CDIL 2.0

Introduction

CDIL is a small library for 68K, PPC, and x86 applications that need to communicate with Newton OS devices. Operations are provided via a C API. The data exchanged is free-form. That is, the CDIL does not impose any sort of data format, nor does it imply any high- or application-level protocols. All it provides is a stream-based communications API for sending data to and receiving data from a Newton OS device. This API turns around and works with transport specific APIs (such as ADSP, TCP/IP, and MNP) to transfer the data.

CDIL concepts

The CDIL uses the concept of a stream-based communications pipe. A pipe can be thought of as a channel through which data is sent. A pipe between two entities (one on the desktop, one on the Newton OS device) is established, used for communications, and then torn down.

The connection is established using a client/server model. The desktop application using the CDIL can be thought of as the server. Its job is to create a pipe and put it into “listening” mode. Once in this mode, the pipe will listen for a connection request from a Newton OS device. Because the desktop device is in the mode of servicing a request from a Newton OS device, it is considered the “server”, while the Newton OS device is considered the “client”. Once the desktop application is informed that there is a Newton OS device requesting a connection, it can accept the connection, thus establishing a two-way path of communications.

Once the connection is established, the desktop application writes bytes into the pipe, which are then transmitted to the Newton OS device. Similarly, bytes sent from the Newton OS device to the desktop are read from the pipe by the desktop application.

Once all data has been exchanged, the desktop and Newton OS devices agree to disconnect via whatever high-level protocol the two are using. Next, the desktop application typically waits for the connection to be broken. The Newton OS application then breaks the connection. The desktop application detects that the connection has been broken, and then closes the connection on its side. At that point, both sides are fully disconnected.

Examples

Using the CDIL is very simple: initialize the library (usually at the start of your application), create a pipe for the desired transport (serial, AppleTalk, TCP, etc.), open a connection on that pipe, read data from and write data to that pipe, close the connection, and close the library (usually at the end of your application).

A working example (sans error checking) is shown below:

Copyright © 1997, Newton, Inc.
All rights reserved

```
CD_Handle   pipe;
CD_State    state;
long        count;
char        dataBuffer[256];

CD_Startup();                       // Initialize the library
CD_CreateADSP(&pipe, NULL, NULL);   // Create a connection object
CD_StartListening(pipe);            // Have that object listen for a
                                    // connection from a Newton OS
                                    // device

while (CD_GetState(pipe) == kCD_Listening)// Wait for a connect request
{
      // If you are displaying a dialog box telling the user to
      // initiate a connection from a Newton OS device, you could
      // also check for clicks on a Cancel button here.
}

if (CD_GetState(pipe) == kCD_ConnectPending)
{
      CD_Accept(pipe);              // Accept the connect request

      MyGetDataToSend(dataBuffer);

      CD_Write(dataBuffer, sizeof(dataBuffer), kCD_DefaultTimeout);

      // This step is optional. We'd execute it if we wanted to
      // ensure that there were 100 bytes available before calling
      // CD_Read, which would otherwise block.
      //

      do {
            CD_Idle(pipe);
            CD_BytesAvailable(pipe, &count);
      } while (count < 100);

      CD_Read(dataBuffer, 100, kCD_DefaultTimeout);
                                    // Assumes we expect 100 bytes back

      CD_Disconnect(pipe);          // Break the connection.
}

CD_Dispose(pipe);                   // Delete the pipe object

CD_Shutdown();                      // Close the library
```


Copyright © 1997, Newton, Inc.
Subject to change *All rights reserved*

Data Types and Values

`CD_Handle`

In order to establish a connection and communicate through it, a client must first call the appropriate CDIL function to create a "pipe". This pipe is of type `CD_Handle`. The pipe is passed as the first parameter to most of the rest of the CDIL functions. It is an opaque data type; there are no user-accessible field in it.

```
typedef enum CD_State
{
      kCD_Uninitialized,
      kCD_Disconnected,
      kCD_Listening,
      kCD_ConnectPending,
      kCD_Connected,
      kCD_DisconnectPending
} CD_State;
```

An enumerated set of values indicating the pipe's current state. You can get the pipe's state by calling CD_GetState at any time. The values have the following meanings:

`kCD_Uninitialized`	A NIL `CD_Handle` was passed to `CD_GetState`.
`kCD_Disconnected`	The pipe was just created, or has had `CD_Disconnect` called on it.
`kCD_Listening`	The pipe has just had `CD_StartListening` called on it.
`kCD_ConnectPending`	The pipe has had `CD_StartListening` called on it and the Newton OS device has indicated its interest in connecting, but the desktop client has not yet called `CD_Accept`.
`kCD_Connected`	The pipe has had `CD_Accept` called on it and is now fully connected. The pipe can now be used in `CD_Read` and `CD_Write` calls.
`kCD_DisconnectPending`	The CDIL has detected that the other end of the pipe has closed. The pipe can no longer be used for communications and should have `CD_Disconnect` called on it.

`kCD_DefaultTimeout`
`kCD_NoTimeout`

Some CDIL functions take timeout values, specified in seconds. You can either specify a custom timeout value, or you can use either `kCD_DefaultTimeout` or `kCD_NoTimeout` to specify standard timeout values. Currently, the default timeout value is 30 seconds.

Copyright © 1997, Newton, Inc.
All rights reserved

Function Reference

```
DIL_Error CD_Startup(void);
```

Initializes the CDIL. This call makes sure that any low-level transport layers (e.g., ADSP, TCP/IP, MNP) are available and properly initialized. If none are available or none can be initilaized, this function returns `kCD_ServiceNotAvailable`.

This function is usually called once at the start of your program. However, you can call it as many times as you want as long as you call `CD_Shutdown` an equal number of times.

```
DIL_Error CD_Shutdown(void);
```

Deinitializes and closes any transport layers opened and initialized in `CD_Startup`.

This function must be called once for every time you called `CD_Startup`. Usually, you just call it once at the end of your program. However, you can call it as many times as you want, as long as you don't call it more times that you've called `CD_Startup`.

```
DIL_Error CD_HasADSP();
DIL_Error CD_HasCTB(const char* toolName);
DIL_Error CD_HasMNPSerial();
DIL_Error CD_HasTCP();
```

These functions return a value indicating whether or not the particular service is available. They provide an indication of whether or not the corresponding `CD_Create…` function will return successfully. These functions can be used when preparing your preferences dialog for display. If the CDIL on your platform is coded to support a service, and a service is available (that is, any underlying libraries can be loaded and/or initialized), its `CD_Has…` function returns `kDIL_NoError`.

```
DIL_Error CD_GetSerialPortName(long index, char* buffer, long* bufLen);
```

Returns a user-displayable string containing the name of a selectable serial port. Examples for the Macintosh would be "Printer Port", "Modem Port", and "Printer-Modem Port". Examples for Windows would be "COM1:", "COM2:", "COM3:", … well, you get the idea.

`index` is a zero-based value indicating the port for which you want a string. `buffer` is a block of `*bufLen` bytes of memory reserved by the caller. If `buffer` is not NULL, `CD_GetSerialPortName` will transfer `*bufLen` bytes into `buffer`. Regardless of the value of `buffer`, it will set `*bufLen` to the number of bytes required to hold the entire string (including a NULL terminator). If `*bufLen` is not large enough, then `CD_GetSerialPortName` will return `kCD_BufferTooSmall`. If `index` becomes too large, then `CD_GetSerialPortName` will return

Copyright © 1997, Newton, Inc.
Subject to change *All rights reserved*

`kCD_IndexOutOfRange` and the contents of `buffer` will be unchanged.

```
DIL_Error CD_CreateADSP(CD_Handle*, const char* name, const char* type);
```

Creates an ADSP-based communications pipe. The address of the variable to recieve the created pipe is passed as the first parameter. The name of the ADSP connection is passed as the second parameter. This string is what appears in the Chooser list on the Newton OS device. If you pass NULL for this parameter, the CDIL will use a default name based on your desktop computer's preferences (for instance, on a Macintosh, it will use the strings specified in the File Sharing control panel). The connection type is passed as the third parameter. This is searched for by the Chooser on the Newton OS device. If you pass NULL for this parameter, the CDIL will use the type specified by the Connection/Dock application.

```
DIL_Error CD_CreateCTB(CD_Handle*, const char* toolName, const char*
                       configString);
```

Creates a CommToolbox-based communications pipe. The name of the tool is passed in `toolName`. The tool-dependent configuration string is passed in `configString`.

```
DIL_Error CD_CreateMNPSerial(CD_Handle*, long port, long baud);
```

Creates a serial communications pipe based on the MNP protocol. MNP is a packet-based protocol that ensures delivery of your data using compression and error correction. The address of the variable to recieve the created pipe is passed as the first parameter. The serial port to use is passed as the second parameter.

The baud rate at which you which to communicate is passed as the third parameter. This value is expressed in terms of bytes per second. Currently, the set of possible values are:

Windows:

110	4800	56000
300	9600	57600
600	14400	115200
1200	19200	128000
2400	38400	256000

Macintosh:

110	2400	19200
300	4800	38400
1200	9600	57600

Note: Not all of these baud rates are compatible with current Newton OS devices. They merely represent what is possible on the desktop platform.

Copyright © 1997, Newton, Inc.
All rights reserved

```
DIL_Error CD_CreateTCP(CD_Handle*, long port);
```

Creates an TCP-based communications pipe. The address of the variable to recieve the created pipe is passed as the first parameter. The TCP port to use is passed as the second parameter. If you pass zero for this parameter, the CDIL will use the port number used by the Connection/Dock application.

```
DIL_Error CD_Dispose(CD_Handle);
```

Disposes of a communications pipe created by CreateADSP, CreateSerial, or CreateTCP. After this call, the reference to the pipe is invalid and should no longer be used.

The pipe passed to `CD_Dispose` can be in any state. If appropriate, the pipe will be disconnected or will remove itself from a listening state before it is deleted.

```
DIL_Error CD_Disconnect(CD_Handle);
```

Returns the specified pipe to a `kCD_Disconnected` state. If the pipe is listening, it will stop listening. If the pipe is connected, it will be disconnected. In all cases, the state of the pipe after making this call is `kCD_Disconnected`. Additionally, any internally buffered data is flushed and can no longer be read with `CD_Read`.

```
DIL_Error CD_StartListening(CD_Handle);
```

Causes a pipe to start listening for a connection request from a Newton OS device. Only pipes in the `kCD_Disconnected` state should be passed to `CD_StartListening`. After the successful completion of this call, the pipe will be in the `kCD_Listening` state.

```
DIL_Error CD_Accept(CD_Handle);
```

Causes a pipe to accept a pending connection. Only pipes in the `kCD_ConnectPending` state should be passed to `CD_Accept`. After the successful completion of this call, the pipe will be fully connected, its state will be `kCD_Connected`, and it can be used to exchange data with a Newton OS application.

```
DIL_Error CD_Read(CD_Handle, void* p, long count);
```

Read bytes from a pipe. `count` number of bytes are read from the pipe and written to the memory location specified by `p`.

Note that a pipe need not be connected in order for bytes to be read from it. It is possible for a pipe to have buffered data received from a Newton OS device before the connection was broken. As long as the pipe's state is `kCD_Connected` or `kCD_DisconnectPending`, clients of the CDIL are still able to retrieve these bytes.

Copyright © 1997, Newton, Inc.
All rights reserved

```
DIL_Error CD_BytesAvailable(CD_Handle, long* count);
```

Returns the number of bytes available for reading from the pipe. Read requests for more bytes than are available run the risk of blocking, as additional bytes need to be read from the Newton OS device.

Note that a pipe need not be connected in order for bytes to be read from it. It is possible for a pipe to have buffered data received from a Newton OS device before the connection was broken. As long as the pipe's state is `kCD_Connected` or `kCD_DisconnectPending`, clients of the CDIL are still able to retrieve these bytes.

```
DIL_Error CD_Write(CD_Handle, const void* p, long count, long timeout);
```

Sends the given bytes to the Newton OS device. Only pipes in the `kCD_Connected` state can send data.

```
DIL_Error CD_FlushOutput(CD_Handle);
```

To increase performance, the CDIL buffers all outgoing data. This data remains in the desktop computer until it is explicitly or implicitly sent to the Newton OS device. To explicitly send the buffered data, call `CD_FlushOutput`. Otherwise, the data will be implicitly sent on the next call to `CD_Idle`, `CD_Read`, `CD_Disconnect`, or `CD_BytesAvailable`.

```
DIL_Error CD_Idle(CD_Handle);
```

This function should be called periodically to allow the CDIL to service any open connections. Typically, this mean detecting changes in the pipe's state and buffering any data received from the Newton OS device. Failure to call `CD_Idle` frequently may result in loss of incoming data.

How frequently you should call this depends on the amount of data to be transferred and how often it is transferred, and needs to be determined by experimentation. If you're losing data, you can try calling `CD_Idle` more frequently. On the other hand, if you call `CD_Idle` more than necessary, it can slow your application.

```
CD_State CD_GetState(CD_Handle);
```

Updates and returns the state of the pipe. Note that in some cases, the returned value reflect the state of the pipe only at the instant in which you made the call. There is no guarantee that two calls to `CD_GetState` made one right after the other will return the same value. In particular, the state can always change from `kCD_Listening` to `kCD_ConnectPending` or `kCD_DisconnectPending`, or from `kCD_Connected` to `kCD_DisconnectPending`.

Copyright © 1997, Newton, Inc.
All rights reserved

```
DIL_Error CD_SetTimeout(CD_Handle, long timeoutInSeconds);
```

When the CDIL pipe is created, it is initialized with a default timeout period of 30 seconds. This timeout period is used to control `CD_Read` and `CD_Write` calls (and, indirectly, any flushing of outgoing data).

You can change this timeout period with `CD_SetTimeout`. "timeoutInSeconds" contains the desired timeout period in seconds. Alternatively, you can specify one of the constants `kCD_DefaultTimeout` (for the default timeout period of 30 seconds), or `kCD_NoTimeout` (for no timeout period; that is, the call will wait indefinitely for the data to be read or written).

For `CD_Read`, if the requested number of bytes are not available after the timeout period, a `kCD_Timeout` error will be returned and no bytes will be transferred.

For `CD_Write`, if no data can be sent after the timeout period, a `kCD_Timeout` error will be returned.

Timeout values are specified on a per-pipe basis.

```
long CD_GetPlatformError(CD_Handle);
```

The CDIL is pretty much a front-end for platform-specific communications services. For instance, on Windows, the CDIL uses the COM serial ports for MNP serial communications, and WinSock for TCP/IP communications. On the Mac, the CDIL uses the Communications Toolbox for MNP serial communications, the AppleTalk drivers for ADSP communications, and Open Transport for TCP/IP communications.

These platform services provide a wide range of error codes. Rather then attempting to map those error codes into a common set of error codes defined by the CDIL, the platform-specific error codes are instead returned by `CD_GetPlatformError`. When a platform-specific communications service encounters an error condition, the CDIL function that caused the error returns `kCD_PlatformError`. If the CDIL client then needs to know the specifics of the error, it can query `CD_GetPlatformError` for the actual error code.

Progression of States

`kCD_Uninitialized`

Meaning: This is the state returned when the CDIL is passed a NIL pipe.
Enter: There is nothing to do to enter this state.
Exit: You exit this state by calling a pipe-creation function, such as `CD_CreateADSP`, `CD_CreateSerial`, or `CD_CreateTCP`. At that point, the pipe's state becomes `kCD_Disconnected`.

`kCD_Disconnected`

Meaning: This is the state returned for a pipe just after it has been created, or after it has been passed to `CD_Disconnect`.
Enter: Create a pipe with `CD_CreateADSP`, `CD_CreateSerial`, or `CD_CreateTCP`, or call

Copyright © 1997, Newton, Inc.
All rights reserved

CD_Disconnect on an already existing pipe.

Exit: You exit this state by calling CD_StartListening. At that point, the pipe's state becomes kCD_Listening.

kCD_Listening

Meaning: This is the state of a pipe that is listening for a connection request from a Newton OS device.

Enter: Call CD_StartListening on an already existing pipe. That pipe's state should be kCD_Disconnected.

Exit: You exit this state in one of three ways: (1) A Newton OS device signals that it is interested in establishing a connection, in which case the pipe's state changes to kCD_ConnectPending; (2) a communications or network error occurs, in which case the pipe's state changes to kCD_Disconnected; (3) the CDIL client cancels the listening operation by calling CD_Disconnect, in which case the pipe's state changes to kCD_Disconnected.

kCD_ConnectPending

Meaning: This is the state of a pipe that has recieved a connection request from a Newton OS device.

Enter: The state of the pipe automatically changes from kCD_Listening to kCD_ConnectPending when the CDIL detects a connection request from a Newton OS device.

Exit: You exit this state in one of two ways: (1) Calling CD_Accept changes the pipe's state to kCD_Connected; (2) calling CD_Disconnect changes the pipe's state changes to kCD_Disconnected.

kCD_Connected

Meaning: The pipe is fully connected to a Newton OS device and can be used for data exchange.

Enter: Call CD_Accept on a pipe whose state is kCD_ConnectPending.

Exit: You exit this state in one of three ways: (1) Calling CD_Disconnect changes the pipe's state to kCD_Disconnected; (2) if the Newton disconnects, the pipe's state changes to kCD_DisconnectPending; (3) if a communications or network error occurs, the state changes to kCD_DisconnectPending.

kCD_DisconnectPending

Meaning: The connection has been broken on the other end. Either the Newton OS device as disconnected, or a communications or network error has occurred. In this state, any buffered data can still be retrieved. The buffered data will be flushed when you call CD_Disconnect.

Enter: The state of the pipe automatically changes from kCD_Connected to kCD_DisconnectPending when the CDIL detects the appropriate conditions.

Exit: You change the state from kCD_DisconnectPending to kCD_Disconnected by calling CD_Disconnect.

Error Codes

Copyright © 1997, Newton, Inc.
Subject to change *All rights reserved*

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

#define kDIL_OutOfMemory            (kDIL_ErrorBase - 1)
#define kDIL_InvalidParameter       (kDIL_ErrorBase - 2)
#define kDIL_InternalError          (kDIL_ErrorBase - 3)
#define kDIL_ErrorReadingFromPipe   (kDIL_ErrorBase - 4)
#define kDIL_ErrorWritingToPipe     (kDIL_ErrorBase - 5)
#define kDIL_InvalidHandle          (kDIL_ErrorBase - 6)

#define kCD_ErrorBase               (kDIL_ErrorBase - 200)

#define kCD_CDILNotInitialized      (kCD_ErrorBase - 1)
#define kCD_ServiceNotSupported     (kCD_ErrorBase - 2)
#define kCD_BadPipeState            (kCD_ErrorBase - 3)
#define kCD_Timeout                 (kCD_ErrorBase - 4)
#define kCD_PipeDisconnected        (kCD_ErrorBase - 5)
#define kCD_IndexOutOfRange         (kCD_ErrorBase - 6)
#define kCD_BufferTooSmall          (kCD_ErrorBase - 7)
#define kCD_PlatformError           (kCD_ErrorBase - 8)

      /* Windows-specific error codes */

#define kCD_TCPCantFindLibraryFns   (kCD_ErrorBase - 20)
#define kCD_TCPInsufficientVersion  (kCD_ErrorBase - 21)
#define kCD_TCPNoSockets            (kCD_ErrorBase - 22)
```

CDIL 1.0 -> 2.0 Conversion Guide

`kCDIL_Uninitialized`	This state previously identified a pipe that had been created but had never been used. Now it identifies a NULL (non-created) pipe.
`kCDIL_InvalidConnection`	This state no longer exists
`kCDIL_Startup`	This state no longer exists
`kCDIL_Listening`	This state is unchanged
`kCDIL_ConnectPending`	This state is unchanged
`kCDIL_Connected`	This state is unchanged
`kCDIL_Busy`	This state no longer exists
`kCDIL_Aborting`	This state no longer exists
`kCDIL_Disconnected`	This state used to identify a pipe that used to be connected, but is no longer. Now it identifies any unconnected pipe, even if it's never been used before.
`kCDIL_Userstate`	This state no longer exists.
`CommErr`	Replaced by `DIL_Error`
`CDInitCDIL`	Renamed to `CD_Startup`

Copyright © 1997, Newton, Inc.
Subject to change *All rights reserved*

`CDDisposeCDIL`	Renamed to `CD_Shutdown`
`CDCreateCDILObject`	Replaced by `CD_Create`… suite of functions
`CDDisposeCDILObject`	Renamed to `CD_Dispose`
`CDPipeInit`	Replaced by `CD_Create`… suite of functions
`CDPipeDisconnect`	Renamed to `CD_Disconnect`
`CDPipeListen`	Renamed to `CD_StartListening`. Function returns immediately; there is no asynchronous operation, so the `timeout`, `completionHook`, and `refCon` parameters have been removed.
`CDPipeAccept`	Renamed to `CD_Accept`
`CDPipeAbort`	Removed
`CDPipeRead`	Renamed to `CD_Read`. The `eom`, `swapSize`, `destEncoding`, `completionHook`, and `refCon` parameters represent functionality that is no longer available and have been removed.
`CDBytesInPipe`	Renamed to `CD_BytesAvailable`. Only the reporting of bytes in the input buffer is supported, so the `direction` parameter has been removed.
`CDPipeWrite`	Renamed to `CD_Write`. The `eom`, `swapSize`, `srcEncoding`, `completionHook`, and `refCon` parameters represent functionality that is no longer available and have been removed.
`CDIdle`	Renamed to `CD_Idle`.
`CDGetPipeState`	Renamed to `CD_GetState`.
`CDSetPipeState`	Removed. User states are no longer supported.
`CDEncryptFunction`	Removed. Encryption is no longer supported at the CDIL level.
`CDDecryptFunction`	Removed. Encryption is no longer supported at the CDIL level.
`CDGetConfigStr`	Removed. Configuration parameters are no longer necessarily specified via a configuration string.
`CDGetPortStr`	Removed. Configuration parameters are no longer necessarily specified via a configuration string.
`CDGetTimeout`	Removed. Timeout values are no longer pipe state variables.
`CDSetApplication`	Removed. The CDIL no longer needs the application's HINSTANCE handle.
`CDFlush`	Removed. This functionality could not be guaranteed for all transport services.
`CDPad`	Removed. This functionality was not required at the CDIL level.
`CDSetPadState`	Removed. This functionality was not required at the CDIL level.

Copyright © 1997, Newton, Inc.
All rights reserved