The High Level Frame Desktop Integration Library (HLFDIL)
Newton Developer Training

The High Level Frame Desktop Integration Library (HLFDIL) is used to move Newton frames and arrays to and from a desktop machine, running either the Macintosh OS or Windows. (In common usage, HLFDIL is usually shortened to Frame Desktop Integration Library, or FDIL, and the two terms are used interchangeably throughout this article.) Before the HLFDIL can be opened a connection between the Newton and the desktop machine must be established using the Communications Desktop Integration Library (CDIL).

The FDIL is used to map the dynamic structure of Newton frames to the static structures used on desktop machines. As with the CDIL, it is implemented in C++ but has a C language API. The basic assumption of the FDIL is that the format of the Newton frames being moved is already known. A C language structure having a one-to-one correspondence with the Newton frame can be defined, and the desktop programmer can define the mapping of slots to fields. However, since there are cases when the programmer may not know the structure of the Newton frame in advance or when additional slots have been added to the frame, there must be a way to handle these unexpected or unknown slots. This is handled by the FDIL by uploading these slots as unbound data, that is, data for which there is not a previously defined memory location of the appropriate size and type. The unbound data is therefore put into memory on the desktop machine in a tree structure which defines where the data is in the NewtonScript frame in relation to other slots in the frame.

Throughout this article, the use of the FDIL to transfer Newton frame data will be discussed. However, it should be remembered that it is also used to transfer Newton array data. It is worth noting that the FDIL cannot be used to transfer simple data such as integers or strings unless they are part of a frame or array. It is recommended that such data be transferred using the CDIL mechanism, or put into a simple NewtonScript frame, such as `fooFrame:={myInteger:3}`.

Opening the FDIL and Creating Objects
After the CDIL connection, or pipe, has been opened, the FDIL may be initialized and used. The FDIL routine FDInitFDIL is called as follows:

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

Any error in initializing the library will be returned. FDIL errors fall in the -28000 range and a complete list may be found at the end of the FDIL section of the document *Newton Desktop Integration Libraries* which can be found on the DIL web page at http://dev.info.apple.com/newton/tools/dils.html.

Next, one or more FDIL objects must be created using the routine FDILCreateObject which is defined as:

```
DILObj *FDCreateObject(short objType,
char *objClass);
```

The first argument describes whether the object being created is an array or a frame, and the second argument is an optional class name which NewtonScript arrays may carry. Each frame or array which is transferred must have a separate object created for it. This includes sub-frames and sub-arrays within a parent frame or array.

For example, for the following NewtonScript frame, three separate FDIL objects must be created before the frame can be defined on the desktop:

```
aFrame:={
name:"foo",
phone:["333-444-5555","101-202-3333"],
address:{street:"123 Main", town:"Fooburg",
state:'GA, postalCode:12345}
}
```

There must be an FDIL object for the main frame, `aFrame`, one for the `phone` sub-array and one for the `address` sub-frame.

The calls to create these FDIL objects would look like this:

```
DILObj *aFrameObj, foneObj, addrObj;

aFrameObj = FDCreateObject(kDILFrame, NULL);
foneObj=FDCreateObject(kDILArray,"phoneClass");
addrObj = FDCreateObject(kDILFrame, NULL);
```

In this example `FDCreateObject` is called twice with the predefined constant `kDILFrame` to create an object for defining a frame and once to create an array object using the `kDILArray` constant for the first argument. In the case of the array, we also supply the class name `"phoneClass"` as it is a named (classed) array. Named arrays are

described on page 2-9 of the *NewtonScript Reference*. (An electronic version of this book is available from the Newton documentation web page noted above.)

As of this writing a known bug exists which will throw an error on the Newton when a string is downloaded, if an array includes an unnamed string (that is, a classless string) as one of its elements. The workaround is simply to give unclassed strings a class of "string" when they are array members. See the routine `DearchiveFrame` in the FDIL Archive Lab solution code for an example of this work around. This bug should be fixed in future releases of the HLFDIL.

Defining Objects

Once you have created an FDIL object, you may begin to bind memory locations to the object. This process describes to the FDIL where data being transferred to or from the Newton will come from or go to. In other words, by binding a memory location on the desktop to a slot in a Newton frame, the FDIL knows where to put data or where to go to get data. The memory locations used in this binding may be a variable on the stack, a global memory location or a dynamically allocated heap location, depending on the use and duration of the data being transferred.

The routine `FDbindSlot` is used to connect the memory location with the Newton slot. Its formal definition is:

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

The first argument (`theObj`) is the object created to define the frame being transferred, and the second (`slotName`) is the name of the slot that is being bound.

The third (`bindVar`) is a pointer to the memory location to which the slot is being connected. When data is to be downloaded to the Newton this will be the location of the data being sent. In the case of an upload from the Newton this is the place where the received data will be put. In the latter case it is up to the programmer to make sure there is enough room to hold the data being received. The `maxLen` argument is used to describe the length of an object being sent to the Newton, and the `curLen` argument describes the length of

an object which is being received from the Newton. In the case of an array or frame, maxLen or curLen should be set to -1.

The varType argument describes what type of data object the slot being bound is expected to be. The following table shows the existing types:

```
kDILPlainArray // as opposed to a classed array
kDILArray // array with class name associated
kDILBoolean // true or false
kDILUnicodeCharacter // 16-bit character
kDILCharacter // 8-bit ASCII
kDILFrame // object is a frame
kDILSmallRect // packs a Newton rect into a long
kDILImmediate // Newton immediate value (not int)
kDILInteger // 30-bit integer
kDILBLOB // not used
kDILNIL // Newton nil value (NULL)
kDILBinaryObject // double or other binary sequence
kDILString // char *
kDILSymbol // handled as char *
```

Of these types several are worth special mention.

kDILArray versus kDILPlainArray - many arrays on the Newton are simply sequences of data but a few have a class associated with them. kDILPlainArray has no class associated with it while kDILArray does.

kDILBoolean and kDILNIL - both have platform-specific definitions for the values true, false and nil.

kDILSmallRect is automatically used when a frame on the Newton which is used to store a rectangle is sent and the rectangle values (top, left, bottom, right) all fit in a single byte value. In this case the values are packed into a single long value.

kDILBLOB is not implemented.

kDILBinaryObject is used for any unspecified binary object (such as sounds, pictures, and so forth) as well as for the Newton Real type. Note, however, that unlike integers or immediates, platform-specific byte ordering must be accounted for.

Make sure you compile using 8-byte doubles if you are going to transfer Real numbers. When transferring Newton Real numbers, you

must set your development environment to use 8-byte double values or it will interpret the reals received as a different value than expected. This is usually an option specific to your system. For example, Metrowerks defaults to 4-byte doubles and the 8-byte double option must be specified in the compiler options in the Preferences menu item. MPW defaults to 8-byte doubles and so no special action must be taken.

`kDILString` includes both classed and unclassed strings.

`kDILSymbol` is a Newton symbol (such as 'mySymbol) transferred to and from the desktop as a C string.

The last argument in `FDbindSlot (objClass)` is the class, if any, of the object. While many objects on the Newton have no class associated with them, many may have a class assigned by the programmer or by the system. For example, while most strings have no class associated with them, they may have one assigned explicitly by the Newton programmer. Conversely, Newton Real numbers always are transferred as objects of type `kDILBinaryObject` with a class of "Real." If an object has no class associated with it, this argument should be a NULL . If it has a class it is transferred with a C string, such as for the class name.

An example taken from the SoupDrink sample code involves downloading a name frame to the Newton Name soup. The following pseudo-NewtonScript describes a card frame and its associated FDIL type:

```
card := {

     cardType:kDILInteger,
     Name: kDILFrame,
     Address: kDILString,
     City: kDILString,
     Region: kDILString,
     Postal_Code: kDILString,
     phones: kDILArray,
     sorton: kDILString
}
```

This is only a code fragment from the SoupDrink example. The variables shown (such as the name strings) are assigned values in earlier code. To see the full code, see the SoupDrink code walkthrough.

In the same way the Name frame and the phones array may be defined as:

```
Name := {

     class: kDILSymbol,
     first: kDILString,
     last: kDILString
}

phones := [kDILString, kDILString, ...]
```

While not shown here, the elements of the phones array each have a separate class associated with them such as "HomePhone," "WorkPhone," and so forth.

The code to create the FDIL object for this structure is:

```
name = FDCreateObject(kDILFrame, NULL);
FDbindSlot(name, "Class", (void *)&pClass, kDILSymbol,
strlen(pClass), -1, NULL);
FDbindSlot(name, "first", (void *)&fName, kDILString,
strlen(fName), -1, NULL);
FDbindSlot(name, "last", (void *)&lName, kDILString,
strlen(lName), -1, NULL);

phones = FDCreateObject(kDILArray, NULL);
FDbindSlot(phones, NULL, (void *)&phoneNo, kDILString,
strlen(phoneNo), -1, "HomePhone"));

entry = FDCreateObject(kDILFrame, NULL);
FDbindSlot(entry, "cardType", (void *)&cardType,
kDILInteger, sizeof(int), -1, NULL) ;
FDbindSlot(entry, "Name", (void *)name, kDILFrame, 0, -1,
NULL);
FDbindSlot(entry, "Address", (void *)&addr, kDILString,
strlen(addr), -1, NULL) ;
FDbindSlot(entry, "City", (void *)&town, kDILString,
strlen(town), -1, NULL) ;
FDbindSlot(entry, "Region", (void *)&state, kDILString,
strlen(state), -1, NULL) ;
FDbindSlot(entry, "Postal_Code", (void *)&zip,
kDILString, strlen(zip), -1, NULL) ;
FDbindSlot(entry, "phones", (void *)phones, kDILArray, 0,
-1, NULL) ;
FDbindSlot(entry, "sorton", (void *)&sName, kDILString,
strlen(sName), 0, "Name");
```

Here the FDIL object's name and phones are bound to slots in the entry object. Note also that the string in the phones array has a class associated with it, HomePhone. Once the entry object is defined it is ready to be used to download it to the Newton.

Transferring Data

Once an FDIL object is created and the slots are defined by binding them to desktop memory, it may be used to transfer data to or from the Newton. This is done by using the routines FDget to upload and FDput to download the object and its associated data. These routines are defined as:

```
objErr FDput (DILObj *entry, short type, CDILPipe *pipe);

objErr FDget (DILObj *entry, short type, CDILPipe *pipe,
long timeOut, CDILPipeCompletionProcPtr
callBack,long refCon);
```

Notice that the CDIL pipe is used here since this is the first time that data will actually be transferred. Note also that the FDget routine may be called asynchronously by passing a procedure pointer in for the fifth argument.

In both routines the first argument (entry) is the master object being transferred. It may have sub-frames and arrays objects bound to it but this is the top level object. The next argument (type) is the FDIL type (kDILFrame or kDILArray) of this master object. The third argument (pipe) is the CDIL pipe being used to transfer the data.

For FDget we have these additional arguments:
The fourth argument (timeOut) defines how long the FDIL should attempt to transfer the object before giving up and returning an error. The timeout value is measured in milliseconds on Windows machines and ticks on Macintosh OS machines.

The next argument (callBack) is a pointer to a callback routine if the FDget call is made asynchronously. If you are making the call synchronously, simply pass NULL for this argument.

The last argument (refCon) is an arbitrary value which can be passed to your completion routine.

SoupDrink downloads the previously defined entry FDIL object to the Newton using this call to FDput:

```
FDput(entry, kDILFrame, ourPipe);
```

The Newton must be in a state to receive a frame or array before it is sent from the desktop machine. See the SoupDrink code for the Newton for an example of code which imports and exports frames.

Unbound Data

Unbound data is data which arrives at the desktop but is not bound to any memory locations by calls to `FDbindSlot()`. This typically occurs in two situations: when the desktop programmer does not know the structure of the Newton data beforehand and when there are previously unknown slots which have been added to a frame.

An example of the first case is a program which allows the programmer to select what will be transferred. SoupDrink does this when it allows the programmer to name a Newton soup which is to be uploaded. In this case SoupDrink provides a universal way to handle all soups by uploading the information as unbound data which it then writes to file.

The second situation occurs either when there is a system or program update or when applications which "tag along" on system soups are added. For example, if a new version of the Names application added new slots to the soup frames stored for a Name card, SoupDrink would get the data from the old slots if it used the entry definition shown above but would not have a place allocated to put new data slots. Similarly, if a new contact application was loaded into the Newton, it might choose to build off of the existing Names application but add additional information to the soup entries. In this case SoupDrink would get the standard data but would not be prepared for the data added to the Names soup entries.

In either case, unbound data provides a way to accept unknown data slots and then to parse them for further disposition. Following is a description of unbound data and how to extract information about the data once it has arrived at the desktop.

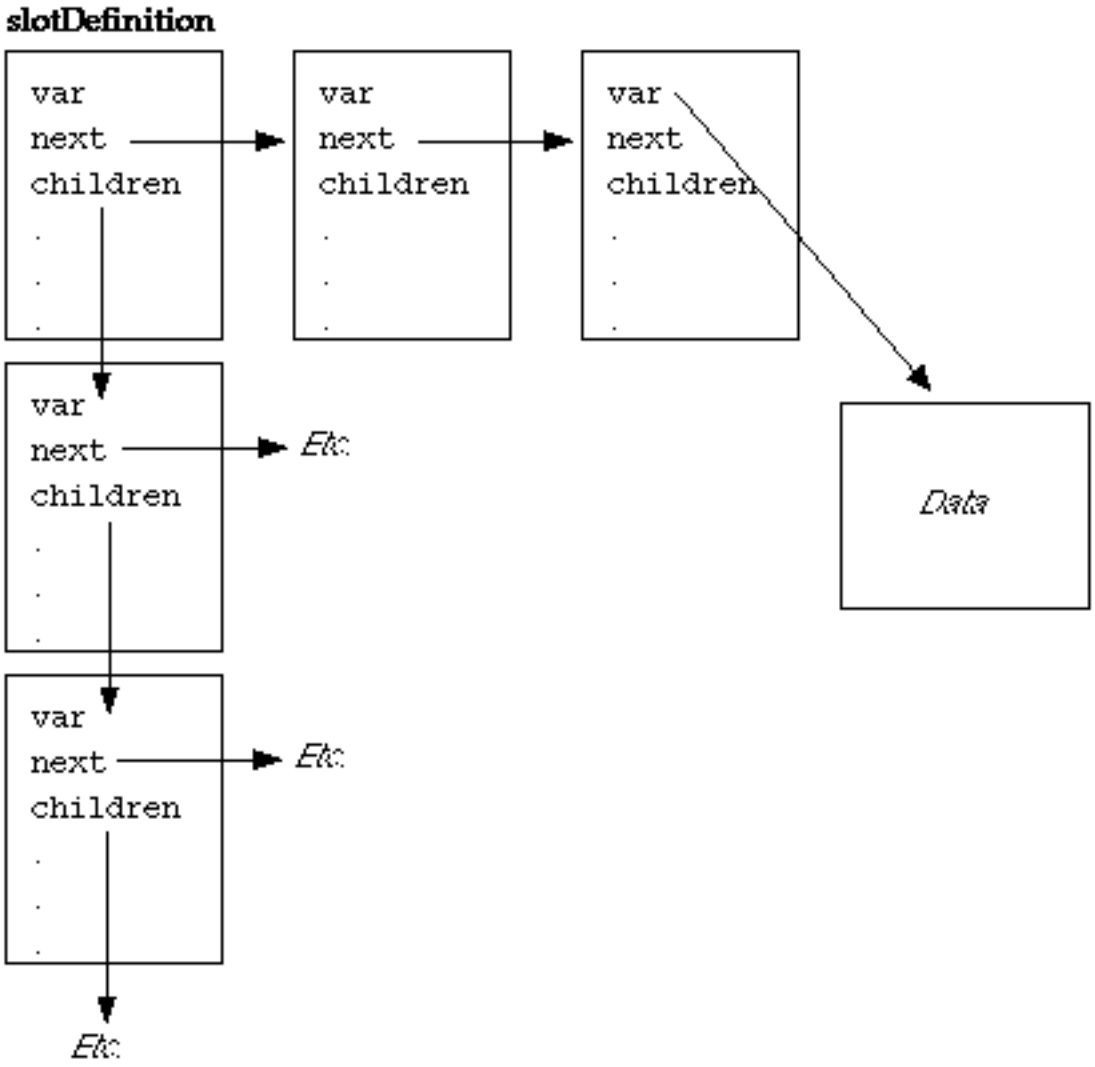
Unbound data is linked together in a chain of data structures of type `slotDefinition`. The `slotDefinition` data type is defined as follows:

```
typedef struct slotDefinition
{
short varType; // Data type of this variable
```

```
void *var; // Actual pointer to data
ulong length; // Length of data (strings)
ulong maxLength; // Maximum Length of data
ulong streamLen; // Length of streaming - internal
ulong bufIndex; // Current buffer index - internal
long namePrec; // Original precedent? - internal
long classPrec; // Original precedent? - internal
char *slotName; // Name of slot for this var
char *oClass; // class of this object
short slotType; // Data type of this slot
ulong truncSize; // Current size of truncated object
long childCnt; // Number of child nodes
long peerCnt; // Number of peer nodes
short dataFilled; // TRUE if data added in this op
long internalFlags; // Internal state flags
short boundData; // for future use
struct slotDefinition *children;
struct slotDefinition *next;
} slotDefinition ;
```

The fields which are significant to understanding the structure of unbound data are the `childCnt`, `peerCnt`, `children` and `next` fields. Unbound data is stored as a series of linked `slotDefinition` structures which may have siblings (peers) or children. In this scheme, `peerCnt` is the number of peers an item has, `childCnt` is the number of children, `next` is a pointer to the `slotDefinition` for the next peer in the list and `children` is a pointer to the start of the chain of `slotDefinitions` for the children of the item.

This is shown in the following diagram:

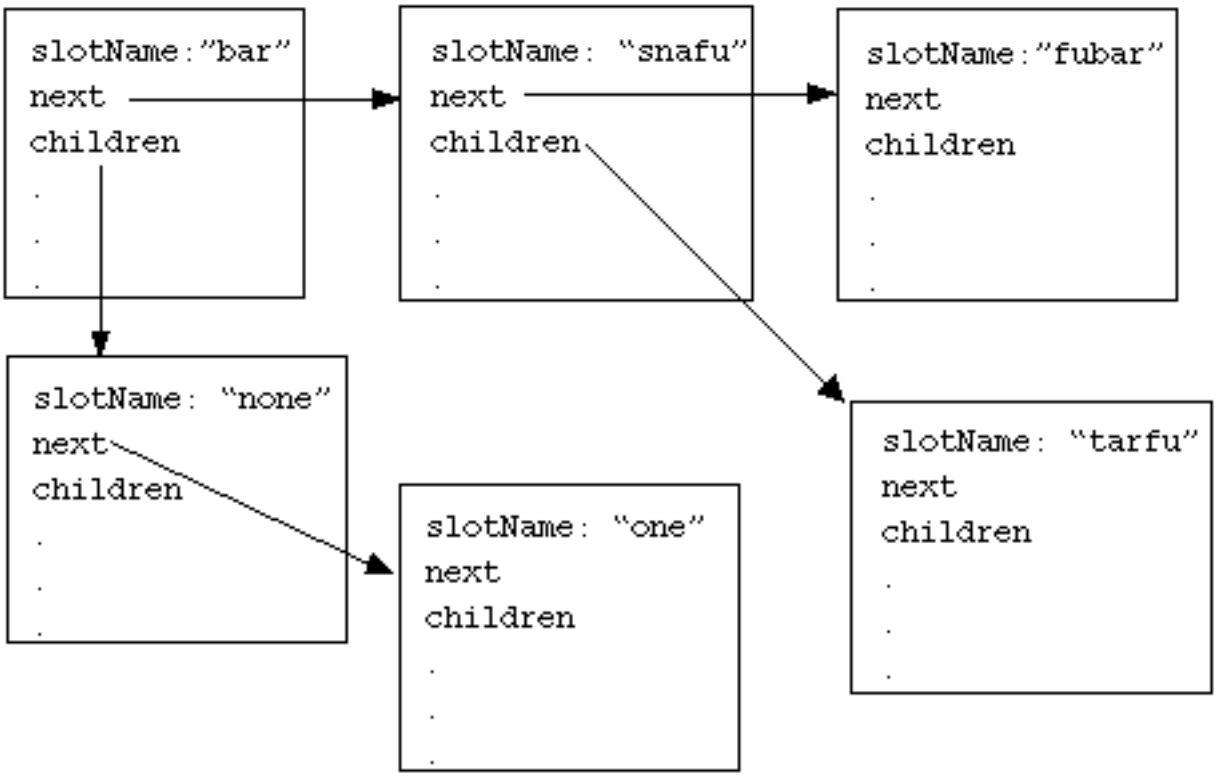


Here the `slotDefinition` structures are linked laterally by the next field while the children field points to the start of the list of all of the item's children. They each have their own list of children which do not intersect.

A more concrete example of this structure is shown below for the following simple frame:

```
foo:= {
bar:{none:nil, one:1},
snafu:{tarfu:2},
fubar: 3
}
```

Unbound data structure for foo frame



Not shown in the second diagram is the var field which points to the actual data associated with the slot. With the addition of this slot we now have enough information to write the following pseudo-code for walking through the unbound data list printing the data:

```
parselist(list)
begin
item:=list[0];
while (item != nil)
begin
if (item !=kDILFrame & item !=kDILArray)
output(item->var)
else
parselist(item->children)
item:=item->next
end
end
```

The only thing remaining is to find the start of the unbound data list and dispose of the memory allocated by the FDIL for the unbound data. The following routines do what is necessary:

```
SlotDefinition *FDGetUnboundList(DILObj *theFDILObj);
objErr FDFreeUnboundList(DILObj *theFDILObj,
SlotDefinition *list);
```

FDGetUnboundList returns a pointer to the list of items in the top level (master frame) of the unbound data while FDFreeUnboundList frees all the memory allocated for unbound data when FDget is called and there is data uploaded which has not been defined using FDbindSlot.

The use of unbound data may be summarized as follows:

Unbound data is the data received from the Newton after a call to FDget returns slots which do not have a specified location in memory connected to them. This may occur because the desktop programmer did not know what the structure of the data would be or because additional data was added to the frame being uploaded. Once transferred to the desktop machine the data is put into a linked list of slotDefinition structures which contain pointers to the next unbound item as well as items which are children of the current item. The unbound list may be parsed and extensive information about the slot associated with the item (including its type, name, class, and the actual data associated with the item) may be extracted from its slotDefinition structure.

The list of items of unbound data are kept in an array and the children field in a slotDefinition is an array of child items. SoupDrink uses this to parse the unbound data by looping through each of these arrays. Both algorithms work since *children is equivalent to children[0] in C.

Destroying Objects and Closing the FDIL

When you are completely finished with an FDIL object, call FDDisposeObject to destroy the object and all associated memory. Note that calling FDDisposeObject does not deallocate memory explicitly allocated by the desktop application. If memory is allocated off of the heap and then bound to a slot in an object, FDDisposeObject will not deallocate the heap memory. It must be explicitly deallocated.

Finally, the FDIL should be closed by calling FDDisposeFDIL. This should be done before closing the CDIL.
The definition of these routines as follows:

```
objErr FDDisposeObject ( DILObj *theObject );
objErr FDDiposeFDIL();
```

*****