# Dante
# Connection Protocol

The Connection protocol is used to communicate between the desktop and Newton.

This document should be read in conjunction with DockProtocol.h which defines the constants and structures referenced here.

**NOTE** This protocol supersedes the 1.0 Newton ROM protocol: refer to the Newton 1.0 Connection Protocol document.

## Protocol Overview

Newton communicates with the desktop by exchanging Newton event commands. The general command structure looks like this:

```
ULong    'newt'     // event header
ULong    'dock'     // event header
ULong    'aaaa'     // specific command
ULong     length    // the length in bytes of the following data
UChar     data[]    // data, if any
```

### Note

- The length associated with each command is the actual length in bytes of the data following the length field.
- Data is padded with nulls to a 4 byte boundary.
- Multi-byte values are in big-endian order.
- Strings are null-terminated 2-byte UniChar strings unless otherwise specified.
- NewtonScript objects are sent in Newton Streamed Object Format (NSOF) (see the Newton Formats document, chapter 4).

## Desktop Applications

Several desktop applications that provide connection services to Newton are available, some of them in Apple's archive. They all implement the protocol defined in this document.

Newton Connection Kit (NCK) 1.0

Protocol: 1

Functions: backup, restore, install

Newton Connection Kit (NCK) 2.0

Protocol: 2

Functions: backup, restore, install

Newton Package Installer (NPI) 1.1 released June 20, 1994

Protocol: 1
Functions: install package only

Newton Backup Utility (NBU) 1.0 released January 25, 1996

Protocol: 2
Functions: backup, restore, install

Newton Connection Utilities (NCU) 1.0 released May 13, 1997

Protocol: 2
Functions: backup, restore, synchronize, install, import, export, keyboard passthrough

Ruby Desktop Connection Library (RDCL) 0.2

Protocol: 2 link
Functions: command line utility

Newton Connection for Mac OS X (NCX) 2.0.1 released July 19, 2013

Protocol: 1 & 2 link
Functions: backup, restore, install, import, export, keyboard passthrough, screenshot

## Newton 2.0 Dante Protocol

The protocol described here is a superset of the 1.0 protocol. Most commands from the 1.0 protocol are still valid, but some have been superseded.

In this protocol, once a session has been established the connection remains open and commands may be issued by either Newton or desktop. This is a departure from the 1.0 Newton protocol in which each session accomplished one function then disconnected.

## Command Summary

The following is a summary of the commands that have been added to the 1.0 protocol, and their four-letter definitions:

```
kDDefaultStore             'dfst'
kDAppNames                 'appn'
kDImportParameterSlipResult 'islr'
kDPackageInfo              'pinf'
kDSetBaseID                'base'
kDBackupIDs                'bids'
kDBackupSoupDone           'bsdn'
kDSoupNotDirty             'ndir'
kDSynchronize              'sync'
kDCallResult               'cres'

kDRemovePackage            'rmvp'
kDResultString             'ress'
kDSourceVersion            'sver'
kDAddEntryWithUniqueID     'auni'
kDGetPackageInfo           'gpin'
kDGetDefaultStore          'gdfs'
kDCreateDefaultSoup        'cdsp'
```

| | |
|---|---|
| kDGetAppNames | 'gapp' |
| kDRegProtocolExtension | 'pext' |
| kDRemoveProtocolExtension | 'rpex' |
| kDSetStoreSignature | 'ssig' |
| kDSetSoupSignature | 'ssos' |
| kDImportParametersSlip | 'islp' |
| kDGetPassword | 'gpwd' |
| kDSendSoup | 'snds' |
| kDBackupSoup | 'bksp' |
| kDSetStoreName | 'ssna' |
| kDCallGlobalFunction | 'cgfn' |
| kDCallRootMethod | 'crmf' |
| kDSetVBOCompression | 'cvbo' |
| kDRestorePatch | 'rpat' |
| | |
| kDOperationDone | 'opdn' |
| kDOperationCanceled | 'opca' |
| kDOpCanceledAck | 'ocaa' |
| kDRefTest | 'rtst' |
| kDUnknownCommand | 'unkn' |

Refer to the Newton 1.0 Connection Protocol document for a list of commands used by the original protocol, and to DockProtocol.h for a full list of dock commands.

## Session Initiation

Every session starts like this:

```
      Desktop                      Newton
                             <  kDRequestToDock
  kDInitiateDocking          >
                             <  kDNewtonName
  kDDesktopInfo              >
                             <  kDNewtonInfo
  kDWhichIcons               >                   optional
                             <  kDResult
  kDSetTimeout               >                   optional
                             <  kDPassword
```

If the password sent from the Newton is wrong, the desktop responds with `kDPWWrong`.

```
  kDPWWrong                  >
                             <  kDPassword
```

The password exchange can occur up to 3 times before the desktop gives up.

```
  kDPWWrong                  >
                             <  kDPassword
```

If the desktop decides that the Newton has had enough guesses, a `kDResult` indicating `kDBadPassword` error can be sent instead of a `kDPWWrong`.

```
  kDPassword                 >
                             <  kDResult
```

If the password sent from the desktop is wrong, the Newton signals a `kDResult` indicating `kDBadPassword` error immediately.

If no password has been specified, the key is returned unencrypted, but the password exchange always takes place.

## Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### kDRequestToDock

**Desktop** < **Newton**

```
                                ULong    'rtdk'
                                ULong    length = 4
                                ULong    protocol version
```

The Newton initiates a session by sending this command to the desktop, which is listening on the network, serial, etc. The protocol version is the version of the messaging protocol that's being used by the Newton ROM. The desktop sends a `kDInitiateDocking` command in response.

### kDInitiateDocking

**Desktop** > **Newton**

```
ULong    'dock'
ULong    length = 4
ULong    session type
```

The session type can be one of {none, settingUp, synchronize, restore, loadPackage, testComm, loadPatch, updatingStores}; see the Session type enum in DockProtocol.h. The Newton responds with information about itself.

### kDNewtonName

**Desktop** < **Newton**

```
                                ULong    'name'
                                ULong    length
                                struct   NewtonInfo
                                UniChar  name[]
```

The Newton's name can be used to locate the proper synchronize file. The version info includes things like machine type (e.g. J1), ROM version, etc; see the `NewtonInfo` struct declaration in DockProtocol.h

### kDDesktopInfo

**Desktop** > **Newton**

```
ULong    'dinf'
ULong    length
ULong    protocol version
ULong    desktop type
ULong    encrypted key 1
ULong    encrypted key 2
ULong    session type
ULong    allow selective sync
NSOF     desktop apps
```

This command is used to negotiate the real protocol version. The protocol version sent with the `kDRequestToDock` command is now fixed at version 9 (the version used by the 1.0 ROMs) so we can support package loading with NPI 1.0, Connection 2.0 and NTK 1.0. Connection 3.0 will send this command with the real protocol version it wants to use to talk to the Newton. The Newton will respond with a number equal to or lower than the number sent to it by the desktop. The desktop can then decide whether it can talk the specified protocol or not.

The desktop type identifies the sender – 0 for Macintosh and 1 for Windows.

The password key is used as part of password verification.

Session type will be the real session type and should override what was sent in `kDInitiateDocking`. In fact, it will either be the same as was sent in `kDInitiateDocking` or `kSettingUpSession` to indicate that although the desktop has accepted a connection, the user has not yet specified an operation.

AllowSelectiveSync is a boolean. The desktop should say no when the user hasn't yet done a full sync and, therefore, can't do a selective sync.

DesktopApps is an array of frames that describes who the Newton is talking with. Each frame in the array looks like this:

```
{ name: "Newton Backup Utility", id: 1, version: 1 }
```

There might be more than one item in the array if the Newton is connecting with a DIL app. The built-in Connection app expects 1 item in the array that has id:
  1: NBU
  2: NCU
It won't allow connection with any other id.

## `kDNewtonInfo`

| Desktop | < | Newton | |
|---|---|---|---|
| | | `ULong` | `'ninf'` |
| | | `ULong` | `length = 12` |
| | | `ULong` | `protocol version` |
| | | `ULong` | `encrypted key 1` |
| | | `ULong` | `encrypted key 2` |

This command is used to negotiate the real protocol version. See `kDDesktopInfo` above for more info. The password key is used as part of password verification.

## `kDWhichIcons`

| Desktop | | > | Newton |
|---|---|---|---|
| `ULong` | `'wicn'` | | |
| `ULong` | `length = 4` | | |
| `ULong` | `icon mask` | | |

This command is used to customize the set of icons shown on the Newton. The `icon mask` indicates which icons should be shown; see the Icon mask enum in DockProtocol.h. For example, to show all icons you would use this:

```
kBackupIcon + kSyncIcon + kInstallIcon + kRestoreIcon + kImportIcon +
kKeyboardIcon
```

## `kDSetTimeout`

| Desktop | | > | Newton |
|---|---|---|---|
| `ULong` | `'stim'` | | |
| `ULong` | `length = 4` | | |
| `ULong` | `timeout in seconds` | | |

This command sets the timeout for the connection (the time the Newton will wait to receive data before it disconnects). This time is typically set to 30 seconds.

## `kDResult`

```
        Desktop          < >        Newton
                  ULong     'dres'
                  ULong     length = 4
                  SLong     error code
```

This command is sent by either Newton or PC in response to any of the commands that don't request data. It lets the requester know that things are still proceeding OK.

### kDPassword

```
        Desktop          < >        Newton
                  ULong     'pass'
                  ULong     length = 8
                  ULong     encrypted key 1
                  ULong     encrypted key 2
```

When sent by the Newton, this command returns the key received in the `kDDesktopInfo` message encrypted using the password.

When sent by the desktop, this command returns the key received in the `kDNewtonInfo` message encrypted using the password.

### kDPWWrong

```
        Desktop           >         Newton
        ULong     'pwbd'
        ULong     length = 0
```

If the password sent from the Newton is wrong, the desktop indicates this with a `kDPWWrong` response. If too many attempts at entering a password have been made, the desktop can instead respond with a `kDResult` command indicating a `kDBadPassword` error.

## Sync and Selective Sync (Backup)

After the session is started (see above) these commands would be sent:

```
        Desktop                     Newton
                            <  kDRequestToSync
    kDGetSyncOptions        >
                            <  kDSyncOptions
    kDLastSyncTime          >    this one’s fake (0) just to get the newton time
                            <  kDCurrentTime
    kDSetCurrentStore       >
                            <  kDResult
    kDLastSyncTIme          >
                            <  kDCurrentTime
```

The following would appear only if syncing system info:

```
    kDGetPatches            >
                            <  kDPatches
```

The following would appear only if syncing 1.x style packages on locked 1.x cards:

```
    kDGetPackageIDs         >
                            <  kDPackageIDList
    kDBackupPackages        >
                            <  kDPackage
    kDBackupPackages        >
```

```
                          <  kDPackage
kDBackupPackages          >
                          <  kDResult
```

Note that the above only syncs 1.x style packages on locked 1.x cards. To complete the package sync the packages soup should also by synced.

The sync would continue like this:

```
kDSetSoupGetInfo          >
                          <  kDSoupInfo
kDLastSyncTIme            >
                          <  kDCurrentTime
kDGetSoupIDs              >
                          <  kDSoupIDs
kdGetChangedIDs           >
                          <  kDChangedIDs
kDDeleteEntries           >
                          <  kDResult
kDAddEntry                >
                          <  kDAddedID
kDReturnEntry             >
                          <  kDEntry
```

Repeat the above for each store and soup followed by:

```
kDOperationComplete       >
```

Optionally the desktop could send this instead of the operation complete:

```
kDSyncResults             >
```

## Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### kDSynchronize

| Desktop | < | Newton |
|---|---|---|
| | | ULong 'sync'<br>ULong length = 0 |

This command is sent to the desktop when the user taps the Synchronize button on the Newton. The user wishes to synchronize Newton data with desktop applications.

### kDGetSyncOptions

| Desktop | > | Newton |
|---|---|---|
| ULong 'gsyn'<br>ULong length = 0 | | |

This command is sent when the desktop wants to get the selective sync or selective restore info from the Newton.

### kDSyncOptions

| Desktop | < | Newton |
|---|---|---|
| | | ULong 'sopt'<br>ULong length |

```
                    NSOF      info frame
```

This command is sent whenever the user on the Newton has selected selective sync. The frame sent completely specifies which information is to be snychronized.

```
{  packages: TRUEREF,
   syncAll: TRUEREF,
   stores: [{store-info}, {store-info}] }
```

Each store frame in the stores array contains the same information returned by the kDStoreNames command with the addition of soup information. It looks like this:

```
{  name: "Treasure Island",
   signature: 159604293,
   totalsize: 15982592,
   usedsize: 3346692,
   kind: "Flash storage card",
   soups: ["Names","Notes",...],
   signatures: [411528, 843359,...],
   info: {store-info-frame}
}
```

If the user has specified to sync all information the frame will look the same except there won't be a soups slot--all soups are assumed.

Note that the user can specify which stores to sync while specifying that all soups should be synced.

If the user specifies that packages should be synced the packages flag will be true and the packages soup will be specified in the store frame(s).

### kDSyncResults

```
Desktop          >          Newton
ULong    'sres'
ULong    length
NSOF     sync results
```

This command can optionally be sent at the end of synchronization. If it is sent, the results are displayed on the Newton. The array looks like this:

```
[["store name", restored, "soup name", count, "soup name" count],
 ["store name", restored, ...]]
```

`restored` is true if the desktop detected that the Newton had been restore to since the last sync.

`count` is the number of conflicting entries that were found for each soup. Soups are only in the list if they had a conflict. When a conflict is detected, the Newton version is saved and the desktop version is moved to the archive file.

## File Browsing

File browsing is used by the Newton to select a file to import, a package to load, or a backup to restore. (For synchronize, the process is completely driven from the desktop side.)

After the session has started (see above) these commands would be sent:

```
Desktop                     Newton
                        < kDRequestToBrowse
```

```
kDGetInternalStore      >                     optional
                        <  kDInternalStore
kDResult                >
                        <  kDGetDevices     Windows only
kDDevices               >                     Windows only
                        <  kDGetFilters     Windows only
kDFilters               >                     Windows only
                        <  kDGetDefaultPath
kDPath                  >
                        <  kDGetFilesAndFolders
kDFilesAndFolders->
```

Note that we must start the transaction with a `kDRequestToDock` to force 1.0 and 2.0 versions of Connection to display the correct message.

When the user changes the path by tapping on a folder, picking a new level from the path popup, or picking a new drive on the drive popup in the Dock browser slip:

```
                        <  kDSetPath
kDFilesAndFolders       >
```

On Windows only, when the user changes the drive by picking a drive on the drive popup, the desktop will change the drive and set the directory to the current directory for that drive, and return the new path to the newton:

```
                        <  kDSetDrive
kDPath                  >
                        <  kDGetFilesAndFolders
kDFilesAndFolders       >
```

On Macintosh only, if the folder is an alias, it's like this:

```
                        <  kDSetPath
kDPath                  >
                        <  kDGetFilesAndFolders
kDFilesAndFolders       >
```

When the user taps on the File Info button:

```
                        < kDGetFileInfo
kDFileInfo              >
```

On Macintosh only, if the selected item is an alias, before doing import, getfileinfo, or setpath:

```
                        < kDResolveAlias    name of alias
kDAliasResolved         >                   0 or 1, 0 => can't resolve
```

On Windows only, if the user picks a new filter from the list:

```
                        < kDSetFilter
kDFilesAndFolders       >
```

## Dock Commands

All commands begin with the `'newt'`, `'dock'` event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### **`kDRequestToBrowse`**

| Desktop | < | Newton |
|---|---|---|

```
ULong     'rtbr'
ULong     length
NSOF      file type
```

This command is sent to a desktop that the Newton wishes to browse files on.  File type can be 'import, 'packages, 'syncFiles or an array of strings to use for filtering.

**SIMON'S NOTE** I have never encountered the array of strings.

## `kDGetDevices` Windows Only

**Desktop** < **Newton**

```
ULong     'gdev'
ULong     length = 0
```

This command asks the Windows desktop system to return an array of device names.

## `kDDevices` Windows Only

**Desktop** > **Newton**

```
ULong     'devs'
ULong     length
NSOF      array of device frames
```

This command returns an array of frames describing devices. These are the devices which will appear in the devices popup in the Windows file browsing dialog. Each frame in the array should look like:

```
{  name: "c:mydisk",
   disktype: 1 }
```

where `disktype` is one of (floppy = 0, hardDrive = 1, cdRom = 2, netDrive = 3): see the Desktop disk drive types enum in DockProtocol.h.

A corresponding icon is displayed in the popup. This may not be possible in which case this slot will be optional.

## `kDGetFilters` Windows Only

**Desktop** < **Newton**

```
ULong     'gflt'
ULong     length = 0
```

This command asks the Windows desktop to send a list of filters to display in the file browser. A `kDFilters` command is expected in response.

## `kDFilters` Windows Only

**Desktop** > **Newton**

```
ULong     'filt'
ULong     length
NSOF      array of strings
```

This command returns an array of filters to the Newton. It's sent in response to a `kDGetFilters` command. The filter should be an array of strings which are displayed in the filter popup. If the filter array is `NILREF` no popup is displayed.

## `kDSetFilter` Windows Only

**Desktop** < **Newton**

```
ULong    'sflt'
ULong    length = 4
ULong    index
```

This command changes the current filter being used. A `kDFilesAndFolders` command is expected in return. `index` is a 0 based long indicating which item in the filters array sent from the desktop should be used as the current filter..

### **`kDSetDrive`** Windows Only

| Desktop | < | Newton |
|---|---|---|

```
ULong    'sdrv'
ULong    length
NSOF     drive string
```

This command asks the desktop to change the drive on the desktop and set the directory to the current directory for that drive. The string contains the drive letter followed by a colon e.g. `"C:"`.

### **`kDGetDefaultPath`**

| Desktop | < | Newton |
|---|---|---|

```
ULong    'dpth'
ULong    length = 0
```

This commands requests the desktop system to return the default path. This is the list that goes in the folder popup for the Mac and in the directories list for Windows.

### **`kDSetPath`**

| Desktop | < | Newton |
|---|---|---|

```
ULong    'spth'
ULong    length
NSOF     array of strings
```

This command tells the desktop that the user has changed the path. The desktop responds with a new list of files and folders. The path is sent as an array of strings rather than an array of frames as all of the other commands are for performance reasons. For the Mac, the array would be something like:

```
[ "Desktop", {name:"My hard disk", whichVol:0}, "Business" ]
```

to set the path to "My hard disk:business:". "Desktop" will always be at the start of the list, since that's the way Standard File works. So if the user wanted to set the path to somewhere in the Desktop Folder he would send something like

```
[ "Desktop", {name:"Business", whichVol:-1} ]
```

to set the path to "My hard disk:Desktop Folder:business:"

The second item in the array, will always be a frame instead of a string and will contain an additional slot `whichVol` to indicate to the desktop whether that item is a name of a volume or a folder in the Desktop Folder and if so its volRefNum.

For Windows the array would be something like:

```
[ "c:\", "business" ]
```

to set the path to "c:\business."

### **`kDPath`**

**Desktop** > **Newton**

```
ULong    'path'
ULong    length
NSOF     array of folder frames
```

This command returns the initial strings for the folder popup in the Mac version of the window and for the directories list in the Windows version. It is also returned after the user taps on a folder alias. In this case the path must be changed to reflect the new location. Each element of the array is a frame that takes the form:

```
{  name: "MacintoshHD",
   type: kDesktopDisk,
   diskType: kHardDrive,
   whichVol: 0 }                  // optional - see below
```

where `type` is one of (desktop = 0, file = 1, folder = 2, disk = 3): see the Desktop file types enum in DockProtocol.h. If the type is `kDesktopDisk`, there is an additional slot `diskType` with the values (floppy = 0, hardDrive = 1, cdRom = 2, netDrive = 3): see the Desktop disk drive types enum in DockProtocol.h. Finally, for the second frame in the array i.e. the one after Desktop, there will be an additional slot `whichVol`, which will be a 0 if the item is disk or a volRefNum if the item is a folder on the desktop.

For example, the Mac might send:

```
[  {name: "Desktop", type: kDesktop},
   {name: "My HD", type: kDesktopDisk, diskType: kHardDrive, whichvol: 0},
   {name: "Business", type: folder} ]
```

or for some folder on the desktop it it might send:

```
[  {name: "Desktop", type: kDesktop},
   {name: "Business", type: kDesktopFolder, whichvol: -1},
   {name: "My Folder", type: kDesktopFolder} ]
```

for Windows it might be:

```
[  {name: "c:\", type: kDesktopFolder},
   {name: "Business", type: kDesktopFolder} ]
```

## kDGetFilesAndFolders

**Desktop** < **Newton**

```
ULong    'gfil'
ULong    length= 0
```

This command requests that the desktop system return the files and folders necessary to open a standard file like dialog.

## kDFilesAndFolders

**Desktop** > **Newton**

```
ULong    'file'
ULong    length
NSOF     array of file/folder frames
```

This command returns an array of information that's used to display a standard file like dialog box on the Newton. Each element of the array is a frame describing one file, folder or device. An individual frame would look like:

```
{ name: "Whatever",
  type: kDesktopFolder,
  disktype: 0,                  // optional if type = disk
  whichVol: 0,                  // optional if name is on the desktop
  alias: nil }                  // optional if it's an alias
```

The frames should be in the order in the array that they are to be displayed in on the Newton. For example, the array might look like this:

```
[ {name: "Applications", type: kDesktopFolder},
  {name: "important info", type: kDesktopFile},
  {name: "System", type: kDesktopFolder}]
```

If the type is a disk, then the frame will have an additional slot `disktype` with the values (floppy = 0, hardDrive = 1, cdRom = 2, netDrive = 3): see the Desktop disk drive types enum in DockProtocol.h. Also, if the current location is the desktop, there is an additional slot `whichVol` to indicate the location of the inidvdual files, folders and disks with the values 0 for disks and a negative number for the volRefNum for files and folders on the desktop.

If the item is an alias there is an `alias` slot. The existence of this slot indicates that the item is an alias.

## kDGetFileInfo

|  |  |  |  |
|---|---|---|---|
| **Desktop** | **<** | **Newton** |  |
|  |  | ULong | 'gfin' |
|  |  | ULong | length |
|  |  | NSOF | filename |

This command asks the desktop to return info about the specified file. See `kDFileInfo` for info about what's returned.

The `filename` is normally a string, but if the selected item is at the Desktop level, a frame

```
{ name:"Business", whichVol:-1 }
```

will be sent instead, to indicate the volRefNum for the file.

## kDFileInfo

|  |  |  |  |
|---|---|---|---|
| **Desktop** |  | **>** | **Newton** |
| ULong | 'finf' |  |  |
| ULong | length |  |  |
| NSOF | info frame |  |  |

This command is sent in response to a `kDGetFileInfo` command. It returns a frame that looks like this:

```
{ kind: "Microsoft Word document", size: 20480,
  created: 3921837, modified: 3434923,
  icon: <binary object of icon>,
  path: "hd:files:another folder:" }
```

|  |  |
|---|---|
| `kind` | is a description of the file. |
| `size` | is the number of bytes (actual, not the amount used on the disk). |
| `created` | is the creation date in Newton date format. |
| `modified` | is the modification date of the file. |
| `icon` | is an icon to display. This is optional. |
| `path` | is the "user understandable" path description |

### `kDGetInternalStore`

```
        Desktop           <         Newton
                                    ULong    'gist'
                                    ULong    length = 0
```

This command requests the Newton to return info about the internal store. The result is described with the `kDInternalStore` command.

### `kDInternalStore`

```
        Desktop           >         Newton
        ULong    'isto'
        ULong    length
        NSOF     store frame
```

This command returns information about the internal store. The info is in the form of a frame that looks like this:

```
{  name: "Internal",
   signature: 27675205,
   totalSize: 3608096,
   usedSize: 535972,
   kind: "Internal"
}
```

This is the same frame returned by `kDStoreNames` except that the store info isn't returned.

### `kDResolveAlias`

```
        Desktop           <         Newton
                                    ULong    'rali'
                                    ULong    length = 0
```

### `kDAliasResolved`

```
        Desktop           >         Newton
        ULong    'alir'
        ULong    length = 4
        ULong    result: 0 or 1
```

This command is sent by the desktop in response to the `kDResolveAlias` command. If the value is 0, the alias can't be resolved. If the data is 1 (or non-zero) the alias can be resolved.

## Restore Originated on Newton

Restore uses the file browsing interface described above. After the user taps the Restore button on the Newton Dock slip, the following commands are used:

```
        Desktop                     Newton
                               <  kDRestoreFile
    kDResult                   >
                               <  kDGetRestoreOptions
    kDRestoreOptions           >
                               <  kDRestoreOptions
    kDSourceVersion            >
```

Selective restore proceeds as a normal restore would except when it wants to restore a package. In this case it does this:

```
kDRestorePackage        >
                        <  kDResult
```

If the user picks a full restore it proceeds like this:

```
                        <  kDRestoreFile
kDResult                >
                        <  kDRestoreAll
kDSourceVersion         >
```

## Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### kDRestoreFile

**Desktop** **<** **Newton**

```
ULong    'rsfl'
ULong    length
NSOF     filename string
```

This command asks the desktop to restore the file specified by the last path command and the filename. If the selected item is at the Desktop level, a frame

```
{ name:"Business", whichVol:-1 }
```

is sent. Otherwise, a string is sent.

### kDGetRestoreOptions

**Desktop** **<** **Newton**

```
ULong    'grop'
ULong    length = 0
```

This command is sent to the desktop if the user wants to do a selective restore. The desktop should return a `kDRestoreOptions` command.

### kDRestoreOptions

**Desktop** **>** **Newton**

```
ULong    'ropt'
ULong    length
NSOF     restore info frame
```

This command is sent to the newton to specify which applications and packages can be restored. It is sent in response to a `kDRestoreFile` command from the Newton.  If the user elects to do a selective restore the Newton returns a similar command to the desktop indicating what should be restored.

The info frame specifies which applications and packages should be restored:

```
{  storeType: kRestoreToNewton,
   packages: ["pkg1",...],
   applications: ["app1",...] }
```

`storeType` indicates whether the data will be restored to a card or internally to the Newton: see the Backup file origin enum in DockProtocol.h.

### kDRestoreAll

```
        Desktop          <        Newton
                                  ULong    'rall'
                                  ULong    length = 4
                                  ULong    merge
```

This command is sent to the desktop if the user elects to restore all information. `merge` is 0 to not merge, 1 to merge.

### kDRestorePackage

```
        Desktop          >        Newton
        ULong    'rpkg'
        ULong    length
        NSOF     array of packages
```

This command sends all the entries associated with a package to the newton in a single array. Packages are made up of at least 2 entries: one for the package info and one for each part in the package. All of these entries must be restored at the same time to restore a working package. A `kDResult` is returned after the package has been successfully restored.

## File Importing

File importing uses the file browsing interface described above. After the user taps the Import button on the Newton Dock slip, the following commands are used:

```
        Desktop                   Newton
                             <  kDImportFile
     kDTranslatorList        >                    when there's more than one
                             <  kDSetTranslator   translator available
     kDImporting             >
                             <  kDResult
```

When the data is ready to be sent to the Newton:

```
     kDSetStoreToDefault     >
                             <  kDResult
     kDSetCurrentSoup        >
                             <  kDResult
     kDAddEntry              >
                             <  kDAddedID
     kDAddEntry              >
                             <  kDAddedID
     kDSoupsChanged          >
```

### Dock Commands

All commands begin with the `'newt'`, `'dock'` event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### kDImportFile

```
        Desktop          <        Newton
```

```
                                   ULong     'impt'
                                   ULong     length
                                   NSOF      filename string
```

This command asks the desktop to import the file specified by the last path command and the filename string. The response to this can be either a list of translators (if there is more than one applicable translator) or an indication that importing is in progress. If the selected item is at the Desktop level, a frame

```
{ name:"Business", whichVol:-1 }
```

is sent. Otherwise, a string is sent.

### kDTranslatorList

**Desktop** **>** **Newton**

```
ULong     'trnl'
ULong     length
NSOF      array of strings
```

This command returns an array of translators that can be used with the specified file. The list can include DataViz translators and tab templates. The array should be in the order that the translators should be displayed in the list.

### kDSetTranslator

**Desktop** **<** **Newton**

```
                                   ULong     'tran'
                                   ULong     length = 4
                                   ULong     index
```

This command specifies which translator the desktop should use to import the file. The translator index is the index into the translator list sent by the desktop in the `kDTranslatorList` command. The desktop should acknowledge this command with an indication that the import is proceeding.

### kDImporting

**Desktop** **>** **Newton**

```
ULong     'dimp'
ULong     length = 0
```

This command is sent to let the Newton know that an import operation is starting. The Newton will display an appropriate message after it gets this message.

### kDSetStoreToDefault

**Desktop** **>** **Newton**

```
ULong     'sdef'
ULong     length = 0
```

This command can be used instead of `kDSetCurrentStore`. It sets the current store to the one the user has picked as the default store (internal or card).

### kDSoupsChanged

**Desktop** **>** **Newton**

```
ULong     'schg'
ULong     length
NSOF      array of soup change info
```

This command returns information about what was imported into the Newton. Each array element specifies a soup and how many entries were added to it. There will typically be only one frame in the array. The frame will look like this:

```
[ {soupName: "Notes", count: 7},
  {soupName: "Names", count: 3} ]
```

**SIMON'S NOTE** This doesn't appear to be used.

## Package Loading

Package loading uses the file browsing interface described above. After the user taps the Load Package button on the Newton Dock slip, the following commands are used:

```
       Desktop                      Newton
                               <  kDLoadPackageFile
    kDLoadPackage              >
                               <  kDResult
```

### Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

#### kDLoadPackageFile

```
       Desktop          <         Newton
                                  ULong     'lpfl'
                                  ULong     length
                                  NSOF      filename
```

This command asks the desktop to load the package specified by the last path command and the filename string. If the selected item is at the Desktop level, a frame

```
{ name:"Business", whichVol:-1 }
```

is sent. Otherwise, a string is sent.

#### kDLoadPackage

```
       Desktop          >         Newton
       ULong     'lpkg'
       ULong     length
       UChar     package data []
```

This command will load a package into the Newton's RAM. The package data should be padded to an even multiple of 4 by adding zero bytes to the end of the package data.

#### kDGetPackageInfo

```
       Desktop          >         Newton
       ULong     'gpin'
       ULong     length
       NSOF      package name
```

The package info for the specified package is returned. See the `kDPackageInfo` command described below Note that multiple packages could be returned because there may be multiple packages with the same title but different package ids. Note that this finds packages only in the current store.

### **`kDPackageInfo`**

```
        Desktop        <        Newton
                                ULong    'pinf'
                                ULong    length
                                NSOF     array of info frames
```

This command is sent in response to a `kDGetPackageInfo` command. An array is returned that contains a frame for each package with the specified name (there may be more than one package with the same name but different package id). The returned frame looks like this:

```
  { name:             "package name passed in",
    packageSize:      123,
    packageID:        123,
    packageVersion:   1,
    format:           1,
    deviceKind:       1,
    deviceNumber:     1,
    deviceID:         1,
    modTime:          49228866,
    isCopyProtected:  true,
    length:           1723,
    safeToRemove:     true }
```

### **`kDRemovePackage`**

```
        Desktop        >        Newton
        ULong    'rmvp'
        ULong    length
        NSOF     package name
```

This command tells the Newton to delete a package. It can be used during selective restore or any other time.

## Functions Initiated by the Desktop While Connected

With the advent of the new protocol, the Newton and the desktop can be connected, but with no command specified. A command can be requested by the user on either the Newton or the Desktop. Commands requested by the Newton user are discussed above. This section describes the commands sent from the Desktop to the Newton in response to a user request on the desktop.

### Dock Commands

All commands begin with the '`newt`', '`dock`' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### **`kDDesktopControl`**

```
        Desktop        >        Newton
        ULong    'dsnc'
        ULong    length = 0
```

To indicate that the desktop is in control, each of the following commands should be preceded by a `kDDesktopControl` command, to which the Newton does not reply. Control is relinquished when the desktop sends a `kDOperationDone` command.

### **kDRequestToSync**

```
    Desktop          >          Newton
    ULong    'ssyn'
    ULong    length = 0
```

This command is sent when the desktop wants to start a sync operation, when both the Newton and the desktop were waiting for the user to specify an operation.

### **kDRequestToRestore**

```
    Desktop          >          Newton
    ULong    'rrst'
    ULong    length = 0
```

This command is sent when the desktop wants to start a restore operation, when both the Newton and the desktop were waiting for the user to specify an operation.

### **kDRequestToInstall**

```
    Desktop          >          Newton
    ULong    'rins'
    ULong    length = 0
```

This command is sent when the desktop wants to start a load package operation, when both the Newton and the desktop were waiting for the user to specify an operation.

During an install session, packages are loaded with the `kDLoadPackage` command, so the command sequence looks like:

```
    Desktop                     Newton
  kDDesktopControl        >
  kDRequestToInstall      >
                          <  kDResult
  kDLoadPackage           >
                          <  kDResult
  kDLoadPackage           >
                          <  kDResult
  kDOperationDone         >
```

An install session can also be cancelled by either Newton or desktop as usual:

```
    Desktop                     Newton
                          <  kDOperationCanceled
  kDOpCanceledAck         >
```

## Remote Query

All of the commands in this section are based on the NewtonScript query functions. Please see the Newton Programmer's Guide for details about the functions performed by the commands. The query command returns an id representing the query's cursor. Each of the other commands take this cursor as a parameter. Entries are returned with the `kDEntry` command.

## Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

### kDQuery

```
        Desktop          >          Newton
        ULong    'qury'
        ULong    length
        NSOF     parameter frame
```

The parameter frame must contain a `querySpec` slot and may contain a `soupName` slot.

Performs the specified query on the current store. The query spec is a full query spec including valid test, etc. functions. The soup name is a string that's used to find a soup in the current store to query. If the soup name is an empty string or a `NILREF` the query is done on the current soup. A `kDLongData` is returned with a cursor ID that should be used with the rest of the remote query commands.

### kDLongData

```
        Desktop          <          Newton
                                    ULong    'ldta'
                                    ULong    length = 4
                                    ULong    long data
```

Newton returns a long value. The interpretation of the data depends on the command which prompted the return of the long value.

### kDCursorGotoKey

```
        Desktop          >          Newton
        ULong    'goto'
        ULong    length = 4 + key size
        ULong    cursor id
        NSOF     key
```

The entry at the specified key location is returned. `NILREF` is returned if there is no entry with the specified key.

### kDEntry

```
        Desktop          <          Newton
                                    ULong    'ntry'
                                    ULong    length
                                    NSOF     soup entry
```

### kDCursorEntry

```
        Desktop          >          Newton
        ULong    'crsr'
        ULong    length = 4
        ULong    cursor id
```

Requests the entry at the current cursor.

### kDCursorMap

```
        Desktop          >          Newton
```

```
ULong    'cmap'
ULong    length = 4 + function size
ULong    cursor id
NSOF     function
```

Applies the specified function to each of the cursor's entries in turn and returns an array of the results. A `kDRefResult` is returned. See MapCursor in NPG.

## kDRefResult

**Desktop** < **Newton**

```
                          ULong    'ref '
                          ULong    length
                          NSOF     result
```

## kDCursorMove

**Desktop** > **Newton**

```
ULong    'move'
ULong    length = 8
ULong    cursor id
ULong    count
```

Moves the cursor forward `count` entries from its current position and returns that entry. Returns `NILREF` if the cursor is moved past the last entry.

## kDCursorNext

**Desktop** > **Newton**

```
ULong    'next'
ULong    length = 4
ULong    cursor id
```

Moves the cursor to the next entry in the set of entries referenced by the cursor and returns the entry. Returns `NILREF` if the cursor is moved past the last entry.

## kDCursorPrev

**Desktop** > **Newton**

```
ULong    'prev'
ULong    length = 4
ULong    cursor id
```

Moves the cursor to the previous entry in te set of entries referenced by the cursor and returns the entry. If the cursor is moved before the first entry `NILREF` is returned..

## kDCursorReset

**Desktop** > **Newton**

```
ULong    'rset'
ULong    length = 4
ULong    cursor id
```

Resets the cursor to its initial state. A `kDResult` of 0 is returned.

## kDCursorResetToEnd

**Desktop** > **Newton**

```
ULong    'rend'
ULong    length = 4
```

```
        ULong     cursor id
```

Resets the cursor to the rightmost entry in the valid subset. A `kDResult` of 0 is returned.

### kDCursorCountEntries

```
        Desktop           >        Newton
        ULong     'cnt '
        ULong     length = 4
        ULong     cursor id
```

Returns the count of the entries matching the query specification. A `kDLongData` is returned.

### kDCursorWhichEnd

```
        Desktop           >        Newton
        ULong     'whch'
        ULong     length = 4
        ULong     cursor id
```

Returns `kDLongData` with a 0 for unknown, 1 for start and 2 for end.

### kDCursorFree

```
        Desktop           >        Newton
        ULong     'cfre'
        ULong     length = 4
        ULong     cursor id
```

Disposes the cursor and returns a `kDResult` with a 0 or error code.

## Keyboard Passthrough

Keyboard passthrough can be initiated by both desktop:

```
        Desktop                    Newton
     kDStartKeyboardPassthrough >
                               <  kDStartKeyboardPassthrough
     kDKeyboardString          >
     kDKeyboardString          >
```

and Newton:

```
        Desktop                    Newton
                               <  kDStartKeyboardPassthrough
     kDKeyboardString          >
     kDKeyboardString          >
```

At any time keyboard passthrough can be cancelled by the desktop:

```
        Desktop                    Newton
     kDOperationCanceled       >
                               <  kDOpCanceledAck
```

or by the Newton:

```
        Desktop                    Newton
                               <  kDOperationCanceled
```

```
kDOpCanceledAck         >
```

### Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

#### kDStartKeyboardPassthrough

**Desktop** **<>** **Newton**

```
            ULong      'kybd'
            ULong      length = 0
```

This command is sent to enter keyboard passthrough mode. It can be followed only by `kDKeyboardChar`, `kDKeyboardString`, `kDHello` and `kDOperationCanceled` commands.

#### kDKeyboardChar

**Desktop** **>** **Newton**

```
ULong     'kbdc'
ULong     length = 4
UniChar   character
UShort    state
```

This command sends 1 unicode character to the Newton for processing.

The keyboard state is defined as follows:
Bit 1 = command key down

**SIMON'S NOTE** The keyboard state appears to be ignored.

#### kDKeyboardString

**Desktop** **>** **Newton**

```
ULong     'kbds'
ULong     length
UniChar   string[]
```

This command sends a string of characters to the Newton for processing. The characters are 2-byte unicode characters in big-endian order and must be null-terminated. If there are an odd number of characters the command should be padded, as usual.

## Miscellaneous Additions

### Dock Commands

All commands begin with the 'newt', 'dock' event header as shown in the general form. For simplicity, that's not shown in the descriptions that follow.

#### kDGetAppNames

**Desktop** **>** **Newton**

```
ULong     'gapp'
ULong     length = 4
```

```
        ULong    what to return
```

This command asks the Newton to send information about the applications installed on the Newton. See the `kDAppNames` description below for details of the information returned. The `what to return` parameter determines what information is returned; see the Info to return with kDAppNames enum in DockProtocol.h.

```
   0: return names and soups for all stores
   1: return names and soups for current store
   2: return just names for all stores
   3: return just names for current store
```

## kDAppNames

**Desktop** < **Newton**

```
                                 ULong    'appn'
                                 ULong    length
                                 NSOF     result frame
```

This command returns the names of the applications present on the newton. It also, optionally, returns the names of the soups associated with each application. The array looks like this:

```
   [{name: "app name", soups: ["soup1", "soup2"]},
    {name: "another app name", ...}, ...]
```

Some built-in names are included. "System information" includes the system and directory soups. If there are packages installed, a "Packages" item is listed. If a card is present and has a backup there will be a "Card backup" item. If there are soups that don't have an associated application (or whose application I can't figure out) there's an "Other information" entry.

The soup names are optionally returned depending on the value received with `kDGetAppNames`.

## kDSetVBOCompression

**Desktop** > **Newton**

```
        ULong    'cvbo'
        ULong    length = 4
        ULong    what to compress
```

This command controls which VBOs are sent compressed to the desktop. VBO can always be sent compressed, never compressed or only package VBOs sent compressed; see the VBO compression enum in DockProtocol.h.

```
   0: don't compress VBOs
   1: compress packages only
   2: compress VBOs
```

## kDRestorePatch

**Desktop** > **Newton**

```
        ULong    'rpat'
        ULong    length
        NSOF     patch data
```

This command is used to restore the patch backed up with `kDGetPatches`. The Newton returns a `kDResult` of 0 (or an error if appropriate) if the patch wasn't installed. If the patch was installed the Newton restarts.

## kDSourceVersion

**Desktop** > **Newton**

```
ULong    'sver'
ULong    length = 16
ULong    version
ULong    manufacturer
ULong    machine type
ULong    patch data
```

This command tells the Newton the version that the subsequent data is from; see the Source OS version enum in DockProtocol.h.

```
kOnePointXData = 1
kTwoPointXData = 2
```

For example, if a 1.x data file is being restored the desktop would tell the Newton that version 1 data is coming. Same for importing a 1.x NTF file. Otherwise, it should indicate that 2.x data is comming. When the connection is first started the version defaults to 2.x. This information is necessary for the Newton to know whether or not it should run the conversion scripts. A `kDResult` command with value 0 is sent by the Newton in response to this command. This commands affects only data added to the Newton with `kDAddEntry` and `kDAddEntryWithUniqueID` commands. In particular, note that data returned by **`kDReturnEntry`** isn't converted if it's a different version than was set by this command.

`manufacturer` and `machine type` tell the Newton the type of Newton that's the source of the data being restored. These are sent at the beginning of a connection as part of the `kDNewtonName` command.

### kDGetPassword

**Desktop** > **Newton**

```
ULong    'gpwd'
ULong    length
NSOF     title string
```

This command displays the password slip to let the user enter a password. The string is displayed as the title of the slip. A `kDPassword` command is returned.

## Protocol Extension Operations

### kDRegProtocolExtension

**Desktop** > **Newton**

```
ULong    'pext'
ULong    length
ULong    command
NSOF     function
```

This command installs a protocol extension into the Newton. The extension lasts for the length of the current connection (in other words, you have to install the extension every time you connect). The function is a NewtonScript closure that would have to be compiled on the desktop. See the Dante Connection (ROM) API IU document for details. A `kDResult` with value 0 (or the error value if an error occurred) is sent to the desktop in response.

### kDRemoveProtocolExtension

**Desktop** > **Newton**

```
ULong    'prex'
ULong    length
ULong    command
```

This command removes a previously installed protocol extension.

## Import Operations

### kDImportParametersSlip

```
Desktop          >          Newton
ULong    'islp'
ULong    length
NSOF     info frame
```

The following is a possible example of  what would be displayed on the Newton following the `kDImportParametersSlip` command:

<missing image>

The slip will, at minimum, display 2 text string fields corresponding to the slip title and a filename.  Up to 5 additional fields, plus the CloseBox, could be  displayed. While the slip is displayed, `kDHello` commands are sent to the desktop. When the user taps on the "Import" button or the CloseBox, a `kDImportParameterSlipResult` is sent to the desktop.  Each of the other 5 fields is shown if the slot defining it exists in the frame parameter.

The frame contains the following slots used to configure the display of the slip:

```
{
  slipTitle: "PDF Import",         //  REQUIRED string for slip title
  fileName: "Results.pdf",         //  REQUIRED name of file being imported
  appListInfo: {
    title: "Import into",          // title above textlist
    listItems: [“Notes”,”Works”],  // name of applications listed in textlist
    selected: [1] },               // array in indexes of items in the
                                      listitems array to select. e.g. [1,3]
                                      would select 1st and 3rd items
  conflictsInfo: {
    text: "string",                // string for labelpicker label
    labelCommands: [“one”,”two”] }, // array of strings
                                   // corresponding to available choices
                                   // in picker list
  datesInfo: {
    title: "string1",              // title above datedurationtextpicker
    text: "string2",               // datedurationtextpicker label
    startTime: 48828712,           // start time (minutes from 1/1/1904)
    stopTime: 48828927 },          // stop time (minutes from 1/1/1904)
  importInfo: {
    title: "Import" ,              // REQUIRED string for button label
    importParametersDoneScript: func() nil } // function object to call
                                   // after button is tapped
}
```

### kDImportParameterSlipResult

```
Desktop          <          Newton
                            ULong    'islr'
                            ULong    length
                            NSOF     result frame
```

This command is sent after the user closes the slip displayed by `kDImportParametersSlip`. The result is a frame containing the following three slots:

```
{
  appList : [“Notes,”Works”],       // strings of the items selected
                                    // in the textlist of applications
  conflicts : "string",             // string of labelpicker entry line
  dates : [48828712, 48828927]      // beginning and ending dates
                                    // of the selected interval
                                    // expressed in minutes
}
```

If the user cancels, the result sent is a `NILREF`.

## Store Operations

### kDSetStoreName

**Desktop** **>** **Newton**

```
ULong    'ssna'
ULong    length
NSOF     name string
```

This command requests that the name of the current store be set to the specified name.

### kDSetStoreSignature

**Desktop** **>** **Newton**

```
ULong    'ssig'
ULong    length
ULong    signature
```

This command sets the signature of the current store to the specified value. A `kDResult` with value 0 (or the error value if an error occurred) is sent to the desktop in response.

### kDGetDefaultStore

**Desktop** **>** **Newton**

```
ULong    'gdfs'
ULong    length = 0
```

This command returns info about the default store. This info is the same as the info returned by the kDGetStoreNames command (see `kDStoreNames` for details). The default store is the one used by `kDLoadPackage`.

### kDDefaultStore

**Desktop** **<** **Newton**

```
                        ULong    'dfst'
                        ULong    length
                        NSOF     store frame
```

This command returns a store info frame describing the default store. This frame contains the same info returned for all stores by the `kDStoreNames` command except that it doesn't include the store info. It contains the name, signature, total size, used size and kind.

### kDCreateDefaultSoup

**Desktop** **>** **Newton**

```
ULong    'cdsp'
ULong    length
UniChar  soup name string
```

This command creates a soup on the current store. It uses a registered soupdef to create the soup meaning that the indexes, etc. are determined by the ROM. A `kDResult` is returned. If no soupdef exists for the specified soup an error is returned.

### kDSetStoreGetNames

**Desktop** > **Newton**

```
ULong    'ssgn'
ULong    length
NSOF     store frame
```

This command is the same as `kDSetCurrentStore` except that it returns the names of the soups on the stores as if you'd send a `kDGetSoupNames` command. It sets the current store on the Newton. A store frame is sent to uniquely identify the store to be set:

```
{  name: "Gilligan’s Island",
   kind: "Flash storage card",
   signature: 734830,
   info: {store-info-frame}      // this one is optional
}
```

A `kDSoupNames` is sent by the Newton in response.

## Soup Operations

### kDSetSoupSignature

**Desktop** > **Newton**

```
ULong    'ssos'
ULong    length
ULong    signature
```

This command sets the signature of the current soup to the specified value. A `kDResult` with value 0 (or the error value if an error occurred) is sent to the desktop in response.

### kDSendSoup

**Desktop** > **Newton**

```
ULong    'snds'
ULong    length = 0
```

This command requests that all of the entries in a soup be returned to the desktop. The Newton responds with a series of `kDEntry` commands for all the entries in the current soup followed by a `kDBackupSoupDone` command. All of the entries are sent without any request from the desktop (in other words, a series of commands is sent). The process can be interrupted by the desktop by sending a `kDOperationCanceled` command. The cancel will be detected between entries. The `kDEntry` commands are sent exactly as if they had been requested by a `kDReturnEntry` command (they are long padded).

### kDBackupSoup

**Desktop** > **Newton**

```
ULong    'bksp'
ULong    length = 4
```

```
        ULong     last known unique id
```

This command is used to backup a soup. The result is a series of commands that includes all entries changed since the last sync time (set by a previous command), all entries with a unique id greater than the one specified, and the unique ids of all other entries to be used to determine if any entries were deleted. The changed entries are sent with `kDEntry` commands. The unique ids are sent with a `kDBackupIDs` command. A `kDBackupSoupDone` command finishes the sequence. If there are any ids > 0x7FFF there could also be a `kDSetBaseID` command. The changed entries and unique ids are sent in unique id sequence. The Newton checks for `kDOperationCanceled` commands occasionally. If the soup hasn't been changed since the last backup a `kDSoupNotDirty` command is sent instead of the ids. A typical sequence could look like this:

```
        Desktop                 Newton
      kDBackupSoup           >
                             <  kDBackupIDs
                             <  kDEntry
                             <  kDEntry
                             <  kDBackupIDs
                             <  kDEntry
                             <  kDSetBaseID
                             <  kDBackupIDs
                             <  kDBackupSoupDone
```

See the definition of the other commands for details.

### kDSoupNotDirty

```
        Desktop          <      Newton
                                ULong    'ndir'
                                ULong    length = 0
```

This command is sent in response to a `kDBackupSoup` command if the soup is unchanged from the last backup.

### kDBackupIDs

```
        Desktop          <      Newton
                                ULong    'bids'
                                ULong    length = -1
                                SShort   encoded ids
```

This command is sent in response to a `kDBackupSoup` command--see that command for command sequence details. The length for this command is always set to -1 indicating that the length is unknown. The ids are specified as a compressed array of 16 bit numbers. Each id should be offset by any value specified by a previous `kDSetBaseID` command (this is how we can specify a 32 bit value in 15 bits). Each id is a number between 0 and 0x7FFF (32767). Negative numbers specify a count of the number of entries above the previous number before the next break (non-contiguous id). The sequence is ended by a 0x8000 word. So, if the Newton contains ids

```
   0, 1, 2, 3, 4, 10, 20, 21, 30, 31, 32
```

the array would look like this:

```
   0, -4, 10, 20, -1, 30, -2, 0x8000
```

Thus we send 8 words instead of 11 longs. Is it worth it? If there are a lot of entries it should be.

### kDSetBaseID

```
        Desktop          <      Newton
```

```
                    ULong     'base'
                    ULong     length = 4
                    ULong     new base id
```

This command sets a new base id for the ids sent with subsequent `kDBackupIDs` commands. The new base is a long which should be added to every id in all `kDBackupIDs` commands until a `kDBackupSoupDone` command is received.

### kDBackupSoupDone

```
Desktop          <          Newton
                            ULong     'bsdn'
                            ULong     length = 0
```

This command terminates the sequence of commands sent in response to a `kDBackupSoup` command.

## Entry Operations

### kDAddEntryWithUniqueID

```
Desktop          >          Newton
ULong     'auni'
ULong     length
NSOF      entry frame
```

This command is sent when the PC wants to add an entry to the current soup. The entry is added with the id specified in the data frame. If the id already exists an error is returned.

**WARNING!** This function should only be used during a restore operation. In other situations there's no way of knowing whether the entry's id is unique. If an entry is added with this command and the entry isn't unique an error is returned.

## General Operations

### kDCallGlobalFunction

```
Desktop          >          Newton
ULong     'cgfn'
ULong     length
NSOF      function name symbol
NSOF      function args array
```

This command asks the Newton to call the specified function and return its result. This function must be a global function. The return value from the function is sent to the desktop with a `kDCallResult` command.

### kDCallRootMethod

```
Desktop          >          Newton
ULong     'crmf'
ULong     length
NSOF      method name symbol
NSOF      method args array
```

This command asks the Newton to call the specified root method. The return value from the method is sent to the desktop with a `kDCallResult` command.

### kDCallResult

```
        Desktop             <          Newton
                                       ULong     'cres'
                                       ULong     length
                                       NSOF      result ref
```

This command is sent in response to a `kDCallGlobalFunction` or `kDCallRootMethod` command. The ref is the return value from the function or method called.

### kDResultString

```
        Desktop             >          Newton
        ULong     'ress'
        ULong     length
        SLong     error code
        NSOF      error string
```

Reports a desktop error to the Newton. The string is included since the Newton doesn't know how to decode all the desktop errors (especially since the Mac and Windows errors are different).

### kDOperationDone

```
        Desktop             >          Newton
        ULong     'opdn'
        ULong     length = 0
```

This command is sent when an operation is completed. It't only sent in situations where there might be some ambiguity. Currently, there are two situations where this is sent. When the desktop finishes a restore it sends this command. When a sync is finished and there are no sync results (conflicts) to send to the Newton the desktop sends this command. Hmm... not quite true.

### kDOperationCanceled

```
        Desktop             < >        Newton
                  ULong     'opca'
                  ULong     length = 0
```

This command is sent when the user cancels an operation. The receiver should return to the "ready" state and acknowledge the cancellation with a `kDOpCanceledAck` command..

### kDOpCanceledAck

```
        Desktop             < >        Newton
                  ULong     'ocaa'
                  ULong     length = 0
```

This command is sent in response to a `kDOperationCanceled`.

### kDHello

```
        Desktop             < >        Newton
                  ULong     'helo'
                  ULong     length = 0
```

This command is sent during long operations to let the Newton or desktop know that the connection hasn't been dropped.

### kDRefTest

```
        Desktop          < >        Newton
                  ULong     'rtst'
                  ULong     length
                  NSOF      object
```

This command is first sent from the desktop to the Newton. The Newton immediately echos the object back to the desktop. The object can be any NewtonScript object (anything that can be sent through object read/write).

This command can also be sent with no ref attached. If the length is 0 the command is echoed back to the desktop with no ref included.

### kDUnknownCommand

```
        Desktop          < >        Newton
                  ULong     'unkn'
                  ULong     length = 4
                  ULong     bad command
```

This command is sent when a message is received that is unknown. When the desktop receives this command it can either install a protocol extension and try again or return an error to the Newton. If the built-in Newton code receives this command it always signals an error. The bad command parameter is the 4 char command that wasn't recognized. The data is not returned.

## Session Termination

### kDDisconnect

```
        Desktop          < >        Newton
                  ULong     'disc'
                  ULong     length = 0
```

This command is sent by either desktop or Newton when the docking operation is complete.

## Compatibility

The protocol version has been incremented so old versions of Newton Connection will no longer work with this version. The reason for this is that although the protocol itself is upwardly compatible, the data structures in other parts of the 2.0 Newton have changed to such a degree that old versions of Newton Connection will no longer work.

However, since it's desirable also to support package downloading from NPI, NCK 1.0 and Connection 2.0 the ROMs will also support the old protocol for downloading packages. To make this work, the 2.0 ROMs will pretend that they are talking the old protocol when they send the `kDRequestToDock` message. If a new connection (or other app) is on the other end the protocol will be negotiated up to the current version. Only package loading is supported with the old protocol.

When a 2.0 ROM Newton is communicating with NPI 1.0, NCK 1.0 or 2.0 Connection the session would look like this:

```
        Desktop                     Newton
                                < kDRequestToDock
```

```
kDInitiateDocking       >                   session type MUST be loadPackage
                        <  kDNewtonName
kDSetTimeout            >                   optional
                        <  kDResult
kDLoadPackage           >
                        <  kDResult
kDDisconnect            >
```