Newton Programming: DILs Overview

An important new addition to Newton communications programming is a set of libraries called the Desktop Integration Libraries or DILs. The first and most important thing to understand about DILs is that *they have nothing to do with programming communications on the Newton. device.* Instead, they are used to create a communications link between Newton devices and *desktop* machines. In other words, they are an aid for writing code for a desktop machine which will communicate with a Newton.

Figure 1 shows this relationship. Essentially, endpoint code on the Newton, whether part of an application or in a transport, transfers data between a Newton device and a desktop machine. On the desktop machine, an application which uses a DIL sends and receives data which is handled by a desktop application. Currently DILs are available for the MacOS and for Windows-based machines.

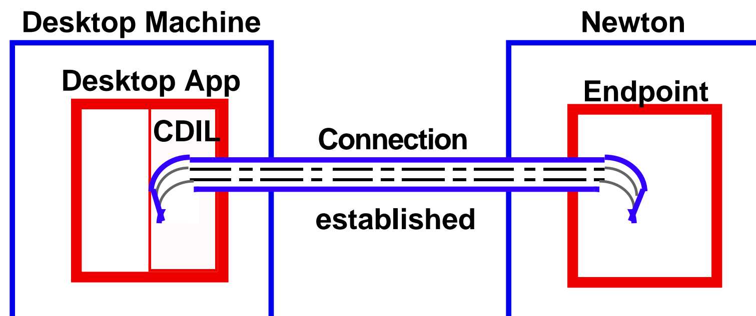

Figure 1: DILs and Newton Devices

The main advantage that DILs provide is that they make it easier to write code for a desktop machine which communicates with a Newton device. In particular, they abstract the Newton connection to a virtual pipe for bytes and provide control over such things as ASCII-to-Unicode conversions and Newton data structures and types such as frames and 30-bit integers.

As shown in Figure 2 there are three DILs which build on one another: CDIL, FDIL and PDIL. The Communications DIL (CDIL) provides basic connectivity to a Newton device and must be used to establish a connection before you can use the

FDIL and PDIL. The Frames DIL (FDIL) provides a relatively simple way to map NewtonScript frames to C structures and also provides a mechanism to handle data which was added dynamically to the frame. The Protocol DIL (PDIL) provides an easy mechanism for synchronizing data between a Newton application and a desktop application. At the time of writing, the PDIL is not yet available but will be available in the future.

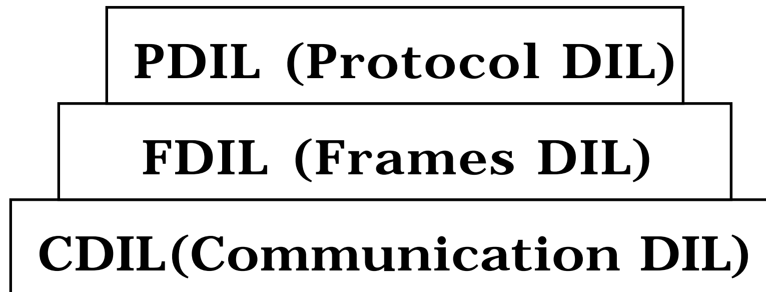

Figure 2: Hierarchical Structure of DILs

All of the DILs are libraries written originally in C++ but called using a C-like syntax with a "magic cookie" object token passed into the calls. On the MacOS side, there are MPW and Metrowerks libraries. On the Windows side, DILs are implemented as DLLs and so should be independent of particular C language implementations.

CDIL

The CDIL essentially has the following phases: initialization, connecting, reading or writing, and disconnecting. This is purposefully very similar to the normal endpoint life cycle. The idea is to create and open a virtual pipe to the Newton device and then communicate using some user-defined protocol by sending and receiving messages or data down the pipe. Figure 3 shows the normal order of calls in using the CDIL.

```
CDInitCDIL()

CDSetAppplication() // Windows only

CDCreateCDILObject()

CDPipeInit()

CDPipeListen()
```

```
CDPipeRead()/CDPipeWrite()

CDPipeDisconnet()

CDDisposeDILObject()

CDDisposeCDIL()
```

Figure 3: CDIL Calls

The `CDInitCDIL()` routine must be called before anything else can be done with the CDIL. On Windows machines the routine `CDSetAppplication()` must be called next. There is no equivalent to this call on the MacOS.

Next, the routine `CDCreateCDILObject()` is called to create a CDIL pipe. It returns a pointer to a pipe which must be used for all subsequent calls which involve that pipe.

`CDPipeInit()` initializes the pipe so that it is "open for business." In particular, it sets the communications options including the media details such as media type (for example, serial, AppleTalk, and so on), and relevant media options (for example, speed of connection, data bits, modem type, and so on).

Next, the pipe uses the `CDPipeListen()` call to wait for a connection from the Newton device. When the Newton device contacts the desktop machine, the application using the CDIL may accept the connection once `CDPipeListen()` returns by calling `CDPipeAccept()`. This allows a connection to be canceled if, for example, the application decides that the actual connect rate was too slow. At any time in this process, the desktop application can cancel an attempted connection by calling `CDPipeAbort()`.

Once a connection is established and working, streams of bytes can be sent and received using the routines `CDPipeRead()` and `CDPipeWrite()`. As with most CDIL routines, these calls may either be made synchronously or asynchronously with a callback routine.

From this point on, the desktop application and the Newton application can engage in an application-specific protocol where there will be an predictable exchange of messages and data via the CDIL's virtual pipeline.

When the decision is made to terminate the connection, the routine `CDPipeDisconnect()` should be called. Once this routine has completed, connection has been broken and both sides must re-establish the connection before data can again be sent or received.

Finally, when the desktop application is completely finished with the pipe, it must call the routines `CDDisposeDILObject()` to tear down the pseudo-object and `CDDisposeCDIL()` to close the CDIL environment. On the MacOS, `CDDisposeCDIL()` closes the Communications Toolbox tool which was opened, and on a Windows machine it closes the appropriate driver.

There are several other additional CDIL calls which may be of use to the desktop programmer. These fall into four categories: encryption, utilities, status, and miscellaneous.

There are two encryption routines: `CDEncryptFunction()` and `CDDecryptFunction()`. These pass callback routines used to the CDIL. These callback routines are called by the CDIL library at the appropriate time to encrypt or decrypt data passing through the pipe. There is no attempt by the CDIL to packetize data and so, if the programmer is using a unit-oriented encryption scheme (for example, cipher block chaining), it is up to the application to buffer the incoming or outgoing data until there is enough data to encrypt or decrypt the block.

The utility routines include such things as `CDFlush()` which is used to flush the contents of the pipe in a given direction, `CDIdle()` which is used to call completion routines passed into asynchronous calls as well as checking on and updating the status of the pipe, and `CDPipeAbort()` which aborts any transactions in a given direction which are pending.

The status routines return information about the pipe. `CDBytesInPipe()` returns the number of bytes currently waiting in a given direction in a particular pipe. `CDConnectionName()` returns the name set by `CDPipeInit()`, `CDGetConfigStr()` returns the media configuration string passed into `CDPipeInit()`, and `CDGetPortStr()` which returns the name of the port the pipe is connected to (for example, "COM2" or "Modem").

`CDGetPipeState() and CDSetPipeState()` get and return the current state of the pipe. Figure 3 shows a list of the possible states of a pipe.

```
kCDIL_Unitialized

kCDIL_InvalidConnection
```

```
kCDIL_Startup

kCDIL_Listening

kCDIL_ConnectPending

kCDIL_Connected

kCDIL_Busy

kCDIL_Aborting

kCDIL_Disconnected

kCDIL_Userstate
```

Figure 3: CDIL Pipe States

The last of the utility class functions is `CDGetTimeout()` which returns the amount of time in milliseconds before a read or write call will time out.

Finally, the miscellaneous category includes two routines which may be useful: `CDPad()` and `CDSetPadState()`. `CDPad()` pads the write buffer so that it is an even multiple of a value passed in as a parameter. This is useful for some packetized protocols or if a unit-oriented encryption scheme is being used. `CDSetPadState()` turns the padding specified by `CDPad()` on or off.

FDIL

The FDIL (also sometimes called HLFDIL for High Level FDIL) is used to support the transfer of NewtonScript objects (frames and arrays) to the desktop. A CDIL connection must be established before FDILs can be used in order to provide the underlying pipeline for FDIL transfers.

Before FDIL calls can be made to move information to or from the Newton device , the FDIL routine `FDInitFDIL()` must be called to initialize the library.

The most common use of the FDIL is to map NewtonScript frames into C structures. If the frame shown in Figure 4 is going to be uploaded to a desktop machine, the desktop application can use FDILs to map this frame into the `fromNewt` C structure shown in the figure.

```
aNewtFrame:={ slot1:'b,

    slot2: {slot3:24,
```

```
        slot4:{ slot5:16,
            slot6:$c}
        }
    slot7: "TROUT"};

struct fromNewt {
    char slot1[5]; // symbols to str
    struct slot2 {
      long subslot3;
      struct subslot4 {
        long subsubslot5;
        char subsubslot6;
      }; // slot4
    }; // slot2
    char slot7[32]; // or max strlen
}
```

Figure 4: Mapping a NewtonScript Frame to a C Struct

To build the mapping between the NewtonScript frame and the C structure, an FDIL object is first created by calling the FDIL routine `FDCreateObject()`. This object acts as the central linkage for all items in the frame or array being sent or received. To map the slots in a frame or elements in an array repeated calls to `FDbindSlot()` are made. These calls match a Newton slot name (or an array object) to a C variable or buffer.

In the case shown, there would be repeated calls to `FDbindSlot()` each of which would specify a slot name of an element in the frame, the address of a memory location the slot value will be copied into, and the FDIL object which keeps all of the frame mappings. Once this binding in completed, the data can be transferred with a single call to `FDget()`.

When the Newton sends the frame data (presumably by calling `Output()`) to the desktop, the FDIL will move the data into the locations on the desktop

machine specified in the bind calls. The FDIL object created keeps track of all bindings previously made so that when `FDget()` is called, the FDIL knows where each piece of incoming data should be put in the desktop machine's memory.

If data is being sent from the desktop machine to the Newton device, the desktop application would call `FDput()` to send the data at the addresses specified by the `FDbindSlot()` calls to the Newton device in a flattened frame format which the Newton OS can understand. In this case, on the Newton side it would be expected that an `inputSpec` would have been established which expected a data form of `'frame`.

`FDget()` and `FDput()` may be called asynchronously. If an asynchronous call of these routines is made, it is important to remember that the memory to which the data is bound must still be available when the completion routine is called. In particular, memory should not be allocated using local variables since the stack of the subroutine making an asynchronous get or put call will disappear when the program exits the subroutine and, on the MacOS, it is also necessary to avoid using references to unlocked handles since heap compaction could cause memory blocks to move.

The steps for creating bound frames are shown in Figure 5.

1. If not previously opened, open CDIL pipeline.
2. Allocate memory on desktop for frame values.
3. Initialize FDIL.
4. Create the FDIL objects for each frame or array object.
5. Bind the slots to map to or from Newton frames to memory locations.
6. Get or put data to FDIL object.
7. When done with FDIL objects, dispose of them.
8. When done with connection, close CDIL pipe.

Figure 5: FDIL Cookbook For Bound Data

This is the easiest and most efficient way to move data to and from the Newton device. However, since NewtonScript frames can change dynamically, there needs to be a way to get information from the Newton

device which the desktop application may not have know about when the binding code was written. This is done by transferring data to the desktop machine into a tree structure which can be parsed by the desktop machine. This is called *unbound data* and is created whenever data is received on the desktop machine which was not actually bound to a location in the desktop memory.

As before, a CDIL connection must be established and then a call to `FDget()` will transfer any data. If any of the data transferred is bound it will be put in the appropriate location. Otherwise it goes into dynamically allocated memory as unbound data.

To access unbound data, the routine `FDGetUnboundList()` gets a pointer to the start of a tree structure which holds it. This tree is organized with a branch for each element of the NewtonScript object. This tree structure is a linked list of elements each of which in turn have a list of sub-elements if the particular branch represents a frame or array. For example, if the Newton frame shown in Figure 4 was brought into the desktop machine in an unbound form, there would be three primary branches. Branch 2 would have two sub-branches and so on. The structure of a generic tree is shown in Figure 6.

The unbound tree shown in Figure 6 is implemented as a linked list of `slotDefinition` structures with the siblings accessed by using the `next` pointer and the children accessed by using the `children` pointer. The `slotType` and `slotName` members describe what type of NewtonScript object a particular piece of unbound data is and what its name is (if it is named). The `varType` member gives the type of the object once it has reached the desktop machine. Meanwhile, the `var` member points to the actual data object on the desktop machine.

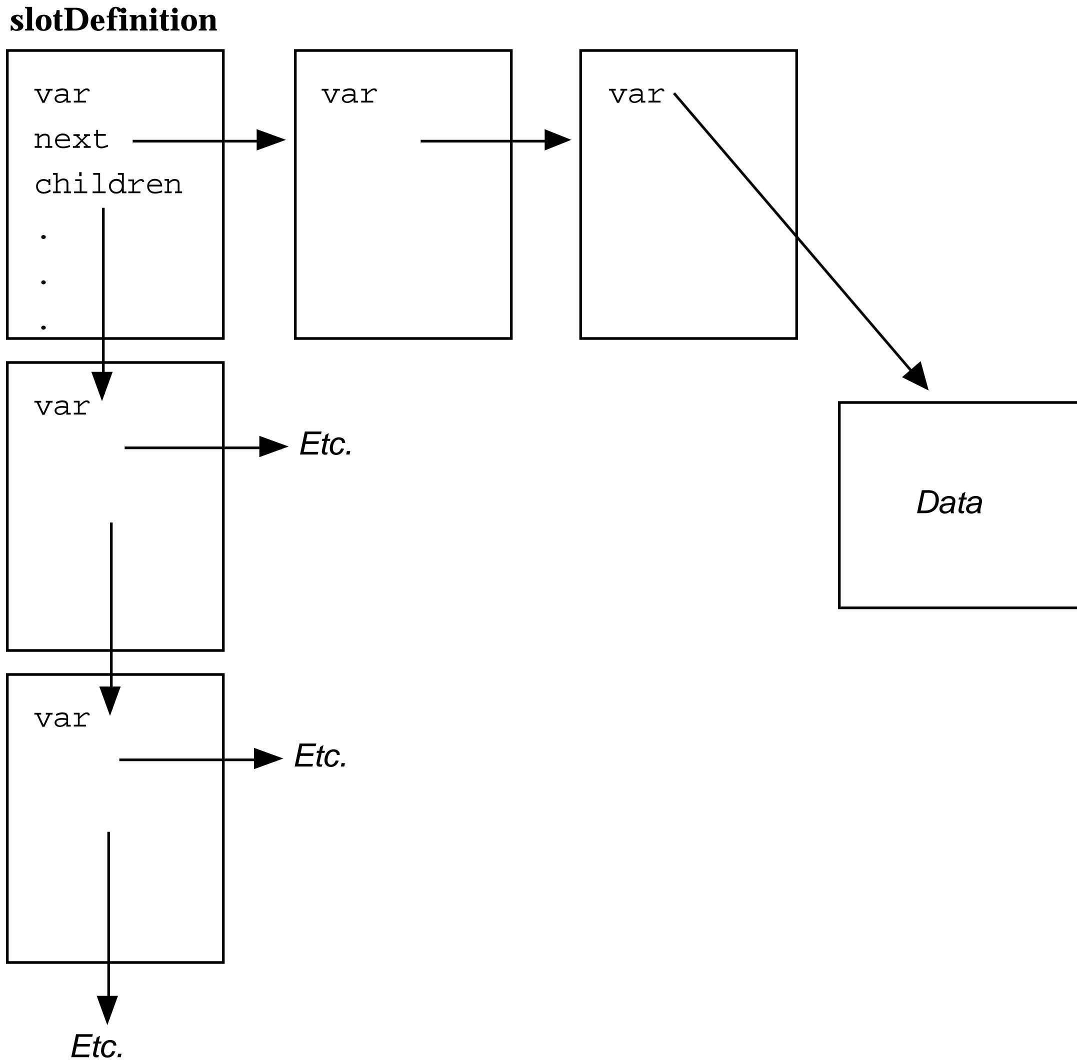

Figure 6: Unbound Data in slotDefinitions

To parse a tree, a program would start with a pointer to the first unbound object returned by `FDGetUnboundList()`. Using this, the program would check whether the object was a leaf (that is, actual data), and if so, copy or use it as desired.

If the object was not a leaf, the children members would be checked and all children of the object evaluated in the same way. Once the children were all evaluated, the next step is to go on to the next sibling object and repeat the

process. When all the siblings of the topmost object have been evaluated, the tree has been completely parsed.

After the desktop application is done with unbound data it should be disposed of by calling `FDFreeUnboundList()`.