ExpectedImmediate`

Character Object Functions

FD_MakeChar

`FD_Handle FD_MakeChar(char` *val*`)`

Creates a character object.

val	An ASCII character.
return value	A character FDIL object.

SEE ALSO

For an example call to this function, see Listing 3-2 (page 3-4).

ERROR CODES

kFD_FDILNotInitialized

FD_MakeWideChar

FD_Handle FD_MakeWideChar(DIL_WideChar *val*)

Creates a character object.

val	A Unicode character.
return value	A character FDIL object.

SEE ALSO

For an example call to this function; see Listing 3-2 (page 3-4).

ERROR CODES

kFD_FDILNotInitialized

FD_IsChar

int FD_IsChar(FD_Handle *obj*)

Determines whether or not an FDIL object is a character object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-2 (page 3-4).

ERROR CODES

kFD_FDILNotInitialized

FD_GetChar

```
char FD_GetChar(FD_Handle obj)
```

Returns the character value stored in the object.

obj	An FDIL character object.
return value	An ASCII character.

SEE ALSO

For an example call to this function, see Listing 3-2 (page 3-4).

ERROR CODES

```
kFD_FDILNotInitialized
kFD_ExpectedChar
```

FD_GetWideChar

```
DIL_WideChar FD_GetWideChar(FD_Handle obj)
```

Returns the character value stored in the object.

obj	An FDIL character object.
return value	A Unicode character.

SEE ALSO

For an example call to this function, see Listing 3-2 (page 3-4).

ERROR CODES

```
kFD_FDILNotInitialized
kFD_ExpectedChar
```

FD_ConvertFromWideChar

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*.

dest	A buffer for the converted ASCII characters.
src	An array of DIL_WideChar objects to translate.
numChars	How many characters to convert.
return value	An error code.

DISCUSSION

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.

SPECIAL CONSIDERATIONS

The characters in *src* are considered to be in big-endian format.

ERROR CODES

kFD_FDILNotInitialized
kFD_NULLPointer
kFD_ExpectedNonNegativeValue

FD_ConvertToWideChar

`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*.

dest	An buffer for the converted `DIL_WideChar` string.
src	An array of ASCII characters to copy.
numChars	How many characters to convert.
return value	An error code.

DISCUSSION

Only *numChars* characters are converted and transferred. No regard is given to NULL terminators.

`FD_ConvertToWideChar` is written such that *dest* and *src* can refer to the start of the same buffer.

SPECIAL CONSIDERATIONS

The characters in *dest* are in big-endian format.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_NULLPointer`
`kFD_ExpectedNonNegativeValue`

FD_SetWideCharEncoding

`DIL_Error FD_SetWideCharEncoding(long encoding)`

Changes the character set to use when converting Unicode and 8-bit characters.

encoding	One of following constants: `kFD_MacEncoding`, `kFD_WindowsEncoding`, or `kFD_DefaultEncoding` (which is

	equal to `kFD_MacEncoding` on Macintosh platforms, and `kFD_WindowsEncoding` on Windows platforms).
return value	An error code.

DISCUSSION

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.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_ValueOutOfRange`

Boolean Object Functions

FD_IsBoolean

`int FD_IsBoolean(FD_Handle` *obj*`)`

Determines whether or not an FDIL object is a Boolean object.

obj	The object to test.
return value	Zero or non-zero.

ERROR CODES

`kFD_FDILNotInitialized`

Nil Object Functions

FD_IsNIL

`int FD_IsNIL(FD_Handle obj)`

Determines whether the given object is the `nil` object.

obj	An FDIL object.
return value	Zero or non-zero.

DISCUSSION

This function is the inverse of `FD_NotNIL`.

ERROR CODES

`kFD_FDILNotInitialized`

FD_NotNIL

`int FD_NotNIL(FD_Handle obj)`

Determines whether the given object is anything but the `nil` object.

obj	An FDIL object.
return value	Zero or non-zero.

DISCUSSION

This function is the inverse of `FD_IsNIL`.

ERROR CODES

`kFD_FDILNotInitialized`

Pointer Object Functions

FD_IsPointerObject

`int FD_IsPointerObject(FD_Handle obj)`

Determines whether or not an FDIL object is a pointer object.

obj	The object to test.
return value	Zero or non-zero.

ERROR CODES

`kFD_FDILNotInitialized`

FD_GetLength

`long FD_GetLength(FD_Handle obj)`

Returns the length of the given object.

obj	An FDIL pointer object.
return value	The length of the object.

DISCUSSION

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.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_ExpectedPointerObject`

FD_SetLength

DIL_Error FD_SetLength(FD_Handle *obj*, long *newSize*)

Sets the length of the object.

obj	An FDIL pointer object.
newSize	The size to set the object's length to.
return value	An error code.

DISCUSSION

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.

SPECIAL CONSIDERATIONS

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 kFD_NIL. If the array is reduced, slots are removed from the end of the array. If those slots contained pointer objects, it is up to you to make sure that the objects are deleted or otherwise handled before the references to them in the array are lost.

All pointers to data within a binary object obtained with FD_GetBinaryData are invalidated if the object's size is changed.

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedPointerObject
kFD_ValueOutOfRange

Binary Object Functions

FD_MakeBinary

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

Creates a raw, unformatted binary object of the given size.

size	The length to make the binary object.
cls	Either NULL in which case the binary object is given a default class, or a string that is passed to FD_MakeSymbol and becomes the object's class.
return value	A binary FDIL object.

DISCUSSION

The contents of the binary object can be accessed with FD_GetBinaryData.

SEE ALSO

For an example call to this function, see Listing 3-3 (page 3-6).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ValueOutOfRange

FD_IsBinary

int FD_IsBinary(FD_Handle *obj*)

Determines whether or not an FDIL object is a binary object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-3 (page 3-6).

ERROR CODES

kFD_FDILNotInitialized

FD_GetBinaryData

void* FD_GetBinaryData(FD_Handle *obj*)

Returns a pointer to the raw binary data stored in the binary object.

obj	An FDIL binary object.
return value	A void* to where the data is stored.

DISCUSSION

FD_GetBinaryData cannot be used to get a pointer to the contents of a large binary object. Instead, use FD_ReadFromLargeBinary and FD_WriteToLargeBinary to access and modify a large binary's contents.

SPECIAL CONSIDERATIONS

Any pointers obtained with FD_GetBinaryData are invalidated by calling FD_SetLength on that binary object.

SEE ALSO

For an example call to this function, see Listing 3-3 (page 3-6).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedBinary

Real Object Functions

FD_MakeReal

```
FD_Handle FD_MakeReal(double val)
```

Creates a real number object from the given value.

val	Any valid IEEE-754 floating point value.
return value	A real FDIL object.

DISCUSSION

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.

SEE ALSO

For an example call to this function, see Listing 3-4 (page 3-7).

ERROR CODES

```
kFD_FDILNotInitialized
kDIL_OutOfMemory
```

FD_IsReal

```
int FD_IsReal(FD_Handle obj)
```

Determines whether or not an FDIL object is a real number object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-4 (page 3-7).

ERROR CODES

kFD_FDILNotInitialized

FD_GetReal

```
double FD_GetReal(FD_Handle obj)
```

Returns the double value stored in the object.

obj	An FDIL real number object.
return value	A double.

SEE ALSO

For an example call to this function, see Listing 3-4 (page 3-7).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedReal

Symbol Object Functions

FD_MakeSymbol

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

Returns a symbol object, creating one if necessary.

str	A NULL-terminated series of less than 254 ASCII characters with values between 32-127, excluding the vertical bar (‘|’) and backslash (‘\’) characters.
return value	A symbol FDIL object.

DISCUSSION

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; where “the same text” implies a case-insensitive comparison.

SEE ALSO

For an example call to this function, see Listing 3-5 (page 3-8).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_NULLPointer
kFD_SymbolTooLong
kFD_IllegalCharInSymbol

FD_IsSymbol

```
int FD_IsSymbol(FD_Handle obj)
```

Determines whether or not an FDIL object is a symbol object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-5 (page 3-8).

ERROR CODES

kFD_FDILNotInitialized

FD_GetSymbol

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

Returns a pointer to the NULL-terminated string of characters of the symbol object.

obj	An FDIL symbol object.
return value	A pointer to the string that is the name of the symbol.

DISCUSSION

The array returned should be treated as read-only.

SEE ALSO

For an example call to this function, see Listing 3-5 (page 3-8).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedSymbol

String Object Functions

FD_MakeString

FD_Handle FD_MakeString(const char* *str*)

Creates a binary object containing a NULL-terminated Unicode string.

str	A NULL-terminated series of ASCII characters; in other words, a "C string."
return value	A string FDIL object.

SEE ALSO

For an example call to this function, see Listing 3-6 (page 3-9).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_NULLPointer

FD_MakeWideString

FD_Handle FD_MakeWideString(const DIL_WideChar* *unicodeStr*)

Creates a binary object containing a NULL-terminated Unicode string.

unicodeStr	NULL-terminates series of Unicode characters
return value	A string FDIL object.

SEE ALSO

For an example call to this function, see Listing 1-6 (page 18).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_NULLPointer

FD_IsString

```
int FD_IsString(FD_Handle obj)
```

Determines whether or not an FDIL object is a string object.

obj	The object to test.
return value	Zero or non-zero.

DISCUSSION

This function returns true if `FD_IsSubclass(obj,"string")` would return true.

SEE ALSO

For an example call to this function, see Listing 3-6 (page 3-9).

ERROR CODES

kFD_FDILNotInitialized

FD_IsRichSting

```
int FD_IsRichString(FD_Handle obj)
```

Determines whether or not an FDIL object is a rich string object.

obj	The object to test.
return value	Zero or non-zero.

DISCUSSION

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 device and may need to detect strings that cannot be completely interpreted.

SEE ALSO

For an example call to this function, see Listing 3-6 (page 3-9).

ERROR CODES

kFD_FDILNotInitialized

FD_GetString

DIL_Error FD_GetString(FD_Handle *obj*, char* *buffer*, long *bufLen*)

Copies over *bufLen* characters from a string object, converting from Unicode to ASCII.

obj	An FDIL string object.
buffer	Pointer to buffer for the C string.
bufLen	The size of the string buffer.
return value	An error code.

DISCUSSION

At most *bufLen* characters are copied over. If *obj* has more than *bufLen* characters, *buffer* points to an array of characters that is not NULL-terminated.

SEE ALSO

For an example call to this function, see Listing 3-6 (page 3-9).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedString
kFD_NULLPointer
kFD_ExpectedNonNegativeValue

FD_GetWideString

`DIL_Error FD_GetWideString(FD_Handle` *obj*`, DIL_WideChar*` *buffer*`, long` *bufLen*`)`

Copies over *bufLen* characters from a string object.

obj	An FDIL string object.
buffer	Pointer to buffer for the string.
bufLen	The size of the string buffer.
return value	An error code.

DISCUSSION

At most *bufLen* characters are copied over. If *obj* has more than *bufLen* characters, *buffer* points to an array of characters that is not NULL-terminated.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_ExpectedString`
`kFD_NULLPointer`
`kFD_ExpectedNonNegativeValue`

FD_ASCIIString

`FD_Handle FD_ASCIIString(FD_Handle` *obj*`)`

Converts a string binary object to a binary object whose data consists of a NULL-terminated array of ASCII characters.

obj	An FDIL string object.
return value	A binary object.

DISCUSSION

To convert a string object to a C string, use this function to do the Unicode to ASCII conversion, then pass this binary object to `FD_GetBinaryData`, and cast the pointer returned to a `char*`. For example:

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

```
FD_Handle asASCII = FD_ASCIIString(myString);
const char* textPtr = (const char*) FD_GetBinaryData(asASCII);
```

SEE ALSO

For an example call to this function, see Listing 3-6 (page 3-9).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedString

Large Binary Object Functions

FD_MakeLargeBinary

FD_Handle FD_MakeLargeBinary(long *size*, const char * *objClass*, long *compressed*)

Creates a large binary object of the given size.

size	The size of the large binary object.
objClass	Either NULL in which case the large binary object is given a default class, or a string that is passed to FD_MakeSymbol and becomes the object's class.
compressed	A value indicating whether to compress the data when storing it, and what compression scheme to use. This compression is done for you; you do not need to supply functions to compress the data. Specify kFD_NoCompression if you do not want the data compressed, and kFD_LZCompression to compress the data.
return value	A large binary FDIL object.

DISCUSSION

The large binary object stores the data using the storage procedures set with FD_SetLargeBinaryProcs, or the default kFD_MemoryStoreProcs.

SEE ALSO

For an example call to this function, see Listing 3-7 (page 3-10).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ErrorCreatingStore
kFD_ValueOutOfRange

FD_IsLargeBinary

```
int FD_IsLargeBinary(FD_Handle obj)
```

Determines whether or not an FDIL object is a large binary object.

obj	The object to test.
return value	Zero or non-zero.

ERROR CODES

kFD_FDILNotInitialized

FD_ReadFromLargeBinary

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

Reads bytes from the large binary object.

obj	An FDIL large binary object.
offset	Where to start reading from, in bytes from the beginning of the binary object.
buffer	Where to store the data.
count	How many bytes to read.
return value	An error code.

SEE ALSO

For an example call to this function, see Listing 3-7 (page 3-10).

ERROR CODES

an error from a user-defined large binary procedure
kFD_FDILNotInitialized
kFD_ExpectedLargeBinary
kFD_ExpectedNonNegativeValue
kFD_NULLPointer
kFD_CouldNotDecompressData
kFD_ErrorReadingFromStore

FD_WriteToLargeBinary

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

Writes bytes to a large binary object.

obj	An FDIL large binary object.
offset	Where to start writing from, in bytes from the beginning of the binary object.
buffer	Where the data is stored.
count	How many bytes to write.
return value	An error code.

SEE ALSO

For an example call to this function, see Listing 3-7 (page 3-10).

ERROR CODES

an error from a user-defined large binary procedure
kFD_FDILNotInitialized
kFD_ExpectedLargeBinary
kFD_ExpectedNonNegativeValue
kFD_NULLPointer
kFD_CouldNotCompressData
kFD_ErrorWritingToStore

FD_SetLargeBinaryProcs

DIL_Error FD_SetLargeBinaryProcs(const FD_LargeBinaryProcs* *procsPtr*)

Sets the default set of procedures to use when creating a large binary object.

procsPtr	A pointer to a struct with function pointers to the functions that create a large binary object and page it in and out of memory. You can pass in the constant kFD_MemoryStoreProcs to store large binary objects in main memory, kFD_DiskStoreProcs to store the object on disk, or kFD_NullStoreProcs to simply discard the data. FD_SetLargeBinaryProcs copies over the struct this pointer points to. This struct need not be permanent data.
return value	An error code.

DISCUSSION

If you do not call this function, the default kFD_MemoryStoreProcs is used.

SEE ALSO

You can create your own large binary procedures, see “Creating Your Own Large Binary Storage Procedures” (page 3-11).

ERROR CODES

kFD_FDILNotInitialized
kFD_NULLPointer

Array Object Functions

FD_MakeArray

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

Creates an array large enough to hold the given number of elements.

size	The initial size, number of slots, of the array.
cls	Either NULL in which case the array's is given a default class, or a string that is passed to `FD_MakeSymbol` and becomes the array's class.
return value	An array FDIL object.

SEE ALSO

For an example call to this function, see Listing 3-9 (page 3-17).

ERROR CODES

`kFD_FDILNotInitialized`
`kDIL_OutOfMemory`
`kFD_ValueOutOfRange`

FD_IsArray

`int FD_IsArray(FD_Handle obj)`

Determines whether or not an FDIL object is an array object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-9 (page 3-17).

ERROR CODES

`kFD_FDILNotInitialized`

FD_InsertArraySlot

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

Inserts the given object into the array at the specified position.

array	An FDIL array object.
pos	Where to insert the item.
item	The item to insert.
return value	An error code.

DISCUSSION

Any objects between that position and the end of the array are moved down in the array to make room. Calling this function with *pos* == `FD_GetSize(`*array*`) -1`, is equivalent to appending an object to the array.

SEE ALSO

For an example call to this function, see Listing 3-9 (page 3-17).

ERROR CODES

```
kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedArray
kFD_ValueOutOfRange
```

FD_AppendArraySlot

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

Appends the given element to the end of the array.

array	An FDIL array object.
item	The item to insert.
return value	An error code.

SEE ALSO

For an example call to this function, see Listing 3-9 (page 3-17).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedArray
kFD_ValueOutOfRange

FD_RemoveArraySlot

FD_Handle FD_RemoveArraySlot(FD_Handle *array*, long *pos*)

Removes the object at the given position in the array.

array	An FDIL array object.
pos	Which item to remove.
return value	The item to removed.

DISCUSSION

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 dispose of it, if desired.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

FD_RemoveArraySlotCount

DIL_Error FD_RemoveArraySlotCount(FD_Handle *array*, long *pos*, long *count*)

Removes *count* slots from the array starting at the given position.

array	An FDIL array object.
pos	Where to begin removing array slots from.
count	How many slots to remove.
return value	An error code.

DISCUSSION

Any objects between that position and the end of the array are moved forward in the array to fill in the vacated slots.

SPECIAL CONSIDERATIONS

The objects in the removed slots are not disposed of. You must address this before losing all references to those objects.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

FD_SetArraySlot

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.

array	An FDIL array object.
pos	Which array slot to set.
item	The new value of that array slot.
return value	The object that used to be in the *pos* array slot.

DISCUSSION

The object being replaced in the array is returned to the caller so that it can dispose of the object. No other array elements are affected, and the size of the array remains unchanged.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

FD_GetArraySlot

FD_Handle FD_GetArraySlot(FD_Handle *array*, long *pos*)

Returns the object in the given slot of the array.

array	An FDIL array object.
pos	Which array slot to access.
return value	The item in that array slot.

SEE ALSO

For an example call to this function, see Listing 3-9 (page 3-17).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedArray
kFD_ValueOutOfRange

Frame Object Functions

FD_MakeFrame

FD_Handle FD_MakeFrame()

Creates an empty frame.

return value	A frame FDIL object.

DISCUSSION

This function creates an empty frame, data can be added to this frame with FD_SetFrameSlot.

SEE ALSO

For an example call to this function, see Listing 3-10 (page 3-18).

ERROR CODES

kFD_FDILNotInitialized

kDIL_OutOfMemory

FD_IsFrame

int FD_IsFrame(FD_Handle *obj*)

Determines whether or not an FDIL object is a frame object.

obj	The object to test.
return value	Zero or non-zero.

SEE ALSO

For an example call to this function, see Listing 3-10 (page 3-18).

ERROR CODES

kFD_FDILNotInitialized

FD_SetFrameSlot

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*.

frame	An FDIL frame object.
slotName	A C string for the slot name.
item	An FDIL object to store in that slot.
return value	An FDIL object or kFD_NIL if the slot does not exist.

DISCUSSION

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 for you to dispose of.

SEE ALSO

For an example call to this function, see Listing 3-10 (page 3-18).

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_ExpectedFrame
kFD_NULLPointer

FD_GetFrameSlot

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

Retrieves the slot identified by *slotName*.

frame	An FDIL frame object.
slotName	A C string for the slot name.
return value	An FDIL object if the slot exists, kFD_NIL otherwise.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_NULLPointer

FD_FrameHasSlot

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

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

frame	An FDIL frame object.
slotName	A C string for the slot name.
return value	Zero or non-zero.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_NULLPointer

FD_RemoveFrameSlot

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

Removes the slot/value pair identified by *slotName*.

frame	An FDIL frame object.
slotName	A C string for the slot name.
return value	The FDIL object in the slot, if the slot exists, kFD_NIL, otherwise.

DISCUSSION

This function does not dispose of the object that was removed from the frame.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_NULLPointer

FD_GetIndFrameSlot

FD_Handle FD_GetIndFrameSlot(FD_Handle *frame*, long *pos*)

Allows traversal of the list of slots in a frame.

frame	An FDIL frame object.
pos	An index into the frame, see DISCUSSION.
return value	The object in the position *pos*.

DISCUSSION

By calling FD_GetIndFrameSlot with values of *pos* ranging from zero to FD_GetLength(*frame*) - 1 (inclusive), you can retrieve the contents of all the slots in the frame.

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.

SEE ALSO

For an example call to this function, see Listing 3-10 (page 3-18).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_ValueOutOfRange

FD_GetIndFrameSlotName

FD_Handle FD_GetIndFrameSlotName(FD_Handle *frame*, long *pos*)

Allows traversal of the list of slots in the frame, getting the name for each one.

frame	An FDIL frame object.
pos	An index into the frame, see DISCUSSION.
return value	An FDIL string object with the slot's name.

DISCUSSION

By calling FD_GetIndFrameSlotName with values of *pos* ranging from zero to FD_GetLength(*frame*)-1 (inclusive), you can retrieve the names of all the slots in the frame.

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.

SEE ALSO

For an example call to this function, see Listing 3-10 (page 3-18).

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedFrame
kFD_ValueOutOfRange

Magic Pointer Object Functions

FD_MakeMagicPointer

`FD_Handle FD_MakeMagicPointer(long` *val*`)`

Creates a magic pointer object.

val	The pointer value.
return value	A magic pointer FDIL object.

DISCUSSION

You should only need to create magic pointer objects if you are creating a Newton development environment.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_ValueOutOfRange`

FD_IsMagicPointer

`int FD_IsMagicPointer(FD_Handle` *obj*`)`

Determines whether or not an FDIL object is a magic pointer object.

obj	The object to test.
return value	Zero or non-zero.

ERROR CODES

`kFD_FDILNotInitialized`

FD_GetMagicPointer

`long FD_GetMagicPointer(FD_Handle` *obj*`)`

Returns the value stored in a magic pointer object.

obj	An FDIL magic pointer object.
return value	A `long`.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedMagicPointer

Library Initialization Functions

FD_Startup

DIL_Error FD_Startup()

Initializes the FDIL.

return value An error code.

DISCUSSION

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.

ERROR CODES

kDIL_OutOfMemory

FD_Shutdown

DIL_Error FD_Shutdown()

Closes the library.

return value An error code.

DISCUSSION

If this is the last call to FD_Shutdown, then all memory allocated by the FDIL since FD_Startup was called is deallocated.

ERROR CODES

kFD_FDILNotInitialized

Object Comparison Function

FD_Equal

`int FD_Equal(FD_Handle obj1, FD_Handle obj2)`

Determines whether or not two objects are equal to each other.

obj1	An FDIL object.
obj2	An FDIL object.
return value	Zero or non-zero.

DISCUSSION

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.

ERROR CODES

`kFD_FDILNotInitialized`

Object Duplication Functions

FD_Clone

`FD_Handle FD_Clone(FD_Handle obj)`

Creates a copy of the given object.

obj	An FDIL object.
return value	The new FDIL object.

DISCUSSION

If the object is an aggregate object, that is, an array or a frame, only the top-level object is cloned. None of the child objects are cloned.

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_PointerObjectIsFree

FD_DeepClone

FD_Handle FD_DeepClone(FD_Handle *obj*)

Creates a copy of the given object.

obj	An FDIL object.
return value	The new FDIL object.

DISCUSSION

If the object is an aggregate object, that is, an array or a frame, all child objects are cloned as well.

ERROR CODES

kFD_FDILNotInitialized
kDIL_OutOfMemory
kFD_PointerObjectIsFree

Object Disposing Functions

FD_Dispose

DIL_Error FD_Dispose(FD_Handle *obj*)

Disposes of an object's allocated memory.

obj	An FDIL object.
return value	An error code.

DISCUSSION

Upon return *obj* is no longer valid, if it used to be a pointer object.

This function simply ignores non-pointer objects, since they contain no data outside the FD_Handle. Symbol objects are not disposed of either, since they are a pooled resource.

This function does a shallow-dispose of an object; that is if the object is an aggregate object such as an array or a frame, memory used by the component objects is not freed. To perform a deep-disposing of an aggregate object, use FD_DeepDispose.

ERROR CODES

kFD_FDILNotInitialized
kFD_PointerObjectIsFree

FD_DeepDispose

DIL_Error FD_DeepDispose(FD_Handle *obj*)

Disposes of an object's allocated memory, and if the object is an array or frame, disposes of any objects contained within them.

obj	An FDIL object.
return value	An error code.

DISCUSSION

Upon return *obj* is no longer valid, if it used to be a pointer object.

This function simply ignores non-pointer objects, since they contain no data outside the FD_Handle. Symbol objects are not disposed of either, since they are a pooled resource.

ERROR CODES

kFD_FDILNotInitialized
kFD_PointerObjectIsFree

Object Printing Function

FD_PrintObject

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

Formats and prints an FDIL object.

obj	The FDIL object to print.
EOLString	The end-of-line sequence used in your development environment.
writeFn	A `DIL_WriteProc` that prints out the formatted text; see "DIL_WriteProc" (page 3-30). As with other functions that call a `DIL_WriteProc`, this function calls your `DIL_WriteProc` with an *amt* parameter that is the number of bytes to be written from the *buf* parameter. This function adds a NULL byte to the end of *buf*, as a convenience, allowing you to treat *buf* as a C string. The NULL byte is added in the $(amt+1)^{th}$ position of *buf*; that is *buf*`[`*amt*`] == 0`.
userData	A pointer that is passed on to your *writeFn*.
return value	An error code.

ERROR CODES

`kFD_FDILNotInitialized`
`kFD_NULLPointer`

Object Streaming Functions

FD_Flatten

DIL_Error FD_Flatten(FD_Handle *obj*, DIL_WriteProc *writeFn*, void* *userData*)

Converts the given object into a flat stream of bytes in Newton Stream Object Format (NSOF) suitable for saving to disk or for transmission to a Newton device.

obj	An FDIL object.
writeFn	A DIL_WriteProc to actually write the streamed bytes; see “DIL_WriteProc” (page 3-30).
userData	A pointer to any data you wish to be passed on to your *writeFn*.
return value	An error code.

DISCUSSION

FD_Flatten just performs the conversion of objects into bytes; the actual disposition of the bytes is determined by the *writeFn* function you provide.

SEE ALSO

This function is discussed in “Object Streaming” (page 3-21).

ERROR CODES

an error returned by a user’s DIL_WriteProc
kFD_FDILNotInitialized
kDIL_OutOfMemory
kDIL_ErrorWritingToPipe
kFD_ErrorReadingFromStore

FD_Unflatten

FD_Handle FD_Unflatten(DIL_ReadProc *readFn*, void* *userData*)

Converts a flat stream of bytes in Newton Stream Object Format (NSOF) into an FDIL object.

readFn	A DIL_ReadProc to actually read the streamed bytes; see “DIL_ReadProc” (page 3-31).
userData	A pointer to any data you wish to be passed on to your *readFn*.
return value	An FDIL object.

DISCUSSION

FD_Unflatten does not care where the bytes come from. It is only responsible for using them to recreate the original objects from which they were formed.

SEE ALSO

This function is discussed in “Object Streaming” (page 3-21).

ERROR CODES

an error returned by a user’s DIL_ReadProc
kFD_FDILNotInitialized
kDIL_OutOfMemory
kDIL_ErrorReadingFromPipe
kFD_UnknownStreamVersion
kFD_StreamCorrupted
kFD_UnsupportedCompression
kFD_UnsupportedStoreVersion
kFD_ErrorCreatingStore
kFD_ErrorWritingToStore

Object Class Functions

FD_GetClass

FD_Handle FD_GetClass(FD_Handle *obj*)

Returns the class of the given object.

obj	An FDIL object.
return value	An FDIL symbol object that is the class of *obj*.

ERROR CODES

kFD_FDILNotInitialized
kFD_PointerObjectIsFree

FD_SetClass

DIL_Error FD_SetClass(FD_Handle *obj*, FD_Handle *oClass*)

Sets the class of given object to the specified class.

obj	An FDIL pointer object.
oClass	An FDIL symbol object for the class, or kFD_NIL.
return value	An error code.

DISCUSSION

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 an object's class to kFD_NIL.

ERROR CODES

kFD_FDILNotInitialized
kFD_ExpectedPointerObject
kFD_InvalidClass

FD_IsSubClass

```
int FD_IsSubClass(FD_Handle obj, const char* class)
```

Returns whether or not an object is an instance of the given object class.

obj	The object to test.
class	The class to test.
return value	Zero or non-zero.

SEE ALSO

This function is discussed in "Object Classes" (page 3-23).

ERROR CODES

```
kFD_FDILNotInitialized
kFD_PointerObjectIsFree
kFD_NULLPointer
```

Error Handling Function

FD_GetError

```
DIL_Error FD_GetError()
```

Returns a value indicating the success or failure of the last operation performed by an FDIL function.

return value	An error code.

DISCUSSION

Robust applications should check the result of `FD_GetError` after calling any FDIL function that can reasonably be expected to fail.

SEE ALSO

"Error Handling" (page 3-21).

Memory Management Functions

FD_AllocatedMemory

```
long FD_AllocatedMemory()
```

Returns the total amount of memory allocated by the FDIL library, including that occupied by created objects and that used by internal data structures.

return value The amount of memory used in bytes.

DISCUSSION

This function can be useful to track how much memory is used by particular objects, or by the FDIL sub-system in general.

SEE ALSO

For an example call to this function, see Listing 3-13 (page 3-25).

ERROR CODES

`kFD_FDILNotInitialized`

FD_IsFree

```
int FD_IsFree(FD_Handle obj)
```

Determines whether the FDIL object refers to a deleted pointer object.

obj The object to test.

return value Zero or non-zero.

DISCUSSION

FDIL objects containing non-pointer objects such as integers or the nil object cause this function to return false, 0.

SPECIAL CONSIDERATIONS

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 occupies the same space previously occupied by the deleted object. The FD_Handle effectively refers 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 not be called in shipping code.

ERROR CODES

kFD_FDILNotInitialized

FD_CheckForMemoryLeaks

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

Reports any undeleted, user-allocated objects, along with the file name and line number within the file containing the function call that allocated that object.

EOLString	The end-of-line sequence used in your development environment.
printFn	The print function to use to print information; see “DIL_WriteProc” (page 3-30).
userData	A pointer passed on to your printing function.
return value	The number of user-allocated objects left undeleted.

ERROR CODES

kFD_FDILNotInitialized

SPECIAL CONSIDERATIONS

This function only exists in the debug version of the DIL.

FDIL Summary

Type Definitions

FD_Handle
DIL_Error
DIL_WideChar
FD_LargeBinaryProcs
DIL_WriteProc
DIL_ReadProc
DIL_StatusProc

Constants

FDIL objects

kFD_NIL
kFD_True

Large Binary Storage Procedures

kFD_MemoryStoreProcs
kFD_DiskStoreProcs
kFD_NullStoreProcs

Large Binary Compression Options

kFD_NoCompression
kFD_LZCompression
kFD_ZippyCompression

Immediate Types

kImmedSpecial
kImmedCharacter
kImmedBoolean
kImmedReserved

Error Codes

kDIL_NoError	kFD_ExpectedPointerObject
kDIL_ErrorBase	kFD_ExpectedImmediate
kDIL_OutOfMemory	kFD_ExpectedMagicPointer
kDIL_InvalidParameter	kFD_ExpectedArray
kDIL_InternalError	kFD_ExpectedFrame
kDIL_ErrorReadingFromPipe	kFD_ExpectedBinary
kDIL_ErrorWritingToPipe	kFD_ExpectedLargeBinary
kFD_ErrorBase	kFD_ExpectedReal
kFD_UnknownStreamVersion	kFD_ExpectedString
kFD_StreamCorrupted	kFD_ExpectedSymbol
kFD_UnsupportedCompression	kFD_ExpectedChar
kFD_CouldNotCompressData	kFD_NULLPointer
kFD_CouldNotDecompressData	kFD_ExpectedPositiveValue
kFD_UnsupportedStoreVersion	kFD_ExpectedNonNegativeValue
kFD_ErrorCreatingStore	kFD_ValueOutOfRange
kFD_ErrorWritingToStore	kFD_SymbolTooLong
kFD_ErrorReadingFromStore	kFD_IllegalCharInSymbol
kFD_FDILNotInitialized	kFD_InvalidClass
kFD_FDILAlreadyInitialized	kFD_PointerObjectIsFree
kFD_ExpectedInteger	kFD_LBReadingFromUnwrittenPage

Functions

Integer Object Functions

```
FD_Handle   FD_MakeInt(long val)
int         FD_IsInt(FD_Handle obj)
long        FD_GetInt(FD_Handle obj)
```

Immediate Object Functions

```
FD_Handle   FD_MakeImmediate(long type, long value)
int         FD_IsImmediate(FD_Handle obj)
DIL_Error   FD_GetImmediate(FD_Handle obj, long* type, long* value)
```

Character Object Functions

```
FD_Handle       FD_MakeChar(char val)
FD_Handle       FD_MakeWideChar(DIL_WideChar val)
int             FD_IsChar(FD_Handle obj)
char            FD_GetChar(FD_Handle obj)
DIL_WideChar    FD_GetWideChar(FD_Handle obj)
DIL_Error       FD_ConvertFromWideChar(char* dest,
                    const DIL_WideChar* src, long numChars)
DIL_Error       FD_ConvertToWideChar(DIL_WideChar* dest,
                    const char* src, long numChars)
DIL_Error       FD_SetWideCharEncoding(long encoding)
```

Boolean Object Function

```
int     FD_IsBoolean(FD_Handle obj)
```

Nil Object Functions

```
int     FD_IsNIL(FD_Handle obj)
int     FD_NotNIL(FD_Handle obj)
```

Pointer Object Functions

```
int         FD_IsPointerObject(FD_Handle obj)
long        FD_GetLength(FD_Handle obj)
DIL_Error   FD_SetLength(FD_Handle obj, long newSize)
```

Binary Object Functions

```
FD_Handle   FD_MakeBinary(long size, const char* cls)
int         FD_IsBinary(FD_Handle obj)
void*       FD_GetBinaryData(FD_Handle obj)
```

Real Object Functions

```
FD_Handle   FD_MakeReal(double val)
```

```
int         FD_IsReal(FD_Handle obj)
double      FD_GetReal(FD_Handle obj)
```

Symbol Object Functions

```
FD_Handle       FD_MakeSymbol(const char* str)
int             FD_IsSymbol(FD_Handle obj)
const char*     FD_GetSymbol(FD_Handle obj)
```

String Object Functions

```
FD_Handle   FD_MakeString(const char* str)
FD_Handle   FD_MakeWideString(const DIL_WideChar* unicodeStr)
int         FD_IsString(FD_Handle obj)
int         FD_IsRichString(FD_Handle obj)
DIL_Error   FD_GetString(FD_Handle obj, char* buffer, long bufLen)
DIL_Error   FD_GetWideString(FD_Handle obj, DIL_WideChar* buffer,
                long bufLen)
FD_Handle   FD_ASCIIString(FD_Handle obj)
```

Large Binary Object Functions

```
FD_Handle   FD_MakeLargeBinary(long size, const char * objClass,
                long compressed)
int         FD_IsLargeBinary(FD_Handle obj)
DIL_Error   FD_ReadFromLargeBinary(FD_Handle obj, long offset,
                void* buffer, long count)
DIL_Error   FD_WriteToLargeBinary(FD_Handle obj, long offset,
                const void* buffer, long count)
DIL_Error   FD_SetLargeBinaryProcs(const FD_LargeBinaryProcs* procsPtr)
```

Array Object Functions

```
FD_Handle   FD_MakeArray(long size, const char* cls)
int         FD_IsArray(FD_Handle obj)
DIL_Error   FD_InsertArraySlot(FD_Handle array, long pos, FD_Handle item)
DIL_Error   FD_AppendArraySlot(FD_Handle array, FD_Handle item)
FD_Handle   FD_RemoveArraySlot(FD_Handle array, long pos)
DIL_Error   FD_RemoveArraySlotCount(FD_Handle array, long pos,
                long count)
FD_Handle   FD_SetArraySlot(FD_Handle array, long pos, FD_Handle item)
FD_Handle   FD_GetArraySlot(FD_Handle array, long pos)
```

Frame Object Functions

```
FD_Handle   FD_MakeFrame()
int         FD_IsFrame(FD_Handle obj)
FD_Handle   FD_SetFrameSlot(FD_Handle frame, const char* slotName,
                FD_Handle item)
FD_Handle   FD_GetFrameSlot(FD_Handle frame, const char* slotName)
int         FD_FrameHasSlot(FD_Handle frame, const char* slotName)
FD_Handle   FD_RemoveFrameSlot(FD_Handle frame, const char* slotName)
FD_Handle   FD_GetIndFrameSlot(FD_Handle frame, long pos)
FD_Handle   FD_GetIndFrameSlotName(FD_Handle frame, long pos)
```

Magic Pointer Object Functions

```
FD_Handle   FD_MakeMagicPointer(long val)
int         FD_IsMagicPointer(FD_Handle obj)
long        FD_GetMagicPointer(FD_Handle obj)
```

Library Initialization Functions

```
DIL_Error   FD_Startup()
DIL_Error   FD_Shutdown()
```

Object Comparison Function

```
int     FD_Equal(FD_Handle obj1, FD_Handle obj2)
```

Object Duplication Functions

```
FD_Handle   FD_Clone(FD_Handle obj)
FD_Handle   FD_DeepClone(FD_Handle obj)
```

Object Disposing Functions

```
DIL_Error   FD_Dispose(FD_Handle obj)
DIL_Error   FD_DeepDispose(FD_Handle obj)
```

Object Printing Function

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

Object Streaming Functions

```
DIL_Error   FD_Flatten(FD_Handle obj, DIL_WriteProc writeFn,
                void* userData)
FD_Handle   FD_Unflatten(DIL_ReadProc readFn, void* userData)
```

Object Class Functions

```
FD_Handle   FD_GetClass(FD_Handle obj)
DIL_Error   FD_SetClass(FD_Handle obj, FD_Handle oClass)
int         FD_IsSubClass(FD_Handle obj, const char* class)
```

Error Handling Function

```
DIL_Error   FD_GetError()
```

Memory Management Functions

```
long    FD_AllocatedMemory()
int     FD_IsFree(FD_Handle obj)
long    FD_CheckForMemoryLeaks(const char* EOLString,
            DIL_WriteProc printFn, void* userData)
```