palmsource™

High-Level Communications

Exploring Palm OS®

Written by Christopher Bey
Technical assistance from Alain Basty and Gavin Peacock

Exploring Palm OS: High-Level Communications
Document Number 3115-003
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

About This Document

This book describes the portions of Palm OS® that are involved in high-level communications functions and are not transport specific. These include:

- Connection Manager, which provides a central mechanism for managing Palm OS communications connections
- Exchange Manager, which supports sending and receiving typed data objects
- Personal Data Interchange, which facilitates the exchange of information using standard **vObjects**

This book focuses on the high-level communications managers. It does not cover lower-level transport-specific managers. For information on these managers, see *Exploring Palm OS: Low-Level Communications*.

Who Should Read This Book

You should read this book if you are a Palm OS software developer and you want to do one of the following:

- Establish a communications connection so that your application can send or receive data.
- Create or configure communications **profiles**, which link lower-level communications components, called **plug-ins**, into a complete path that can be used to establish a connection.
- Send or received typed data objects from an application without having to manage the communications connection. The Exchange Manager manages all communications details for you.
- Register your application to receive data objects of a particular type when they arrive on the handheld device.
- Enable your application to read vObjects from an input stream or write vObjects to an output stream. The vObject standard allows applications to exchange standardized data

types such as vCard (virtual business cards) and vCal (calendar and schedule information).

The APIs described in this book are optional, though they can greatly enhance the capabilities of an application. Beginning Palm OS developers may want to delay reading this book until they gain a better understanding of the fundamentals of Palm OS application development. Instead, consider reading *Exploring Palm OS: Programming Basics* to gain a good understanding of event management and *Exploring Palm OS: User Interface* to learn about events generated by standard UI controls. Read this book when you find that you need to enable your application with communications functionality.

What This Book Contains

This book contains the following information:

- Part I, "Connection Manager," contains information on the Connection Manager:
  - Chapter 1, "Connections," on page 3 describes how to establish, manage, and configure connections that use communications components called plug-ins.
  - Chapter 2, "Connection Manager Plug-ins," on page 31 describes the Connection Manager plug-ins that are built into the Palm OS®.
  - Chapter 3, "Connection Manager Reference," on page 49 describes the APIs for working with connections, and for managing and configuring connection profiles.
- Part II, "Exchange Manager," contains information on the Exchange Manager:
  - Chapter 4, "Object Exchange," on page 105 explains how to send and receive typed data objects such as MIME data, databases, or database records.
  - Chapter 5, "Exchange Manager Reference," on page 159 describes the APIs for sending and receiving typed data.

- Part III, “Personal Data Interchange,” contains information on Personal Data Interchange:
  - Chapter 6, “Personal Data Interchange,” on page 217 explains how to read and write vObjects.
  - Chapter 7, “Personal Data Interchange Reference,” on page 247 describes the APIs for reading and writing vObjects.
  - Chapter 8, “Unified Data Access Manager Reference,” on page 285 describes APIs for abstracting read and write access to different kinds of source and destination media, including memory and the Exchange Manager.

Changes to This Book

3115-003

- Minor bug fixes.

3115-002

- Minor bug fixes and editorial corrections.

3115-001

- Initial version.

The *Exploring Palm OS* Series

This book is a part of the *Exploring Palm OS* series. Together, the books in this series document and explain how to use the APIs exposed to third-party developers by the fully ARM-native versions of Palm OS, beginning with Palm OS Cobalt. Each of the books in the *Exploring Palm OS* series explains one aspect of the Palm operating system and contains both conceptual and reference documentation for the pertinent technology.

As of this writing, the complete *Exploring Palm OS* series consists of the following titles:

- *Exploring Palm OS: Programming Basics*
- *Exploring Palm OS: Memory, Databases, and Files*
- *Exploring Palm OS: User Interface*

- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Application Porting Guide*

Additional Resources

- Documentation

  PalmSource publishes its latest versions of documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/
- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training
- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/support/kb/

Part I
Connection Manager

The Connection Manager provides a central mechanism for managing Palm OS® communication connections at a high level.

1

Connections

The Connection Manager API provides a central mechanism for managing Palm OS® communications connections at a high level. This chapter contains the following sections that describe how to use the Palm OS Connection Manager:

- "About the Connection Manager" on page 3, explains concepts you need to know before you can begin using the Connection Manager.
- "Using the Connection Manager" on page 18, describes how to use the functions in the Connection Manager to accomplish common tasks.

For information about the built-in Connection Manager plug-ins, see Chapter 2, "Connection Manager Plug-ins," on page 31.

For detailed information about Connection Manager data types, constants, and functions, see Chapter 3, "Connection Manager Reference," on page 49.

About the Connection Manager

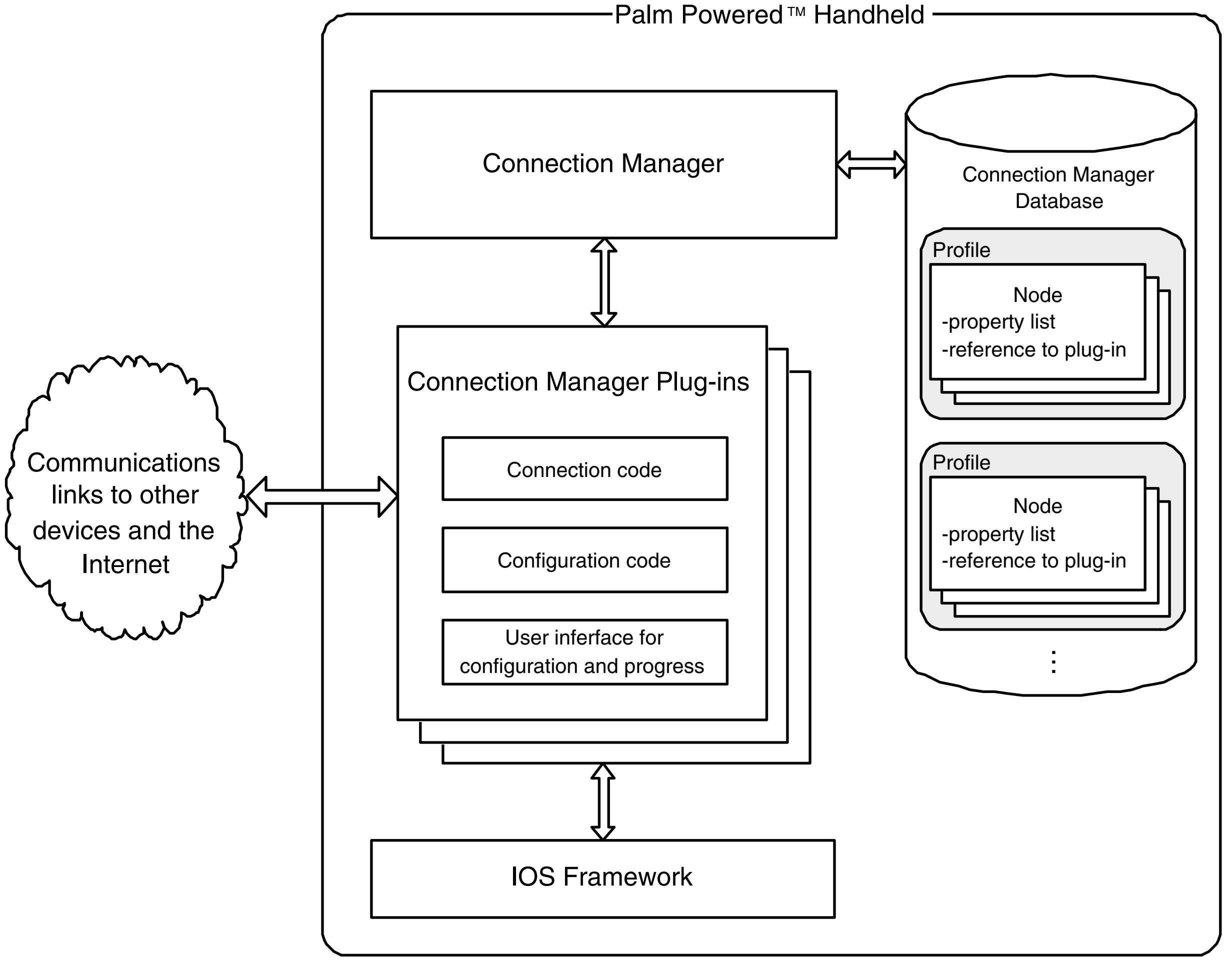
This section explains concepts you need to know before you can begin using the Connection Manager. It discusses the following topics:

- Overview
- Terminology
- Connection Profiles
- Security Considerations
- Persistent Connections
- Graph Management

Overview

The Connection Manager manages at a high level all connections from the Palm Powered™ handheld to external devices. Figure 1.1 shows how the Connection Manager interacts with other components in the communications system.

Figure 1.1 Connection Manager architecture



Communications components are installed in the Palm OS in the form of Connection Manager plug-ins, which handle the work of making, maintaining, and terminating communications connections at a low level. A Connection Manager plug-in is a piece of code that is responsible for configuring and connecting one or more communications components that implement one or more protocols. A plug-in also contains the user interface parts that are necessary to configure its components. Examples of plug-ins include Bluetooth, TCP/IP, PPP, USB, and IrDA.

Connection Manager plug-ins are built on the IOS (Input/Output Subsystem) framework and typically make use of lower-level IOS drivers and modules. Connection Manager plug-ins that are built into the Palm OS are described in Chapter 2, "Connection Manager Plug-ins," on page 31. Licensees and hardware developers can find more information about developing and installing plug-ins in the book *Network Driver Design Guide* in the Palm OS Platform Development Kit (PDK).

The Connection Manager controls the communications plug-ins, requesting them to display configuration dialogs to the user, make connections, and terminate connections.

The Connection Manager provides a user interface framework that invokes configuration dialogs supplied by plug-ins, handles transitions between forms, allows editing of configurations and profiles, and includes progress indicators and error display.

The main functions of the Connection Manager include:

- Managing communications plug-ins, which provide specific communications facilities and protocols to the Palm OS
- Managing communications profiles, a kind of communications stack that links together various plug-ins to provide a complete communications path between an application and a remote service
- Establishing connections by using profiles

The Connection Manager only manages connections and does not send or receive data. You must use APIs provided by the plug-ins or the system, such as the STDIO API, to send and receive data once a connection is established.

Terminology

This section defines terms important to understanding the Connection Manager.

availability: The Connection Manager determines the availability of a profile by querying each plug-in in the profile. If all plug-ins are available then the profile is available. Availability can change depending on system configuration and external circumstances. For example, a Bluetooth connection may be unavailable if there are no other Bluetooth devices within range.

communications component: A piece of code involved in a communications stack. It can implement a protocol or offer services like compression and encryption. A Connection Manager plug-in can represent one or more communications components, and a communications component can be associated with multiple plug-ins.

Connection Manager database: A database that stores records that reference plug-ins, interfaces, and connection profiles. All configuration information is stored in this database.

Connection Manager server: Most of the Connection Manager code runs in the Connection Manager server, a thread that runs in the system process, along with the plug-ins. The Connection Manager library transparently communicates with the server to accomplish operations requested by API calls.

connection type: The protocol used by the top-level component in a profile is the connection type of the profile.

edge: A hierarchical relationship between two components, where one component requires another lower-level one to make a connection. A component can have multiple edges to other components if there are multiple options available. The search algorithm uses edges to find related nodes.

graph: The hierarchical arrangement of nodes (plug-ins or interfaces) from the connection database that describes all the components that can be used for a particular connection type. It may contain multiple paths, each potentially used for a different profile.

interface: A graph node that is an abstraction of plug-ins with similar interfaces. An interface node has no associated code module or user interface components; it is simply an object in the Connection Manager database that relates to other plug-ins.

macro profile: A special profile that is used as a macro reference in other profiles. Macro profile names must begin with the special character `kCncMacroSpecialChar`, which is the open curly brace "{". An example of a macro profile is: `{MyMacro}="IPIF/ILL"`. The reference, {MyMacro} in this example, can be used in another profile and is expanded to its defined value. Macro profiles are not attached to a communications component.

node: A plug-in or interface in the Connection Manager graph, with its associated configuration parameters.

object: A connection profile, plug-in, or interface. A plug-in object doesn't contain the plug-in, but simply references it.

path: A sequence of consecutive edges in the graph.

plug-in: A Connection Manager plug-in is a piece of code that is responsible for configuring and connecting one or more communications components that implement one or more protocols. A plug-in also contains the user interface parts that are necessary to configure its components.

profile: An object that defines an ordering of plug-ins and configuration settings of each referenced plug-in. It is typically stored in the Connection Manager database, but can also be private to an application. A profile has a string equivalent (see **profile string**).

profile string: A string equivalent of a connection profile. It's a list of plug-in names, with some values for their configuration parameters.

record: A connection profile stored in the Connection Manager database.

session: A communication session between the Connection Manager library and the Connection Manager server. A session is automatically opened when calling any Connection Manager functions that need a session. A session is automatically closed when the application that opened it exits. A session identifier can be used to poll a connection.

subprofile: A profile that is referenced by another profile. A subprofile is just a normal profile that is called a subprofile when it is referenced by another profile.

template: A special profile which can be copied to create a new profile.

usability: The user determines the usability of a profile by checking it in a list of profiles for a connection type. Only the checked profiles in the list are usable. When an automatic connection of that type is requested, the Connection Manager considers only usable profiles.

Connection Profiles

A connection **profile** is an object that defines a sequence of plug-in and interface nodes from the Connection Manager graph that forms a complete path from a topmost component to a lower-level component that can establish a physical connection. For example, a profile might consist of these ordered components: TCP/IP, PPP, Dialer. This means that the desired protocol, TCP/IP, requires the use of the PPP component, which in turn requires the use of the Dialer component, which establishes a modem connection.

When making a connection, the application generally interacts only with the topmost component, which defines the connection type—TCP/IP, in this example. The application doesn't need to know about the lower-level connections.

Profiles can be created by the user via the Connection application, programmatically, or by software installation. The Connection Manager also allows modification and management of existing profiles via the Connection application or programmatically. An application can also create and manipulate private connection profiles that are visible only to itself (see "Profile Strings" on page 12).

The Connection Manager database stores records that reference plug-ins, interfaces, and profiles.

A connection profile ID is a unique `uint32_t` value that represents a stored connection profile. This is the typical way to reference stored profiles. Other objects stored in the Connection Manager database, such as plug-ins and interfaces, also have unique IDs.

A connection profile has a priority that is used to rank the profile in a list of profiles that have the same connection type. The highest priority profile of a type (such as TCP/IP profiles) is returned first from searches, and profiles are listed from highest to lowest priority in the list of profiles with the same connection type.

Connection profiles also have an availability. The availability indicates if the profile can be connected at the current time (which is not always possible to determine). For example, a Bluetooth plug-in might be able to detect if the device is within range of a Bluetooth access point, and would set this field accordingly.

The Connection Manager features a fallback mechanism that attempts to make a connection using alternate profiles of the same connection type if the highest priority profile has an unavailable status. Profiles of the same connection type are tried in highest to lowest priority order, and only those that have a status of available are used; or, if the availability of a profile is unknown, it is tried. The user can configure the fallback mechanism via the Connection application.

The following subsections contain more details on profiles:

- Subprofiles
- Profile Configuration
- Automatic and Manual Mode Profiles
- Profile Strings
- Macro Profiles
- Templates

Subprofiles

A profile can contain references to other connection profiles, called subprofiles because they are referenced from another profile. When the profile is needed for connection or other purposes, the references are automatically expanded to the full plug-in or interface sequence that they represent.

For example, you might have a TCP/IP profile where the IP/PPP part of the profile is always the same and the dialing part changes as the user travels to different countries. You could make this change by inserting a reference to a subprofile that contains the correct dialing configuration.

You can expand and compress such subprofiles referenced within a profile. To expand all subprofiles within a profile, use `CncProfileUngroup()`. To recompress a profile that was previously expanded, use `CncProfileRegroupSubmit()`. This function resubmits each subprofile to the Connection Manager database as a separate profile and replaces each one in the main profile with a reference.

You can also substitute one subprofile for another in a locked expanded profile by calling `CncSubProfileAssign()`.

Profile Configuration

Each node in a profile has an associated property list that contains configuration parameters for its component. Because the configuration parameters are stored in the profiles, different profiles that use the same component can specify different parameter values for it.

To be usable, profiles must be configured; that is, each plug-in in the profile must have at least those parameters set that are needed to make a connection. There are three agents that can assign parameters to a node (that is, set parameters in the property list of the plug-in):

- Installation code: The plug-in installation code can define default parameters or find parameter values in the property lists of other nodes during plug-in installation.
- User: The user can set parameters when creating or changing a profile via the Connection application; or the user can be asked to supply missing parameters by the plug-in during a connection attempt.
- Application: An application can programmatically set plug-in parameters.

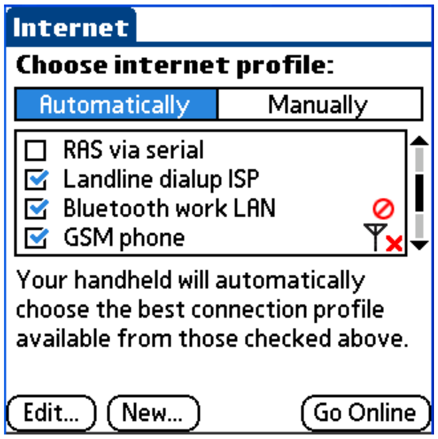
The Connection application serves as a central mechanism for managing and configuring all connections on the handheld. It manages choosing the node path to form complete profiles. And it provides a user interface framework that can display dialogs supplied by plug-ins, for the user to supply configuration parameters or choose among options.

Note that the Connection application itself is not visible in the Launcher. The Internet, Phone, and other communication configuration applications just sublaunch the Connection application, which shows only the relevant profiles.

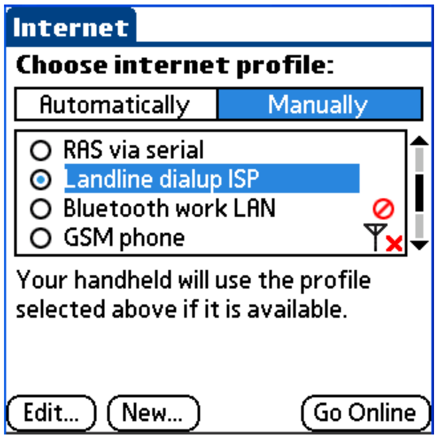
Automatic and Manual Mode Profiles

Profiles can be automatic or manual. In a list of automatic profiles for a particular connection type, the Connection Manager selects a profile to use depending on its usability, availability, and priority. For example, the Internet panel lists several network profiles in priority order. If the user selects Choose Profile Automatically, the system selects the highest priority profile that is usable and available when a network connection is requested.

The user determines usability by checking the boxes next to the profiles that are usable, as shown in the following figure. The Connection Manager determines availability internally, by querying each plug-in in a profile to see if it is available. The red symbols in the figure below indicate unavailable profiles. The circle with a slash is a generic "unavailable" symbol (for profiles that don't supply their own), and the antenna with red X is the unavailable symbol for phone profiles.


Internet
Choose internet profile:
[✓ Automatically] Manually
☐ RAS via serial
☑ Landline dialup ISP
☑ Bluetooth work LAN
☑ GSM phone
Your handheld will automatically choose the best connection profile available from those checked above.

If the user selects Choose Profile Manually, the user must choose a specific profile from the list; this profile is always used when a connection of that type is requested. If the selected profile is not available at the time a connection is requested, then no connection is made.



What makes a profile automatic or manual is a flag (`kCncManualModeOption` in `CncInfoType.options`) that is set in the first plug-in in the profile. This is the plug-in to which the profile is attached. Because all profiles of the same connection type have the same first plug-in, the profiles in a particular list are either all automatic or all manual.

Profile Strings

A connection profile object can be represented textually by a connection profile string that lists the node names and their properties.

An application can have a private profile, not stored in the Connection Manager database or visible to other applications, by saving a profile string.

The general format of a profile string is:

```
"node1:properties/node2:properties/..."
```

where *node1* and *node2* are plug-in and interface nodes in the Connection Manager graph and *properties* are property lists associated with the nodes. Each node is separated from the next by a forward slash (/).

Property lists are pairs of parameter names and values separated by commas. Note the following when writing parameters:

- Binary parameters are encoded using a textual hexadecimal representation enclosed in square brackets (for example, [645a3100]). (The usual 0x prefix is not used.)

- String parameters are enclosed in single quotes.
- Special characters in the names or values of parameters, as well as the equal sign and quote symbols, must be represented with quoted-printable notation. Quoted-printable notation consists of an equal sign followed by a two digit hexadecimal representation of the character's value (for example, =3D). For details on quoted-printable notation, refer to RFC 2045 at: http://www.ietf.org/rfc/rfc2045.txt
- Integers begin with a digit.
- Hexadecimal numbers begin with 0x (for example, 0x4a).

Here is an example of creating a string profile and then storing it in a profile object:

```
sprintf(profile,
"SerialMgr:name='PortCOM1',crea=0x%08X/Serial:Baud=%d,FCtl=0x%X,Bits=0x%X",
'com1', B9600, CRTSCTS, CS8);
profileId = CncProfileDecode("SerialMgrPort1Profile", profile);
```

Macro Profiles

A macro profile is a special type of subprofile whose name begins with the special character `kCncMacroSpecialChar`, which is the open curly brace "{". For example, {MyMacro} could be the name of a macro profile that is defined as follows:

```
{MyMacro} = "IPIF/ILL"
```

When it appears in a profile, a macro name is automatically expanded to its full definition on connection, control, and search requests. You can also expand the macros in a profile by calling `CncProfileUngroup()`.

Say there is a profile defined as follows:

```
MyProfile ="{MyMacro}/DLE"
```

When the profile is expanded, it becomes `"IPIF/ILL/DLE"`, using the definition of MyMacro above.

Macro profiles are not attached to any plug-in or interface. They are used as references in profiles or templates.

The two special macros {REPLACE} and {USING} are used in template profiles created by plug-in developers. Normal profiles don't need to use these.

Templates

Templates are special profiles used when creating a new profile. When creating a new profile, the user typically chooses a template that is copied to a real profile.

A template contains the properties and default settings applicable to the connection type. These can then be customized by the user in the Connection application.

Template profiles are generally created only by plug-in developers.

Link Objects

Link objects are used to associate different names to the same profile. They act like symbolic links or shortcuts; they reference another object and when they are found in a search, the real profile they point to is returned.

This is useful if you create default profiles with internationalized names but also want such profiles to be referenced by others with a well known, unchanging name. For example a link named "RS232 at 115200bps" can point to the profile named "RS232 à 115200bps" in French. If a search finds the profile ID of "RS232 at 115200bps," the profile ID of "RS232 à 115200bps" will be returned.

Security Considerations

The Connection Manager addresses security in a number of ways. The Connection Manager server, which does all the actual work, runs in the system process, separate from the application process where applications run. Access to the system process is restricted, though plug-ins, which also run in the system process, do have access so they can read and send passwords to establish a connection, for example.

Connection passwords, along with other sensitive parameters, can be designated as write-only by plug-ins. Such passwords and parameters cannot be read by applications. Also, the user interface for a plug-in in the Connection application can give the user the choice not to store the password (they must enter it each time).

Plug-in installation is protected by the Security Manager. Plug-in code must be signed to guarantee its authenticity.

Persistent Connections

A persistent connection is a connection that is maintained even when applications switch. For example, an application may open a type of connection that is persistent, then the user may switch to a different application, and the connection is still maintained even though the application that opened it has stopped running.

Persistency is a property of a plug-in, and thus of the profiles attached to that plug-in. All built-in network profiles are persistent. Plug-in developers may define other types of persistent connections.

An application can determine if a persistent network connection exists by checking the `kCncIsConnectedOption` flag in the `CncInfoType.options` field of a profile, as shown in Listing 1.1.

Listing 1.1 Checking for a network connection

```
// Returns true if there is a network connection
Boolean IsOnline(void)
{
  Boolean isOnline = false;
  uint32_t profileID; // ID of the network interface
  status_t err; // Result code
  CncInfoType* profileList = NULL; // the profiles list
  int16_t profileCount = 0; // the profiles count
  int16_t index;

  // get the updated profile list
  // kCncNetOutgoingInterface is defined in NetCnc.h (TCP/IP)
  profileID = CncObjectGetIndex(kCncNetOutgoingInterface);
  if (0 == profileID) return false;
  err = CncObjectFindAll(profileID, kCncFindDefault,
                        &profileCount, &profileList);
  if (errNone != err || 0 == profileCount) return false;
  // update online feature
  for (index = 0; index < profileCount; index++)
    {
```

```
    if ((profileList[index].options &
         kCncIsConnectedOption)!= 0)
      {
       isOnline = true;
       break;
      }
    }
  // free the returned list
  MemPtrFree(profileList);
  return isOnline;
}
```

An application can also register for the `cncNotifyConnectionStateEvent` notification, which is broadcast whenever the connection state of a persistent profile changes (a persistent profile is connected or disconnected on error or on user request, or the availability of the profile changes).

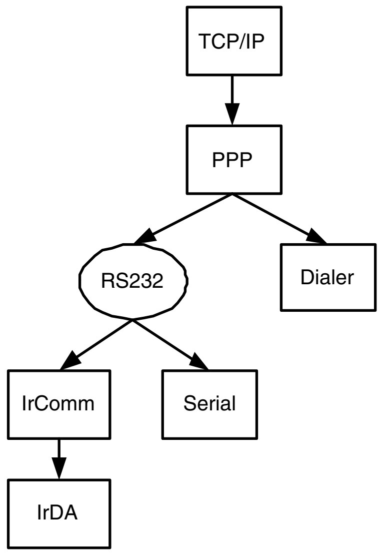
Graph Management

Most application developers won't need to modify the Connection Manager graph. This section is provided as an overview in case it is necessary.

The Connection Manager maintains a graph of connections between all of the installed communication components, which appear as nodes in the graph. This graph is maintained in the Connection Manager database.

The graph represents hierarchical relationships between components, where a higher component requires the use of a component at the next lower level to establish a connection. Figure 1.2 shows a simplified example of such a graph.

Figure 1.2 Connection Manager graph



Nodes in the graph can be plug-ins (boxes in the figure) or interfaces (ovals in the figure).

An **interface** is a node that is an abstraction of similar plug-ins that it logically groups together. For example, the IrComm and serial plug-ins both provide the same RS-232 interface, so they can be abstracted by a general RS-232 interface node. An interface node has no associated code module or user interface components; it is simply an object in the Connection Manager database that relates to other plug-in objects.

An interface can be used to logically group profiles (which are attached to nodes in the graph; see "Adding a Profile to a Connection Type List" on page 25.) For example, a developer might use `CncInterfaceNew()` to create a top-level telnet interface node just to attach some telnet-specific profiles to.

Typically, interfaces are used by plug-in developers and you probably won't need to create them.

To establish a connection, the system requires a complete path from a top component in the graph, which defines the connection type, to the bottom component of any branch. There may be several different possible paths from the top component to the bottom component of a branch, each forming a possible connection profile. A communications component can be used in more than one profile.

Components can be linked together by creating relationships, called **edges**, between them. An edge is simply a hierarchical relationship between two components. It means that the “upper” component requires the “lower” component to make a connection. The upper component may have multiple edges to lower components, meaning that there is a choice of which one to use.

The search algorithm also uses edges internally. For example, if an edge exists between the “RS232” and “Serial” nodes, then searching for “RS232/*” returns profiles attached to the “RS232” node as well as profiles attached to the “Serial” node.

The Connection Manager includes functions to create (`CncEdgeNew()`) and delete (`CncEdgeDelete()`) edges between components, but these are generally unnecessary to use. Edges are typically created or deleted only by plug-in developers.

Using the Connection Manager

The Connection Manager manages all connections at a high level from the Palm Powered™ handheld to external devices.

The Connection Manager is a shared library that the system automatically loads when needed and unloads when not needed. You don’t need to do anything to load or initialize the library.

This section explains how to use the Connection Manager in your application. It covers:

- Making a Connection
- Creating a Profile
- Changing a Profile
- Finding Profiles
- Managing Profiles
- Configuring Components
- Invoking a Function in a Profile Plug-In

Many Connection Manager functions operate on profiles, which are identified by a profile ID number. You can obtain the ID of a profile by passing the name of the profile to `CncObjectGetIndex()` or by searching for it with `CncProfileFindFirst()`, which searches based on name or a partial profile string. Of course, if you create a new profile with `CncProfileNew()` or `CncProfileDecode()`, then that function returns the profile ID.

Making a Connection

You can make a connection in the following ways:

- From an existing stored profile
- By creating a profile dynamically and connecting from it

These methods are discussed in the following sections.

Connecting From a Stored Profile

To make a connection from an existing profile stored in the Connection Manager database, call `CncProfileConnect()`. You pass this function the ID of a stored profile and it makes the connection by requesting each plug-in in the profile to establish a connection, starting with the lowest level. Once the topmost plug-in in the profile has successfully completed a connection, this function returns an IOS file descriptor for the connection.

Applications use the file descriptor to identify the connection for reading and writing data, configuring the connection, or closing it. These tasks can be performed by functions in the STDIO library, which provides a uniform, generic, POSIX-like STDIO interface. For more information, refer to Part VI, "IOS STDIO," in *Exploring Palm OS: Low-Level Communications*.

Normally, the call to `CncProfileConnect()` is synchronous, and blocks until the connection is established. You can call this function in asynchronous mode, however, by setting the `kCncConnectAsynchronous` flag. In this case, you must call `CncConnectReceiveState()` to receive connection progress and termination messages. The calling application may want to use `IOSPoll()` on the session file descriptor (returned by `CncGetSession()`) to know when to call this function.

Here's an example of how to use `IOSPoll()` to poll the file descriptor for a Connection Manager response, and allow normal user interface event processing to continue while polling.

```
#define kNfds 2
EventType event;
struct pollfd fdList[kNfds]; // the fd list
int32_t oNfds; // returned count of fds
status_t err;
CncConnectionStateType state;
fdList[0].fd = CncGetOrOpenSession(); // The CM session fd
fdList[1].fd = EvtGetEventDescriptor(); // The event queue fd
fdList[0].events = POLLIN; // wait for a normal message
fdList[1].events = POLLIN;
while(1) { // loop and poll
  err = IOSPoll(fdList, kNfds, -1, &oNfds);
  if (err != errNone)
    break;
  if (fdList[0].revents & POLLIN) {
    // process connection manager notifications
    err = CncConnectReceiveState(&state);
    // check state and exit loop as needed
  } else if (fdList[1].revents & POLLIN) {
    // process UI events
    do {
      EvtGetEvent(&event, 0);
      // process events normally
    } while(EvtEventAvail());
    EvtFinishLastEvent();
  }
}
```

If you don't have the ID of a profile to connect, you can use the alternative function, `CncProfileFindConnect()`. This function searches the stored profiles by name or partial profile string.

The `CncProfileConnect()` and `CncProfileFindConnect()` functions can have multiple user interface side effects. For example, the user can be prompted to complete or configure the connection profile, if parameter values are missing or the profile does not specify a complete connection path. Some user interface side effects can be allowed or disallowed via the `flags` parameter to these functions.

Creating a Profile Dynamically and Connecting

An application may want to use a private profile, not storing it in the Connection Manager database. To do this, the application can privately construct and store a profile string without using any

Connection Manager functions. To connect using it, pass the string to `CncProfileFindConnect()`. When passed a complete profile string, this function creates a profile object from it and connects using that profile.

Note that calling any connection function can have multiple user interface side effects, as noted in the previous section.

Canceling or Disconnecting a Profile

If you want to cancel the connection process before it is completed, you can call `CncProfileDisconnect()`. You can also call this function to disconnect a persistently connected profile.

Creating a Profile

Users generally create profiles through the Connection application. An application can create a profile programmatically as well.

To create a new empty profile, call `CncProfileNew()`. To add items such as plug-ins, interfaces, and other profiles to it, call `CncProfileInsertItem()`. To submit the changes to the Connection Manager database and unlock the profile, call `CncProfileSubmit()`.

You can also create a profile by passing a string to `CncProfileDecode()`.

You can create a profile dynamically, when you want to make a connection. For details, see the previous section, "Configuring Components."

If you create a profile but don't submit it to the Connection Manager database, it won't be saved after your application exits. However, it is associated with the client session while your application is running and it will be found by the find functions described in "Finding Profiles" on page 23.

For information on configuring the individual communications components in the profile, see "Configuring Components" on page 26.

Changing a Profile

You can change a profile in the following ways:

- Add new components such as plug-ins and interfaces
- Delete components from the profile, or the profile itself
- Launch the configuration application

These tasks are discussed in the following sections.

Adding Components to a Profile

To add components such as plug-ins, interfaces, and other profiles to a profile, call `CncProfileInsertItem()`. Adding a component inserts a reference to it into the sequence of components that comprise a profile.

One profile can even be inserted into another as a subprofile; for details, see "Subprofiles" on page 9.

Changes you make to a profile are not saved until you submit the changes to the Connection Manager database by calling `CncProfileSubmit()`.

Deleting Components and Profiles

To delete a plug-in or interface component and its associated parameters from a profile, call `CncProfileDeleteItem()`.

Changes you make to a profile are not saved until you submit the changes to the Connection Manager database by calling `CncProfileSubmit()`.

To delete a profile from the Connection Manager database, call `CncObjectDelete()`. Removing a plug-in or interface object also removes all edges and profiles that use it. Removing a profile also removes profiles that reference it.

Launching the Configuration Application

To allow the user to complete or edit a profile or configure components, you can launch the Connection application by calling `CncProfileEdit()`. By setting a flag, you can control what form the application displays on launch. There are several options, such as the form that lists connection types, the form that lists profiles available for a particular connection type, the configuration form, the profile creation form, and the profile deletion form.

Finding Profiles

There may be many different profiles stored in the Connection Manager database. You can always find a profile object from its name by using `CncObjectGetIndex()`, but if you don't know its name, you can use the find functions.

There are two ways to find profiles:

- Use `CncObjectFindAll()` to find all profiles that are attached to a particular plug-in or interface. This function returns all found profiles in an array.
- Use `CncProfileFindFirst()` to find a profile based on its name or partial profile string, then iterate through other matching profiles with `CncProfileFindNext()`. This method is explained in more detail in the following paragraphs.

To begin a search for a profile based on its name or a partial profile string, call `CncProfileFindFirst()`. You can pass this function a partial profile string such as "NetOut/*", to find profiles that begin with it. This example finds network profiles. The "/*" characters are a wildcard at the end that signify a partial string. For details about profile strings, see "Profile Strings" on page 12.

The `CncProfileFindFirst()` function returns the first profile that matches the search string and creates a search object that you can query repeatedly by calling `CncProfileFindNext()` to return the next matching profile.

Both `CncProfileFindFirst()` and `CncProfileFindNext()` return locked profiles, so if you don't want to edit the returned profiles, you might want to call `CncProfileUnlock()` in the search loop.

It's good practice to call `CncProfileFindClose()` to clean up memory when you are done with a search, after you have called `CncProfileFindNext()` for the last time. This function frees the memory used for the search object allocated by `CncProfileFindFirst()`.

Note that you can also use the `CncProfileFindFirst()` function to create a new profile. If you specify a complete profile string for the search string, this function creates and returns the ID of a new profile object. However, the profile is automatically deleted when the Connection Manager session closes, unless you call `CncProfileSubmit()` to save it.

If you create a profile but don't submit it to the Connection Manager database, it isn't saved, but it is associated with the client session while your application is running and it will be found by the find functions described in this section. Other applications won't find it, because each application using the Connection Manager server has its own exclusive client session. Only after it is submitted can other applications find it.

Managing Profiles

Profile management involves tasks such as:

- Getting information about profiles and items
- Adding a profile to the list of those available for a particular connection type
- Locking, unlocking, and submitting changes

These tasks are discussed in the following sections.

Getting Information About Profiles and Items

Before you can work with profiles and the communications components in them, you'll need to obtain their IDs, which are unique identifiers assigned to all Connection Manager objects. You can obtain the ID of a profile by passing the name of the profile to `CncObjectGetIndex()` or by searching for it with `CncProfileFindFirst()`, which searches based on its name or a partial profile string.

Most operations on the items in a profile require the index of the item in the profile, which you can obtain by passing its name to `CncProfileGetItemIndex()`. From the index, you can obtain an item's ID by calling `CncProfileGetItemId()`.

To determine the number of items in a profile, use `CncProfileGetLength()`.

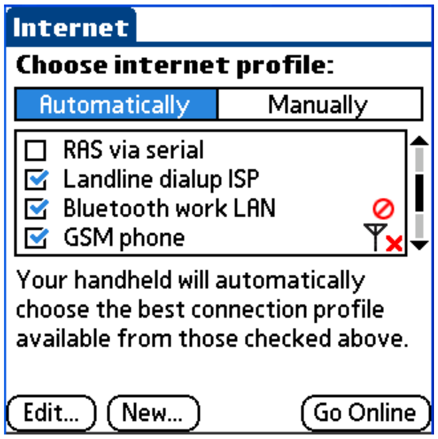
An information structure, `CncInfoType`, is associated with each profile and item object, and is stored in the Connection Manager database. This structure contains information such as the object's name, version, type, and for profile objects, the priority and availability. You can use the functions `CncObjectGetInfo()` and `CncObjectSetInfo()` to get and set the information in this structure.

Adding a Profile to a Connection Type List

On a Palm OS device, users can request connections by type. The type of a connection is identified by the topmost plug-in of its profile. All the profiles that share that topmost plug-in are of the same type. Some of the top-level plug-ins typically include TCP/IP (network), Serial, Bluetooth, and Phone. Each of these connection types might have several associated profiles.

For each connection type, the associated Connection Manager panel lists the various profiles available, so the user can choose which to use. For example, to make a TCP/IP connection, the user might have the choice of profiles that use a dial-up connection to an ISP, a Bluetooth connection to an ISP through a LAN access point, or a GPRS connection through a mobile phone via an infrared link.

Here's an example of a list of possible network connection profiles shown in the Internet panel:



☐ RAS via serial
☑ Landline dialup ISP
☑ Bluetooth work LAN
☑ GSM phone

Your handheld will automatically choose the best connection profile available from those checked above.

To add a new profile to the list for a particular connection type (called “attaching” a profile), use the function `CncProfileAttach()`. For example, calling this function to attach a profile to the TCP/IP plug-in would add it to the list shown above.

The order in which the profile is inserted into the list is determined by its priority (see the `priority` field of `CncInfoType`) relative to the existing profiles in the list. The list is ordered from highest to lowest priority and determines the order in which profiles are returned by `CncProfileFindNext()` when searching for profiles of the same connection type.

You can change the position of a profile in the list by calling `CncObjectMoveItem()`. Note that this doesn't change the priority of the profile, however. The priority is used only once —to initially place the profile in the list.

Locking, Unlocking, and Submitting Profiles

Before you can perform most operations on a profile, such as editing it or configuring its components, you must lock the profile. This prevents other threads from trying to modify the profile at the same time. You can lock a profile by calling `CncProfileLock()`. (Note that `CncProfileUngroup()` also locks a profile.)

If you create a new profile with `CncProfileNew()`, the returned profile is automatically locked.

To save the changes when you are done changing a profile, call `CncProfileSubmit()`. This function submits the changes to the Connection Manager database and unlocks the profile.

If you want to unlock a profile without saving any changes, call `CncProfileUnlock()`.

Locks are counted and if you lock a profile more than once, but only unlock it once, it is still locked.

Locks are cleaned up by the Connection Manager. If your application exits and leaves a profile locked, it will automatically be unlocked.

Configuring Components

The communications components referenced by a profile typically have various parameters that can be set to control some aspect of the communications operation. For example, a TCP/IP plug-in might have parameters that specify the IP address, DNS address, timeout, and so forth.

Values for these parameters are stored in the items that are in a particular profile in the Connection Manager database. You can retrieve the parameter values for an item by calling

`CncProfileGetParameters()` and set the parameter values by calling `CncProfileSetParameters()`. Both of these functions use parameter arrays that you can allocate by calling `CncParametersInit()`.

When you are done with the parameter array returned by `CncProfileGetParameters()`, call `CncParametersFree()` to free the memory used by the array.

To allow the user to complete or edit a profile or configure components, you can launch the Connection application by calling `CncProfileEdit()`. By setting a flag, you can control what form the application displays on launch. There are several options, such as the configuration form, profile creation form, partial profile completion form, etc.

Invoking a Function in a Profile Plug-In

An application may want to programmatically invoke a function in a plug-in associated with a profile. You can do this by calling `CncObjectControl()`. Using this call, you can send a request to the control function in a single plug-in or in all plug-ins in a profile. The control function in a plug-in accepts a request parameter that tells it what to do, and a parameter block that is associated with the request.

Plug-in developers define the requests that a plug-in can respond to. This feature allows plug-ins to expose proprietary features.

One request that all plug-ins should respond to is the `kCncControlAvailability` request, which the Connection application uses to query all plug-ins about their availability.

Summary of Connection Manager

Connection Manager Functions	
Session Open and Close	
`CncCloseSession()`	`CncGetSession()`
`CncGetOrOpenSession()`	

Connection Manager Functions

Profile Management

- CncEdgeDelete()
- CncEdgeNew()
- CncObjectControl()
- CncObjectMoveItem()
- CncProfileAttach()
- CncProfileCopy()
- CncProfileEdit()
- CncProfileInsertItem()
- CncProfileLock()
- CncProfileNew()
- CncProfileUnlock()

Connecting

- CncConnectReceiveState()
- CncProfileConnect()
- CncProfileDisconnect()
- CncProfileFindConnect()

Searching

- CncObjectFindAll()
- CncProfileFindClose()
- CncProfileFindFirst()
- CncProfileFindNext()

Profile/String Conversion

- CncProfileDecode()
- CncProfileEncode()

Plug-in Registration

- CncRegisterPluginModule()

Database Record Operations

- CncInterfaceNew()
- CncObjectDelete()
- CncObjectGetIndex()
- CncObjectGetInfo()
- CncObjectSetInfo()
- CncProfileSubmit()

Parameter Configuration

- CncParametersFree()
- CncParametersInit()

Profile Item Operations

- CncProfileDeleteItem()
- CncProfileGetItemId()
- CncProfileGetItemIndex()
- CncProfileGetLength()
- CncProfileGetParameters()
- CncProfileSetParameters()

Connection Manager Functions	
Subprofile Operations	
`CncProfileRegroupSubmit()`	`CncSubProfileAssign()`
`CncProfileSubmit()`	

2

Connection Manager Plug-ins

This chapter documents the Connection Manager plug-ins that are commonly built into the Palm OS®. If you need to construct custom Connection Manager profiles, you will need to know what plug-ins are available and what their parameters are. This chapter describes the following plug-ins and groups of plug-ins:

- Network Plug-ins
- Serial Plug-ins
- USB Plug-in
- Infrared Plug-in
- Bluetooth Plug-in
- Telephony Plug-ins

A particular Palm OS device may not have all of the Connection Manager plug-ins described here, or may have additional plug-ins, depending on the communication hardware installed in the system.

For information about setting and getting plug-in parameters, see "Configuring Components" on page 26. You can also work with plug-in parameters in profile strings, and this is described in "Profile Strings" on page 12.

Note that plug-in development is not covered in the SDK. Licensees and hardware developers can find more information about developing and installing plug-ins in the book *Network Driver Design Guide* in the Palm OS Platform Development Kit (PDK).

Network Plug-ins

All network profiles start with the NetOut interface. This interface serves as the top-level network component in the system. It is not a plug-in, but rather an interface, below which all other plug-ins in the TCP/IP stack reside.

Below the NetOut interface, the TCP/IP stack includes the following plug-ins:

- IPIF Plug-in
- ILL Plug-in
- PPP Plug-in
- Script Plug-in
- DLE Plug-in

Some parameters of the network plug-ins can reside in more than one plug-in in the network stack.

Some parameters of the network plug-ins are dynamic; that is, they are set by the plug-in after the connection is active.

IPIF Plug-in

The IPIF (IP Interface) plug-in resides in the layer below the NetOut interface in the TCP/IP stack. The IPIF plug-in manages the configuration of IP networking interfaces and other associated configuration information such as network routes, domain name resolver configuration entries, etc. It also controls DHCP when automatic configuration is enabled.

The IPIF plug-in has a user interface that is accessed via the TCP/IP tab in the Connection application for Internet profiles. This tab allows the user to set certain configuration parameters such as automatic (IP address is supplied by DHCP) or manual mode (IP address is supplied by the user), DNS servers, and DNS suffixes.

The complete list of parameters that you can set or get for the IPIF plug-in is shown in Table 2.1.

Table 2.1 IPIF plug-in parameters

Parameter	Type	Description
'LoIP'	string	Local IP address, in dotted notation. If this parameter is missing, automatic mode is assumed and the DHCP client is started for this connection. This value is set by the IP Address field in the TCP/IP tab if Manual mode is chosen.
'ReIP'	string	Remote IP address for PPP links, in dotted notation. Set only in manual mode. If this parameter is present, a PPP link is assumed and a default route is added, using this address as a default gateway.
'_MTU'	unsigned integer	Maximum transmit unit in bytes. Defaults to the value provided by the lower-level layer (DLPI or PPP).
'NetM'	string	Subnet mask; formatted as a dotted IP address. Set only in manual mode; in automatic mode it is computed automatically. This value is set by the Subnet Mask field in the TCP/IP tab if Manual mode is chosen.
'Brod'	string	Broadcast address. Set only in manual mode; in automatic mode, it is computed automatically.
'IFFS'	unsigned integer	Interface bits to set, which override the default IP configuration set by the SIOCSIFFLAGS ioctl command. Set only in manual mode.
'IFFC'	unsigned integer	Interface bits to clear, which override the default IP configuration set by the SIOCSIFFLAGS ioctl command. Set only in manual mode.
'IFId'	unsigned integer	Interface ID for use with the 'LLNa' parameter to build the interface name.
'LLNA'	string	Link name for use with the 'IFId' parameter to build the interface name.

Table 2.1 IPIF plug-in parameters *(continued)*

Parameter	Type	Description
'Mtic'	unsigned integer	Interface metric (defaults to 1 if not available). A metric is a value that is assigned to an IP route for a particular network interface that identifies the cost that is associated with using that route. For example, the metric can be valued in terms of link speed, hop count, or time delay. Higher metrics have the effect of making a route less favorable.
'DNSs'	string	List of DNS servers formatted as a list of dotted IP addresses, separated by spaces. This value is set by the DNS Servers form in the TCP/IP tab.
'GWys'	string	List of default gateways formatted as a list of dotted IP addresses, separated by spaces. This value is set by the Gateway field in the TCP/IP tab if Manual mode is chosen.
'Doms'	string	List of DNS domain suffixes separated by spaces. This value is set by the DNS Suffixes form in the TCP/IP tab.

ILL Plug-in

The ILL (IP Link Layer) plug-in implements a Data Link Provider Interface (DLPI). It resides below the IPIF plug-in in the TCP/IP stack. Those interested in the DLPI specification can find it at: http://www.opengroup.org/onlinepubs/9638599/toc.htm

The ILL plug-in has no user configurable parameters and thus has no user interface.

The complete list of parameters that you can set or get for the ILL plug-in is shown in Table 2.2.

Table 2.2 ILL plug-in parameters

Parameter	Type	Description
'DDev'	string	DLPI device name.
'DPPA'	unsigned integer	DLPI physical point of attachment for style 2 DLPI drivers.
'_ARP'	unsigned integer	Nonzero if ARP (Address Resolution Protocol) can be done on the link; zero if it cannot, such as with PPP links. Read-only.
'LLNA'	string	Link name for use with the 'IFId' parameter to build the interface name.
'LLAd'	binary	Link hardware MAC address. This parameter is always dynamic; that is, it is set by the plug-in after the connection is active. Read-only.

PPP Plug-in

The PPP plug-in implements a Point-to-Point (PPP) link and performs PPP negotiation. It resides below the ILL plug-in in the TCP/IP stack.

The PPP plug-in has a user interface that is accessed via the PPP tab in the Connection application for Internet profiles. This tab allows the user to set certain configuration parameters such as user name, password, timeout, Maximum Receive Unit (MRU) size, authentication type, etc.

The complete list of parameters that you can set or get for the PPP plug-in is shown in Table 2.3.

Table 2.3 PPP plug-in parameters

Parameter	Type	Description
'LoIP'	string	Local IP address, in dotted notation. If this parameter is missing, automatic mode is assumed and the IP address is obtained from the server for this connection. This value is set by the IP Address field in the TCP/IP tab if Manual mode is chosen.
'ReIP'	string	Remote IP address for PPP links, in dotted notation. Set only in manual mode. If this parameter is present, a PPP link is assumed and a default route is added, using this address as a default gateway.
'_MTU'	unsigned integer	Maximum transmit unit in bytes. Defaults to 1500.
'_MRU'	unsigned integer	Maximum receive unit in bytes. Defaults to 1500.
'Mtic'	unsigned integer	Interface metric (defaults to 30 if not available). A metric is a value that is assigned to an IP route for a particular network interface that identifies the cost that is associated with using that route. For example, the metric can be valued in terms of link speed, hop count, or time delay. Higher metrics have the effect of making a route less favorable.
'DNSs'	string	List of DNS servers formatted as a list of dotted IP addresses, separated by spaces. This value is set by the DNS Servers form in the TCP/IP tab.
'DDev'	string	DLPI device name.
'ConT'	unsigned integer	Timeout in milliseconds when connecting. The default is 10 seconds (10000).
'TrmT'	unsigned integer	Timeout in milliseconds when disconnecting. The default is 3 seconds (3000).

Table 2.3 PPP plug-in parameters *(continued)*

Parameter	Type	Description

'LCOp'	string	LCP (Link Control Protocol) options. The following options are valid (multiple options can be specified with the OR operator, and the default is all options ORed together):
		`ACFC`
		Address and control field compression.
		`PFC`
		Protocol field compression.
		`Magic`
		Magic number (for loopback detection).

'Auth'	string	Allowed authentication options. The following options are valid (multiple options can be specified with the OR operator):
		`CHAP`
		Challenge Handshake Authentication Protocol.
		`MSCHAP`
		Microsoft Challenge Handshake Authentication Protocol.
		`PAP`
		Password Authentication Protocol.
		`All`
		Allow all options (default).
		`EncryptedOnly`
		Allow all options that use encrypted passwords (`CHAP`, `MSCHAP`).

'ACCM'	unsigned integer	PPP asynchronous characters map. Defaults to 0x000A0000 (XON/XOFF characters).

Table 2.3 PPP plug-in parameters *(continued)*

Parameter	Type	Description
'User'	string	Username.
'Pass'	string	Password (write-only).

Script Plug-in

The Script plug-in is associated with PPP and provides login script capability for PPP connections.

The Script plug-in has only one parameter, listed in Table 2.4.

Table 2.4 Script plug-in parameters

Parameter	Type	Description
'LogS'	binary	Login script.

Login script commands that you can use are listed in Table 2.5.

Table 2.5 Login script commands

Function	Command	Parameters	Example
Send	s	string	s go PPP
Wait for	w	string	w password:
Delay	d	seconds	d 1
Get IP	g	none	g
Prompt	a	string	a Enter Name:
Wait for prompt	f	string	f ID:
Send CR	n	none	n
Send UserID	u	none	u
Send Password	x	none	x
End	e	none	e

When setting the value of the LogS parameter, the characters in the script command string must be encoded into ASCII hex values and enclosed in square brackets. Each script command must end with the null character (00). For example, for the command "s go PPP", you would set this value: [7320676f2050505000]

The final script command string in the set must end with an additional null character, so there must be two nulls at the end (0000).

In the parameters to various login script commands, you can use the special escape sequences listed in Table 2.6. Each escape sequence, when encountered in a script command, expands to the value shown.

Table 2.6 Login script escape sequences

Escape sequence string	Description
$USERID	Expands to the user name.
$PASSWORD	Expands to the password.
^c	If c is one of the ASCII characters in the sequence '@' through '_', then this equals a byte value of 0 through 31, respectively. If c is one of the ASCII characters in the sequence 'a' through 'z', then this equals a byte value of 1 through 26, respectively. If c is any other character, this equals that character.
<cr>	Carriage return (0x0D)
<lf>	Line feed (0x0A)
\"	" (double quotation mark)
\^	^ (circumflex accent)
\<	< (less-than sign)
\\	\ (backslash)

DLE Plug-in

The DLE plug-in provides the Ethernet framing interface and resides at the lowest level, above the network hardware.

The DLE plug-in has no user configurable parameters and thus has no user interface.

The complete list of parameters that you can set or get for the DLE plug-in is shown in Table 2.7.

Table 2.7 DLE plug-in parameters

Parameter	Type	Description
'DDev'	string	DLPI device name.
'DPPA'	unsigned integer	DLPI physical point of attachment for style 2 DLPI drivers.
'_ARP'	unsigned integer	Nonzero if ARP (Address Resolution Protocol) can be done on the link; zero if it cannot, such as with PPP links. Read-only.
'DevN'	string	Name of IOS driver that will be opened when connecting the profile.

Examples of Network Profile Strings

Here is an example of a profile using the 'foo' device. It will be configured using DHCP:

P1 = “NetOut/IPIF/ILL/DLE:DevN='foo'”

Here is an example of a subprofile:

“Mydevice” = “ILL/DLE:DevN='foo'”

The following profile is equivalent to P1:

P2 = “NetOut/IPIF/Mydevice”

Here is an example of a manually configured profile with the local IP address set. The subnet mask and broadcast address will be automatically set:

P3 = “NetOut/
IPIF:LoIP='192.168.1.10',GWys='192.168.1.1',DNSs='192.168.2
.1',Doms='bar.org'/Mydevice”

Here is an example of a PPP connection using a direct IrDA link:

P4 = “NetOut/IPIF/ILL/PPP:User='foo',Pass='bar'/
Serial:DevN='ircomm'”

Here is an example of a PPP connection with the local IP address manually set and a script:

P5 = “NetOut/IPIF/ILL/
PPP:User='foo',Pass='bar',LoIP='192.168.2.10'/
Script:LogS=[...........]/Serial:DevN='ircomm'”

The following profile is equivalent to P5 (it will produce the same effect but is not creatable via the user interface):

P6 = “NetOut/IPIF:LoIP='192.168.2.10'/ILL/
PPP:User='foo',Pass='bar'/Script:LogS=[...........]/
Serial:DevN='ircomm'”

Serial Plug-ins

There is a serial interface node and a serial plug-in that provide an interface to serial communication hardware:

- Serial Interface
- Serial Plug-in

Serial Interface

All serial profiles must start with the SerialMgr interface. This interface serves as the top-level serial component in the system. It is not a plug-in, but rather an interface, below which all other serial plug-ins must reside.

Profiles define a number of parameters for the SerialMgr interface, and these are listed in Table 2.8.

Table 2.8 SerialMgr interface parameters

Parameter	Type	Description
'crea'	unsigned integer	The port ID that is returned by `SrmGetDeviceInfo()` in `DeviceInfoType.serDevCreator`. This ID can be used to open a port. Profiles defining IrComm and RfComm will use the legacy IDs 'ircm' and 'rfcm', respectively.
'feat'	unsigned integer	Defines the port capabilities (`serDevCradlePort`, `serDevRS232Serial`, etc.), which are returned by `SrmGetDeviceInfo()` in `DeviceInfoType.serDevFtrInfo`.
'mbrt'	unsigned integer	Maximum baud rate supported by the port, which is returned by `SrmGetDeviceInfo()` in `DeviceInfoType.serDevMaxBaudRate`.
'hsbr'	unsigned integer	Always set this parameter to zero.
'hnam'	string	The name to be used for this port in the user interface. This name is returned by `SrmGetDeviceInfo()` in `DeviceInfoType.serDevPortInfoStr`. This name must be localized.
'lpid'	unsigned integer	Optional. Defines the mapping of this port to a Serial Manager logical port ID (for example, `serPortCradlePort`; constants are defined in SerialMgrLib.h). There must be only one profile defining each logical port.

Here is an example of a profile string for a port based on a device named “SERIAL”:

“SerialMgr:crea=1234,feat=0x00001000,mbrt=115200,hsbr=300,hnam='Cradle port',lpid=0/
Serial:Baud=9600,FCtl=0x80000000,Bits=0x30,DevN='SERIAL'”

Serial Plug-in

The Serial plug-in provides an interface to serial communication hardware. It registers itself with the Connection Manager under the name “Serial”.

The Serial plug-in has a user interface that is accessed via the Serial tab in the Connection application. This tab allows the user to set certain configuration parameters such as the device name, baud rate, number of data and stop bits, parity, etc.

The complete list of parameters that you can set or get for the Serial plug-in is shown in Table 2.9.

Table 2.9 Serial plug-in parameters

Parameter	Type	Description
'Baud'	unsigned integer	Baud rate. Standard baud rate constants are defined in termios.h. All baud rates are not supported by all serial hardware.
'FCtl'	unsigned integer	Flow control configuration. Specify `CTSFLOW`, `RTSFLOW`, or `CRTSCTS` (constants are defined in termios.h).
'Bits'	unsigned integer	Number of data bits. Specify `CS5`, `CS6`, `CS7`, or `CS8` (constants defined in termios.h).
'Stop'	unsigned integer	Number of stop bits. For 1 stop bit specify 0, or for 2 stop bits specify `CSTOPB` (constants defined in termios.h).
'Prty'	unsigned integer	Parity setting. For no parity specify 0, for even parity specify `PARENB`, or for odd parity specify `PARENB`|`PARODD` (constants defined in termios.h).
'DevN'	string	Name of IOS driver that will be opened when connecting the profile.

USB Plug-in

The USB plug-in provides an interface to USB hardware. It registers itself with the Connection Manager under the name “USBSerial”.

The complete list of parameters that you can set or get for the USB plug-in is shown in Table 2.10.

Table 2.10 USB plug-in parameters

Parameter	Type	Description
'Func'	unsigned integer	32-bit creator code that defines what the function of the USB connection is, for example, 'sync' for HotSync.
'DevN'	string	Name of IOS USB serial driver that will be opened when connecting the profile. This must be USBSERIAL.

The USB plug-in uses the other serial plug-in parameters when creating the default USB profiles so that the profiles will mimic a serial interface. These serial paramters have no effect on the connection, however.

Infrared Plug-in

The Infrared plug-in provides an interface to infrared (IR) hardware. It registers itself with the Connection Manager under the name "IRIF".

The Infrared plug-in configures itself automatically and has no parameters.

Here is an example profile string for PPP over infrared:

```
"NetOut/IPIF/PPP:User='foo',Pass='bar'/IRIF"
```

Bluetooth Plug-in

The Bluetooth plug-in provides an interface to Bluetooth hardware. It registers itself with the Connection Manager under the name "Bluetooth".

The Bluetooth plug-in has a user interface that is accessed via the Bluetooth Connection application. This tab allows the user to set certain configuration parameters such as the Bluetooth device name, whether to authenticate the connection, and whether to encrypt the data.

The complete list of parameters that you can set or get for the Bluetooth plug-in is shown in Table 2.11.

Table 2.11 Bluetooth plug-in parameters

Parameter	Type	Description
'addr'	string	48-bit Bluetooth device address of the remote device formatted as six one-byte values expressed in hex and separated by colons (for example '98:76:54:ab:CD:ef'). If this parameter is not specified, then a device discovery operation is performed.
'cod'	string	Class-of-device. Used for filtering the devices that are displayed to the user if a discovery procedure is used to determine the remote device address. Valid values are: 'computer', 'pda', 'phone', 'modem', or 'lan'. If no class-of-device parameter is specified, then the discovery procedure displays devices of all classes. Developers are discouraged from specifying a value for 'cod', 'cod2', and 'cod3' because there is no reliable correlation between a remote device’s class of device and the services it provides.
'cod2'	string	A second class-of-device that takes the same values as 'cod'.
'cod3'	string	A third class-of-device that takes the same values as 'cod'.
'prot'	string	Bluetooth protocol to use, either 'l2cap' or 'rfcomm'. If not specified, then 'rfcomm' is used.
'intf'	string	Which standard interface, if any, to present above the Bluetooth protocol entity. Only the value 'serial' is supported.

Table 2.11 Bluetooth plug-in parameters *(continued)*

Parameter	Type	Description
'sci'	string	Service class identifier to look for in the `ServiceClassIDList` attribute of the remote service record. If more than one service class identifier is specified (by using the sci*N* parameters), they are searched in sequence one at a time. The search ends as soon as one is found. Valid values are: 'SerialPort', 'LANAccessUsingPPP', 'DialupNetworking', 'OBEXObjectPush'; or a 128-bit UUID expressed by exactly 32 hex digits, for example, '0123456789abcdef0123456789abcdef'. (If specifying a UUID, the hex digits are not bracketed or preceded by 0x.) If PPP is used above Bluetooth and no sci*N* parameter is specified, and the class-of-device is 'phone' or 'modem', then sci='DialupNetworking'. If the class-of-device is 'lan' or 'computer', then sci='LANAccessUsingPPP'. If Serial or Telephony are used above Bluetooth and no sci*N* parameter is specified, then sci='SerialPort'.
'sci2'	string	Service class id to look for if that specified by 'sci' is not found.
'sci3'	string	Service class id to look for if those specified by 'sci' and 'sci2' are not found.
'auth'	string	Authentication required. Specify either 'yes' (default if 'encr' is set to 'yes') or 'no' (default if 'encr' is set to 'no').
'encr'	string	Encryption required; implies authentication is required. Specify either 'yes' or 'no' (default).

To determine the values for parameters whose values are not explicitly specified, the Bluetooth plug-in uses any contextual information present in the profile. That is, it looks for those values in plug-ins and interfaces to the left of Bluetooth in the profile string.

You should never programmatically create a profile containing only the Bluetooth plug-in (a Bluetooth terminal profile) because these must always be kept in sync with the Bluetooth favorites list and the Bluetooth cache information. The recommended way to create a

Bluetooth terminal profile is to sublaunch the Bluetooth panel to add a new favorite device. Once the device has been added to the favorites list, you can still programmatically add or modify profile parameters, except for 'addr' and all 'cod*N*' parameters, which must never be changed.

Telephony Plug-ins

There are two telephony plug-ins that provide an interface to phone hardware:

- Phone Plug-in
- DataCall

Phone Plug-in

The phone plug-in registers itself with the Connection Manager under the name “Phone”.

The Phone plug-in has a user interface that is accessed via the Phone tab in the Connection application. This tab allows the user to choose the phone driver.

The Phone plug-in has only one parameter, listed in Table 2.12.

Table 2.12 Phone plug-in parameters

Parameter	Type	Description
'Drvr'	unsigned integer	The creator of the phone driver to use.

DataCall

The DataCall plug-in registers itself with the Connection Manager under the name “DataCall”.

The DataCall plug-in has a user interface that is accessed via the DataCall tab in the Connection application. This tab allows the user to set certain parameters such as the phone connection type and phone number to dial for analog connections.

The complete list of parameters that you can set or get for the DataCall plug-in is shown in Table 2.13.

Table 2.13 DataCall plug-in parameters

Parameter	Type	Description
'Type'	unsigned integer	Only the value 0, for analog connections, is supported.
'Dial'	string	The phone number to dial, for analog connections.

3

Connection Manager Reference

The Connection Manager allows other applications to access, add, modify, and delete connection profiles contained in the Connection Manager database, and to make and terminate connections.

This chapter provides reference material for the Connection Manager API declared in the header file `CncMgr.h`:

- Connection Manager Structures and Types
- Connection Manager Constants
- Connection Manager Notifications
- Connection Manager Functions

For more information on the Connection Manager, see Chapter 1, "Connections," on page 3.

Connection Manager Structures and Types

This section describes data structures and types used by Connection Manager functions.

CncConnectionStateType Struct

Purpose Contains information about the connection state. This structure is returned by the `CncConnectReceiveState()` function.

Declared In `CncMgr.h`

Prototype

```
typedef struct CncConnectionStateTag {
   int32_t asyncId;
   uint32_t profileId;
   status_t error;
   int32_t fd;
   uint16_t state;
   uint16_t prgResoureId;
   uint32_t prgModuleCreator;
   uint32_t prgModuleType;
   uint16_t prgStringId;
   uint16_t prgIconId;
} CncConnectionStateType;
```

Fields

`asyncId`
: Asynchronous operation ID used instead of the file descriptor (in the `fd` field) for asynchronous connection operations. This identifies which asynchronous operation this structure refers to.

`profileId`
: ID of the connected profile.

`error`
: Error returned by a plug-in or the Connection Manager. Possible Connection Manager errors include: `cncErrMemory`, `cncErrInvalidParam`, `cncErrObjectTableFull`, `cncErrObjectNotFound`, `cncErrCommunication`, `cncErrMandatoryParameterNotFound`. For details on these errors, see "Error Codes" on page 60.

`fd`
: File descriptor of the IOS connection.

`state`
: Indicates the connection state. Possible values include:

  `#define kCncConnectedState 0`
  : Connected.

  `#define kCncConnectingState 1`
  : Connecting.

  `#define kCncDisconnectedState 2`
  : Disconnected.

  `#define kCncDisconnectingState 3`
  : Disconnecting.

`prgResoureId`
: Resource ID of the code module associated with this connection.

`prgModuleCreator`
: Creator code of the database that contains progress information for this connection.

`prgModuleType`
: Type code of the database that contains progress information for this connection.

`prgStringId`
: ID of the string containing progress status for this connection. This value, along with `prgIconId`, is useful for an application that wants to connect without using the default progress dialog. The application can use this information to update its own status display.

`prgIconId`
: ID of the icon associated with progress status for this connection.

CncControlType Struct

Purpose Contains information about an object. This structure is used by the `CncObjectControl()` function.

Declared In `CncMgr.h`

Prototype

```
typedef struct CncControlTag {
   uint16_t size;
   uint16_t session;
   status_t error;
   uint32_t Id;
   int16_t index;
   int16_t count;
   uint32_t data;
} CncControlType;
```

Fields

`size`
: Size of the whole data structure passed in the *`controlP`* parameter to `CncObjectControl()`. Must be supplied by the caller of `CncObjectControl()`.

`session`
: Session identifier to use for `CncSrv`... functions. The Connection Manager supplies this value to plug-ins as they are called.

`error`
Error returned from the plug-in function call. This value is set by the plug-in.

`Id`
Object ID of a plug-in or profile. Must be supplied by the caller of `CncObjectControl()`. If you specify a plug-in, only that plug-in's control function is called. If you specify a profile, the control function for each plug-in in the profile is called.

`index`
Index of the current plug-in in the profile, or -1 if the `Id` field specifies a single plug-in. The Connection Manager supplies this value to plug-ins as they are called.

`count`
Number of plug-ins in the profile, or -1 if the `Id` field specifies a single plug-in. The Connection Manager supplies this value to plug-ins as they are called.

`data`
Function-specific data to be passed to the function and/or returned from it.

CncEditMode Typedef

Purpose Specifies a mode of operation for the Connection application.

Declared In `CncMgr.h`

Prototype `typedef uint32_t CncEditMode;`

Constants `#define kCncTechnologyMode ((CncEditMode) 1)`
Display a list of communications technologies that can be configured. This lists consists of all profiles that begin with the `kCncTechnologiesRoot` interface.

`#define kCncProfileMode ((CncEditMode) 2)`
Display a list of profiles associated with a communications technology, for example network connections.

`#define kCncNewMode ((CncEditMode) 4)`
Display the profile creation form.

`#define kCncDeleteMode ((CncEditMode) 8)`
Display the profile deletion form.

```
#define kCncEditMode ((CncEditMode) 16)
```
Display the configuration form, for setting component parameters. Each plug-in in the profile is displayed in a separate tab so that the user can configure each plug-in that defines a configuration form.

```
#define kCncFullMode (kCncTechnologyMode |
  kCncProfileMode | kCncNewMode | kCncDeleteMode |
  kCncEditMode)
```
Reserved for future use.

```
#define kCncPanelMode (kCncProfileMode |
  kCncNewMode | kCncDeleteMode | kCncEditMode)
```
Reserved for future use.

```
#define kCncAppSwitchMode ((CncEditMode) 0x8000)
```
The configuration application is launched via an application switch; otherwise, the application is sublaunched.

```
#define kCncNoDoneButtonMode ((CncEditMode)
  0x4000)
```
The configuration application must not display the Done button.

See Also `CncProfileEdit()`

CncFindOptionsType Typedef

Purpose Specifies the kind of object to find. These constants can be ORed.

Declared In `CncMgr.h`

Prototype `typedef uint32_t CncFindOptionsType;`

Constants

```
#define kCncFindAllObjects ((CncFindOptionsType)
  kCncAllObjects)
```
Find all objects.

```
#define kCncFindPluginObjects
  ((CncFindOptionsType) kCncPluginObject)
```
Find only plug-in objects.

```
#define kCncFindInterfaceObjects
  ((CncFindOptionsType) kCncInterfaceObject)
```
Find only interface objects.

`#define kCncFindProfileObjects ((CncFindOptionsType) kCncProfileObject)`
Find only profile objects.

`#define kCncFindTemplateObjects ((CncFindOptionsType) kCncTemplateObject)`
Find only template objects.

`#define kCncFindLinkObjects ((CncFindOptionsType) kCncLinkObject)`
Find only link objects.

`#define kCncFindAvailableOnly ((CncFindOptionsType) 0x100)`
Find only objects that have availability different from `kCncNotAvailable` and `kCncPercentBaseAvailability` (0%); these are all available objects.

`#define kCncFindUsableOnly ((CncFindOptionsType) 0x200)`
Find only objects that have the `kCncUsableOption` flag.

`#define kCncFindInvisible ((CncFindOptionsType) 0x400)`
Find objects that are both visible (have the `kCncVisibleOption` flag) and invisible (without the `kCncVisibleOption` flag). If the `kCncFindInvisible` flag is not set, then invisible objects are not found.

`#define kCncFindDefault (kCncFindProfileObjects)`
Find only profile objects, which is the default.

`#define kCncFindAllCountMax 128`
Maximum count of objects returned by `CncObjectFindAll()`.

See Also `CncObjectFindAll()`, `CncProfileFindFirst()`

CncInfoType Struct

Purpose Contains information about a Connection Manager object (plug-in, interface, or profile).

Declared In `CncMgr.h`

Prototype

```
typedef struct CncInfoTag {
   char name[kCncMaxNameLength + 1];
   uint32_t objectId;
   uint8_t version;
   uint8_t type;
   uint16_t count;
   uint8_t priority;
   uint8_t availability;
   uint16_t options;
   uint32_t manualId;
} CncInfoType;
```

Fields

`name`
: Name of the object (read/write field).

`objectId`
: Object unique ID (read-only field).

`version`
: Version of the object (read-only field).

`type`
: Type of the object (read/write field). One of the constants listed in "Object Types" on page 63.

`count`
: For a profile, the number of plug-ins and interfaces in it; for a plug-in or interface, the number of profiles that include it (read-only field).

`priority`
: Priority number of the profile (read/write field). 0 is the lowest priority. The priority is used only when a profile is first added to a list of similar profiles, to determine where in the list to insert it. Profiles of the same connection type are listed in order from high priority to low priority.

  This value is set by the user via the Connection application, or programmatically by `CncObjectSetInfo()`.

`availability`
: Indicates if the profile can connect at this time (read/write field). By default, all profiles have an availability of `kCncProfileAvailable`. Possible values include:

  `#define kCncProfileUnknownAvailability 0`
  : Unknown availability. The Connection manager will try to dynamically determine the availability when asked by querying each plug-in in the profile.

  `#define kCncProfileAvailable 1`
  : Profile can connect.

  `#define kCncPercentBaseAvailability 100`
  : The profile is available with a defined percentage, as for a phone signal strength for example. A value between `kCncPercentBaseAvailability` and `kCncPercentBaseAvailability+100` represents a percentage of availability between 0 and 100.

  `#define kCncProfileNotAvailable 255`
  : Profile cannot connect.

`options`
: Flags that set various characteristics of the object. These constants can be ORed together.

  `#define kCncHasUIOption 0x8000`
  : Plug-in has a user interface form and a form handler.

  `#define kCncHasConnectOption 0x4000`
  : Plug-in has a connect callback function.

  `#define kCncHasFriendlyNameOption 0x2000`
  : Plug-in has a friendly name

  `#define kCncIsConnectedOption 0x1000`
  : Profile is connected.

  `#define kCncUsableOption 0x0001`
  : Object is usable for obtaining connect information.

  `#define kCncVisibleOption 0x0002`
  : Object is visible. When this flag is cleared, the object becomes “invisible” to searches and won’t be found by the find functions. Generic template profiles that are never connected or searched are typically made

invisible; they only appear in the template list to be copied as the basis of a new profile.

`#define kCncUnsearchableOption 0x0004`
Object is searchable by search operations.

`#define kCncReadOnlyOption 0x0008`
Object is read-only and cannot be changed.

`#define kCncReplaceOnUsingOption 0x0010`
Profile must be copied even if it is referenced by a {USING} macro.

`#define kCncUndeletableOption 0x0020`
Object cannot be deleted.

`#define kCncPersistentOption 0x0040`
The profiles attached to the plug-in or interface are persistent.

`#define kCncManualModeOption 0x0080`
For Internet profiles, the Connection Manager uses the ID specified in the `manualId` field when trying to connect. If this flag is not set, the Connection Manager uses the first available and usable Internet profile. The configuration application changes this option when the user taps the Automatic/Manual buttons in the Internet profiles list.

`#define kCncUserCanChangeModeOption 0x0100`
User can change the manual/automatic mode for this plug-in or interface. This option is used by the configuration application.

`#define kCncTestableOption 0x0200`
This plug-in is testable. If a plug-in has this option set, then the Connection application displays a "Test" button in place of the "Go Online" button for profiles attached to this plug-in. Tapping this button causes the Connection application to call `CncObjectControl()` and to send the `kCncControlTest` request to the selected profile.

`manualId`
ID of the Internet profile to connect, if `kCncManualModeOption` is set in the `options` field (read/write field).

See Also `CncObjectFindAll()`, `CncObjectGetInfo()`, `CncObjectSetInfo()`

CncParameterType Struct

Purpose Contains information about a parameter of a profile item. This structure is used by the `CncProfileGetParameters()` and `CncProfileSetParameters()` functions.

Declared In `CncMgr.h`

Prototype

```
typedef struct CncParameterTag {
   uint32_t name;
   uint16_t size;
   uint8_t type;
   uint8_t reserved;
   union {
      int32_t asInteger;
      char *asString;
      uint8_t *asBinary;
   } value;
} CncParameterType;
```

Fields

`name`
Parameter name.

`size`
Size of the parameter value in bytes.

`type`
Parameter type; one of the constants listed in "Parameter Types" on page 63.

`reserved`
Reserved for system use.

`value`
Parameter value.

Connection Manager Constants

This section describes constants used by Connection Manager functions.

Connection Options

Purpose Flags that control how a connection is made.

Declared In `CncMgr.h`

Constants

`#define kCncConnectProgressUI 1`
Allows the Connection Manager to display the progress indicator during the connection process.

`#define kCncConnectChooserUI 2`
Reserved for future use.

`#define kCncConnectDisableFallback 4`
Disables the automatic fallback mechanism. The fallback mechanism tries to make a connection using the first available and usable profile, and if it fails, the next one is tried, and so on.

`#define kCncConnectDisableReconnection 8`
Reserved for future use.

`#define kCncConnectAsynchronous 16`
Causes the function to return immediately without waiting for the connection completion. The caller must call `CncConnectReceiveState()` to determine connection progress and status.

See Also `CncProfileConnect()`, `CncProfileFindConnect()`

Control Requests

Purpose Specify the type of request to send to a plug-in via the `CncObjectControl()` function.

Declared In `CncMgr.h`

Constants

`#define kCncControlTest 0x00000080`
Requests the plug-in to perform a test function. Built-in plug-ins currently don't support such a function.

`#define kCncControlUserChange 0x00000081`
: Notifies the plug-in that a profile it is part of has changed. Some plug-ins may want to do some kind of update in this case. The Connection application sends this request code to all plug-ins in a profile when the profile is changed.

`#define kCncControlAvailability 0x00000082`
: Requests the plug-in to check its availability. The Connection Manager uses this request code to check the availability of plug-ins in a profile. All built-in plug-ins support this request.

Error Codes

Purpose Error codes returned by Connection Manager functions.

Declared In `CncMgr.h`

Constants

`cncErrInvalidParam`
: One or more of the function parameters is invalid.

`cncErrOpenFailed`
: The Connection Manager failed to open a session with the Connection Manager server.

`cncErrObjectTableFull`
: There are too many locked profiles (from all sessions). Unlock some profiles and try again.

`cncErrInvalidPluginModule`
: The specified plug-in module is invalid.

`cncErrMemory`
: Not enough free memory to perform the requested operation.

`cncErrNotImplemented`
: The requested operation is not implemented.

`cncErrObjectNotFound`
: The specified object cannot be found in the database.

`cncErrCannotAllocateObject`
: Cannot allocate a new object in the database.

`cncErrObjectFull`
: Cannot add a new item in the object because the item count limit is reached.

`cncErrIndexOutOfRange`
: The specified index is out of range.

`cncErrDatabase`
: The Connection Manager database cannot be opened or created.

`cncErrCommunication`
: The application cannot communicate with the Connection Manager.

`cncErrPluginModuleInitFailed`
: Failed to initialize the plug-in module.

`cncErrInvalidObject`
: The specified object is invalid.

`cncErrObjectAlreadyExists`
: Cannot create a new object with the same name as an existing object.

`cncErrMandatoryParameterNotFound`
: A parameter required to make a connection is not defined for a plug-in.

`cncErrModuleAlreadyLoaded`
: Internal error.

`cncErrNoPluginForm`
: A plug-in is involved in the configuration application but does not have a configuration form.

`cncErrSessionTableFull`
: There are too many locked profiles (from the current session). Unlock some profiles in the current session and try again.

`cncErrExclusiveObject`
: Tried to lock an object while another client has exclusive access to it.

`cncErrObjectInUse`
: Attempt to delete a locked object.

`cncErrAlreadyDisconnecting`
Attempt to disconnect a profile while the profile is already in a disconnecting or disconnected mode.

`cncErrUndeletableObject`
Attempt to delete an object that has the `kCncUndeletableOption` option flag.

`cncErrReadOnlyObject`
Attempt to change an object that has the `kCncReadOnlyOption` option flag.

Object Information Flags

Purpose Specify which fields in the `CncInfoType` structure are set by the `CncObjectSetInfo()` function.

Declared In `CncMgr.h`

Constants

`#define kCncNameInfoFlag 1`
The `name` field.

`#define kCncPriorityInfoFlag 2`
The `priority` field.

`#define kCncAvailabilityInfoFlag 4`
The `availability` field.

`#define kCncOptionsInfoFlag 8`
The `options` field.

`#define kCncOptionsSetInfoFlag 16`
Set to 1 (with an OR operation) the bits specified in `infoP.options`.

`#define kCncOptionsClearInfoFlag 32`
Set to 0 (with an AND NOT operation) the bits specified in `infoP.options`.

`#define kCncOptionsInvertInfoFlag 64`
Invert (with an XOR operation) the bits specified in `infoP.options`.

`#define kCncToggleTemplateInfoFlag 128`
Changes a profile that is not a template into a template; or changes a template profile into a regular profile. This flag

toggles the setting of the `infoP.type` field between `kCncProfileObject` and `kCncTemplateObject`.

`#define kCncManualIdInfoFlag 256`
Sets the default profile attached to a top-level plug-in; this profile is then used in manual mode.

`#define kCncToggleLinkInfoFlag 512`
Changes a profile into a link; or changes a link into a regular profile. This flag toggles the setting of the `infoP.type` field between `kCncProfileObject` and `kCncLinkObject`.

Object Types

Purpose Specify the type of an object.

Declared In `CncMgr.h`

Constants `#define kCncPluginObject 0x01`
Plug-in object.

`#define kCncInterfaceObject 0x02`
Interface object.

`#define kCncProfileObject 0x04`
Profile object.

`#define kCncTemplateObject 0x08`
Template object.

`#define kCncLinkObject 0x10`
Template object.

Parameter Types

Purpose Specify the type of value stored for a parameter in a connection profile.

Declared In `CncMgr.h`

Constants `#define kCncUndefinedParameterType 0`
Undefined type. When setting parameters, use this type to remove a parameter from a profile.

`#define kCncIntegerParameterType 1`
Integer type (`int32_t`).

`#define kCncStringParameterType 2`
String type (zero terminated).

`#define kCncBinaryParameterType 3`
Binary data type.

Profile Move Constants

Purpose Specify how to move a profile in a list of profiles for a particular connection type.

Declared In `CncMgr.h`

Constants

`#define kCncMoveUp -1`
Move the profile one position closer to the top of the list.

`#define kCncMoveDown +1`
Move the profile one position lower in the list.

`#define kCncMoveAsDefault -128`
Move the profile to the top of the list, so it becomes the default profile for the connection type.

`#define kCncMoveLast 127`
Move the profile to the end of the list.

See Also `CncObjectMoveItem()`

Connection Manager Notifications

cncNotifyConnectionStateEvent

Purpose Broadcast by the Connection Manager whenever the connection state of a persistent profile changes (a persistent profile is connected or disconnected on error or on user request, or the availability of the profile changes).

Declared In `NotifyMgr.h`

Prototype `#define cncNotifyConnectionStateEvent 'cncc'`

Parameters None.

Comments This notification carries no data, so notification clients will need to query the persistent profiles they are interested in to update their state information.

Connection Manager Functions

CncCloseSession Function

Purpose Closes the interprocess communication channel session with the Connection Manager server.

Declared In `CncMgr.h`

Prototype `void CncCloseSession(void)`

Parameters None.

Returns None.

Comments Generally, you do not need to call this function because the Connection Manager closes the session when your application exits.

You might want to use this function to close a session in order to free the session resources if your application no longer needs to use the Connection Manager, but continues running.

If a session is not open, this function does nothing.

See Also `CncGetOrOpenSession()`

CncConnectReceiveState Function

Purpose Blocks until a connection status message from the Connection Manager server is available and then returns it.

Declared In `CncMgr.h`

Prototype
```
status_t CncConnectReceiveState
    (CncConnectionStateType *stateP)
```

Parameters
← *`stateP`*
Pointer to a `CncConnectionStateType` structure that contains the current state of the connection.

Returns Returns `errNone` if the function was successful or returns an error code if not successful.

Comments When you call this function to connection check status, make sure that the ID returned in *`stateP`*→`asyncID` matches the asynchronous ID returned by your connection call. This ensures that you are getting the status of the correct asynchronous operation, in case there are more than one in progress.

After calling this function, wait until *`stateP`*→`state` is `kCncConnectedState` before attempting to send or receive data. This signals that the connection process is finished and the *`stateP`*→`fd` field is set to the file descriptor of the connection. Other intermediate states can be used to update a custom progress UI.

The calling application may want to use `IOSPoll()` on the session file descriptor (returned by `CncGetSession()`) to know when to call this function.

Before calling `CncConnectReceiveState()`, the application must have previously called `CncProfileConnect()` or `CncProfileFindConnect()` in asynchronous mode.

For an example of how to poll using `IOSPoll()`, see “Connecting From a Stored Profile” on page 19.

CncEdgeDelete Function

Purpose Deletes an edge (a relationship between two nodes) from the Connection Manager graph.

Declared In `CncMgr.h`

Prototype
```
status_t CncEdgeDelete(uint32_t fromId,
    uint32_t toId)
```

Parameters

→ *`fromId`*
: Start node of the edge; that is, the ID of a plug-in or interface that defines the start of an edge. You can use `CncObjectGetIndex()` to obtain an object ID.

→ *`toId`*
: End node of the edge; that is, the ID of a plug-in or interface that defines the end of an edge.

Returns Returns the following result codes:

`errNone`
: No error; the edge is deleted.

`cncErrObjectNotFound`
: The specified start or end nodes do not exist.

CncEdgeNew Function

Purpose Adds a Connection Manager graph edge (relationship) between two nodes (plug-ins or interfaces) in the Connection Manager database.

Declared In `CncMgr.h`

Prototype
```
status_t CncEdgeNew(uint32_t fromId,
    uint32_t toId)
```

Parameters

→ *`fromId`*
: Start node of the edge; that is, the ID of a plug-in or interface that defines the start of the edge. The start node represents a plug-in or interface that uses the node identified by *`toId`*; it's "above" the end node in a communications stack. You can use `CncObjectGetIndex()` to obtain an object ID.

→ *toId*
End node of the edge; that is, the ID of a plug-in or interface that defines the end of the edge, below the node identified by *fromId*, hierarchicaly.

Returns Returns the following result codes:

`errNone`
No error; the edge is created.

`cncErrObjectNotFound`
The specified start or end nodes do not exist.

`cncErrObjectFull`
A new edge cannot be added to *toId*.

`cncErrMemory`
Cannot allocate memory for the edge.

Comments Use this function in an application that needs a private interface node to link an interface with one or more nodes of the Connection Manager graph.

CncGetOrOpenSession Function

Purpose Returns the existing interprocess communication channel session with the Connection Manager server. If a session is not already open, this function opens a new session.

Declared In `CncMgr.h`

Prototype `int32_t CncGetOrOpenSession(void)`

Parameters None.

Returns Returns the file descriptor of the session. A returned value of -1 means that a session could not be opened with the Connection Manager server.

Comments The session ID is a file descriptor. Use this function if you want to call asynchronous functions and want to use `IOSPoll()` to wait for Connection Manager server replies.

Generally, you do not need to call this function to open a session because the Connection Manager automatically opens a session when your application makes Connection Manager calls that need a session.

See Also `CncCloseSession()`, `CncGetSession()`

CncGetSession Function

Purpose Returns the existing interprocess communication channel session with the Connection Manager server.

Declared In `CncMgr.h`

Prototype `int32_t CncGetSession(void)`

Parameters None.

Returns Returns the file descriptor of the session. A returned value of -1 means that a session is not open with the Connection Manager server.

Comments The session ID is a file descriptor. Use this function if you want to call asynchronous functions and want to use `IOSPoll()` to wait for Connection Manager server replies. The `CncGetOrOpenSession()` function is a more convenient way to obtain the session file descriptor.

See Also `CncGetOrOpenSession()`

CncInterfaceNew Function

Purpose Creates an empty interface node and adds it to the Connection Manager database.

Declared In `CncMgr.h`

Prototype
```
status_t CncInterfaceNew(const char *nameStr,
   uint32_t *Id)
```

Parameters

→ *`nameStr`*
Pointer to the interface name.

← *`Id`*
Pointer to the interface ID that is created. If the name specified by *`nameStr`* already exists, a pointer to that existing object ID is returned.

Returns Returns the following result codes:

`errNone`
  No error.

`cncErrMemory`
  Memory allocation error.

`cncErrCannotAllocateObject`
  The Connection Manager object table is full.

`cncErrObjectAlreadyExists`
  An object already exists with the name *`nameStr`*.

Comments An application can use this function to create a private interface node in the Connection Manager graph. After calling this function, an application would typically call `CncEdgeNew()` to create relationships between the interface node and one or more plug-ins.

CncObjectControl Function

Purpose Calls the control function in a single plug-in or in every plug-in in a profile.

Declared In `CncMgr.h`

Prototype

```
status_t CncObjectControl(uint32_t objectId,
   uint32_t request, CncControlType *controlP)
```

Parameters

→ `objectId`
  ID of the object you want to control. This can be plug-in or a profile. If a profile is specified, the `controlF` function is called for each plug-in in the profile.

→ `request`
  The request code, which is passed to the control function in the plug-in. This code tells the plug-in what to do. Universally defined request codes are listed in "Control Requests" on page 59. Plug-ins can also define their own request codes.

↔ *`controlP`*
  Parameters that are passed to the control function; see `CncControlType`.

Returns Returns the following result codes:

`errNone`
: No error; the object is deleted.

`cncErrInvalidParam`
: One or more function parameters are invalid.

`cncErrObjectTableFull`
: There are too many locked profiles.

`cncErrCannotAllocateObject`
: A new object cannot be allocated.

`cncErrSessionTableFull`
: The object has too many profiles locked in the current session.

Comments This function serves as a general mechanism for calling directly into the plug-ins associated with a profile, for control or any other purpose. Each plug-in should define a `controlF` function that is called when `CncObjectControl()` is invoked.

The *`request`* parameter is designed to pass a request code telling the plug-in what to do. The *`controlP`* parameter can contain custom parameters needed for the request.

The *`controlP`* parameter can be of a type other than `CncControlType`, but it must have the same first fields as `CncControlType`. For example, the structure used for the `kCncControlAvailability` request is defined as follows:

```
typedef struct CncControlAvailabilityTag {
    CncControlType control;
    uint8_t availability;
    uint32_t moduleCreator;
    uint32_t moduleType; }
    CncControlAvailabilityType;
```

A new session to the Connection Manager server is created to handle the requests from this function, to avoid conflicting with the client session.

CncObjectDelete Function

Purpose Deletes a plug-in, interface, or profile from the Connection Manager database.

Declared In `CncMgr.h`

Prototype
```
status_t CncObjectDelete(uint32_t nodeId)
```

Parameters

→ *`nodeId`*
ID of the object to delete. After deletion, the ID is no longer valid.

Returns Returns the following result codes:

`errNone`
No error; the object is deleted.

`cncErrObjectNotFound`
The specified object ID does not exist.

Comments Removing a plug-in or interface also removes all edges and profiles that use it. Removing a profile also removes profiles that reference it.

CncObjectFindAll Function

Purpose Searches all items associated with an object.

Declared In `CncMgr.h`

Prototype
```
status_t CncObjectFindAll(uint32_t objectId,
   CncFindOptionsType options, int16_t *countP,
   CncInfoType **infoArrayP)
```

Parameters

→ *`objectId`*
ID of the object in which to search associated items. For example, specify a profile ID to search all plug-ins associated with that profile.

→ *`options`*
Specifies the kind of object to search for; see `CncFindOptionsType`.

← *`countP`*
Number of items found.

↔ *infoArrayP*
An array of CncInfoType objects for the found items. When you are done with this array, free it with `MemPtrFree()`. If you don't want the array of items passed back, set this parameter to `NULL` on input.

Returns Returns the following result codes:

`errNone`
No error; the object is deleted.

`cncErrMemory`
Not enough memory to allocate the array of returned objects.

`cncErrInvalidParam`
One or more function parameters are invalid.

Comments This function can be used to find a set of profiles at once, instead of looping with `CncProfileFindFirst()`, `CncProfileFindNext()`, and `CncProfileFindClose()`. Also, `CncObjectFindAll` automatically allocates memory that it needs.

This function not only finds profiles in the database but also finds profiles created but not yet submitted in the current session.

Returned objects are unlocked.

Example To find all plug-ins registered with the Connection Manager, call this function like this:

```
uint32_t profileID;
status_t err;
CncInfoType* profileList = NULL; // the profiles list
int16_t profileCount = 0; // the profiles count
profileID = CncObjectGetIndex(kCncPluginsRoot);
if (0 == profileID) return false;
err = CncObjectFindAll(profileID, kCncFindPluginObjects,
                       &profileCount, &profileList);
```

To find all Internet connection profiles, use this code:

```
uint32_t profileID;
status_t err;
CncInfoType* profileList = NULL; // the profiles list
int16_t profileCount = 0; // the profiles count
// kCncNetOutgoingInterface is defined in NetCnc.h (TCP/IP)
profileID = CncObjectGetIndex(kCncNetOutgoingInterface);
if (0 == profileID) return false;
```

```
err = CncObjectFindAll(profileID, kCncFindDefault,
                       &profileCount, &profileList);
```

See Also CncProfileFindFirst()

CncObjectGetIndex Function

Purpose Gets the ID of a plug-in, interface, or profile, by name.

Declared In `CncMgr.h`

Prototype `uint32_t CncObjectGetIndex(const char *nameStr)`

Parameters → *nameStr*
Pointer to the name of the object to return.

Returns Returns the ID of the object. If the name is not found, returns 0.

Comments This function parses all of the Connection Manager database named records until the name is found or all records are examined.

CncObjectGetInfo Function

Purpose Gets information from the database record of a plug-in, interface, or profile.

Declared In `CncMgr.h`

Prototype
```
status_t CncObjectGetInfo(uint32_t recordId,
    CncInfoType *infoP)
```

Parameters → *recordId*
ID of the object to return information about.

↔ *infoP*
Pointer to a `CncInfoType` structure.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
The specified object ID does not exist.

`cncErrInvalidParam`
The *`infoP`* pointer is not valid (`NULL`).

See Also `CncObjectSetInfo()`

CncObjectMoveItem Function

Purpose Changes the order of a profile in the list of profiles attached to a communications component.

Declared In `CncMgr.h`

Prototype
```
status_t CncObjectMoveItem(uint32_t itemId,
   int16_t newIndexRelative)
```

Parameters

→ *`itemId`*
The ID of the item to move.

→ *`newIndexRelative`*
An offset by which to move the item, such as -1, +3, or one of the `Profile Move Constants`. Specify a negative number to move the item closer to the top of the list and a positive number to move it lower in the list.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
The specified object ID does not exist.

`cncErrInvalidParam`
One or more function parameters are invalid.

`cncErrMemory`
Not enough free memory to perform the operation.

`cncErrObjectTableFull`
There are too many locked profiles.

`cncErrIndexOutOfRange`
The value specified for *`newIndexRelative`* is out of range.

`cncErrSessionTableFull`
The object has too many profiles locked in the current session.

Comments This function can also be used to make a profile the default one in a list by moving it to the top of the list.

This function does not change the priority of a profile, only its order in the list.

CncObjectSetInfo Function

Purpose Sets information in the database record of a plug-in, interface, or profile.

Declared In `CncMgr.h`

Prototype
```
status_t CncObjectSetInfo(uint32_t recordId,
   CncInfoType *infoP, uint32_t flags)
```

Parameters

→ *`recordId`*
ID of the object to set information about.

→ *`infoP`*
Pointer to `CncInfoType` structure that contains the information to set.

→ *`flags`*
Flags indicating which fields to set. Specify values from the Object Information Flags.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
The specified object ID does not exist.

`cncErrInvalidParam`
The *`infoP`* pointer is not valid (`NULL`).

See Also `CncObjectGetInfo()`

CncParametersFree Function

Purpose Frees all binary and string parameters in an array of parameters. Use this function after calling `CncProfileGetParameters()` to free allocated memory.

Declared In `CncMgr.h`

Prototype

```
void CncParametersFree
    (CncParameterType parameters[])
```

Parameters

↔ *`parameters`*
An array of `CncParameterType` structures.

Returns None.

See Also `CncParametersInit()`

CncParametersInit Function

Purpose Initializes a `CncParameterType` array.

Declared In `CncMgr.h`

Prototype

```
void CncParametersInit
    (CncParameterType parameters[], int32_t n)
```

Parameters

↔ *`parameters`*
An array of `CncParameterType` structures.

→ *`n`*
The number of elements to initialize (typically the size of the array).

Returns None.

Comments This function sets the last element in the array of parameters to the constant `kCncParameterTableEnd`. The last array element is required to have this value when setting parameters, so you should set the value of *`n`* to one greater than the number of parameters you want to set.

See Also `CncProfileGetParameters()`, `CncProfileSetParameters()`

CncProfileAttach Function

Purpose Inserts a profile into the list of profiles attached to a plug-in or interface.

Declared In `CncMgr.h`

Prototype

```
status_t CncProfileAttach(uint32_t fromId,
   uint32_t toId)
```

Parameters

→ *`fromId`*
ID of the plug-in or interface to which you want to attach the profile.

→ *`toId`*
ID of the profile to attach.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
One of the specified object IDs does not exist.

Comments A plug-in or interface can have several different profiles attached to it, and these are kept in a list for the plug-in or interface. These are typically the profiles that have that plug-in or interface as the topmost component in their sequence, and this defines their connection type. For example, the TCP/IP plug-in might have three attached profiles; these each have TCP/IP as the topmost component.

The order in which `CncProfileAttach()` inserts the profile into the list is determined by the profile's priority (see the `priority` field of `CncInfoType`) relative to the existing profiles in the list. The list is ordered from highest to lowest priority.

CncProfileConnect Function

Purpose Makes a connection using a stored connection profile.

Declared In `CncMgr.h`

Prototype

```
int32_t CncProfileConnect(uint32_t profileId,
   uint32_t flags, status_t *error)
```

Parameters

→ *`profileId`*
ID of the connection profile to use to establish a connection.

→ *`flags`*
Flags that control how the connection is made. See "Connection Options" on page 59.

← *`error`*
`errNone` if the connection is successful, or a connection error returned by lower-level communications services.

Returns If `error` is `errNone`, returns the file descriptor of the connection. Or, if called in asynchronous mode, returns the asynchronous operation ID, which uniquely identifies this particular asynchronous operation so that when you call `CncConnectReceiveState()` to determine the status, you can be sure it's for this operation and not some other.

Comments This function can be called in asynchronous mode with the `kCncConnectAsynchronous` flag. In this case, the caller must call `CncConnectReceiveState()` to get connection progress and termination messages. The caller can use `IOSPoll()` to poll the Connection Manager session file descriptor in order to know when to call `CncConnectReceiveState()`.

When you call `CncConnectReceiveState()` to check status, make sure that the ID returned in *`stateP`*→`asyncID` matches the ID returned by `CncProfileConnect()`. This ensures that you are getting the status of the correct asynchronous operation, in case there are more than one in progress.

This function can have multiple user interface side effects. For example, the user can be prompted for a password or a progress dialog can be displayed. Some side effects are allowed or disallowed via the *`flags`* parameter.

See Also `CncProfileDisconnect()`, `CncProfileFindConnect()`

CncProfileCopy Function

Purpose Creates a new profile by copying a subset of items from an existing profile.

Declared In `CncMgr.h`

Prototype

```
status_t CncProfileCopy(uint32_t *profileId,
   int16_t from, int16_t to)
```

Parameters

↔ *`profileId`*
: On input, a pointer to the ID of a profile to copy. On output, the ID of the copy of the profile.

→ *`from`*
: The index of the item in the profile at which to begin copying.

→ *`to`*
: The index of the last item to copy.

Returns Returns the following result codes:

`errNone`
: No error.

`cncErrObjectNotFound`
: The specified object ID does not exist.

`cncErrInvalidParam`
: One or more function parameters are invalid.

`cncErrMemory`
: Not enough free memory to perform the operation.

`cncErrObjectTableFull`
: There are too many locked profiles.

`cncErrIndexOutOfRange`
: The value specified for *`from`* or *`to`* is out of range.

`cncErrCannotAllocateObject`
: Cannot allocate a new object in the database.

`cncErrSessionTableFull`
: There are too many locked profiles (from the current session). Unlock some profiles in the current session and try again.

Comments Use this function to copy a subset of items from a profile and create a new profile from them. All items beginning with the index *`from`* and through (and including) the index `to` are copied to the new profile.

The newly created copy is locked.

CncProfileDecode Function

Purpose Decodes a profile string into a new profile and submits it to the Connection Manager database.

Declared In `CncMgr.h`

Prototype

```
uint32_t
   CncProfileDecode(const char *profileName,
   const char *pathStr)
```

Parameters

→ *`profileName`*
Pointer to the name of the profile.

→ *`pathStr`*
Pointer to the profile path string.

Returns The ID of the unlocked profile, or 0 if an error occurs.

Comments Use this function along with `CncProfileEncode()` to manipulate string representations of connection profiles.

You can use this function to easily create a new connection profile, such as a private connection profile used by an application.

This function submits the newly created profile to the Connection Manager database and automatically attaches the profile to the first plug-in (or interface) in the profile. That is, the profile is added to the list of profiles available for the connection type that corresponds to the first plug-in. (This function calls `CncProfileAttach()` internally.)

CncProfileDeleteItem Function

Purpose Deletes an item (plug-in, interface, or a nested profile) and its parameters from a locked profile.

Declared In `CncMgr.h`

Prototype
```
status_t CncProfileDeleteItem(uint32_t lockedId,
   uint32_t itemIndex)
```

Parameters

→ *`lockedId`*
ID of a locked profile.

→ *`itemIndex`*
The zero-based index of an item to delete in the profile.

Returns The ID of the locked profile, or 0 if an error occurs.

Comments The profile must first be locked with `CncProfileLock()` or `CncProfileNew()`.

CncProfileDisconnect Function

Purpose Disconnects a persistently connected profile or cancels the connection process.

Declared In `CncMgr.h`

Prototype
```
status_t CncProfileDisconnect(uint32_t profileId,
   uint8_t kind)
```

Parameters

→ *`profileId`*
ID of the connection profile to disconnect or cancel.

→ *`kind`*
The reason for the disconnection; this is passed to the plug-in. You can specify the following values:

`#define kCncUserCancelled 1`
The user canceled the connection being made.

`#define kCncSystemCancelled 2`
The system canceled the connection being made.

`#define kCncUserDisconnected 3`
The user disconnected the persistent connection.

`#define kCncSystemDisconnected 4`
The system disconnected the persistent connection.

Returns Returns the following result codes:

`errNone`
: No error.

`cncErrObjectNotFound`
: The ID specified in *`profileId`* cannot be found.

`cncErrAlreadyDisconnecting`
: The connection is already being disconnected.

Comments The profile must be connected or in the process of connecting, otherwise this function returns `cncErrObjectNotFound`.

See Also `CncProfileConnect()`

CncProfileEdit Function

Purpose Launches the Connection application and opens the configuration form associated with the first plug-in of a profile.

Declared In `CncMgr.h`

Prototype

```
status_t CncProfileEdit(uint32_t *Id,
   CncEditMode launchMode,
   CncEditMode enabledModes,
   CncEditParametersType *params)
```

Parameters

↔ *`Id`*
: ID of the profile to configure; the configuration form associated with the first plug-in in the profile is opened if *`launchMode`* is `kCncEditMode`. The ID can be for a plug-in to create or choose a profile beginning with this plug-in.

→ *`launchMode`*
: Identifies the mode in which the Connection application is to open. See the `CncEditMode` type for valid values.

→ *`enabledModes`*
: Reserved for future use.

↔ *`params`*
: Reserved for future use.

Returns Returns `errNone` if the ID parameter is valid and references the selected profile. Do not assume that this ID is the same as the one passed in; the user may have switched to another profile or created a new one.

Comments If the `kCncAppSwitchMode` bit is set in *`launchMode`*, the Connection application is launched via `SysUIAppSwitch()`. The current application is first closed, then the Connection application is launched.

If the `kCncAppSwitchMode` bit is not set, the Connection application is sublaunched via `SysAppLaunch()` (the Connection application is launched as a subroutine of the current application).

If the `kCncNoDoneButtonMode` bit is set in *`launchMode`*, the Connection application does not display a **Done** button. If the **Done** button is shown in the Connection application, tapping it returns to the calling application, in the case of a `SysAppLaunch()`.

CncProfileEncode Function

Purpose Encodes a profile into an external string representation.

Declared In `CncMgr.h`

Prototype

```
char* CncProfileEncode(uint32_t profileId,
    int16_t index)
```

Parameters

→ *`profileId`*
ID of the profile to encode.

→ *`index`*
The index of a single plug-in within the profile that you want to encode. If you specify a valid plug-in index, then just the string representation of that plug-in is encoded. Specify -1 to encode the whole profile into a string.

Returns The external string representation of the profile (or plug-in), or `NULL` if an error occurs. If non-`NULL`, you must free this by calling `MemPtrFree()` when you are done with it.

Comments Use this function along with `CncProfileDecode()` to manipulate string representations of connection profiles.

You can use this function to help manage an application's private connection profile.

CncProfileFindClose Function

Purpose Ends a profile search in the database.

Declared In `CncMgr.h`

Prototype
```
void CncProfileFindClose(uint32_t searchId)
```

Parameters
→ *`searchId`*
ID of the search, returned by `CncProfileFindFirst()`.

Returns None.

Comments This function releases any internal data structures used by the find algorithm.

CncProfileFindConnect Function

Purpose Finds a profile, or creates a profile dynamically, and makes a connection using it.

Declared In `CncMgr.h`

Prototype
```
int32_t CncProfileFindConnect(char *profileStr,
   uint32_t flags, uint32_t *profileIdP,
   status_t *error)
```

Parameters
→ *`profileStr`*
Pointer to the profile search string (profile name, complete profile string, or partial profile string); for example, "NetOut/*". If you specify a complete profile string, this function creates a new profile from it rather than searching the database.

→ *`flags`*
Flags that control how the connection is made. See "Connection Options" on page 59.

↔ *`profileIdP`*
Returns a pointer to the ID of the found profile, if a search was performed. If you don't need the ID to be returned, you can set this parameter to `NULL` on input.

← *`error`*
`errNone` if the connection is successful, or a connection error returned by lower-level communications services.

Returns If `error` is `errNone`, returns the file descriptor of the connection. Or, if called in asynchronous mode, returns the asynchronous operation ID, which uniquely identifies this particular asynchronous operation so that when you call `CncConnectReceiveState()` to determine the status, you can be sure it's for this operation and not some other.

Comments This function internally calls `CncProfileFindFirst()` to search for or create a profile string. If *`profileStr`* is a profile name or partial profile string (with "/*" at the end), then `CncProfileFindConnect()` finds the first matching profile and makes a connection with it.

If *`profileStr`* is a complete profile string (without "/*" at the end) then this function creates a new profile from it and makes a connection from it. The profile is not saved in the Connection Manager database unless you call `CncProfileSubmit()`; if you don't call `CncProfileSubmit()`, the profile is automatically deleted when the Connection Manager session closes.

This function can be called in asynchronous mode with the `kCncConnectAsynchronous` flag. In this case, the caller must call `CncConnectReceiveState()` to get connection progress and termination messages. The caller can use `IOSPoll()` to poll the Connection Manager session file descriptor in order to know when to call `CncConnectReceiveState()`.

When you call `CncConnectReceiveState()` to check status, make sure that the ID returned in *`stateP`*→`asyncID` matches the ID returned by `CncProfileFindConnect()`. This ensures that you are getting the status of the correct asynchronous operation, in case there are more than one in progress.

This function can have multiple user interface side effects. For example, the user can be prompted for a password or a progress dialog can be displayed. Some side effects are allowed or disallowed via the *`flags`* parameter.

The Connection Manager uses the usability and availability properties of profiles to determine if it can connect one that it finds. It won't try to connect a profile that is not usable or available. However, if the first plug-in of a profile is in manual mode, the Connection Manager attempts to connect the profile identified by `CncInfoType.manualId` regardless of its usability or availability.

Manual mode is indicated for a profile when the `CncInfoType.options` field of its first plug-in has the `kCncManualModeOption` flag set. For more information on automatic and manual mode, see "Automatic and Manual Mode Profiles" on page 11.

An application can use this function to make a connection using a private connection profile that it has stored as a string.

See Also `CncProfileConnect()`, `CncProfileDisconnect()`

CncProfileFindFirst Function

Purpose Begins a profile search in the database and current session, or creates a new profile.

Declared In `CncMgr.h`

Prototype

```
uint32_t CncProfileFindFirst(char *searchStr,
   CncFindOptionsType options,
   uint32_t *searchIdP, status_t *errP)
```

Parameters

→ *`searchStr`*
Pointer to the search string (profile name, complete profile string, or partial profile string); for example, "NetOut/*". If you specify a complete profile string, this function creates a new profile from it rather than searching the database.

→ *`options`*
Specifies the kind of object to search for; see `CncFindOptionsType`.

← *`searchIdP`*
Pointer to the new search ID (used by other search functions). If you specify `NULL` on input, this indicates you don't want the search ID to be returned. In this case, this function internally calls `CncProfileFindClose()`, to end the search before returning.

← *`errP`*
`errNone` if the operation is successful, or an error code if not successful.

Returns The ID of the first profile matching the search string, or 0 if no profile is found. The found profile is locked.

Comments If *`searchStr`* is a profile name, this function returns that profile ID and the value of *`searchIdP`* is set to 0.

If *`searchStr`* is a complete profile string (without "/*" at the end) then this function creates a new profile from it and returns the ID of the newly created profile. The value of *`searchIdP`* is set to 0. If you do not call `CncProfileSubmit()`, the profile is automatically deleted when the Connection Manager session closes.

If *`searchStr`* is a partial profile string (with "/*" at the end), this function searches for the first matching profile. The value of *`searchIdP`* is set to a non-zero value and references the search. This search ID must be passed to `CncProfileFindNext()` and `CncProfileFindClose()`. It's good practice to call `CncProfileFindClose()` to clean up memory when you are done with the search; it needs to be called only when *`searchIdP`* returns a non-zero value.

To find the next profile that matches the search criteria, call `CncProfileFindNext()`; except if you've specified a `NULL` value for *`searchIdP`* on input, in which case the search is closed and no search ID is returned.

This function also finds profiles created but not yet submitted in the current session.

See Also `CncObjectFindAll()`

CncProfileFindNext Function

Purpose Continues a profile search in the database and current session.

Declared In `CncMgr.h`

Prototype
```
uint32_t CncProfileFindNext(uint32_t searchId,
    status_t *errP)
```

Parameters

→ *`searchId`*
Search ID returned by `CncProfileFindFirst()`.

← *`errP`*
`errNone` if the operation is successful, or an error code if not successful.

Returns The ID of the next profile matching the search criteria established by `CncProfileFindFirst()`, or 0 if no more matching profiles are found. The found profile is locked.

Comments The `CncProfileFindFirst()` function must be called before using `CncProfileFindNext()`.

It's good practice to call `CncProfileFindClose()` to clean up memory when you are done with the search, after you have called `CncProfileFindNext()` for the last time.

This function also finds profiles created but not yet submitted in the current session.

See Also `CncObjectFindAll()`

CncProfileGetItemId Function

Purpose Gets the ID of an item in a profile (a plug-in or interface object) from its index.

Declared In `CncMgr.h`

Prototype

```
uint32_t CncProfileGetItemId(uint32_t lockedId,
    int16_t index)
```

Parameters

→ *`lockedId`*
ID of a locked profile. A profile is locked by `CncProfileLock()` or `CncProfileNew()`.

→ *`index`*
`Index (zero-based) of the item in the profile.`

Returns The ID of the plug-in or interface object, or 0 if an error occurs.

See Also `CncProfileGetItemIndex()`

CncProfileGetItemIndex Function

Purpose Gets the index of an item in a profile (a plug-in or interface object) from the item name.

Declared In `CncMgr.h`

Prototype

```
status_t
   CncProfileGetItemIndex(uint32_t lockedId,
   int16_t *index, char *nameStr)
```

Parameters

→ *`lockedId`*
: ID of a locked profile. You can obtain the ID of a profile by calling `CncObjectGetIndex()` or `CncProfileFindFirst()`. A profile is locked by `CncProfileLock()` or `CncProfileNew()`.

↔ *`index`*
: On input, a pointer to the index (zero-based) of the item at which to begin the search. Use 0 to search the whole profile. On output, a pointer to the index (zero-based) of the found item, if no error occurs.

→ *`nameStr`*
: Pointer to the name of the item to search for.

Returns Returns the following result codes:

`errNone`
: No error.

`cncErrObjectNotFound`
: The specified profile ID does not exist.

`cncErrInvalidParam`
: The *`indexP`* pointer is not valid.

See Also `CncProfileGetItemId()`

CncProfileGetLength Function

Purpose Returns the number of items (plug-in and interface objects) in a profile.

Declared In `CncMgr.h`

Prototype `uint32_t CncProfileGetLength(uint32_t lockedId)`

Parameters → *`lockedId`*

ID of a locked profile. A profile is locked by `CncProfileLock()` or `CncProfileNew()`. You can obtain the ID of a profile by calling `CncObjectGetIndex()` or `CncProfileFindFirst()`.

Returns The number of items in the profile, or -1 if *`lockedId`* is not found.

CncProfileGetParameters Function

Returns parameters of an item (plug-in or interface object) in a profile.

Declared In `CncMgr.h`

Prototype

```
status_t CncProfileGetParameters
    (uint32_t lockedId, int16_t itemIndex,
    int8_t method, CncParameterType parameters[])
```

Parameters → *`lockedId`*

ID of a locked profile. A profile is locked by `CncProfileLock()` or `CncProfileNew()`. You can obtain the ID of a profile by calling `CncObjectGetIndex()` or `CncProfileFindFirst()`.

→ *`itemIndex`*

Index (zero-based) of an item (plug-in or interface) in the profile.

→ *`method`*

Flags that determine the scope of the parameter search:

`kCncGetParametersItemOnly`
Only the item is searched.

`kCncGetParametersInherited`
The item and previous items (above in the hierarchy) are searched. If the parameter appears multiple times,

the first value found is returned. The profile is parsed right to left (upwards in the hierarchy).

`kCncGetParametersWholeProfile`
: The whole profile is searched. If the parameter appears multiple times, the first value found is returned. The profile is parsed left to right (downwards in the hierarchy).

`kCncGetParametersActive`
: The profile is searched for dynamic parameters that are available only in the current Connection Manager session when a persistent profile is connected. For example, network plug-ins update a network profile with the assigned IP address. Such dynamic parameters exist in the profile in the Connection Manager session, but are not saved to the profile in the database.

↔ *`parameters`*
: On input, an array of `CncParameterType` structures that contain the parameter names and types to get; the array previously should have been initialized by `CncParametersInit()`. On output, an array of those parameter values.

Returns Returns the following result codes:

`errNone`
: No error; the *`parameters`* structures are updated with the parameter values.

`cncErrObjectNotFound`
: The specified profile ID does not exist.

`cncErrInvalidParam`
: The *`itemIndex`* or *`parameters`* parameter is invalid.

Comments After you are done using the parameter array, you should call `CncParametersFree()` to free the memory.

Example

```
CncParameterType myParams[3];
CncParametersInit(myParams, 3);
myParams[0].name = 'Fctl';
myParams[1].name = 'port';
```

```
index = 0;
if (CncProfileGetItemIndex(Id, &index, "rs232") == errNone)
{
  CncProfileGetParameters(Id, index, myParams,
kCncGetInherited);
  FlowControl = myParams[0].value.asString;
  PortNumber = myParams[1].value.asInteger;
  // Do something else here
}
CncParametersFree(myParams);
```

See Also `CncProfileSetParameters()`

CncProfileInsertItem Function

Inserts an item into a locked profile. Items can be plug-ins, interfaces, or other profiles.

Declared In `CncMgr.h`

Prototype

```
status_t CncProfileInsertItem(uint32_t lockedId,
    int16_t *atIndex, uint32_t itemId,
    CncParameterType parameters[])
```

Parameters

→ *lockedId*
ID of a locked profile. A profile is locked by `CncProfileLock()` or `CncProfileNew()`. You can obtain the ID of a profile by calling `CncObjectGetIndex()` or `CncProfileFindFirst()`.

→ *atIndex*
Index of an item in the profile at which to insert the new item. Specify 0 to add the item at the beginning, or `kCncInsertAtEndIndex` to add the item at the end of the profile.

→ *itemId*
ID of the item to insert in the profile.

→ *parameters*
An array of `CncParameterType` structures that contain parameters associated with the item being inserted. Can be `NULL` if there are no parameters.

Returns Returns the following result codes:

`errNone`
: No error.

`cncErrObjectNotFound`
: The specified profile ID does not exist.

`cncErrInvalidParam`
: The *`itemID`* or *`parameters`* parameter is invalid.

Comments To submit the changes to the Connection Manager database and unlock the profile, call `CncProfileSubmit()`.

If you pass a *`parameters`* array to this function, note that the array is not freed by this function. After you are done using the array, you should call `CncParametersFree()` to free the memory.

CncProfileLock Function

Purpose Locks a stored profile.

Declared In `CncMgr.h`

Prototype `status_t CncProfileLock(uint32_t Id)`

Parameters → *`Id`*
: ID of a profile to lock. You can obtain the ID of a profile by calling `CncObjectGetIndex()`.

Returns Returns the following result codes:

`errNone`
: No error; the profile is locked.

`cncErrObjectNotFound`
: The specified profile ID does not exist.

Comments Use this function to lock a profile before editing it. To submit the changes to the Connection Manager database and unlock the profile, call `CncProfileSubmit()`. To only unlock the profile, call `CncProfileUnlock()`.

CncProfileNew Function

Purpose Creates a new empty, locked profile.

Declared In `CncMgr.h`

Prototype
```
status_t CncProfileNew(char *profileName,
   uint32_t *lockedId)
```

Parameters

→ *`profileName`*
Pointer to the user-visible name of the profile; for example, "My Fast Internet Connection".

← *`lockedId`*
Pointer to the ID of the newly created profile. If the name specified by *`profileName`* already exists, a pointer to that existing object ID is returned.

Returns Returns the following result codes:

`errNone`
No error; the *`lockedId`* parameter is set to the ID of the newly created profile.

`cncErrMemory`
There is not enough memory available to create the new object.

`cncErrCannotAllocateObject`
The Connection Manager object table is full.

`cncErrObjectAlreadyExists`
An object already exists with the name *`profileName`*.

Comments An application can use this function to create a new empty profile. To add items, call `CncProfileInsertItem()`. To submit the changes to the Connection Manager database and unlock the profile, call `CncProfileSubmit()`. To only unlock the profile, call `CncProfileUnlock()`.

CncProfileRegroupSubmit Function

Purpose Resubmits to the Connection Manager database all the subprofiles within an expanded profile as separate profiles. The main profile is recompressed by replacing all the subprofiles with references.

Declared In `CncMgr.h`

Prototype

```
status_t
   CncProfileRegroupSubmit(uint32_t lockedId)
```

Parameters

→ *`lockedId`*
ID of a locked, expanded profile to recompress.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
The specified profile ID is not found.

Comments This function works only if the profile was expanded by a call to `CncProfileUngroup()`, where the *`regroupTags`* parameter was set to true.

All individual subprofiles that exist in the main profile are saved in the Connection Manager database as separate profiles and are unlocked. The main profile is recompressed, submitted to the database, and unlocked.

If there are no subprofiles in the specified profile, then this function has the same effect as `CncProfileSubmit()`.

See Also `CncSubProfileAssign()`

CncProfileSetParameters Function

Purpose Sets the parameters of an item (plug-in or interface object) in a profile.

Declared In `CncMgr.h`

Prototype

```
status_t
    CncProfileSetParameters(uint32_t lockedId,
    int16_t itemIndex,
    CncParameterType parameters[])
```

Parameters

→ *`lockedId`*
: ID of a locked profile. A profile is locked by `CncProfileLock()` or `CncProfileNew()`. You can obtain the ID of a profile by calling `CncObjectGetIndex()` or `CncProfileFindFirst()`.

→ *`itemIndex`*
: Index (zero-based) of an item in the profile.

→ *`parameters`*
: Array of `CncParameterType` structures that contain the parameters to set; the array previously should have been initialized by `CncParametersInit()`. The last item in the parameter array must have the special name `kCncParameterTableEnd`. Note that this item name is set automatically when the array is initialized by `CncParametersInit()`.

Returns Returns the following result codes:

`errNone`
: No error; the *`parameters`* structures are updated with the parameter values.

`cncErrObjectNotFound`
: The specified profile ID does not exist.

`cncErrInvalidParam`
: The *`itemIndex`* or *`parameters`* parameter is invalid.

Comments To remove a parameter from a profile, use the type `kCncUndefinedParameterType`.

To submit the changes to the Connection Manager database and unlock the profile, call `CncProfileSubmit()`.

Example

```
CncParameterType myParams[4];
char* myName = "SomeName";

CncParametersInit(myParams, 4);
myParams[0].name = 'uart';
myParams[0].type = kCncIntegerParameterType; // set an int
myParams[0].value.asInteger = 0;
myParams[1].name = 'DevN';
myParams[1].type = kCncStringParameterType; // set a string
myParams[1].value.asString = myName;
myParams[2].name = 'baud';
myParams[2].type = kCncUndefinedParameterType; // remove this
CncProfileSetParameters(Id, 1, myParams);
```

See Also `CncProfileGetParameters()`

CncProfileSubmit Function

Purpose Submits a changed profile to the Connection Manager database and unlocks the profile.

Declared In `CncMgr.h`

Prototype `status_t CncProfileSubmit(uint32_t` *`lockedId`*`)`

Parameters

→ *`lockedId`*
ID of a profile to submit and unlock.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrInvalidParam`
The specified profile ID is not a locked profile.

See Also `CncProfileUnlock()`

CncProfileUngroup Function

Purpose Locks a profile and expands all subprofiles and macros used in it.

Declared In `CncMgr.h`

Prototype
```
status_t CncProfileUngroup(uint32_t profileId,
   Boolean regroupTags)
```

Parameters

→ *`profileId`*
ID of a profile to expand.

→ *`regroupTags`*
Specify true to allow all macros to be compressing back into macro form later. This option inserts BEGIN and END tags around each expanded portion, to allow future recompression.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
The specified profile ID is not found.

Comments Each subprofile and macro is expanded by copying the subprofile or macro definition into the profile in place of its reference.

If there are no subprofiles or macros in the specified profile, then this function has the same effect as `CncProfileLock()`.

See Also `CncProfileRegroupSubmit()`, `CncSubProfileAssign()`

CncProfileUnlock Function

Purpose Unlocks a locked profile without submitting changes to the Connection Manager database.

Declared In `CncMgr.h`

Prototype
```
status_t CncProfileUnlock(uint32_t lockedId)
```

Parameters

→ *`lockedId`*
ID of a profile to unlock.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrInvalidParam`
The specified profile ID is not a locked profile.

See Also `CncProfileSubmit()`

CncRegisterPluginModule Function

Purpose Registers a code module that contains one or more plug-ins with the Connection Manager and adds the description to the database.

Declared In `CncMgr.h`

Prototype
```
status_t CncRegisterPluginModule(uint32_t dbType,
   uint32_t dbCreator, uint16_t rsrcId)
```

Parameters

→ *`dbType`*
Type of the code module.

→ *`dbCreator`*
Creator ID of the code module.

→ *`rsrcId`*
Code resource ID of the code module.

Returns The ID of the newly added plug-in code module, or 0 if an error occurs. If the code module is already registered, this function returns the ID of the registered module without affecting the Connection Manager database.

Comments This function does not need a session with the Connection Manager. A module can use this function at boot time, before the Connection Manager thread is running.

The referenced code module is sublaunched twice. The code must respond to the `sysCncPluginLaunchCmdGetPlugins` launch code to pass back the plug-in definitions, and then to `sysCncPluginLaunchCmdRegister` if more initialization is needed after the plug-ins are registered with the Connection Manager. For more information on these launch commands, refer to *Exploring Palm OS: Programming Basics*.

CncSubProfileAssign Function

Purpose Changes a subprofile in an expanded profile to a different subprofile, which is also expanded within the main profile.

Declared In `CncMgr.h`

Prototype
```
status_t CncSubProfileAssign(uint32_t lockedId,
    int16_t itemIndex, uint32_t newRefId)
```

Parameters

→ *`lockedId`*
ID of a locked, expanded profile within which to change a subprofile.

→ *`itemIndex`*
Index of the subprofile to change. The index refers to the index of the BEGIN tag that marks the subprofile. The first BEGIN block within a profile has an index of 0.

→ *`newRefId`*
ID of a profile to substitute as a subprofile into the main profile in place of the subprofile identified by *`itemIndex`*.

Returns Returns the following result codes:

`errNone`
No error.

`cncErrObjectNotFound`
The specified profile ID is not found.

Comments This function works only if the profile was expanded by a call to `CncProfileUngroup()`, where the *`regroupTags`* parameter was set to true.

See Also `CncProfileRegroupSubmit()`

Part II
Exchange Manager

The Exchange Manager manages the sending and receiving of typed data objects.

4 Object Exchange

The simplest form of communication for a Palm OS® application to implement is the sending and receiving of typed data objects, such as MIME data, databases, or database records.

You use the Exchange Manager to send and receive typed data objects. The Exchange Manager interface is independent of the transport mechanism. You can use Bluetooth, email, IR, SMS, or any other protocol that has an Exchange Manager plug-in called an **exchange library**.

This chapter describes how applications use the Exchange Manager to send and receive typed data objects. It covers the following topics:

About the Exchange Manager

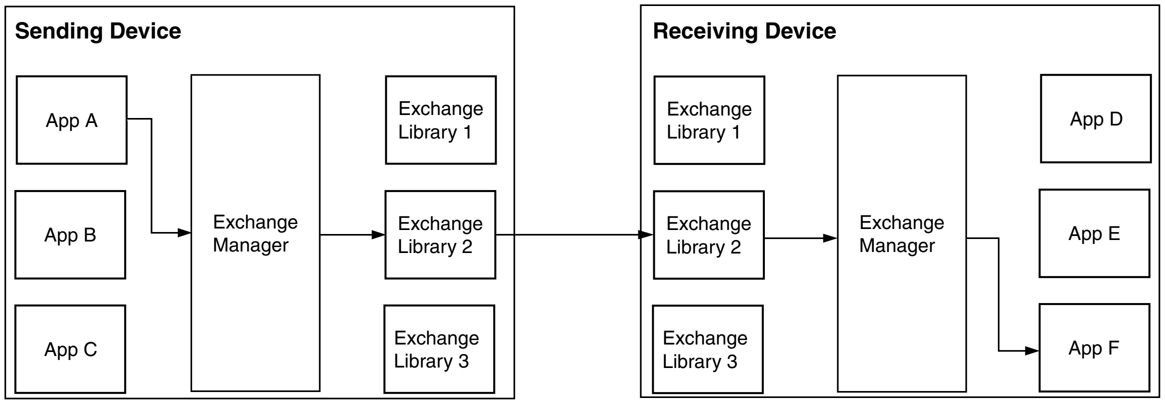
This section explains concepts you need to know before you can begin using the Exchange Manager. It discusses the following topics:

- Exchange Libraries
- Typed Data Objects

Exchange Libraries

The Exchange Manager works in conjunction with an exchange library. Each **exchange library** is transport-dependent and performs the actual communication with the remote device. When an application makes an Exchange Manager call, the Exchange Manager forwards the request to the appropriate exchange library. The Exchange Manager's main duty is to maintain a registry of which libraries implement each protocol and which applications receive each type of data. See Figure 4.1.

Figure 4.1 Object exchange using Exchange Manager



The list of available exchange libraries depends on the particular device hardware and on what other software the user has installed. Some typically available libraries include: IR Library (IrDA), Local Exchange Library, SMS (Short Messaging System) Library, Bluetooth Library, and HotSync® Exchange Library.

As other exchange libraries become available, users can install them on their Palm Powered™ handhelds and use the communications functionality they provide.

Typed Data Objects

The Exchange Manager sends and receives typed data objects. A **typed data object** (or **object**) is a stream of bytes plus some information about its contents. The content information includes any of: a creator ID, a MIME data type, or a filename.

The object itself can be in any format, but it's best to use a standardized data format rather than a proprietary one if you have a choice. Table 4.1 lists the standardized data formats that the built-in Palm OS applications can receive.

Table 4.1 Built-in applications and standard data types

Application	Data Type
Address Book	vCards (vcf file extension, text/x-vCard MIME type). Palm OS supports vCard version 2.1 and most features of version 3.0 (except for the '\' 'n' sequence in properties).
Datebook	vCalendars (vcs file extension, text/x-vCalendar MIME type). Palm OS supports vCalendar version 1.0.
Launcher	Palm OS databases (prc, pdb, oprc, and pqa file extensions, application/x-pilot and application/vnd.palm MIME types)
Memo	Plain text (txt file extension, text/plain MIME type)
ToDo	Not explicitly registered, but receives vCalendar objects from Datebook as appropriate

NOTE: The MIME type application/vnd.palm has been registered with the IANA and is preferred over the application/x-pilot MIME type.

More information on the vCard and vCalendar formats is available at http://www.imc.org/pdi/. For text, the basic MIME text format is described in RFC 822 (http://www.ietf.org/rfc/rfc822.txt). Palm OS builds on that with support for the quoted printable format (for international character sets) in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt) and multipart MIME (for categories) in RFC 2046 (http://www.ietf.org/rfc/rfc2046.txt). Palm OS doesn't implement everything in these RFCs, but it does generate and read content that is compliant with these standards.

If you want your application to receive objects, you must first register with the Exchange Manager for the type of data you want to receive. See "Registering for Data" for instructions on how to do so. You can override the built-in applications by registering for any data type listed in Table 4.1 and becoming the default application for that type (but only with the user's permission). See "Setting the Default Application" for more information.

If you only want to send data, you do not have to register. Your application can send data of the types listed in Table 4.1, and the Exchange Manager ensures that the appropriate application receives it.

Initializing the Exchange Socket Structure

The Exchange Manager, exchange library, and application use an exchange socket structure (`ExgSocketType`) to communicate with each other. This structure is passed from the application to the Exchange Manager to the exchange library and vice versa. (The use of the term "socket" in the Exchange Manager API is not related to the term "socket" as used in sockets communication programming.) When your application sends data, you must create this structure and initialize it with the appropriate information. When you receive data, this structure provides information about the connection and the incoming data.

The `ExgSocketType` structure you use must identify two important pieces of information:

- the exchange library that should do the sending (see "Identifying the Exchange Library")

- the type of data being sent (see "Identifying the Type of Data")

The socket structure defines other fields that you may use to provide other information if you want. See the description of the `ExgSocketType` structure for complete details.

IMPORTANT: When initializing the `ExgSocketType` structure, set all unused fields to 0.

Identifying the Exchange Library

The `ExgSocketType` structure identifies the library to be used via a Uniform Resource Locator (URL) in the `name` field.

When your application sends data, it should always identify which exchange library to use. (If you do not specify an exchange library, the IR Library is used to maintain backward compatibility.)

The URL scheme specifies which exchange library to use. The **scheme** is the part of the URL that appears before the colon (:). For example, the scheme in the following URL is "http"

```
http://www.palmos.com
```

When you pass the preceding URL to a web browser, the scheme tells the browser to connect to the server using the HTTP protocol. Similarly, when you pass the Exchange Manager a URL, the scheme tells the Exchange Manager which exchange library to use. For example, the following URL tells the Exchange Manager to connect to a remote Palm Powered device using the IR Library:

```
_beam:BusinessCard.vcf
```

Multiple exchange libraries can register for the same scheme.

On Palm OS, a URL has the following format (in BNF notation):

`[?][`*`scheme`*`{;`*`scheme`*`}:]`*`filename`*

where:

`?`

If more than one exchange library is registered for the provided schemes, the Exchange Manager has the user select the exchange library by displaying the Send With dialog.

`scheme{;scheme}`

The URL schemes that identify which exchange library should be used. If more than one exchange library is registered for the scheme, the default exchange library is selected unless the URL begins with a question mark.

As shown, multiple schemes may be provided, separated by semicolons. Multiple schemes are only supported in conjunction with the question mark. For example, the string "?_send;_beam:" has the Exchange Manager display a Send With dialog that lists all exchange libraries that support either the _send scheme or the _beam scheme.

`filename`

The name of the file to send. Typically, this file also has an extension that is used, if necessary, to determine which application should receive the data. See "Identifying the Type of Data" for more information about the file extension.

Palm OS URL schemes all begin with the underscore (_) character. Standard schemes, such as mailto, are supported without the underscore.

Palm OS defines some URL prefixes that any application can use to connect with the installed exchange libraries. A URL prefix is everything up to and including the colon character. Table 4.2 describes the prefix constants. Note that you generally only need to use `exgBeamPrefix` or `exgSendPrefix` unless a specific transport is required.

Table 4.2 Exchange Library URL Prefixes

Exchange Library	URL Prefix
IR Library	`exgBeamPrefix`
Local Exchange Library	`exgLocalPrefix`
SMS Library	`kSmsScheme`:
Bluetooth Library	`_btobex:`
Mobile Mail Exchange Library	`exgMobileMailPrefix` and `exgMailtoScheme:`
HotSync Exchange Library	`exgDesktopPrefix`

Table 4.2 Exchange Library URL Prefixes *(continued)*

Exchange Library	URL Prefix
Any library that supports the _send scheme (user's choice)	`exgSendPrefix`
Any library that supports the _send or _beam scheme (user's choice)	`exgSendBeamPrefix`
Any library that supports the _get scheme (user's choice)	`exgGetPrefix`

The section "Implementing the Send Command" on page 122 provides more information on using `exgSendPrefix` or `exgSendBeamPrefix`.

The section "HotSync Exchange" on page 138 provides more information on using `exgDesktopPrefix`.

The section "Attachment Support Guidelines" on page 142 provides more information on using `exgGetPrefix`.

For more information on the SMS exchange library, refer to Chapter 5, "SMS Exchange Library Reference," in *Exploring Palm OS: Telephony and SMS*.

For more information on the Bluetooth exchange library, refer to Chapter 12, "Bluetooth Exchange Library Support," in *Exploring Palm OS: Low-Level Communications*.

Identifying the Type of Data

When your application sends data, the exchange socket structure (`ExgSocketType`) identifies the type of data being sent. It can do so with one of the following values:

- A MIME type in the `type` field.
- A file extension for the file in the `name` field. That is, you might supply `MyDB.pdb` as the value of the `name` field. The part after the last period (.) is the extension.

In most cases, the data type determines which application receives the data on the remote side. (If the `target` field is specified, it determines which application receives the data instead of the data type as described below.) The Exchange Manager maintains a registry of applications and the types of data each application can receive. When the Exchange Manager receives an object, it checks the exchange socket for the data type. It checks the `type` field first, and if it is not defined or if no application is registered to receive that MIME type, it checks the `name` field for a file extension. This is discussed in more detail in the "Registering for Data" section.

Note that you may also directly specify which application should receive the data. To do so, place the creator ID in the `target` field. You do not have to specify a MIME type or file extension in this instance. If the `target` field is nonzero, the Exchange Manager checks to see if an application is registered for that creator ID and, if so, delivers the data directly to that application.

If the target application does not exist, the Exchange Manager searches the registry as usual. Use the `target` field only if you know that you are communicating with a Palm Powered device and want to explicitly specify which application should receive the data.

An application can register for another application's creator ID and receive all objects targeted to that creator ID, but only with the user's permission. See "Setting the Default Application" for more details.

Registering for Data

In most cases, applications that want to receive data from the Exchange Manager must register for the MIME type and/or file extension that they want to receive.

To do so, call `ExgRegisterDatatype()` and pass it five parameters:

- Your application's creator ID.
- A constant that identifies the type of data you want to register to receive: `exgRegExtensionID` for file extensions, `exgRegTypeID` for MIME types, `exgRegCreatorID` for creator IDs (see "Setting the Default Application"), or `exgRegSchemeID` for URL schemes (see "Requesting a URL"). Alternatively, you can register for direct delivery of data (bypassing an email application) by specifying one of these constants: `exgRegDirectExtensionID`,

`exgRegDirectCreatorID`, or `exgRegDirectTypeID`. Or, you can indicate that the application supports data viewing by specifying one of these constants: `exgRegViewExtensionID`, `exgRegViewCreatorID`, or `exgRegViewTypeID`.

- A string that lists the MIME types or file extensions.
- A string containing descriptions of the data you are registering to receive; these are displayed to preview the data in the exchange dialog under certain circumstances.
- A flag value of zero.

For example:

```
ExgRegisterDatatype(beamerCreator,
  exgRegExtensionID, BitmapExt, "bitmap", 0);
```

General Registration Guidelines

Follow these guidelines when registering for data:

- Register as early as possible.

  To ensure that your application can receive data at any time after it is installed, call `ExgRegisterDatatype()` in response to the `sysAppLaunchCmdSyncNotify` and `sysAppLaunchCmdSystemReset` launch codes. The `sysAppLaunchCmdSyncNotify` launch code is sent to your application upon its first installation and any time the HotSync® operation modifies the application's database. The `sysAppLaunchCmdSystemReset` is sent to your application when the system is reset.

- It's best to use a standardized data format rather than a proprietary one if you have a choice.
- Provide user-friendly descriptive information for the *`descriptionsP`* parameter of `ExgRegisterDatatype()`. The descriptions are used in dialogs displayed by Exchange Manager to identify applications or libraries. Use information that describes the type of information handled, such as pictures, sounds, contact information, etc. Don't use MIME types or file extensions because they are not meaningful to the average user.

- Multiple applications can register to receive the same data type, however, an application **must not** automatically register itself as the default application without prompting the user for permission. The section "Setting the Default Application" describes this further. An application might check when it is launched if it is still the default application (with `ExgGetDefaultApplication()`). However, it must honor the user's choices for handling a particular data type by saving this information in its database and by providing a "Don't ask me again" option regarding changing the default.
- When registering for file extensions, do not include the period (.) as part of the extension. Register for "TXT", for example, not ".TXT".
- Do **not** make multiple calls if you want to register for more than one MIME type or more than one file extension.

  Instead, make one call for all file extensions and one call for all MIME types. Pass a single string containing file extensions or MIME types separated by a tab (\t) character. For example, the following call registers the application for two file extensions, TXT and DOC:

  ```
  ExgRegisterDatatype(myCreator,
  exgRegExtensionID,"TXT\tDOC", "plain text", 0);
  ```
- Applications that want to serve as display applications for specific kinds of data should register to receive data in view mode by using one of the view mode constants: `exgRegViewExtensionID`, `exgRegViewCreatorID`, or `exgRegViewTypeID`. For more details on supporting view mode, see "Viewing Attachments" on page 143.

Setting the Default Application

Because multiple applications can register for the same data type, the Exchange Manager supports the concept of a default application that receives all objects of a particular data type. To set the default application, call the function `ExgSetDefaultApplication()`. There is one default application per data type in the registry.

Suppose a device receives a vCard object, and it has three applications registered to receive vCards. The Exchange Manager checks the registry to see if any of these applications is assigned as the default. If so, the default application receives all vCards (unless the exchange socket structure's `target` field is set). If none of the three applications is the default, the Exchange Manager chooses one, and that application receives all vCards.

Do **not** automatically register as the default application without the user's permission. It's imperative that you allow users to choose which application is the default. To do so, you could display a panel via a menu option that shows users the applications that can receive the same type of data as your application, show them which is the default, and allow them to select a different default. Use `ExgGetRegisteredApplications()` to get a list of all applications registered to receive the same data type as yours, and use `ExgGetDefaultApplication()` to retrieve the current default, if any. See Listing 4.2 to see how an application performs this task for the `mailto` URL scheme.

Listing 4.1 Initializing a List of Registered Applications

```
void PrvSetMailAppsList(int32_t listSelection)
{
   ControlPtr ctl;
   ListPtr lst;
   uint32_t defaultID;

   ctl = GetObjectPtr(PrefDefaultMailTrigger);
   lst = GetObjectPtr(PrefDefaultMailList);

   // crIDs, appCnt, appNames are all global variables.
   // Get the list of creator IDs if we don't have it already.
   if(!crIDs) {
      ExgGetRegisteredApplications(&crIDs, &appCnt, &appNames, NULL,
         exgRegSchemeID, "mailto");
      if(appCnt) {
         MemHandle tmpH = SysFormPointerArrayToStrings(appNames, appCnt);
         if(tmpH)
            appNamesArray = MemHandleLock(tmpH);
         else
            return;
      }
      else
         return;
   }

   if(appNamesArray)
      LstSetListChoices(lst, appNamesArray, appCnt);
   LstSetHeight(lst, appCnt < 6 ? appCnt : 6);

   if(listSelection == -1)
   {
      uint16_t i;
      ExgGetDefaultApplication(&defaultID, exgRegSchemeID, "mailto");

      for(i=0;i<appCnt;i++) {
         if(crIDs[i] == defaultID)
            LstSetSelection(lst, i);
      }
   }
   else
      LstSetSelection(lst, listSelection);

   CtlSetLabel(ctl, appNamesArray[LstGetSelection(lst)]);
}
```

To become the default application for a data type that a built-in Palm OS application is registered to receive (see Table 4.1), you must perform some extra steps to ensure that you can receive that type of object when it is beamed from a device running Palm OS 3.X. You must register for the built-in application's creator ID and become the default application for that creator ID.

On Palm OS 3.X, the built-in applications always set their creator IDs in the `target` field when sending data, causing the data to always be sent to that application. On Palm OS 4.0 and higher, the built-in applications still register to receive the same type of data, but they do not set the `target` field when sending. This means that if your application is registered for the same data type and is the default application, it receives the data from devices running Palm OS 4.0 and higher as expected, but if the data is sent from a device running Palm OS 3.X, you still won't receive that data because it is specifically targeted for the built-in application.

To solve this problem, the `ExgRegisterDatatype()` function supports registering for another application's creator ID. Listing 4.2 shows how an application that receives vCards might set the default application after allowing the user to select the default from a list, assuming the list is initialized with code similar to that in Listing 4.1.

Note that, as with all data types, your application won't receive the data targeted for the other application unless yours is the default application for that creator ID.

Listing 4.2 Setting the default application for vCards

```
uint32_t PilotMain (uint16_t cmd, void *cmdPBP, uint16_t launchFlags)
{
   ...
   // Register for vCard MIME type, extension, and Address Book's creator ID.
   // At this point, we are not the default application so we do not receive
   // vCards. We still must register upon install so that our application
   // appears in the preferences list when the user chooses the default
   // application for vCards.
   case sysAppLaunchCmdSyncNotify:
   case sysAppLaunchCmdSystemReset:
      char addressCreatorStr[5];

      // Create a string from Address Book's creator ID.
      MemMove(addressCreatorStr, sysFileCAddress, 4);
      addressCreatorStr[4] = chrNull;

      ExgRegisterDatatype(crID, exgRegTypeID, "text/x-vCard", "vCard", 0);
      ExgRegisterDatatype(crID, exgRegExtensionID, "vcf", "vCard", 0);
      ExgRegisterDatatype(crID, exgRegCreatorID, addressCreatorStr, NULL, 0);
      ...
}

static void PrefApply (void)
{
   MemHandle h;
   FieldType *fld;
   ControlType *ctl;
   uint16_t listItem;

   // Set the default vCard app
vif(appCnt && crIDs)
   {
      uint32_t crID;
      char addressCreatorStr[5];

      // Create a string from Address Book's creator ID.
      MemMove(addressCreatorStr, sysFileCAddress, 4);
      addressCreatorStr[4] = chrNull;

      listItem = LstGetSelection(GetObjectPtr(PrefDefaultAppList));
      crID = crIDs[listItem];
      ExgSetDefaultApplication(crID, exgRegTypeID, "text/x-vCard");
      ExgSetDefaultApplication(crID, exgRegExtensionID, "vcf");
      ExgSetDefaultApplication(crID, exgRegCreatorID, addressCreatorStr);
   }
}
```

Registering to Receive Unwrapped Data

In rare circumstances, you can register to receive data that is sent enclosed in another object.

For example, suppose you have a stock quote application that wants to receive vStock objects. If the device is sent an email message that has the vStock object attached, your application may want to register to receive the vStock object directly rather than having the email application receive it. To do so, call `ExgRegisterDatatype()` and pass one of the direct delivery constants (`exgRegDirectCreatorID`, `exgRegDirectExtensionID`, or `exgRegDirectTypeID`) as the second parameter.

If you want to register to receive an object when it is sent as part of another object, you probably also want to receive it when it is sent by itself. This requires two calls to `ExgRegisterDatatype()`: one with one of the direct delivery constants, and one without.

```
ExgRegisterDatatype(myCreator, exgRegDirectExtensionID, "TXT\tDOC",
  "plain text", 0);
ExgRegisterDatatype(myCreator, exgRegExtensionID, "TXT\tDOC", "plain text", 0);
```

Thus, you might make four calls to `ExgRegisterDatatype()`:

- one call to register for the file extensions
- one call to register for file extensions that are sent as part of another object
- one call to register for MIME types
- one call to register for MIME types that are sent as part of another object

As mentioned previously, it's rare for an application to register to receive unwrapped data directly. It's more common for one application (such as an email application) to receive the entire compound object and then unwrap and disperse the enclosed objects using the Local Exchange Library. See "Sending and Receiving Locally" and "Attachment Support Guidelines" for more information.

Sending Data

This section describes how to send data using the Exchange Manager. It discusses the following topics:

- Sending a Single Object
- Sending Multiple Objects
- Implementing the Send Command

For information about sending data as an attachment from a messaging application such as email or SMS, see "Attachment Support Guidelines" on page 142.

Sending a Single Object

The most common use of the Exchange Manager is to send or receive a single object. To send an object, do the following:

1. Create and initialize an `ExgSocketType` data structure with information about which library to use and the data to be sent. See "Initializing the Exchange Socket Structure" for more information.
2. Call `ExgPut()` to establish the connection with the exchange library.
3. Call `ExgSend()` one or more times to send the data.

   In this function, you specify the number of bytes to send. You may need to call it multiple times if you don't send all the data in the first call.
4. Call `ExgDisconnect()` to end the connection.

   A zero (0) return value indicates a successful transmission. However, this doesn't necessarily mean that the receiver kept the data. If the target application for an object doesn't exist on the receiving device, the data is discarded; or the user can decide to discard any received objects.

Note that the `ExgSend()` function blocks until it returns. However, most libraries provide a user interface dialog that keeps the user informed of transmission progress and allows them to cancel the operation.

The Exchange Manager automatically displays error dialogs as well, if errors occur. You must check for error codes from Exchange Manager routines, but you don't need to display an error dialog if you get one because the Exchange Manager handles this for you.

For example, Listing 4.3 shows how to send the current draw window from one Palm Powered handheld to another Palm Powered handheld.

Listing 4.3 Sending data using Exchange Manager

```
status_t SendData(void)
{
   ExgSocketType exgSocket;
   uint32_t size = 0;
   uint32_t sizeSent = 0;
   status_t err = 0;
   BitmapType *bmpP;

   // copy draw area into the bitmap
   SaveWindow();
   bmpP = PrvGetBitmap(canvasWinH, &size, &err);
   // Is there data in the field?
   if (!err && size) {
      // important to init structure to zeros...
      MemSet(&exgSocket,sizeof(exgSocket),0);
      exgSocket.description = "Beamer picture";
      exgSocket.name = "Beamer.pbm";
      exgSocket.length = size;
      err = ExgPut(&exgSocket);
      if (!err) {
         sizeSent = ExgSend(&exgSocket,bmpP,size,&err);
         ExgDisconnect(&exgSocket,err);
      }
   }
   if (bmpP) MemPtrFree(bmpP);
   return err;
}
```

Sending Multiple Objects

If the exchange library supports it, you can send multiple objects in a single connection. To send multiple objects, do the following:

1. Create and initialize an `ExgSocketType` data structure with information about which library to use and the data to be sent. See "Initializing the Exchange Socket Structure" for more information. You might also supply a value for the `count` field to specify how many objects are to be sent.
2. Call `ExgConnect()` to establish the connection with the exchange library.
3. For each object, do the following:
   a. Call `ExgPut()` to signal the start of a new object.
   b. Call `ExgSend()` one or more times to send the data.

      You may need to call it multiple times if you don't send all the data in the first call.
4. Call `ExgDisconnect()` to end the connection.

   A zero (0) return value indicates a successful transmission. However, this doesn't necessarily mean that the receiver kept the data. If the target application for an object doesn't exist on the receiving device, the data is discarded; or the user can decide to discard any beamed objects.

The `ExgConnect()` call is optional. Some exchange libraries, such as the IR Library, support the sending of multiple objects but do not support `ExgConnect()`. If `ExgConnect()` returns an error, the first call to `ExgPut()` initiates the connection. You should only continue to send objects if the first `ExgPut()` call succeeds. See Listing 4.4. Libraries that support the `ExgConnect()` call also support sending multiple objects without using `ExgConnect()`.

Listing 4.4 Sending multiple objects

```
Boolean isConnected = false;
err = ExgConnect(&exgSocket);      //optional
if (!err)
   isConnected = true;
if (!err || err == exgErrNotSupported) {
   while (/* we have objects to send */) {
      err = ExgPut(&exgSocket);
      if (!isConnected && !err)
         isConnected = true; //auto-connected on first put.
      sizeSent = ExgSend(&exgSocket,dataP,size,&err);
      if (err)
         break;
   }
}
if (isConnected)
   ExgDisconnect(&exgSocket, err);
```

Implementing the Send Command

The built-in applications support a Send menu command. The purpose of this command is to allow the user to send data using any available transport mechanism.

The Exchange Manager defines a _send URL scheme. The intent is that any exchange library that supports sending is registered for the _send scheme. Currently, the Bluetooth, HotSync, SMS, and Mobile Mail libraries are registered for this scheme on release ROMs. The IR Library is **not** registered for the _send scheme.

To implement the Send command in your application, construct a URL that has the prefix `exgSendPrefix`, and send the data in the normal manner. You can also use the `exgSendBeamPrefix` instead so that the user can select from all exchange libraries registered for either sending or beaming (which includes the IR Library). Both of these prefixes begin with a question mark, causing the Exchange Manager to display a dialog if it finds more than one exchange library registered for the specified schemes.

For an example of how to implement the Send command, see the Memo application example code distributed with the Palm OS SDK.

Receiving Data

To have your application receive data from the Exchange Manager, do the following:

1. Register for the type of data you want to receive. See "Registering for Data" on page 112 for more information.

2. Handle the launch code `sysAppLaunchCmdExgAskUser` if you want to control the user confirmation dialog that is displayed. See "Controlling the Exchange Dialog" on page 123 for more information.
3. Handle the launch code `sysAppLaunchCmdExgReceiveData` to view and/or receive the data. See "Receiving the Data" on page 126 for more information on receiving the data, and see "Viewing Attachments" on page 143 for more information about viewing the data; it's best to follow the guidelines in "Put with View Mode" on page 147.
4. If you want, handle `sysAppLaunchCmdGoTo` to display the record.

Controlling the Exchange Dialog

When the Exchange Manager receives an object and decides that your application is the target for that object, it sends your application a series of launch codes. The first launch code your application receives, in most cases, is `sysAppLaunchCmdExgAskUser`.

NOTE: The Exchange Manager allows the exchange library to turn off the user confirmation dialog. In this case, your application does not receive the `sysAppLaunchCmdExgAskUser` launch code.

The Exchange Manger sends this launch code because it is about to display the exchange dialog, which asks the user to confirm the receipt of data. The launch code is your opportunity to accept the data without confirmation, reject the data without confirmation, or replace the exchange dialog.

Responding to this launch code is optional. If you don't respond, the Exchange Manager calls `ExgDoDialog()` to display the exchange dialog.

The `ExgDoDialog()` function allows you to specify that the dialog display a category pop-up list. This pop-up list allows the user to receive the data into a certain category in the database, but the pop-up list is not shown by default. If you want the exchange dialog to display the pop-up list, you must respond to `sysAppLaunchCmdExgAskUser` and call `ExgDoDialog()` yourself. Pass a pointer to an `ExgDialogInfoType` structure. The `ExgDialogInfoType` structure is defined as follows:

```
typedef struct {
  uint16_t     version;
  DmOpenRef  db;
  uint16_t     categoryIndex;
} ExgDialogInfoType;
```

→ *`version`*
Set this field to 0 to specify version 0 of this structure.

→ *`db`*
A pointer to an open database that defines the categories the dialog should display.

← *`categoryIndex`*
The index of the category in which the user wants to file the incoming data.

If `db` is valid, the function extracts the category information from the specified database and displays it in a pop-up list. Upon return, the `categoryIndex` field contains the index of the category the user selected, or `dmUnfiledCategory` if the user did not select a category.

If the call to `ExgDoDialog()` is successful, your application is responsible for retaining the value returned in `categoryIndex` and using it to file the incoming data as a record in that category. One way to do this is to store the `categoryIndex` in the socket's `appData` field (see `ExgSocketType`) and then extract it from the socket in your response to the launch code `sysAppLaunchCmdExgReceiveData`. See Listing 4.5 for an example.

Listing 4.5 Extracting the category from the exchange socket

```
uint16_t categoryID = (ExgSocketType *)cmdPBP->appData;

/* Receive the data, and create a new record using the
   received data. indexNew is the index of this record. */
if (category != dmUnfiledCategory){
   uint16_t attr;
   status_t err;
   err = DmRecordInfo(dbP, indexNew, &attr, NULL, NULL);

   // Set the category to the one the user specified, and
   // mark the record dirty.
   if ((attr & dmRecAttrCategoryMask) != category) {
     attr &= ~dmRecAttrCategoryMask;
     attr |= category | dmRecAttrDirty;
     err = DmSetRecordInfo(dbP, indexNew, &attr, NULL);
   }
}
```

Some of the Palm OS built-in applications (Address Book, Memo, and ToDo) use this method of setting the category on data received through beaming. Refer to the example code provided in the Palm OS SDK for these applications for a more complete example of how to use `ExgDoDialog()`.

When you explicitly call `ExgDoDialog()`, you must set the `result` field of the `sysAppLaunchCmdExgAskUser` launch code's parameter block to either `exgAskOk` (upon success) or `exgAskCancel` (upon failure) to prevent the system from displaying the dialog a second time.

Getting the Object Description

The user might need more information about the object being received, so the Exchange Manager displays information about the object in the exchange dialog. Some exchange libraries do not transmit information for the exchange socket's `description` field, so the Exchange Manager must provide another means of supplying the user with information about the data being received.

The Exchange Manager displays the first item that it locates in the following list:

- The data's description from the exchange socket's `description` field
- The filename in the socket's `name` field
- The receiving application's description as stored in the exchange registry (you pass this description to `ExgRegisterDatatype()` when registering)

- The MIME type in the socket's `type` field
- The file extension in the socket's `name` field

If you want to support viewing the data in an application, the Exchange Manager can launch a display application with the launch code `sysAppLaunchCmdExgReceiveData`. Display applications should register both file extension(s) and MIME type(s) of data that they can handle. Ideally, display applications should register for receiving data in view mode by registering with one or more of the view mode data type constants: `exgRegViewExtensionID`, `exgRegViewCreatorID`, or `exgRegViewTypeID`. Refer to "General Registration Guidelines" on page 113.

For detailed information about supporting data viewing, see "Viewing Attachments" on page 143; particularly, you should follow the guidelines in "Put with View Mode" on page 147. This information applies not only to email attachments, but to any incoming data.

Receiving the Data

If the Exchange Manager receives `exgAskOk` in response to the exchange dialog or the `sysAppLaunchCmdExgAskUser` launch code, the next step is to launch the application with `sysAppLaunchCmdExgReceiveData`. This launch code tells the application to actually receive the data.

To respond to this launch code, do the following:

1. Call `ExgAccept()` to accept the connection.
2. Call `ExgReceive()` one or more times to receive the data.

   In this function you specify the number of bytes to receive, and `ExgReceive()` returns the number of bytes that were received. You may need to call it multiple times if data is remaining to be received after the first and subsequent calls.

   Note that in the socket structure, the `length` field may not be accurate, so in your receive loop you should be flexible in handling more or less data than `length` specifies.
3. If you want your application launched again with the `sysAppLaunchCmdGoTo` launch code, place your application's creator ID in the `ExgSocketType`'s

goToCreator field and supply the information that should be passed to the launch code in the gotoParams field. (The ExgSocketType structure is the parameter block for the sysAppLaunchCmdExgReceiveData launch code.)

4. Call ExgDisconnect() to end the connection.

   A zero (0) return value indicates a successful transmission.

After your application returns from sysAppLaunchCmdExgReceiveData, if the goToCreator specifies your application's creator ID and if the exchange library supports it, your application is launched with sysAppLaunchCmdGoto. In response to this launch code, your application should launch, open its database, and display the record identified by the recordNum field (or matchCustom field) in the parameter block. The Exchange Manager always does a full application launch with sysAppLaunchCmdGoto, so your application has access to global variables; however, if you also use this launch code to implement the global find facility, you may not have access to global variables in that instance. The example code in Listing 4.6 checks to see if globals are available, and if so, calls StartApplication to initialize them.

Listing 4.6 Responding to sysAppLaunchCmdGoto

```
case sysAppLaunchCmdGoto:
  if (launchFlags & sysAppLaunchFlagNewGlobals) {
     err = StartApplication();
     if (err) return err;
     GoTo(cmdPBP, true);
     EventLoop();
     StopApplication();
  } else {
     GoTo(cmdPBP, false);
}
```

Not all exchange libraries support using the sysAppLaunchCmdGoto launch code after the receipt of data.

Because Palm OS supports multiple object exchange, there is no guarantee that your application is the one that is launched at the end of a receipt of data. If multiple objects are being received, it is possible for another application to receive data after yours and to set the `goToCreator` field to its own creator ID. In this case, the last application to set the field is the one that is launched.

Listing 4.7 shows a function that receives a data object and sets the `goToCreator` and `goToParams`.

Listing 4.7 Receiving a data object

```
status_t ReceiveData(ExgSocketPtr exgSocketP)
{
   status_t err;
   MemHandle dataH;
   uint16_t size;
   uint8_t *dataP;
   int16_t len;
   uint16_t dataLen = 0;

   if (exgSocketP->length)
      size = exgSocketP->length;
   else
      size = ChunkSize;
   dataH = MemHandleNew(size);
   if (!dataH) return -1;  //
   // accept will open a progress dialog and wait for your receive commands
   err = ExgAccept(exgSocketP);
   if (!err){
      dataP = MemHandleLock(dataH);
      do {
         len = ExgReceive(exgSocketP,&dataP[dataLen], size-dataLen,&err);
         if (len && !err) {
            dataLen+=len;
            // resize block when we reach the limit of this one...
            if (dataLen >= size) {
               MemHandleUnlock(dataH);
               err = MemHandleResize(dataH,size+ChunkSize);
               dataP = MemHandleLock(dataH);
               if (!err) size += ChunkSize;
            }
         }
      }
      while (len && !err);

      MemHandleUnlock(dataH);

      ExgDisconnect(exgSocketP,err); // closes transfer dialog

      if (!err) {
         exgSocketP->goToCreator = beamerCreator;
         exgSocketP->goToParams.matchCustom = (uint32_t)dataH;
      }
   }
   // release memory if an error occured
   if (err) MemHandleFree(dataH);
   return err;
}
```

Sending and Receiving Databases

It’s common to want to send and receive an entire database using the Exchange Manager. For example, you might want to allow your application’s users to share their versions of the PDB file associated with your application by beaming that file to each other.

Sending and receiving a database involves the extra steps of flattening the database into a byte stream when sending and un-flattening it upon return.

In addition to the process documented in this section, you can now also use the Data Manager function `DmBackupUpdate()` to flatten a database into a bye stream, and `DmRestoreUpdate()` to restore a database from a byte stream. These functions are more flexible than the Exchange Manager functions.

Sending a Database

To send a database, do the following:

1. Create and initialize an `ExgSocketType` data structure with information about which library to use and the data to be sent. See “Initializing the Exchange Socket Structure” for more information.
2. Call `ExgPut()` to establish the connection with the exchange library.
3. Call `ExgDBWrite()` and pass it a pointer to a callback function in your application that it can use to send the database. You make the call to `ExgSend()` in that function.
4. Call `ExgDisconnect()` to end the connection.

The `ExgDBWrite()` function takes as parameters the local ID of the database to be sent and a pointer to a callback function. You may also pass in the name of the database as it should appear in a file list and any application-specific data you want passed to the callback function. In this case, you would pass the pointer to the exchange socket structure as the application-specific data. If you need any other data, create a structure that contains the exchange socket and pass a pointer to that structure instead.

The write callback function is called as many times as is necessary to send the data. It takes three arguments: a pointer to the data to be sent, the size of the data, and the application-specific data passed as the second argument to `ExgDBWrite()`.

Listing 4.8 shows an example of how to send a database. The `SendMe()` function looks up the database creator ID and passes it to the `SendDatabase()` function. The `SendDatabase()` function creates and initializes the exchange socket structure and then passes all that information along to the `ExgDBWrite()` function. The `ExgDBWrite()` function locates the database in the storage heap, translates it into a stream of bytes and passes that byte stream as the first argument to the write callback function `WriteDBData()`. `WriteDBData()` forwards the exchange socket and the data stream to the `ExgSend()` call, sets its size parameter to the number of bytes sent (the return value of `ExgSend()`), and returns any error returned by `ExgSend()`.

Listing 4.8 Sending a database

```
// Callback for ExgDBWrite to send data with Exchange Manager
status_t WriteDBData(const void* dataP, uint32_t* sizeP, void* userDataP)
{
   status_t err;

   *sizeP = ExgSend((ExgSocketPtr)userDataP, (void*)dataP, *sizeP, &err);
   return err;
}

status_t SendDatabase (LocalID dbID, CharPtr nameP, CharPtr descriptionP)
{
   ExgSocketType exgSocket;
   status_t err;

   // Create exgSocket structure
   MemSet(&exgSocket, sizeof(exgSocket), 0);
   exgSocket.description = descriptionP;
   exgSocket.name = nameP;

   // Start an exchange put operation
   err = ExgPut(&exgSocket);
   if (!err) {
      err = ExgDBWrite(WriteDBData, &exgSocket, NULL, dbID);
      err = ExgDisconnect(&exgSocket, err);
   }
   return err;
}

// Sends this application
status_t SendMe(void)
{
   status_t err;
   // Find our app using its internal name
   LocalID dbID = DmFindDatabase(0, "Beamer");
   if (dbID)
      err = SendDatabase(dbID, "Beamer.prc", "Beamer application");
   else
      err = DmGetLastErr();
   return err;
}
```

Note that there is nothing about `ExgDBWrite()` that is tied to the Exchange Manager, so it may be used to send a database using other transport mechanisms as well. For example, if you wanted to transfer a database from your Palm Powered handheld to your desktop PC using the serial port, you could use `ExgDBWrite()` to do so.

Receiving a Database

The Launcher application receives databases with the `.prc` or `.pdb` file extension. If you want your application to be launched when the database is received, you can use a different extension and handle receiving the database within your application. For example, a book reader application might want to be launched when the user is beamed a book. In this case, the book reader application might use an extension such as `.bk` for the book databases.

You receive a database by responding to the same launch codes that you do for receiving any other data object (see "Receiving Data"); however, your response to the `sysAppLaunchCmdExgReceiveData` launch code is a little different:

1. Call `ExgAccept()` to accept the connection.
2. Call `ExgDBRead()` and pass it a pointer to a callback function in your application that it can use to read the database. You make the call to `ExgReceive()` in that function.
3. Call `ExgDisconnect()` to end the connection.

The `ExgDBRead()` function takes as parameters two pointers to callback functions. The first callback function is a function that is called multiple times to read the data. The second function is used if the database to be received already exists on the device.

Requesting Data

This section describes how to use the Exchange Manager to request data. It covers:

- Sending a Get Request for a Single Object
- Responding to a Get Request
- Two-Way Communications
- Requesting a URL

Some exchange libraries may allow you to request data from a remote device through a call to `ExgGet()`. If supported, you can use `ExgGet()` to implement two-way communications between two Palm Powered devices.

NOTE: The only standard exchange library that supports `ExgGet()` is the Local Exchange Library; currently there are no transports that support remote get requests.

For information on using `ExgGet()` to attach a document to a message from a messaging application, see "Sending an Attachment from a Messaging Application" on page 150.

Sending a Get Request for a Single Object

To request data from a remote device, do the following:

1. Create and initialize an exchange socket structure (`ExgSocketType`) as described in "Initializing the Exchange Socket Structure"section. The data structure should identify the exchange library and the type of data that your application wants to receive.
2. Call `ExgGet()` to establish the connection and request the data.

   In response, the exchange library establishes a connection with the remote device, and upon return has data that your application should receive. If the remote device is a Palm Powered device, the exchange library obtains this data from an application on the remote side using the process described in the "Responding to a Get Request" section.
3. Call `ExgReceive()` one or more times to receive the data.
4. Call `ExgDisconnect()` to end the connection.

Responding to a Get Request

When the Exchange Manager receives a get request, it launches the appropriate application with the launch code `sysAppLaunchCmdExgGetData`. Applications can register their support for the Get mechanism by registering to handle the `_get` scheme.

NOTE: Since no standard transports support remote get requests, this section describes how to support a local get request.

Your response to the `sysAppLaunchCmdExgGetData` launch code should be to send the requested data:

1. Present a document selection screen so that the user can choose an object to send.
2. Set the name, type, and description fields in the socket.
3. Call `ExgAccept()`. (Do not call `ExgPut()`.)
4. Call `ExgSend()` one or more times.

5. Call `ExgDisconnect()` when finished.

For more information on supporting the Get mechanism, see "Sending an Attachment from a Messaging Application" on page 150.

See the "Sending a Single Object" section for more information on sending objects.

Two-Way Communications

You can use `ExgGet()` and `ExgPut()` in combination with the `ExgConnect()` call to have your application perform two-way communication. For example, you may want to implement two-way communication in a multiuser game.

In such a situation, one device acts as a client and the other acts as a server. The client calls `ExgConnect()`, which tells the exchange library that a connection is established to perform multiple operations, such as the sending of multiple objects. The client then calls `ExgGet()` or `ExgPut()` repeatedly and calls `ExgDisconnect()` when finished. On the server device, the appropriate application is launched for each of these requests. The server also calls `ExgDisconnect()` when it is done sending or receiving each object. The swapping of client and server roles is not supported.

Remember that not all exchange libraries support `ExgConnect()` and `ExgGet()`. If either one of these returns an error, your application should assume that this feature is not available.

Getting the Sender's URL

For some applications, you might need to know the URL that addresses the remote device from which you are receiving data. This is especially useful for games and other two-way communications. You can get the URL after calling `ExgAccept()` by calling `ExgControl()` and passing the `exgLibCtlGetURL` operation code.

Not all exchange libraries support this operation. The Bluetooth exchange library does support it, and you can find more information in Chapter 12, "Bluetooth Exchange Library Support," in *Exploring Palm OS: Low-Level Communications*.

Requesting a URL

In addition to requesting data with an `ExgGet()` call, you can request a URL with a `ExgRequest()` call. The idea behind the `ExgRequest()` call is to follow the model of pull technology. You could, for example, implement a web browser if you had an exchange library that supported the HTTP protocol. You could then send an `ExgRequest()` call with an exchange socket containing a URL such as `http://www.palmos.com` and receive the web page in response.

The fundamental differences between `ExgRequest()` and `ExgGet()` are:

- `ExgRequest()` does not automatically send the data back to the application that requested it. With `ExgRequest()`, when the exchange library receives the requested data, it has the Exchange Manager send it to the default application for that data type.
- Applications can register for URLs sent using `ExgRequest()`. `ExgRequest()` first looks for an exchange library that handles the URL scheme. If it cannot find one, it looks for an application instead. If it finds an application, it launches it with the `sysAppLaunchCmdGoToURL` launch code.

  For example, suppose an application that handles email registers for the `mailto` URL scheme. If another application wants to implement an email command, it could do so by calling `ExgRequest()` and passing an exchange socket with a URL that begins with `mailto`. In response to this command, the Exchange Manager launches the application that handles email, allowing the user to compose the email.

For information on another method of implemented email and attaching a document to a message, see "Sending an Attachment from a Display Application" on page 154.

Sending and Receiving Locally

Most of this chapter has described how to use the Exchange Manager to send data to a remote device and receive data from a remote device.

You may also use the Exchange Manager to exchange data with other applications on the local device. To do so, use the Local Exchange Library. You might want to do so in the following circumstances:

- You might have an application that creates some sort of event in the Datebook application. Your users might have an application that they use in place of the built-in Datebook. To ensure that the appointment is sent to the user's chosen application, you can send that data as a vCalendar object using the Local Exchange Manager. This way, whichever application is the default in the Exchange Manager registry is the one that receives your vCalendar.
- Your application receives compound data objects, such as email messages that contain attachments intended for other applications. As described in the "Registering to Receive Unwrapped Data" section, exchange libraries can "unwrap" a compound object and deliver the objects it contains directly; however, doing so is the exception the rule.

  It's much more common for the email message to be sent to the email application and have the attachments delivered to the appropriate applications only when the user requests it. In response to a user request, the email application extracts the attached object and uses the Local Exchange Library to send it to the application that should receive it, for viewing and/or storage. For detailed guidelines on handling attachments, see "Attachment Support Guidelines" on page 142.
- Your application exchanges data with a remote device, and you want to debug the code that interacts with the Exchange Manager. In this case, using the Local Exchange Library causes your application to send data in loopback mode, where it is also the recipient of the data.

To use the Local Exchange Library, do the following:

1. Use a URL in the `name` field of the [`ExgSocketType`] structure to identify the Local Exchange Library. Begin the URL with the constant string `exgLocalPrefix`.
2. If you want to suppress the exchange dialog, create and initialize an [`ExgLocalSocketInfoType`] structure and assign it to the socket's `socketRef` field.

```
typedef struct {
  Boolean freeOnDisconnect;
  Boolean noAsk;
  ExgPreviewInfoType *previewInfoP;
  ExgLocalOpType op;
  FileHand tempFileH;
} ExgLocalSocketInfoType;
```

where the following are parameters you might want to set:

`freeOnDisconnect`	Determines whether the structure is freed when the `ExgDisconnect()` call is made. The default is `true`. In general, code that allocates a structure should be responsible for freeing that structure. Therefore, if you have allocated `ExgLocalSocketInfoType`, you should set this field to `false` and explicitly free the structure when you are finished with it.
`noAsk`	Set to `true` to disable the display of the exchange dialog. For example, if you want to create a vCalendar object and send it to the datebook application in response to a user command, you probably want to set `noAsk` to `true` so that the user does not have to confirm the receipt of the data they just requested you to send.
`previewInfoP`	A pointer to an [`ExgPreviewInfoType`] structure, used to display a preview of the data. The preview feature is deprecated and is maintained only for backward compatibility.

All other fields are set by the Local Exchange Library. If you don't create this structure, the library does it for you; therefore, you only need to create this structure if you want to supply non-default values for the `noAsk` or `previewInfoP` fields.

3. You can suppress the display of the progress dialogs that the exchange libraries typically display by setting the `noStatus` field of the `ExgSocketType` structure to `true`.
4. Send and receive data in the normal manner. See "Sending Data" and "Receiving Data" for details.

Interacting with the Launcher

When you beam an application from the Launcher, other databases can be automatically beamed with it. If the application has an associated overlay database, the overlay is beamed along with the application. You do not have to perform any extra work to allow this to happen. This bundling behavior is available only when beaming from the launcher, not if an application manually beams an application.

In addition to beaming overlays, you can set up a record database so that the Launcher beams it along with the application database and the overlay. For example, a dictionary application might have its dictionary data in an associated database. When a user beams the dictionary application to another user, the dictionary data should be beamed along with the application itself. To allow this to happen, you set the bit `dmHdrAttrBundle` in the database's attributes.

If you beam an application plus databases to a device running Palm OS 4.0 or higher, the user sees a single confirmation message. If you beam the application to a device running Palm OS 3.X, the device receives only the application database and displays an alert saying that it cannot receive the other databases.

HotSync Exchange

HotSync Exchange allows a Palm OS Cobalt device and a desktop computer running the Palm Desktop to exchange files in their native formats. For example, HotSync Exchange enables installation of

JPEG (.JPG) files to a handheld image viewer via the standard HotSync desktop install tool and the HotSync exchange library, provided the viewer application has registered with the Exchange Manager for the .JPG extension. This feature eliminates the need for the viewer application developer to write a custom conduit to pack JPEG data into the Palm OS database format.

Similarly, the viewer application can send JPEG files via the Exchange Manager directly to the desktop where HotSync stores them in standard .JPG format.

HotSync Exchange also supports bundled install, which is the installation of an application and its data files in a single HotSync session. Bundled install requires the newly installed application to register for its data types with the Exchange Manager when it receives the `sysAppLaunchCmdSyncNotify` launch code. This allows the HotSync exchange library to deliver the data files when it receives the subsequent `sysNotifySyncFinishEvent`.

For more information about the desktop side of HotSync Exchange, refer to the book *Introduction to Conduit Development*.

Note that HotSync Exchange is supported only by Palm OS devices running Palm OS Cobalt.

Sending Files with HotSync Exchange

Applications use the HotSync exchange library to send files to a HotSync desktop. The HotSync exchange library supports the following Exchange Manager schemes:

- The desktop scheme (_desktop). This scheme supports direct exchange of a file to a HotSync desktop.
- The _send scheme. This scheme allows users to send data using any transport that supports this scheme, such as the HotSync exchange library.

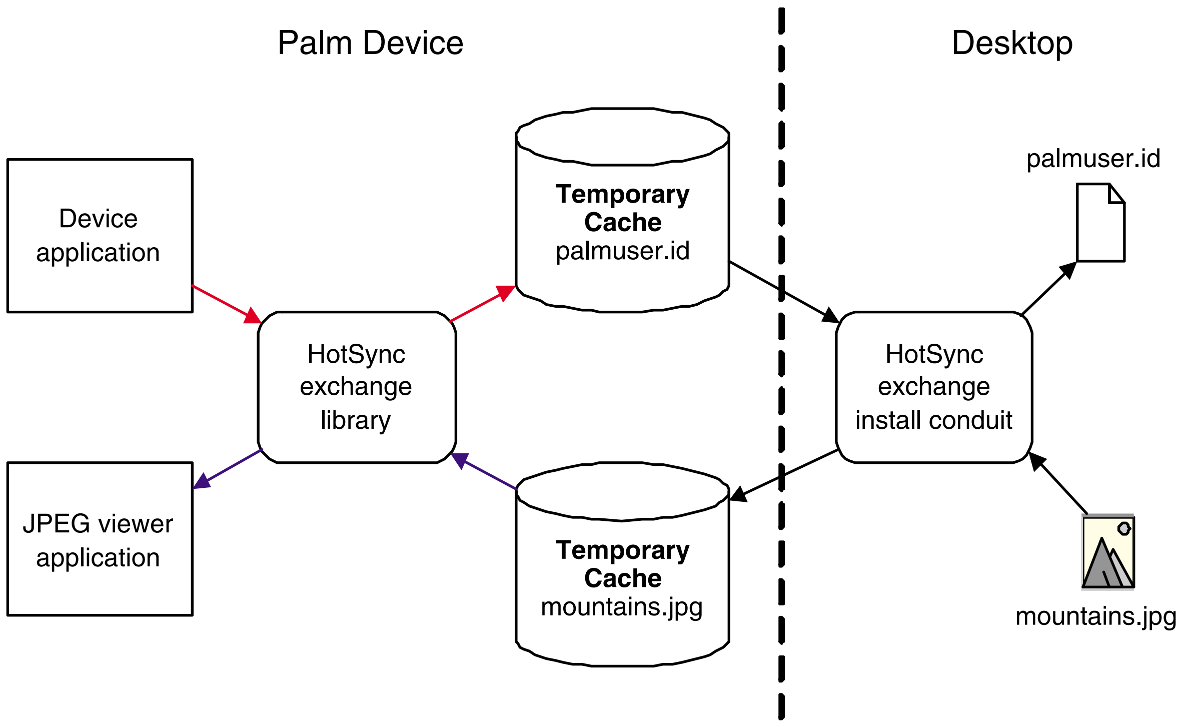
These schemes support the exchange of files during a HotSync operation to a directly connected desktop.

Neither scheme supports the specification of a target desktop in the Exchange URL, so the file will, by default, be sent during the next HotSync operation to any desktop. However, the user may select a specific target desktop for each file pending HotSync Exchange via the HotSync client user interface, if desired (see below). This will cause the file to be sent during the next HotSync operation with the selected desktop.

Example

The following example illustrates the data flow involving the HotSync exchange library and the desktop. Say that a device application wants to transfer palmuser.id to the desktop. The user also wants to install the picture mountain.jpg to the handheld to view on a registered JPEG picture viewer.

Figure 4.2 HotSync Exchange example



Prior to the HotSync Operation

Before the HotSync operation the following operations are performed:

The user queues the file mountain.jpg for install to the handheld during a subsequent HotSync. The file is queued in a user-specific download folder on the desktop.

The device application uses the Exchange Manager API to do the following (represented by red lines in the figure):

1. Initialize the `ExgSocketType` structure with the file name palmuser.id.
2. Call `ExgPut()`. This causes the HotSync exchange library to allocate a temporary cache to store the data. It also stores information about the pending transaction in a catalog of pending desktop exchanges.
3. Call `ExgSend()` to fill the database with the data.
4. Call `ExgDisconnect()` to close the cache. The data is then queued until it can be successfully sent during a HotSync operation.

During the HotSync Operation

During the HotSync Exchange operation, the conduit performs several actions, represented by the black lines in the figure.

The HotSync Exchange conduit first checks if there are any pending handheld-to-desktop transfer requests for the local desktop by examining the handheld exchange catalog mentioned above. Since the conduit finds an entry, it proceeds to create a desktop file with the associated file name (palmuser.id) in the user's directory on the desktop that contains the contents of the associated temporary cache. Afterwards, the conduit deletes the catalog entry and the temporary cache from the handheld.

The conduit then checks to see if there are any files to be installed to the device. It finds mountains.jpg and creates a cache named mountains.jpg on the handheld.

After the HotSync Operation

After the HotSync operation the following operations are performed (represented by blue lines in the figure):

The system launches newly installed applications and applications whose data has been modified by the HotSync operation. At this point newly installed applications can register supported file types with the Exchange Manager. (So, if the HotSync operation had installed a JPEG viewer, it would now register for .JPG files). When the HotSync exchange library receives the `sysNotifySyncFinishEvent` notification, it searches for temporary HotSync caches. On finding mountains.jpg, it opens the cache and calls `ExgNotifyReceive()`. This causes the Exchange Manager to launch the viewer for JPEG files and transfer the file mountains.jpg to it. The application may choose to convert and store it in a Palm OS database. When the application calls `ExgDisconnect()`, the HotSync exchange library deletes the temporary cache created by the conduit.

The HotSync exchange library disables the confirm receipt dialog that is normally displayed when data is sent via the Exchange Manager. Thus there is no user interface on the handheld device during a successful desktop to handheld exchange.

Attachment Support Guidelines

This section outlines how Palm OS applications should interoperate to exchange email attachments. On other platforms attachments are stored intermediately on a file system for the handover between a messaging application (email, instant messaging, etc.) and a display application (word processor, image viewer, etc.). On Palm OS, which doesn't provide a file system, the Exchange Manager enables data exchange between applications.

The following sections describe how applications should use the Exchange Manager to handle attachments. These guidelines were designed with the following goals:

- Leveraging existing capabilities of today's applications.
- Ensuring backwards compatibility with Palm OS 4 by using existing standard mechanisms.
- Providing a good user experience.

Application providers are strongly encouraged to follow these guidelines to enable a consistent user experience on Palm OS.

The following topics are covered:

- Viewing Attachments
- Sending an Attachment from a Messaging Application
- Sending an Attachment from a Display Application
- Email Application Guidelines

The Attachment Support sample code shows how to implement the guidelines covered in this section. It is available in the Developer Knowledge Base at http://kb.palmsource.com/

Viewing Attachments

Messaging applications should leverage dedicated display applications already available on the device rather than implementing their own content viewers. Passing data from the messaging application to a display application is facilitated by the Exchange Manager.

NOTE: The data viewing mechanism described here can be used for more than just viewing email attachments. It can be used as a general purpose mechanism to view any data incoming via the Exchange Manager. Follow the guidelines for display applications to support data viewing in a display application.

There are two methods of exchanging attachments:

- Regular Put: This is the standard put mechanism (using `ExgNotifyReceive()` or `ExgPut()`). It is backwards compatible with older applications that don't know about attachment support.
- Put with View Mode: This method defines an enhanced view mode. It enables display applications to distinguish between data sent for temporary viewing versus data sent to be accepted into the database.

  Additionally, this method defines a return mechanism to the messaging application. This allows display applications to execute a full launch of one or more other applications before returning control to the messaging application.

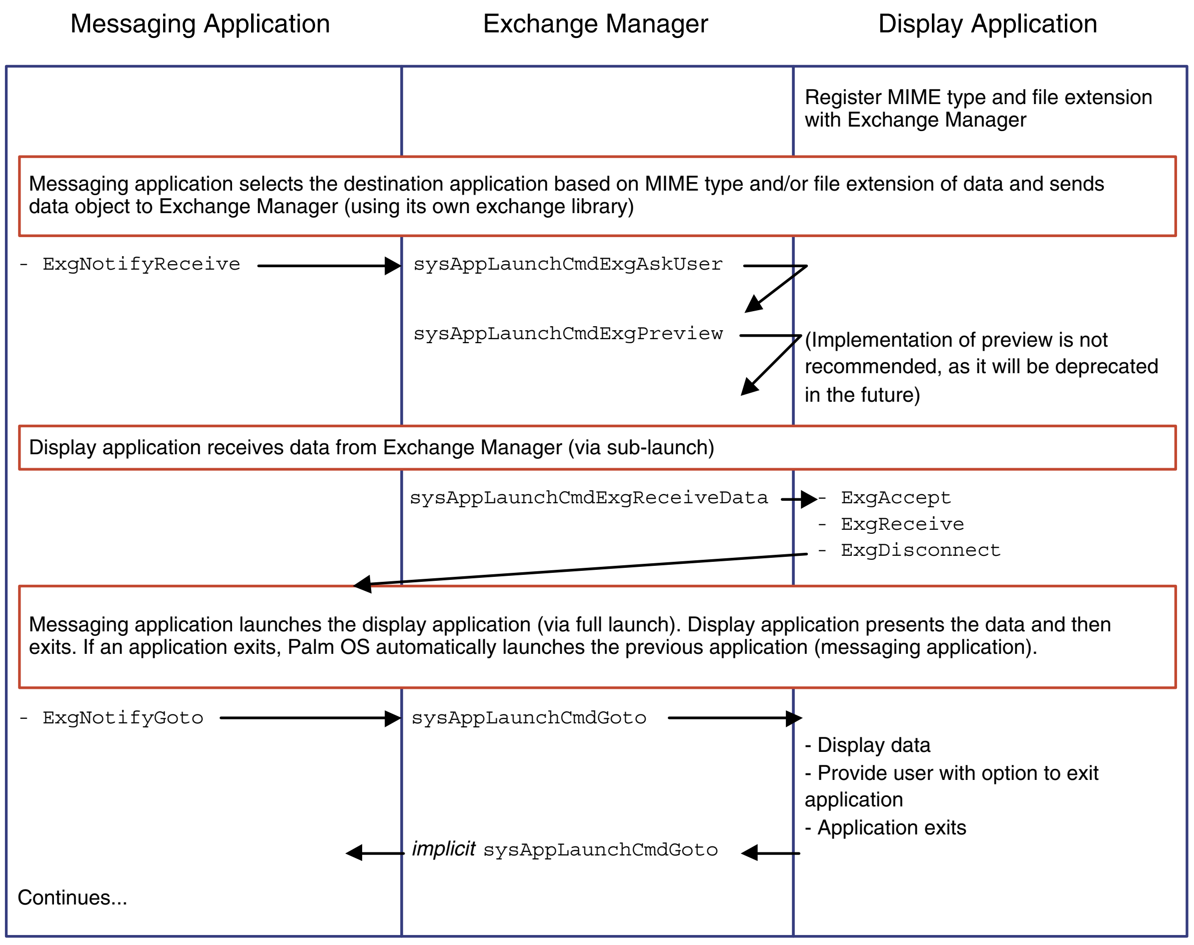
Regular Put

This method, which has been around for several years, enables an application to send data to another application. Older display applications, which are not attachment aware, will simply accept the data into their database and not return to the messaging application. (The user needs to return via the launcher). Although this behavior doesn't present an optimal user experience, it provides a way to take advantage of older applications.

Newer applications, which follow the guidelines outlined in this section, provide the user with the option to return to the messaging application (for example, via a Done button). The recommended method is that the display application simply exits. The system will automatically launch the previous application (implicit launch of the messaging application).

The interaction between the messaging application, Exchange Manager, and the display application for a regular Put operation is shown in Figure 4.3. Specific guidelines for messaging applications and display applications follow the figure.

Figure 4.3 Regular Put operation


Messaging Application
Exchange Manager
Display Application
Register MIME type and file extension with Exchange Manager
Messaging application selects the destination application based on MIME type and/or file extension of data and sends data object to Exchange Manager (using its own exchange library)
- ExgNotifyReceive
sysAppLaunchCmdExgAskUser
sysAppLaunchCmdExgPreview
(Implementation of preview is not recommended, as it will be deprecated in the future)
Display application receives data from Exchange Manager (via sub-launch)
sysAppLaunchCmdExgReceiveData
- ExgAccept
- ExgReceive
- ExgDisconnect
Messaging application launches the display application (via full launch). Display application presents the data and then exits. If an application exits, Palm OS automatically launches the previous application (messaging application).
- ExgNotifyGoto
sysAppLaunchCmdGoto
- Display data
- Provide user with option to exit application
- Application exits
implicit sysAppLaunchCmdGoto
Continues...

Guidelines for Messaging Applications

To handle attachment viewing with the regular Put method, messaging applications should follow these guidelines:

- Messaging applications are strongly encouraged to implement their own exchange library to support the Put mechanism (minimal implementation of accept/receive/ disconnect).

  Although Put can be implemented without an exchange library (by using `ExgPut()`, `ExgSend()`, and `ExgDisconnect()`), this implementation is slow, especially for large attachments, because the Exchange Manager first reads and copies all the data before handing it over to the display application.

- Messaging applications can query the Exchange Manager registry to determine if a dedicated application is available to handle a specific content type (use `ExgGetRegisteredTypes()`). If no application can handle the attachment, the messaging application may convert the content into a different format that can be displayed on the device. The conversion might take place either on the client side or, in the case of a distributed email solution, on a server or desktop computer. For example, if no application has registered with the Exchange Manager to handle HTML content, the email application could convert it to text and display it itself.

- Whenever possible, messaging applications should include the MIME type information (besides the file extension) when sending data to the Exchange Manager. File extensions do not uniquely identify the content of a document. In some cases, the same file extension has been used for different file formats (for example, ".doc"). Refer to IANA (http:// www.iana.org/assignments/media-types/) for the official list of registered MIME types.

  Some notes on MIME types: MIME type information in the Content-Type field of email messages isn't always reliable. Some email programs use non-registered MIME types (for example, application/powerpoint instead of application/ vnd.ms-powerpoint) or use a generic MIME type (for example, application/octet) when attaching documents. It is recommended that email applications analyze the Content-

Type field carefully (including file extensions), and map it to the correct MIME type before passing this information to the Exchange Manager. Developers of messaging applications are encouraged to present a list of applications that can handle the attachment, instead of simply using the default application. (The `ExgGetRegisteredApplications()` function allows messaging applications to query for applications that have registered for a specific MIME type or file extension).

- Messaging applications should send only one data object (document) at a time to a document application in a single connection. Although the exchange library is capable of handling multiple objects in a single connection, the assumption is that a display application can display only one document at a time.
- For regular Put, the default Exchange Manager dialog ("Do you want to accept…") should not be disabled. (Use `ExgNotifyReceive()(…,0)` in the exchange library).
- Messaging applications are responsible for transport-related encoding and decoding of the content. All content exchanged with a display application happens in binary format.

Guidelines for Display Applications

To handle attachment viewing with the regular Put method, display applications should follow these guidelines:

- Display applications should register both file extension(s) and MIME type(s) of data that they can handle. Refer to "General Registration Guidelines" on page 113.
- Applications should be prepared to receive invalid data and deal with this situation gracefully.
- Applications that accept data or documents into their databases should avoid unnecessary duplication. For instance, if a user views an attachment several times, the data should not be duplicated.
- Applications should provide the user with an option to return to the messaging application (for example, a Done button).
- Display applications and messaging applications exchange data in its normal binary format. Messaging applications will

encode/decode the data according to their transport requirements.

Put with View Mode

This method defines an extension to the standard Put mechanism. It allows display applications to implement a special view mode where data is displayed, but not automatically added to the application's database. During view mode, the default Exchange Manager "Accept…" dialog can be disabled, which saves the user an extra step.

Moreover, view mode defines a mechanism to return to the messaging application across an arbitrary number of nested launches of other applications. This is useful, for instance, when a messaging application hands a .zip or .tar file to a decompression utility application that, after the user chooses a specific document, sends the content to the appropriate display application.

Implementation of the view mode requires collaboration of both the messaging and display applications. Display applications indicate support for view mode by using the following exchange registry IDs during registration:

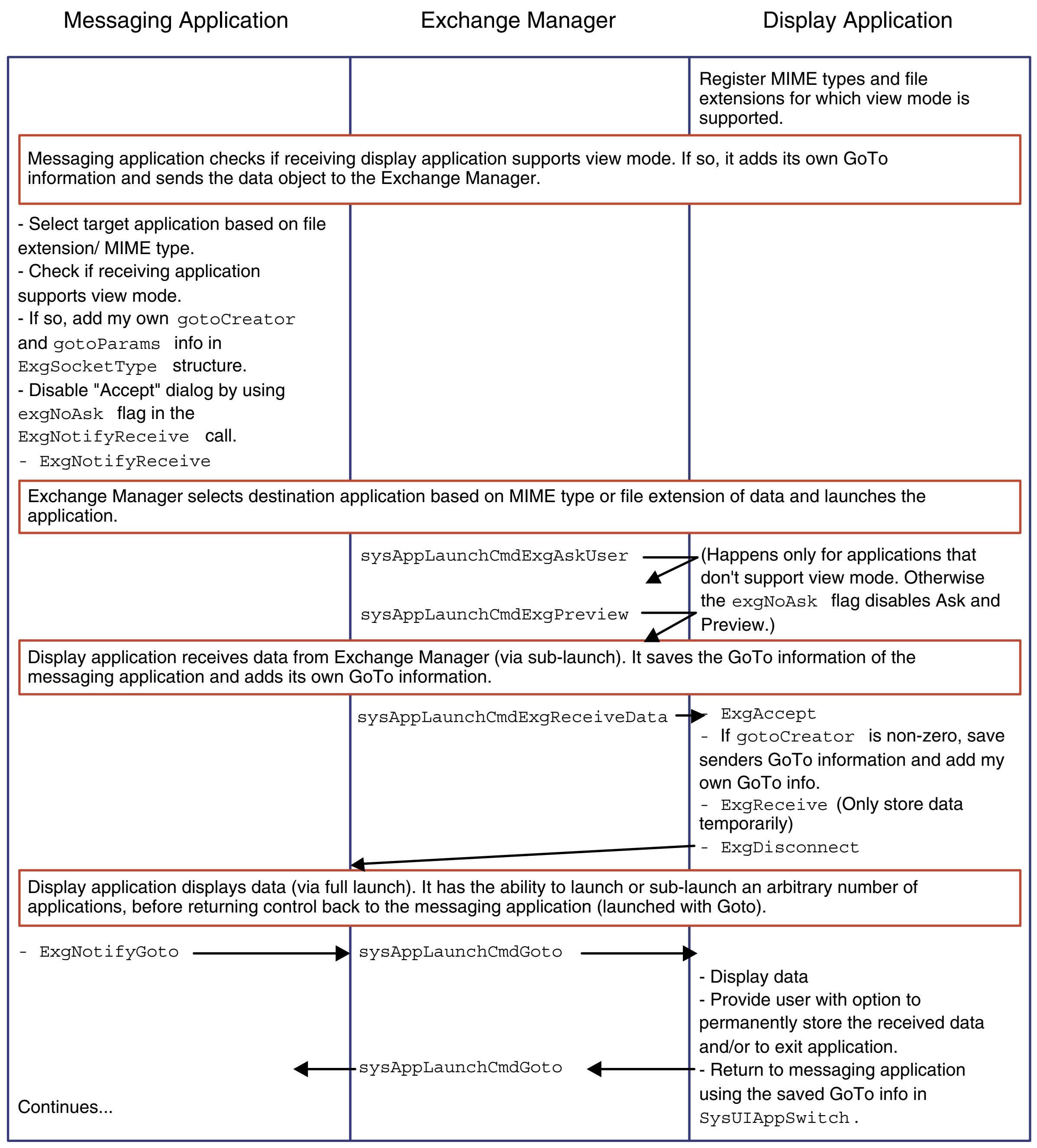
```
// New Exchange registry IDs for View registry. Don't change values!
#define exgRegViewExtensionID 0xff8d // filename ext. registry for View
#define exgRegViewTypeID 0xff8e // MIME type registry for View

ExgRegisterDatatype(…,exgRegViewExtensionID,…,…,…);
ExgRegisterDatatype(…,exgRegViewTypeID,…,…,…);
```

These two exchange registry IDs, one for file extensions and the other for MIME types, allow display applications to support view mode only for a subset of the content they usually accept.

Before a messaging application calls `ExgNotifyReceive()`, it must first check if the receiving application supports view mode for the content that is to be sent. If the display application supports view mode, the messaging application must add its own GoTo information to the `ExgSocketType` structure. Figure 4.4 shows the interaction between the applications and the Exchange Manager in more detail.

Figure 4.4 Put with view mode


Messaging Application
Exchange Manager
Display Application
Register MIME types and file extensions for which view mode is supported.
Messaging application checks if receiving display application supports view mode. If so, it adds its own GoTo information and sends the data object to the Exchange Manager.
- Select target application based on file extension/ MIME type.
- Check if receiving application supports view mode.
- If so, add my own gotoCreator and gotoParams info in ExgSocketType structure.
- Disable "Accept" dialog by using exgNoAsk flag in the ExgNotifyReceive call.
- ExgNotifyReceive
Exchange Manager selects destination application based on MIME type or file extension of data and launches the application.
sysAppLaunchCmdExgAskUser
sysAppLaunchCmdExgPreview
(Happens only for applications that don't support view mode. Otherwise the exgNoAsk flag disables Ask and Preview.)
Display application receives data from Exchange Manager (via sub-launch). It saves the GoTo information of the messaging application and adds its own GoTo information.
sysAppLaunchCmdExgReceiveData
ExgAccept
- If gotoCreator is non-zero, save senders GoTo information and add my own GoTo info.
- ExgReceive (Only store data temporarily)
- ExgDisconnect
Display application displays data (via full launch). It has the ability to launch or sub-launch an arbitrary number of applications, before returning control back to the messaging application (launched with Goto).
- ExgNotifyGoto
sysAppLaunchCmdGoto
- Display data
- Provide user with option to permanently store the received data and/or to exit application.
- Return to messaging application using the saved GoTo info in SysUIAppSwitch.
sysAppLaunchCmdGoto
Continues...

Guidelines for Messaging Applications

Messaging applications are strongly encouraged to support view mode.

In addition to the guidelines defined in "Regular Put" on page 143, messaging applications that want to take advantage of display applications with view mode support must follow these guidelines:

- Messaging applications must check if the receiving application supports view mode, before adding their own GoTo information. The default "Accept" dialog can be disabled only if the target application supports view mode, resides on the same device, and if GoTo information is added.

```
ExgGetRegisteredApplications(…, …, …,
exgRegViewExtensionID,…)
ExgGetRegisteredApplications(…, …, …, exgRegViewTypeID,…)
…
if (supportsView) {
  add_my_GoTo_info();
  ask = exgNoAsk;
}
ExgNotifyReceive(…, ask);
```

- If the receiving application does not support view mode, the `gotoCreator` field and the `goToParams` structure fields in the `ExgSocketType` structure passed to `ExgNotifyReceive()` (or `ExgPut()`) must be set to zero and the "Accept" dialog must not be disabled. The reason for this is that some applications (for example, certain built-in PIM applications) that don't support view mode, use some `goToParams` fields to determine certain states. Supplying wrong information could lead to unwanted behavior.
- When a messaging application is relaunched after view mode, it should return to the same state it was in when the display application was launched. Especially, the same message should be displayed and any splash screens should be skipped.

Guidelines for Display Applications

In addition to the guidelines defined in "Regular Put" on page 143, display applications that want to implement view mode support must follow these guidelines:

- Support for view mode must be indicated by registering with the Exchange Manager like this:

```
// New Exchange registry IDs for View registry. Don't change values!
#define exgRegViewExtensionID 0xff8d // filename extension ID
#define exgRegViewTypeID 0xff8e // MIME type registry

ExgRegisterDatatype(…,exgRegViewExtensionID,…,…,…);
ExgRegisterDatatype(…,exgRegViewTypeID,…,…,…);
```

- Always check if your application is called in view mode by checking if `goToCreator` information is available (in the `ExgSocketType` structure).
- When applications receive data in view mode, the data should be stored temporarily and not automatically accepted into the application database. Instead, the user should be given the option to save the data. To save memory space, temporary data should be removed before the application exists.
- Applications that support view mode must provide an option to return to the messaging application (for example, a Done button). Returning to the messaging application must be implemented by actively launching the messaging application (with `SysUIAppSwitch()` or the `AppLaunchWithCommand()` macro), using the GoTo information provided by the messaging application.

Sending an Attachment from a Messaging Application

Traditionally, when the user composes a message, messaging applications provide an option to select a document and attach it to the outgoing message. This has not been straightforward on Palm Powered devices in the past, since the Palm OS doesn't provide a file system. These guidelines define a standard way to implement this feature using the Exchange Manager's Get mechanism. This mechanism isn't limited to attachment support, but rather defines a general import mechanism.

The process of selecting an attachment includes two steps:

1. Let the user select the application where the data is stored.
2. Launch this application and let the user pick from a list of data objects.

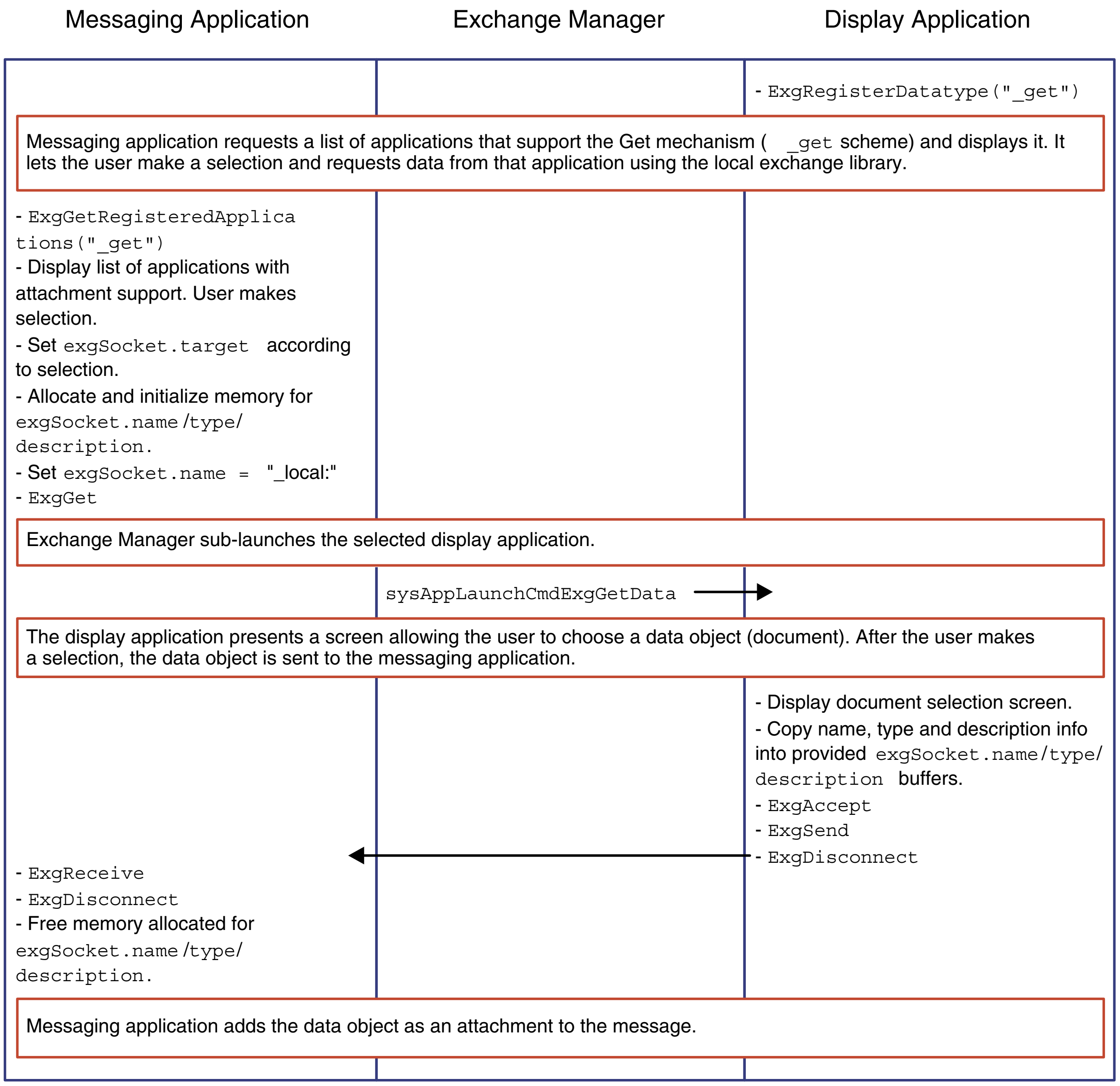
The idea is to enable users to perform a data lookup by application (in contrast to a path lookup on other platforms). This fits the Palm OS model where data is associated with one particular application. The application that stores the data is responsible for providing the user interface in step 2. This makes it possible to present a user interface that is optimized for that particular data type. For instance, a calendar application can present information in an appropriate calendar context rather then as a simple list, allowing for a better user experience.

The Get mechanism requires implementation for both the messaging application as well the display application. Although the Get mechanism has been available since Palm OS 4, it hasn't been widely used yet by applications, due to lack of guidelines and usage scenarios. However, with more and more applications supporting the exchange of native file formats across different platforms, this situation will change rapidly.

To leverage the existing Get mechanism for the purpose of supporting attachments, a new `_get` exchange scheme has been defined (similar to `_local` or `_send`). Display applications must register this scheme with the Exchange Manager to let other applications know that they support the Get mechanism.

Figure 4.5 shows the interaction between the applications and the Exchange Manager.

Figure 4.5 Sending attachment from messaging application


Messaging Application
Exchange Manager
Display Application
- ExgRegisterDatatype("_get")
Messaging application requests a list of applications that support the Get mechanism (_get scheme) and displays it. It lets the user make a selection and requests data from that application using the local exchange library.
- ExgGetRegisteredApplications("_get")
- Display list of applications with attachment support. User makes selection.
- Set exgSocket.target according to selection.
- Allocate and initialize memory for exgSocket.name/type/description.
- Set exgSocket.name = "_local:"
- ExgGet
Exchange Manager sub-launches the selected display application.
sysAppLaunchCmdExgGetData
The display application presents a screen allowing the user to choose a data object (document). After the user makes a selection, the data object is sent to the messaging application.
- Display document selection screen.
- Copy name, type and description info into provided exgSocket.name/type/description buffers.
- ExgAccept
- ExgSend
- ExgDisconnect
- ExgReceive
- ExgDisconnect
- Free memory allocated for exgSocket.name/type/description.
Messaging application adds the data object as an attachment to the message.

NOTE: There are bugs in some versions of Palm OS that prevent using the Get mechanism as documented here. The Attachment Support sample code shows how to work around these problems. It is available in the Developer Knowledge Base at http://kb.palmsource.com/

Guidelines for Messaging Applications

To handle attachment sending, messaging applications should follow these guidelines:

- Messaging Applications must use the `_get` scheme to identify the available display applications that implement attachment support.
- Messaging applications should offer a way to add multiple attachments to a message by repeating the attachment selection process.
- Messaging application should never try to read/write directly from/to the databases of other applications. Database formats might change or databases might be encrypted. Using the Exchange Manager not only prevents database corruption by other applications but also ensures interoperability between 68K and ARM applications.
- Before calling `ExgGet()`, messaging applications must:
  - allocate buffers for the `exgSocket.name`, `exgSocket.type`, and `exgSocket.description` fields. The size of these buffers are:

    `description`: `exgMaxDescriptionLength + 1`

    `type`: `exgMaxTypeLength + 1`

    `name`: `exgMaxTypeLength + 1` (same as for type)
  - initialize all buffers with zeros
  - set `exgSocket.name` to "_local:"

Guidelines for Display Applications

To handle attachment sending from a messaging application, display applications should follow these guidelines:

- Applications must register the `_get` scheme with the Exchange Manager if they implement attachment support. Refer to "General Registration Guidelines" on page 113.
- Applications must not send more then one data object for every `ExgGet()` request.
- When responding to a `sysAppLaunchCmdExgGetData` launch code, an application should copy the file name, type, and description information of the content into the supplied

exgSocket.name, exgSocket.type, and exgSocket.description fields. Memory for these fields has been allocated by the calling application. The usable size of these fields is:

  - name: exgMaxTypeLength (same as type)
  - type: exgMaxTypeLength
  - description: exgMaxDescriptionLength

Applications should first check if memory was allocated (check for NULL pointers) before writing to the buffers. An application should never allocate or reallocate memory for these fields itself.

If the exgSocket.name field contains a scheme (for example, "_local:"), this scheme information should not be overwritten. Rather, name information should be appended to the scheme (for example, "_local:myname.txt").

- Display applications should provide a familiar document selection screen, similar to the document list users might see when they usually launch the display application.
- When responding to a sysAppLaunchCmdExgGetData launch code, applications may support requests for a specific record (for example, via an application-specific URL scheme passed in the name field of the ExgSocketType structure). Such a request should be handled without a user interface.

A display application may be launched with a full launch in response to a get request. This implementation is shown in detail in the Attachment Support sample code. It is available in the Developer Knowledge Base at http://kb.palmsource.com/

Sending an Attachment from a Display Application

Any display application should be able to send data to another device using any messaging application as a transport. For instance, a user could send a JPEG image from an image viewer to another person via email. This feature is implemented via the Exchange Manager using the _send URL scheme. Because the send capability is has been available since Palm OS 4, many existing display applications already support it.

Guidelines for Messaging Applications

To handle attachment sending from a display application, messaging applications should follow these guidelines:

- Messaging applications must provide an exchange library that supports the `_send` URL scheme and register the scheme with the Exchange Manager (see "General Registration Guidelines" on page 113.). Email applications will prompt the user for additional information (recipient information, subject line, etc.) and add the received data as an attachment to the message.
- Messaging applications should be able to accept any content type to act as a transport mechanism.

Guidelines for Display Applications

To handle attachment sending, display applications should follow these guidelines:

- Display applications must use the `_send` URL scheme to send data to the Exchange Manager. The Send mechanism is a general mechanism that provides the user with a choice of all the transport mechanisms available on the device that support the `_send` scheme (for example, Bluetooth, email, etc.).

  The sample code below shows how to send or beam data. The question mark in the `name` field means that on OS 4.0 and higher, it'll display a dialog to let the user choose which transport to use (for example, IR, Bluetooth, SMS, etc.).

```
status_t err;
ExgSocketType exgSocket;

// Fill out exgSocket with the description of the data and the
// package's "address"
MemSet(&exgSocket, sizeof(exgSocket), 0);
exgSocket.description = "SuperApp data";
exgSocket.name = "?_send;_beam:SuperAppData.sad";
exgSocket.type = "application/x-superappdata";

err = ExgPut(&exgSocket); // open the connection
ExgSend(&exgSocket, theData, theDataSize, &err); // send the data
err = ExgDisconnect(&exgSocket, err); // that's all!
```

- Display applications must set at least the type information (`exgSocket.type`). Name information (`exgSocket.name`), including file extension and description information (`exgSocket.description`), should also be provided. This allows messaging applications to format outgoing messages correctly, with the proper MIME content-type fields.

Email Application Guidelines

Email applications should follow these guidelines to support the `mailto` scheme and the helper APIs.

Support for Mailto URL Scheme

Applications like a web browser must be able to launch an email application to start composing a new message. In contrast to the `_send` scheme, the `mailto` URL automatically implies email as a transport medium and allows applications to provide additional information, like recipients, subject line, etc.

Therefore:

- Email applications must implement the `mailto` URL scheme and register the scheme with the Exchange Manager. For more information on the `mailto` scheme, see RFC 2368 at http://www.ietf.org/rfc/rfc2368.txt
- The application must display the message content before a message is sent. This prevents malicious applications from sending uncontrolled spam without the user's knowledge.

Support for Helper API

Messaging applications should register their services with the helper API as described in "Helper Notifications" on page 66 in *Exploring Palm OS: Programming Basics*. In particular, email applications should support the mail service class.

Summary of Exchange Manager

Exchange Manager Functions	
Sending Data	
`ExgSend()`	`ExgDBWrite()`
`ExgPut()`	
Receiving Data	
`ExgReceive()`	`ExgDBRead()`
`ExgAccept()`	
Registering for Data	
`ExgRegisterDatatype()`	`ExgRegisterData()`
`ExgSetDefaultApplication()`	
Requesting Data	
`ExgGet()`	`ExgRequest()`
Connecting and Disconnecting	
`ExgDisconnect()`	`ExgConnect()`
Displaying the Exchange Dialog	
`ExgDoDialog()`	
Obtaining Registry Information	
`ExgGetTargetApplication()`	`ExgGetRegisteredTypes()`
`ExgGetRegisteredApplications()`	`ExgGetDefaultApplication()`
Querying the Exchange Library	
`ExgControl()`	
For Exchange Library Use Only	
`ExgNotifyReceive()`	`ExgNotifyGoto()`
`ExgNotifyPreview()`	

5

Exchange Manager Reference

This chapter describes the Exchange Manager API declared in the header file `ExgMgr.h` and the Exchange Local Library API declared in the header file `ExgLocalLib.h`. It discusses the following topics:

- Exchange Manager Data Structures
- Exchange Manager Constants
- Exchange Manager Launch Codes
- Exchange Manager Functions
- Application-Defined Functions

For more information on the Exchange Manager, see Chapter 4, "Object Exchange," on page 105.

Exchange Manager Data Structures

ExgAskParamType Struct

Purpose The `ExgAskParamType` structure is the parameter block for the `sysAppLaunchCmdExgAskUser` launch code.

Declared In `ExgMgr.h`

Prototype

```
typedef struct {
   ExgSocketPtr socketP;
   ExgAskResultType result;
   uint8_t reserved;
   uint16_t padding_1;
} ExgAskParamType;
```

Fields

↔ *socketP*
Socket pointer (see ExgSocketType).

← *result*
One of the ExgAskResultType enumerated values.

→ *reserved*
Reserved for future use.

→ *padding_1*
Padding; not used.

ExgCtlGetURLType Struct

Purpose The `ExgCtlGetURLType` structure identifies the URL of a remote device as returned by the `ExgControl()` function with the `exgLibCtlGetURL` operation code.

Declared In `ExgMgr.h`

Prototype

```
typedef struct ExgCtlGetURLType{
   ExgSocketType *socketP;
   char *URLP;
   uint16_t URLSize;
   uint16_t padding;
} ExgCtlGetURLType;
```

Fields

`socketP`
Pointer to the socket structure (see ExgSocketType).

`URLP`
Pointer to the URL string.

`URLSize`
Size of the URL string.

`padding`
Padding; not used.

ExgGoToType Struct

Purpose The `ExgGoToType` structure defines the `goToParams` field of the `ExgSocketType` structure. Applications that want to be launched after the data is received place their creator IDs in the `goToCreator` field and define the `goToParams` field. The values in this structure are copied to the `sysAppLaunchCmdGoTo` launch code's parameter block.

Declared In `ExgMgr.h`

Prototype

```
typedef struct {
   LocalID dbID;
   uint32_t recordNum;
   uint32_t uniqueID;
   uint32_t matchCustom;
} ExgGoToType;
```

Fields

`dbID`
: The local ID of the database that contains the added record.

`recordNum`
: The index of the record that was added.

`uniqueID`
: The unique ID of the record that was added. This field is not used.

`matchCustom`
: Application-specific information.

ExgLocalSocketInfoType Struct

Purpose The `ExgLocalSocketInfoType` structure identifies information specific to the Local Exchange Library. The `socketRef` field of the `ExgSocketType` structure is set to this structure when you send and receive data using the Local Exchange Library. The Local Exchange Library creates this structure if it does not already exist. You only need to create it if you want to supply non-default values for the `noAsk` or `previewInfoP` fields.

Declared In `ExgLocalLib.h`

Prototype

```
typedef struct {
   Boolean freeOnDisconnect;
   Boolean noAsk;
   ExgPreviewInfoType *previewInfoP;
   void *tempFileH;
   status_t err;
   ExgLocalOpType op;
} ExgLocalSocketInfoType;
```

Fields

`freeOnDisconnect`
: Determines whether the structure is freed when the `ExgDisconnect()` call is made. The default is `true`. In general, code that allocates a structure should be responsible for freeing that structure. Therefore, if you have allocated `ExgLocalSocketInfoType`, you should set this field to `false` and explicitly free the structure when you are finished with it.

`noAsk`
: Set to `true` to disable the display of the exchange dialog. For example, if you want to create a vCalendar object and send it to the Datebook application in response to a user command, you probably want to set `noAsk` to `true` so that the user does not have to confirm the receipt of the data they just requested you to send.

`previewInfoP`
: A pointer to an `ExgPreviewInfoType` structure. The preview feature is deprecated and is maintained only for backward compatibility.

`tempFileH`
: A temporary buffer that the Local Exchange Library uses. Do not set this field directly; the Local Exchange Library should set it.

`err`
: The error code returned from the Local Exchange Library. Do not set this field directly; the Local Exchange Library should set it.

`op`
: The operation in progress. Do not set this field directly. The Local Exchange Library sets this field to one of the following constants:

  `exgLocalOpNone`
  : No operation in progress.

  `exgLocalOpPut`
  : A send is in progress.

  `exgLocalOpAccept`
  : A receive is in progress.

`exgLocalOpGet`
A get is in progress.

`exgLocalOpGetSender`
The library is receiving information from the sending application during a get operation.

ExgPreviewInfoType Struct

Purpose The `ExgPreviewInfoType` structure provides information to the `ExgNotifyPreview()` function. The preview feature is deprecated and is maintained only for backward compatibility. Applications should no longer implement preview via this mechanism, but should follow the data viewing guidelines described in "Put with View Mode" on page 147.

Declared In `ExgMgr.h`

Prototype

```
typedef struct {
   uint16_t version;
   uint16_t padding1;
   ExgSocketType *socketP;
   uint16_t op;
   uint16_t padding2;
   char *string;
   uint32_t size;
   RectangleType bounds;
   uint16_t types;
   uint16_t padding3;
   status_t error;
} ExgPreviewInfoType;
```

Fields

`version`
Set this field to 0 to specify version 0 of this structure.

`padding1`
Padding; not used.

`socketP`
A pointer to the socket structure (see `ExgSocketType`). The `name` field must identify the exchange library from which data should be received, and the `target` or `type` field should be defined as well.

`op`
: One of the following constants:

  `exgPreviewDialog`
  : Display a modal dialog containing the preview. This constant is only used in situations where one application launches another to display data.

  `exgPreviewDraw`
  : The preview is a graphic.

  `exgPreviewLongString`
  : The preview is a long string.

  `exgPreviewQuery`
  : Ask the application which preview operations it supports. The answer is returned in the `types` field. If the application does not support any preview modes, the `error` field contains `exgErrNotSupported`.

  `exgPreviewShortString`
  : The preview is a short string.

`padding2`
: Padding; not used.

`string`
: A buffer into which the application places the string preview if `exgPreviewLongString` or `exgPreviewShortString` is specified in `op`.

`size`
: The allocated size of the `string` field.

`bounds`
: The bounds of the rectangle in which the application draws the graphic if `exgPreviewDraw` is specified in `op`.

`types`
: Upon return from an `exgPreviewQuery` operation, a bit field identifying the types of previews the application supports.

`padding3`
: Padding; not used.

`error`
The error code returned from the application. If `errNone`, the preview operation was successful.

ExgSocketType Struct

Purpose The `ExgSocketType` structure defines an Exchange Manager socket, which is passed to most Exchange Manager functions. The `ExgSocketPtr` type points to a `ExgSocketType` structure.

Declared In `ExgMgr.h`

Prototype

```
typedef struct ExgSocketTag {
   uint16_t libraryRef;
   uint16_t socketRefSize;
   uint32_t socketRef;
   uint32_t target;
   uint32_t count;
   uint32_t length;
   uint32_t time;
   uint32_t appData;
   uint32_t goToCreator;
   ExgGoToType goToParams;
   uint16_t localMode:1;
   uint16_t packetMode:1;
   uint16_t noGoTo:1;
   uint16_t noStatus:1;
   uint16_t preview:1;
   uint16_t cnvFrom68KApp:1;
   uint16_t acceptedSocket:1;
   uint16_t reserved:9;
   uint8_t componentIndex;
   uint8_t padding_1;
   char *description;
   char *type;
   char *name;
} ExgSocketType;
typedef ExgSocketTag *ExgSocketPtr;
```

Note that when data is received, some of the fields in this structure may not have values. When you are sending data, it is recommended that you provide values for all of these fields, but you should not rely on receiving values for the fields marked optional.

Fields

`libraryRef`
: The exchange library in use. When an application or library receives a socket, this field is already assigned. This is no longer used for ARM code development, but is kept for compatibility with 68K exchange libraries.

  When sending data, applications may identify the exchange library they want to connect with by providing a URL in the `name` field, and should use 0 for the `libraryRef` field.

`socketRefSize`
: Size of the data block referenced by `socketRef`, if `socketRef` is a pointer to a data block.

`socketRef`
: The connection identifier. This value is supplied by the exchange library when a connection is established. It contains any necessary library-specific data. For an ARM exchange library, this *must* be a single data block without pointer references to other blocks.

`target`
: The creator ID of the application that should receive the message.

`count`
: The number of objects in this connection, usually 1 (optional).

`length`
: The total byte count for all objects being sent (optional).

`time`
: The last modified time of object (optional).

`appData`
: Application-specific information (optional). For an ARM exchange library, do *not* pass pointer values here; only the 32-bit value itself is safe.

`goToCreator`
: The creator ID of the application to launch using the `sysAppLaunchCmdGoTo` launch code after the item is received if `noGoTo` is 0. The value is assigned by the application that receives the object. See the Comments section in `ExgDisconnect()` for more information.

`goToParams`
: If `goToCreator` is specified, then this field contains data that is copied into the launch code's parameter block. See `ExgGoToType`.

`localMode`
: Deprecated; set to 0. To use local exchange, specify a URL with the `exgLocalPrefix`. (Note that this flag still works for backward compatibility.)

`packetMode`
: Deprecated; set to 0.

`noGoTo`
: Set to 1 to disable launching the application with `sysAppLaunchCmdGoTo`. The default is 0.

`noStatus`
: If `true`, the exchange library should not display a progress dialog. If `false`, the library can display a progress dialog. The default is `false`.

  The Exchange Manager sets and clears this bit at various times while data is received. Applications may also want to set this bit if they use the Local Exchange Library and want to prevent the progress dialog from being displayed during a send.

`preview`
: If `true`, a preview is in progress. The `ExgNotifyPreview()` function sets this bit while the preview takes place and clears it when the preview is finished. Exchange libraries should not discard any data while a preview is in progress because the full data must be sent later if the receiving user accepts it. The preview feature is deprecated and is maintained only for backward compatibility.

`cnvFrom68KApp`
: If set to 1, indicates that the socket was created by a 68K application. This lets the library know that it may need to convert some data (such as the `socketRef` data).

`acceptedSocket`
: If set to 1, indicates that this socket was passed into `ExgAccept()`.

`reserved`
: Reserved system flags.

`componentIndex`
: The Exchange Manager C++ component holding this socket (if any). Nonzero for valid sockets.

`padding_1`
: Padding; not used.

`description`
: A pointer to the text description of the object (optional).

`type`
: A pointer to the MIME type of the object (optional).

`name`
: The name of the object being sent. This can be a URL whose scheme identifies the exchange library to connect with.

  If the name has a colon, it is treated as a URL.

Exchange Manager Constants

ExgAskResultType Enum

Purpose The `ExgAskResultType` enum defines possible values for the `result` field of the `ExgAskParamType` launch code parameter block.

Declared In `ExgMgr.h`

Prototype

```
typedef enum {
   exgAskDialog,
   exgAskOk,
   exgAskCancel
} ExgAskResultType;
```

Constants

`exgAskDialog`
: The Exchange Manager should display a dialog that prompts the user to confirm the receipt of data. See `ExgDoDialog()`.

`exgAskOk`
: Accept the data.

`exgAskCancel`
Reject the data.

Registry ID Constants

Purpose The registry ID constants are used in the Exchange Manager registry. Exchange libraries register for the URL prefixes they handle, and applications register for the types of data they receive. The registry ID constants specify which type of data is being registered for.

Declared In `ExgMgr.h`

Constants `#define exgRegCreatorID 0xfffb`
Register for a creator ID. The `target` field of the `ExgSocketType` contains the creator ID of the application that should receive the data. Typically, the application with the matching creator ID receives the data, but it is possible for one application to register for another's creator ID and receive data in its place.

`#define exgRegSchemeID 0xfffc`
Register for a URL scheme. Typically, only exchange libraries register for URL schemes. Applications can register for URL schemes, but they only receive the URL when `ExgRequest()` is called. If the `name` field of the `ExgSocketType` contains a colon (:), the portion of the URL before the colon is the URL scheme. The default library registered for URLs with that scheme will handle the message.

`#define exgRegExtensionID 0xfffd`
Register for a filename extension. If the `name` field of the `ExgSocketType` contains a period (.), the portion of the name after the last period is the filename extension. The application registered to handle files of that extension will handle the message.

`#define exgRegTypeID 0xfffe`
Register for a MIME type. If the `type` field of the `ExgSocketType` contains a value, the application registered to receive that MIME type handles the message.

`#define exgRegDirectCreatorID 0xffeb`
: Register for a creator ID for direct delivery. The `target` field of the `ExgSocketType` contains the creator ID of the application that should directly receive the data instead of an email application. (This is used instead of the `exgUnwrap` flag and must *not* be used with it.)

`#define exgRegDirectExtensionID 0xffed`
: Register for a filename extension for direct delivery. If the `name` field of the `ExgSocketType` contains a period (.), the portion of the name after the last period is the filename extension. The application registered to handle files of that extension will handle the message, instead of an email application. (This is used instead of the `exgUnwrap` flag and must *not* be used with it.)

`#define exgRegDirectTypeID 0xffee`
: Register for a MIME type for direct delivery. If the `type` field of the `ExgSocketType` contains a value, the application registered to receive that MIME type handles the message instead of an email application. (This is used instead of the `exgUnwrap` flag and must *not* be used with it.)

`#define exgRegViewCreatorID 0xff8b`
: Register for a creator ID for view mode. The `target` field of the `ExgSocketType` contains the creator ID of an application that can be used to view the data.

`#define exgRegViewExtensionID 0xff8d`
: Register for a filename extension for view mode. If the `name` field of the `ExgSocketType` contains a period (.), the portion of the name after the last period is the filename extension. The application registered to handle files of that extension can be used to view the data.

`#define exgRegViewTypeID 0xff8e`
: Register for a MIME type. If the `type` field of the `ExgSocketType` contains a value, the application registered to receive that MIME type can be used to view the data.

Predefined URL Schemes

Purpose The Exchange Manager provides these predefined URL schemes, for which exchange libraries can register.

Declared In `ExgMgr.h` and `ExgLocalLib.h`

Constants

`#define exgBeamScheme "_beam"`
The URL scheme for Beam commands. By default, the IR Library handles this scheme.

`#define exgSendScheme "_send"`
The URL scheme for Send commands. The purpose of the Send command is to provide a choice of transport mechanisms to the user; therefore, any exchange library that sends data should register for this scheme.

`#define exgGetScheme "_get"`
The URL scheme for the Get mechanism.

`#define exgLocalScheme "_local"`
The URL scheme for the Local Exchange Library.

Predefined URL Prefixes

Purpose The Exchange Manager provides these prefixes, which can be used to construct URLs appropriate for the `name` field of the `ExgSocketType` structure. When sending data, applications provide a URL to identify the exchange library that should transport the data.

Declared In `ExgMgr.h`

Constants

`#define exgBeamPrefix exgBeamScheme ":"`
The URL to beam data.

`#define exgSendPrefix "?" exgSendScheme ":"`
A URL for the general Send command. Because this URL begins with a question mark (?), the Exchange Manager displays a dialog with a list of exchange libraries registered for the `exgSendScheme`. The user then chooses the desired exchange library.

```
#define exgSendBeamPrefix "?" exgSendScheme ";"
  exgBeamScheme ":"
```
A URL for the general Send command. The Exchange Manager displays a dialog with a list of exchange libraries registered for either the `exgSendScheme` or the `exgBeamScheme`.

`#define exgLocalPrefix exgLocalScheme ":"`
The URL for using the Local Exchange Library.

`#define exgGetPrefix exgGetScheme ":"`
The URL for using the Get mechanism.

Exchange Manager Launch Codes

sysAppLaunchCmdExgAskUser

Purpose The Exchange Manager sends this launch code to an application when data has arrived for that application. This launch code allows the application to tell the Exchange Manager not to display the exchange dialog, which it uses to have the user confirm the receipt of data. If the application does not handle this launch code, the default course of action is that the Exchange Manager displays the exchange dialog.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdExgAskUser 27`

Parameters The launch code's parameter block pointer references a `ExgAskParamType` structure.

Comments Applications may want to respond to this launch code under these circumstances:

- To reject all incoming data or to reject data under certain circumstances without first prompting the user. To reject incoming data, set the `result` field of the parameter block to `exgAskCancel` and then return.
- To receive incoming data without confirmation. To automatically receive incoming data, set the result field to `exgAskOk`.
- To provide a user confirmation dialog with extra functionality. This is described in more detail below.

The Exchange Manager allows applications to provide extra functionality in the exchange dialog. You can have the dialog include a category pop-up list from which the user chooses a category in which to file the incoming data. If you want to provide a category pop-up list, call the `ExgDoDialog()` function in response to this launch code and pass it a database that contains the categories to be listed. See the description of this function for more information.

Applications may also bypass the call to `ExgDoDialog()` altogether and provide their own dialogs.

If an application responds to this launch code, it must set the `result` field in the parameter block to the appropriate value. Possible values are defined by the `ExgAskResultType` enum.

If you don't use the default version of the dialog, return `exgAskOk` if the user confirmed or `exgAskCancel` if the user canceled. If you don't set the `result` field properly, two dialogs are displayed.

See Also `sysAppLaunchCmdExgPreview`, and Chapter 6, "Common Launch Codes," in *Exploring Palm OS: Programming Basics*

sysAppLaunchCmdExgGetData

Purpose The Exchange Manager sends this launch code when the exchange library requests data to be sent to a remote device. That is, an application on a remote device has called the `ExgGet()` function to request data, and the Exchange Manager has determined that the launched application should handle the request.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdExgGetData 59`

Parameters The parameter block sent with this launch code is a pointer to the `ExgSocketType` structure corresponding to the Exchange Manager connection on which the data is to be sent. You pass this socket pointer to `ExgAccept()`. For more details, see "Responding to a Get Request" on page 133.

Comments To respond to this launch code, applications should accept a connection with `ExgAccept()`, use `ExgSend()` to send the data, and call `ExgDisconnect()` when finished.

See Also Chapter 6, "Common Launch Codes," in *Exploring Palm OS: Programming Basics*

sysAppLaunchCmdExgPreview

Purpose Sent after `sysAppLaunchCmdExgAskUser` to have the application display the preview in the exchange dialog.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdExgPreview 57`

Parameters The launch code's parameter block pointer references a `ExgPreviewInfoType` structure.

Comments This launch command is deprecated and is maintained only for backward compatibility. Applications should no longer implement preview via this mechanism, but should follow the data viewing guidelines described in "Put with View Mode" on page 147.

See Also Chapter 6, "Common Launch Codes," in *Exploring Palm OS: Programming Basics*

sysAppLaunchCmdExgReceiveData

Purpose The Exchange Manager sends this launch code following the `sysAppLaunchCmdExgAskUser` and `sysAppLaunchCmdExgPreview` launch codes to notify the application that it should receive the data (assuming that the application or the user has indicated the data should be received).

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdExgReceiveData 26`

Parameters The parameter block sent with this launch code is a pointer to the `ExgSocketType` structure corresponding to the Exchange Manager connection on which the data is arriving. Pass this pointer to the `ExgAccept()` function to begin receiving the data. For more details, see "Receiving the Data" on page 126.

Comments The application should use Exchange Manager functions to receive the data and store it or do whatever it needs to with the data. Specifically, most applications should respond to this launch code by calling `ExgAccept()` to accept the connection and then `ExgReceive()` to receive the data.

Note that the application may not be the active application, and thus may not have globals available when it is launched with this launch code. You can check if you have globals by using this code in the `PilotMain()` function:

```
Boolean appIsActive = launchFlags & sysAppLaunchFlagSubCall;
```

The `appIsActive` value is `true` if your application is active and globals are available; otherwise, you won't be able to access any of your global variables during the receive operation.

See Also Chapter 6, "Common Launch Codes," in *Exploring Palm OS: Programming Basics*

Exchange Manager Functions

ExgAccept Function

Purpose Accepts a connection from a remote device.

Declared In `ExgMgr.h`

Prototype `status_t ExgAccept (ExgSocketPtr` *`socketP`*`)`

Parameters → *`socketP`*
A pointer to the socket structure (see `ExgSocketType`).

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find default exchange library
`exgErrNotSupported`	A preview is in progress, and the exchange library doesn't support preview mode

Other error codes depend on the exchange library.

Comments Applications call this function when launched with the `sysAppLaunchCmdExgReceiveData` or `sysAppLaunchCmdExgPreview` launch code. The launch code contains `socketP` in its parameter block. Applications should pass this socket to `ExgAccept()` to accept the connection, then call

`ExgReceive()` one or more times to receive the data, and then call `ExgDisconnect()` to disconnect.

NOTE: Don't create the socket on the receiving side of an exchange. The socket is passed to you in the command parameter block of the `sysAppLaunchCmdExgReceiveData` or `sysAppLaunchCmdExgPreview` launch code.

See Also `ExgConnect()`, `ExgPut()`, `ExgGet()`

ExgConnect Function

Purpose Establishes a connection with a remote socket.

Declared In `ExgMgr.h`

Prototype `status_t ExgConnect (ExgSocketPtr` *`socketP`*`)`

Parameters → *`socketP`*
A pointer to the socket structure (see `ExgSocketType`). Specify a URL in the `name` field and 0 for the `libraryRef` field.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find exchange library
`exgErrNotSupported`	The library doesn't support the operation specified in `socketP`
`exgErrUserCancel`	The user cancelled the connection operation
`exgMemError`	There isn't enough free memory to respond to the request
`exgErrNotEnoughPower`	The battery does not have enough power to perform the operation

Other error codes depend on the exchange library.

Comments Applications can call this function to initiate a connection for sending multiple objects or for performing two-way communications. Some exchange libraries support sending multiple objects but do not support this call. See "Sending Multiple Objects" on page 121 for more information.

Before calling this function, the application must initialize the `socketP` parameter. The socket should identify the exchange library to connect with by providing a URL in the `name` field. The default exchange library registered for that type of URL handles the connection.

To provide users with a choice of transport mechanisms, specify a URL that begins with a question mark (?). The Exchange Manager displays a dialog with a list of all exchange libraries that respond to URLs of the specified type. If only one exchange library is registered for this URL scheme, no dialog is displayed.

For example, many applications support a Send command. This command generates a URL with the prefix `exgSendPrefix` (see Predefined URL Prefixes). The Exchange Manager displays a dialog containing a list of libraries registered for that URL scheme. The user selects an exchange library, and that library's `ExgLibConnect()` function is called.

If the library is not specified by URL, the Exchange Manager by default uses the IR Library; however, if the `localMode` flag is set, the Local Exchange Library is used instead.

In addition to specifying the library, you can set the `count` field in `socketP` before making this call to indicate the number of objects that are going to be sent. Use a `count` of 0 if the number of objects isn't known in advance.

If no error is returned from `ExgConnect()`, applications can follow this call either by sending multiple objects or requesting data from the remote device or both. To send an object, call `ExgPut()` at the beginning of each object and call `ExgSend()` one or more times per object to send the data. To request data from the remote device, use `ExgGet()` (and then use `ExgReceive()` to receive the requested data). You can use these calls in combination with each other to support two-way communications. After all of the objects have been sent and received, call `ExgDisconnect()` to disconnect.

IMPORTANT: Not all exchange libraries support the sending of multiple objects or using `ExgGet()` to request data.

See Also `ExgPut()`, `ExgAccept()`, `ExgGet()`

ExgControl Function

Purpose Requests that an exchange library perform an operation.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgControl (ExgSocketType *socketP,
   uint16_t op, void *valueP,
   uint16_t *valueLenP)
```

Parameters

→ *`socketP`*
A pointer to the socket structure (see `ExgSocketType`). Specify a URL in the `name` field and 0 for the `libraryRef` field.

→ *`op`*
A constant identifying the operation that the exchange library should perform. See the Comments section for more information.

↔ *`valueP`*
Upon entry, a parameter that the exchange library requires to perform the operation, if any. Most operations do not require an input parameter. Upon return, contains the result of the operation.

↔ *`valueLenP`*
The size of the `valueP` buffer. The size is updated upon return to show the actual length of the content returned.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find the requested exchange library
`exgErrNotSupported`	The exchange library does not support the requested operation

Other error codes depend on the exchange library.

Comments The Exchange Manager uses this function to request information from the exchange library. Applications may also call this function.

The Exchange Manager defines and uses a set of operation constants that it might send to any exchange library. These constants begin with the prefix `exgLibCtlGet`. The type of the variable pointed to by `valueP` depends on the type of operation to be performed. Table 5.1 lists and describes the predefined Exchange Manager operations.

Table 5.1 ExgControl operations for all exchange libraries

Operation exgLibCtlGet...	value Data Type	Description
`Preview`	Boolean. Output only.	Returns `true` if the exchange library supports preview mode or `false` if not. If the exchange library does not respond to this operation, it is assumed to support preview mode. The preview feature is deprecated and is maintained only for backward compatibility.
`Title`	String buffer of size `exgTitleBufferSize` bytes. Output only.	Returns the name of the exchange library as it should appear in the Send dialog. All exchange libraries must respond to this operation.

Table 5.1 ExgControl operations for all exchange libraries

Operation exgLibCtlGet...	value Data Type	Description
`URL`	`ExgCtlGetURLType` structure.	Returns the URL that addresses the remote device from which you receive data. You can get the URL after calling `ExgAccept()`.
`Version`	`uint16_t`. Output only.	Returns the version of the exchange library API that this library implements. The constant `exgLibAPIVersion` defines the current version number. If the exchange library does not respond to this operation, the library supports the version of the Exchange Library API defined in Palm OS 4.0.

An exchange library may also define its own operations. Operations specific to an exchange library are numbered starting at `exgLibCtlSpecificOp`.

The `socketP` passed to this function must identify an exchange library with a URL in the `name` field. The Comments section in `ExgConnect()` describes how an application should identify the exchange library.

ExgDBRead Function

Purpose Converts a Palm OS database from its internal format and writes it to storage RAM.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgDBRead (ExgDBReadProcPtr readProcP,
   ExgDBDeleteProcPtr deleteProcP,
   void *userDataP, DatabaseID *dbIDP,
   Boolean *needResetP, Boolean keepDates)
```

Parameters

→ *readProcP*
A pointer to a function that reads in the database and passes it to `ExgDBRead()`. See `ExgDBReadProcPtr()` for details.

→ *deleteProcP*
A pointer to a function that is called if a database with an identical name already exists on the device. See `ExgDBDeleteProcPtr()` for details.

→ *userDataP*
A pointer to any data you want to pass to either the `readProcP` or `deleteProcP` functions. Often, this parameter is used to pass the `ExgSocketType` that is required by many Exchange Manager functions.

← *dbIDP*
The ID of the database that `ExgDBRead()` created on the local device.

← *needResetP*
Set to `true` by `ExgDBRead()` if the `dmHdrAttrResetAfterInstall` attribute bit is set in the received database.

→ *keepDates*
Specify `true` to retain the creation, modification, and last backup dates as set in the received database header. Specify `false` to reset these dates to the current date.

Returns Returns `errNone` if successful; otherwise, returns one of the data manager error codes (`dmErr...`) or a callback-specific error code. (If the `readProcP` function returns an error, it is also returned by `ExgDBRead()`.)

Comments This function converts data received from an exchange library or from any other transport mechanism into a Palm OS database and stores that database in the storage heap. It is not required that you use this function in conjunction with Exchange Manager calls. That is, it's possible to use this function to perform other operations, such as converting a database created on the desktop computer to a Palm OS formatted database in the storage heap.

The primary use of this function, however, is to receive a database that has been beamed onto the device. In this case, call `ExgDBRead()` in response to the launch code `sysAppLaunchCmdExgReceiveData` after calling `ExgAccept()` to accept the connection. Place the call to `ExgReceive()` in the read callback function you passed as the `readProcP` parameter. Pass the `ExgSocketType` structure returned from `ExgAccept()` in the `userDataP` parameter so that you have access to it in the read callback function.

The read callback function performs the actual reading of data. `ExgDBRead()` calls the read callback function multiple times. Each time, the `sizeP` parameter contains the number of bytes `ExgDBRead()` expects the data returned in `dataP` to contain. It's important for the read callback function to set the number of bytes (in `sizeP`) that it actually placed in `dataP` if it's not the same as what `ExgDBRead` expected. `ExgDBRead()` stops calling the read callback function after 0 is returned in `sizeP`.

The callback function you pass in `deleteProcP` handles the case where the database being read already exists on the device. It is called only in that circumstance. The callback function may want to close the database if it is open, change the existing database's name, or delete the existing database to allow an overwrite. See `ExgDBDeleteProcPtr()` for more information.

See Also ExgDBWrite()

ExgDBWrite Function

Purpose Converts a given Palm OS database from its internal format on the local device and writes it using a function you supply.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgDBWrite
    (ExgDBWriteProcPtr writeProcP,
    void *userDataP, const char *nameP,
    DatabaseID dbID)
```

Parameters

→ *writeProcP*
A pointer to a function that writes out the database identified by `dbID`. See `ExgDBWriteProcPtr()` for details.

→ *userDataP*
A pointer to any data you want to pass to the `writeProcP` function. Often, this parameter is used to pass the `ExgSocketType` that is required by many Exchange Manager functions.

→ *nameP*
A pointer to the name of the database that you want `ExgDBWrite()` to write. This database is passed to `writeProcP`.

→ *dbID*
The ID of the database that you want `ExgDBWrite()` to pass to `writeProcP`. If you don't supply an ID, then `nameP` is used to search for the database by name.

Returns Returns `errNone` if successful; otherwise, returns one of the data manager error codes (`dmErr...`) or a callback-specific error code. (If the `writeProcP` function returns an error, it is also returned by `ExgDBWrite()`.)

Comments This function converts a Palm OS formatted database on the storage heap into a stream of bytes that can be sent over the Internet or over any other transport mechanism. It is not required that you use this function in conjunction with Exchange Manager calls.

The primary use of this function, however, is to write a database that is going to be beamed onto another device. In this case, call `ExgDBWrite()` after establishing the connection with `ExgPut()`. Place the call to `ExgSend()` in the write callback function you passed as the `writeProcP` parameter. Pass the `ExgSocketType` structure returned from `ExgSend()` in the `userDataP` parameter so that you have access to it in the write callback function.

The write callback function performs the actual writing of data. `ExgDBWrite()` calls the write callback function multiple times. Each time, the `sizeP` parameter contains the number of bytes of `dataP` that are to be written. If the write callback function didn't handle it all, it's important that it set in `sizeP` the number of bytes that it did handle successfully. `ExgDBWrite()` stops calling the write callback function after 0 is returned in `sizeP`.

See Also ExgDBRead()

ExgDisconnect Function

Purpose Terminates an Exchange Manager transfer and disconnects.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgDisconnect (ExgSocketPtr socketP,
   status_t error)
```

Parameters

→ *socketP*
A pointer to the socket structure (see `ExgSocketType`) identifying the connection to terminate.

→ *error*
Any error that occurred. This parameter tells the exchange library why the connection is being broken. Normally the error code from `ExgSend()` or `ExgReceive()` is passed in here.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find default exchange library
`exgMemError`	Couldn't read data to send
`exgErrUserCancel`	User cancelled transfer

Other error codes depend on the exchange library.

Comments Applications must call this function when finished sending data or receiving data. It terminates the connection made with `ExgConnect()`, `ExgAccept()`, `ExgPut()`, or `ExgGet()`.

In the `error` parameter, pass any error that occurs during the application loop, including errors returned from other Exchange Manager functions. This ensures that the connection is shut down knowing that it failed rather than succeeded.

It's especially important to check the result code from this function, since this will tell you if the transfer was successful. An `errNone` return value means that the item was delivered to the destination successfully. It does not mean that the user on the other end actually kept the data.

`ExgDisconnect()` is used after sending and receiving. When receiving, the application can insert its creator ID into the `goToCreator` field in the socket structure and add other goto information in the `goToParams` field. After the application returns from the `sysAppLaunchCmdExgReceiveData` launch code, the exchange library may call `ExgNotifyGoto()`, which launches the `goToCreator` application with the standard launch code `sysAppLaunchCmdGoTo`.

IMPORTANT: Placing your creator ID in the `goToCreator` field is not a guarantee that you receive this launch code, because Palm OS supports the sending of multiple objects at once. Thus, another application might overwrite the `goToCreator` field after your application has disconnected, making that application the recipient of the launch code.

Note that some exchange libraries wait to establish a connection until `ExgDisconnect()` is called. The IR Library, for example, buffers the data that it receives and then waits until `ExgDisconnect()` to actually send this data unless `ExgConnect()` is called to establish a multi-object send connection.

The Exchange Manager does not automatically launch the `goToCreator` application, if one was provided, upon return from this function. Exchange libraries that want this behavior must explicitly call `ExgNotifyGoto()`.

See Also `ExgReceive()`, `ExgSend()`

ExgDoDialog Function

Purpose Displays a dialog that allows users to accept or reject the receipt of data.

Declared In `ExgMgr.h`

Prototype
```
Boolean ExgDoDialog (ExgSocketType *socketP,
   ExgDialogInfoType *infoP, status_t *errP)
```

Parameters

→ *`socketP`*
A pointer to the socket structure (see `ExgSocketType`) identifying the connection.

Applications can obtain the socket structure from the `sysAppLaunchCmdExgAskUser` launch code parameter block.

↔ *`infoP`*
A pointer to an `ExgDialogInfoType` structure (see the Comments section below).

← *`errP`*
`errNone` if no error, or the error code if an error occurred. Currently, no errors are returned.

Returns Returns `true` if the user clicks the OK button on the dialog, or `false` otherwise.

Comments This function displays the exchange dialog, which prompts the user to accept or reject incoming data.

By default, the Exchange Manager calls this function if the receiving application doesn't handle the `sysAppLaunchCmdExgAskUser` launch code or if it returns `exgAskDialog` from the launch code handler. When the Exchange Manager calls `ExgDoDialog()`, the dialog displays a message similar to "Do you want to accept 'John Doe' into Address Book?" and allows the user to accept or reject the data. If the user clicks OK, the data should be received as an unfiled record.

The Exchange Manager attempts to display a preview of the data in the exchange dialog to provide users with enough information to determine if they want to accept or reject the data. To display the preview data, it calls `ExgNotifyPreview()`. See the `ExgNotifyPreview()` function's description for more information.

Applications may also want to allow users to select a category in which to accept the incoming data. To do so, handle `sysAppLaunchCmdExgAskUser` to call `ExgDoDialog()` directly and pass it a pointer to an `ExgDialogInfoType` structure. The `ExgDialogInfoType` structure is defined as follows:

```
typedef struct {
  uint16_t version;
  DmOpenRef db;
  uint16_t categoryIndex;
} ExgDialogInfoType;
```

→ *`version`*
Set this field to 0 to specify version 0 of this structure.

→ *`db`*
A pointer to an open database that defines the categories the dialog should display.

← *`categoryIndex`*
The index of the category in which the user wants to file the incoming data.

If `db` is valid, the function extracts the category information from the specified database and displays it in a pop-up list. Upon return, the `categoryIndex` field contains the index of the category the user selected, or `dmUnfiledCategory` if the user did not select a category.

If the call to `ExgDoDialog()` is successful, your application is responsible for retaining the value returned in `categoryIndex` and using it to file the incoming data as a record in that category. One way to do this is to store the `categoryIndex` in the socket's `appData` field (see `ExgSocketType`) and then extract it from the socket in your response to the launch code `sysAppLaunchCmdExgReceiveData`. For example:

```
if (cmd == sysAppLaunchCmdExgReceiveData) {
  uint16_t category =
    (ExgSocketPtr)cmdPBP->appData;
  /* other declarations */

/* Receive the data, and create a new record
  using the received data. indexNew is the
  index of this record. */
```

```
    if (category !- dmUnfiledCategory) {
      uint16_t attr;
      status_t err;
      err = DmRecordInfo(dbP, indexNew, &attr,
        NULL, NULL);

      // Set the category to the one the user
      // specified, and mark the record dirty.
      if ((attr & dmRecAttrCategoryMask) !=
        category) {
        attr &= ~dmRecAttrCategoryMask;
        attr |= category | dmRecAttrDirty;
        err = DmSetRecordInfo(dbP, indexNew,
          &attr, NULL);
      }
    }
  }
```

Some of the Palm OS built-in applications (Address Book, Memo, and ToDo) use this method of setting the category on data received through beaming. Refer to the example code for these applications provided in the SDK for a more complete example of how to use `ExgDoDialog`.

When you explicitly call `ExgDoDialog()`, you must set the `result` field of the `sysAppLaunchCmdExgAskUser` launch code's parameter block to either `exgAskOk` (upon success) or `exgAskCancel` (upon failure) to prevent the system from displaying the dialog a second time.

ExgGet Function

Purpose Establishes a connection and requests an object from a remote or local device.

Declared In `ExgMgr.h`

Prototype `status_t ExgGet (ExgSocketPtr` *`socketP`*`)`

Parameters → *`socketP`*

A pointer to the socket structure (see `ExgSocketType`). Specify a URL in the `name` field and 0 for the `libraryRef` field. The `target` or `type` fields should identify the data being requested.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find default exchange library
`exgErrUserCancel`	The user cancelled the operation
`exgMemError`	There is not enough free memory to perform the operation
`exgErrNotSupported`	The exchange library doesn't support this function

Other error codes depend on the exchange library.

Comments Applications use this function to request data (initiate a send) from a remote or local device. Not all exchange libraries support this operation; among the exchange libraries built into the Palm OS only the Local Exchange library supports it.

Before calling this function, the application must initialize the `socketP` parameter. The socket should identify the exchange library to connect with by providing a URL in the `name` field. The default exchange library registered for the URL's scheme handles the connection. The socket should also specify what data it is requesting by providing values for at least one of the `target`, `name`, and `type` fields. Specifying the data in the `name` field is the most common method.

To provide users with a choice of transport mechanisms, the application can provide a URL that begins with a question mark (?). The Exchange Manager displays a dialog with a list of all exchange libraries that respond to URLs of the specified type. If only one exchange library is registered for this URL scheme, no dialog is displayed.

If the library is not specified by URL, the Exchange Manager by default uses the IR Library; however, if the `localMode` flag is set, the Local Exchange Library is used instead.

Applications can use `ExgGet()` to initiate a send from the Local Exchange Library. For more information, see "Sending and Receiving Locally" on page 136.

If no error is returned, applications should follow this call with one or more calls to `ExgReceive()`, to receive the data, or `ExgDisconnect()`, to disconnect.

See Also `ExgPut()`, `ExgConnect()`

ExgGetDefaultApplication Function

Purpose Retrieves the default application for the specified type of data or the default exchange library for URLs with the specified scheme.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgGetDefaultApplication
   (uint32_t *creatorIDP, uint16_t id,
   const char *dataTypeP)
```

Parameters

← *creatorIDP*
A pointer to the creator ID of the default application or default exchange library.

→ *id*
The registry ID constant identifying the type of data in `dataTypeP`. See Registry ID Constants.

→ *dataTypeP*
A pointer to a string that contains the type of data for which to retrieve the default application or library. If `dataTypeP` is a file extension, do not include the period (.). If it is a URL, do not include the colon (:).

Returns Returns `errNone` if a match was found or `exgErrNoKnownTarget` if there is no default application or library for this type of data.

Comments You might use this function to see which application on this device will receive a particular type of data or to see which library on this device handles URLs of a particular scheme.

For example, to find out which application receives TXT files on this device, do the following:

```
uint32_t creatorID;
status_t error;
error = ExgGetDefaultApplication(&creatorID,
  exgRegExtensionID, "TXT");
```

```
if (!error) {
    //creatorID contains default application.
```

To find out which exchange library handles URLs that use the beam prefix, do the following:

```
uint32_t creatorID;
status_t error;
error = ExgGetDefaultApplication(&creatorID,
  exgRegSchemeID, exgBeamScheme);
if (!error) {
    //creatorID contains default library.
```

It's possible to have several applications registered to receive the same type of data, but none of them is the default. When the Exchange Manager receives an object of that type, it selects an application to receive the data, and it selects that same application every time. The selected application effectively becomes the default for the data type even though it is not explicitly set as the default. If this is the case, the `ExgGetDefaultApplication()` function returns the creator ID of this de-facto default application.

See Also `ExgGetRegisteredApplications()`, `ExgGetRegisteredTypes()`, `ExgRegisterDatatype()`, `ExgSetDefaultApplication()`

ExgGetRegisteredApplications Function

Purpose Retrieves a list of all applications registered to receive data of a specified type.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgGetRegisteredApplications
    (uint32_t **creatorIDsP, uint32_t *numAppsP,
    char **namesP, char **descriptionsP,
    uint16_t id, const char *dataTypeP)
```

Parameters

← *`creatorIDsP`*
An array of the creator IDs of the applications registered to receive objects of this type. Pass `NULL` for this parameter if you only want to know how many applications are registered for this type.

← *`numAppsP`*
The number of applications registered to receive objects of this type. This is the number of elements in the `creatorIDsP` array, the `namesP` array, and the `descriptionsP` array.

← *`namesP`*
A packed list of strings, suitable for passing to `SysFormPointerArrayToStrings()`, containing the names of the applications or libraries. Each string is no more than `exgMaxTitleLen` characters. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`descriptionsP`*
A packed list of strings, suitable for passing to `SysFormPointerArrayToStrings()`, containing the descriptions of the applications or libraries. Descriptions are specified when the applications or libraries register for data. Each string is no more than `exgMaxDescriptionLength` characters. Pass `NULL` for this parameter if you don't want to retrieve it.

→ *`id`*
The registry ID constant identifying the type of data in `dataTypeP`. See Registry ID Constants.

→ *`dataTypesP`*
A pointer to a tab-delimited, null-terminated string listing the items to register. (Use `\t` for the tab character.) Each item in the string must be no more than `exgMaxTypeLength` characters. There can be no more than 16 types total.

Returns Returns `errNone` upon success or `exgMemError` if the function cannot allocate space for the creator IDs, names, or descriptions.

IMPORTANT: This function allocates enough space for the `creatorIDsP`, `namesP`, and `descriptionsP` arrays as long as you do not pass `NULL` for the parameters. You are still responsible for freeing these arrays.

Comments You might use this function to see which applications on this device can receive a particular type of data or to see which libraries on this device handle URLs of a particular scheme. You can also use it to built a list of choices from which the user can select a default application or default exchange library for a particular data type or URL scheme.

The Exchange Manager itself uses `ExgGetRegisteredApplications()` to find exchange libraries when it is given a URL that begins with a question mark (?). It displays the returned list to the user in the Send With dialog.

See Also `ExgGetDefaultApplication()`, `ExgGetRegisteredTypes()`, `ExgRegisterDatatype()`, `ExgSetDefaultApplication()`

ExgGetRegisteredTypes Function

Purpose Retrieve a list of all data types for which a registration exists.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgGetRegisteredTypes
    (char **dataTypesP, uint32_t *sizeP,
    uint16_t id)
```

Parameters

← *`dataTypesP`*
: A packed list of strings, suitable for passing to `SysFormPointerArrayToStrings()`, containing a sorted list of data types for which a registration exists. Each string is no more than `exgMaxTypeLength` characters.

← *`sizeP`*
: The number of elements in the `dataTypesP` array.

→ *`id`*
: The type of data to search for. For example, you can search for all registered creator IDs, all registered MIME types, and so on.

Returns Returns `errNone` upon success or `exgMemError` if the function cannot allocate space for the data types array.

IMPORTANT: This function allocates enough space for the `dataTypesP` array as long as you do not pass `NULL` for the parameter. You are still responsible for freeing this array.

Comments This function could be used to create an application that allows users to choose the default application for each data type.

See Also `ExgGetDefaultApplication()`, `ExgGetRegisteredApplications()`, `ExgRegisterDatatype()`, `ExgSetDefaultApplication()`

ExgGetTargetApplication Function

Purpose Retrieves the application that should receive a specific message. This function does not search for libraries.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgGetTargetApplication
   (ExgSocketType *socketP, Boolean unwrap,
   uint32_t *creatorIDP, char *descriptionP,
   uint32_t descriptionSize)
```

Parameters

→ *`socketP`*
A pointer to the socket structure (see `ExgSocketType`). The structure should contain values for the `target`, `type`, or `name` fields.

→ *`unwrap`*
If `true`, only an application that registered to receive the data type with the `exgUnwrap` flag set should be the target application. If `false`, the target application should be an application that registered with the `exgUnwrap` flag clear. Note that usage of the `exgUnwrap` flag is deprecated.

← *`creatorIDP`*
The creator ID of the application that should receive this object.

↔ *`descriptionP`*
The application's description from the registry, if any.

→ *`descriptionSize`*
The size of the `descriptionP` buffer.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrTargetMissing`	The `target` field contains a creator ID, but the application with that creator ID does not exist
`exgErrNoKnownTarget`	No application is registered to receive the data type

Comments The Exchange Manager uses this function to determine which application should be launched to receive incoming data. Applications and libraries may call this function as well.

`ExgGetTargetApplication()` determines the target application by doing the following:

- If the `socketP->target` field contains a creator ID, the Exchange Manager searches the registry to see if an application is registered for that creator ID as the default application. If the registry does not contain an entry for the creator ID, it checks to see if the application identified by the creator ID is installed on this device. If an application is found for the `target`, that is the application returned.
- If the `socketP->type` field contains a MIME type, the Exchange Manager searches the registry for an application registered to receive objects of that type. If one is found, that is the application returned.
- If the `socketP->name` field contains a period (.), the portion after the last period is taken to be the file extension. The Exchange Manager searches the registry for an application registered to receive a file with the specified extension. If one is found, that is the application returned. If not, `exgErrNoKnownTarget` is returned.

If more than one application is registered for the target, type, or file extension, this function returns the one that is registered as the default. If no application is registered as the default, then a specific application is chosen. The Exchange Manager chooses this same application each time. That is, each time a file with a TXT extension is sent with no target or MIME type specified, the `ExgGetTargetApplication()` returns the same application to handle the receipt.

Set the `unwrap` parameter to `true` if the object was sent as part of another object, such as a vStock object that was sent as an attachment to an email message. In this case, the Exchange Manager searches for an application that registered to receive the target, the type, or the file extension of the vStock object with the `exgUnwrap` flag set. If an application is found, the vStock object is delivered, and the exchange library should discard the object that contained it (the email message). If there is no application registered to receive the data with the `exgUnwrap` flag set, this function returns `exgErrNoKnownTarget`. In this case, the exchange library should call `ExgNotifyReceive()` again passing the entire email message instead of just the vStock attachment.

See Also `ExgSetDefaultApplication()`, `ExgNotifyPreview()`, `ExgNotifyReceive()`, `ExgRegisterDatatype()`

ExgNotifyGoto Function

Purpose Launches the target application using `sysAppLaunchCmdGoTo`.

Declared In `ExgMgr.h`

Prototype

```
status_t ExgNotifyGoto (ExgSocketType *socketP,
   uint16_t flags)
```

Parameters

→ *`socketP`*
A socket identifying the object to deliver (see `ExgSocketType`). The `goToCreator` field contains the application to be launched, and the `goToParams` field contains data for the launch code's parameter block.

→ *`flags`*
Not currently used. Pass 0 for this parameter.

Returns Returns one of the following error codes:

`errNone`	Success or the `goToCreator` field is empty

`dmErr...` (one of the data manager error codes)	The specified application could not be found
`memErrNotEnoughSpace`	Not enough memory available to create the launch code's parameter block

Comments Exchange libraries call this function if they want to support immediate display of the received object. Applications do not call this function.

Most exchange libraries should call `ExgNotifyGoto()` after the return from `ExgNotifyReceive()` so that the user can inspect the newly received data. If the exchange library is most often used by a single application that does not require the launch code, this call to `ExgNotifyGoto()` can be skipped. For example, the SMS Library does not call `ExgNotifyGoto()`. SMS messages are received by the SMS Messenger application, which does not launch upon receiving data.

`ExgNotifyGoto()` launches an application only if one is specified in the `goToCreator` field and the `noGoTo` parameter is `false`. If a `goToCreator` is not specified, it is not considered an error. This gives the application a way to override the default behavior.

See Also `ExgNotifyReceive()`, `ExgDisconnect()`

ExgNotifyPreview Function

Purpose Gets a description of the data to be received for the exchange dialog.

Declared In `ExgMgr.h`

Prototype
```
status_t ExgNotifyPreview
    (ExgPreviewInfoType *infoP)
```

Parameters ↔ *`infoP`*
An `ExgPreviewInfoType` structure containing information about the operation.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrNotSupported`	The exchange library doesn't support preview mode
`exgErrNoKnownTarget`	There is no application registered to receive the type of object

Other error codes depend on the application.

Comments The `ExgDoDialog()` function calls this function to get a description of the data to show in the exchange dialog. Exchange libraries might want to call this function in certain circumstances. An application rarely calls this function, but it may do so if it displays its own dialog in response to the launch code `sysAppLaunchCmdExgAskUser`.

The preview feature is deprecated and is maintained only for backward compatibility. Applications should no longer implement preview via this mechanism, but should follow the data viewing guidelines described in "Put with View Mode" on page 147.

This function provides to the exchange dialog the first of the following object descriptions that it finds:

- the data's description from `socketP->description`
- the filename in `socketP->name`
- the target application's description as stored in the exchange registry
- the MIME type in `socketP->type`
- the file extension in `socketP->name`

See Also `ExgNotifyReceive()`, `ExgDisconnect()`

ExgNotifyReceive Function

Purpose Delivers an object to the appropriate application using the registry.

Declared In `ExgMgr.h`

Prototype
```
status_t ExgNotifyReceive
    (ExgSocketType *socketP, uint16_t flags)
```

Parameters

↔ *`socketP`*
A pointer to the socket structure (see `ExgSocketType`).

→ *`flags`*
A bit field. Pass 0 or a combination of the following constants (OR the constants together to specify more than one):

`exgUnwrap`
The object being delivered should only be handled by an application that registered to receive it with the `exgUnwrap` flag set.

`exgNoAsk`
Do not ask the user to confirm receipt of data. If this constant is passed, the target application does not receive the `sysAppLaunchCmdExgAskUser` launch code, and the Exchange Manager does not call `ExgDoDialog()` to display the user confirmation dialog.

`exgGet`
Specifies that this is a request for the application to send data rather than to receive data.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrTargetMissing`	The `target` field contains a creator ID, but the application with that creator ID does not exist
`exgErrNoKnownTarget`	No application is registered to receive the data type
`exgErrUserCancel`	The user cancelled the operation

Other error codes depend on the application that is launched.

Comments Exchange libraries call this function to initiate a receive operation on the receiving device. Applications do not call this function.

The `ExgNotifyReceive()` function uses `ExgGetTargetApplication()` to determine which application should receive the data, then sends that application the appropriate launch codes.

If the `flags` parameter is 0, a receive operation is assumed. The `ExgNotifyReceive()` function does the following:

1. It sends the application the `sysAppLaunchCmdExgAskUser` launch code.
2. If the application returns `exgAskDialog` or does not respond to the launch code, it calls `ExgDoDialog()`, which sends the application the `sysAppLaunchCmdExgPreview` launch code to have the application receive preview data for the dialog.
3. It sends the application the `sysAppLaunchCmdExgReceiveData` launch code to tell the application to receive the data.

If the `flags` field contains the `exgNoAsk` flag, the first and second steps are skipped.

If the `flags` field contains `exgGet`, this function is a request for data to send to the remote device, not a request to receive data from the remote device. In this case, `ExgNotifyReceive()` launches the target application with the `sysAppLaunchCmdExgGetData` launch code.

If the `flags` field has the `exgUnwrap` bit set, it means that the object to be received was sent as part of another object, and it should only be sent to an application that registered to receive it with the `exgUnwrap` flag set. For example, if the exchange library receives an email message with an attached vStock object, the exchange library may call `ExgNotifyReceive()` with the `exgUnwrap` flag set and a socket that describes the vStock data type to see if there is an application that registered to receive it directly. If no application is registered to receive vStock objects with the `exgUnwrap` flag set, `ExgNotifyReceive()` returns `exgErrNoKnownTarget`. The exchange library should then call `ExgNotifyReceive()` again, but this time without the `exgUnwrap` flag and with a socket that describes the email message data type. This second call sends the object to the application registered to receive the email message rather than its vStock attachment. That application may extract the vStock attachment from the message and use the Local Exchange Library to send it to an application registered to receive vStock objects normally (without the `exgUnwrap` flag).

See Also `ExgNotifyPreview()`

ExgPut Function

Purpose Initiates the transfer of data to the destination device.

Declared In `ExgMgr.h`

Prototype `status_t ExgPut (ExgSocketPtr` *`socketP`*`)`

Parameters → *`socketP`*

Pointer to the socket structure (see `ExgSocketType`). Specify a URL in the `name` field. The structure should also contain a value for the `target` or `type` fields.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find default exchange library
`exgMemError`	Not enough memory to initialize transfer
`exgErrNotEnoughPower`	The battery does not have enough power to perform the operation

Other error codes depend on the exchange library.

Comments Applications call this function to start a send operation.

If the connection does not already exist, this function establishes one. You must create and initialize an `ExgSocketType` structure containing information about the data to send and the destination application. All unused fields in the structure **must** be set to 0.

If no error is returned, this call **must** be followed by `ExgSend()`, to begin sending data, or `ExgDisconnect()`, to disconnect. You may want to call `ExgSend()` multiple times to send the data in chunks.

The socket's `name` field must identify by URL the library that performs the transfer. The socket should also specify what data is being sent by providing values for at least one of the `target` and `type` fields.

To provide users with a choice of transport mechanisms, the application can provide a URL that begins with a question mark (?). The Exchange Manager displays a dialog with a list of all exchange libraries that respond to URLs of the specified type. If only one exchange library is registered for this URL scheme, no dialog is displayed.

For example, many applications support a Send command. This command generates a URL with the prefix `exgSendPrefix` (see Predefined URL Prefixes). The Exchange Manager displays a dialog containing a list of libraries registered for that URL scheme. The user selects an exchange library, and that library's `ExgLibSend()` function is called.

If the library is not specified by URL, the Exchange Manager by default uses the IR Library; however, if the `localMode` flag is set, the Local Exchange Library is used instead.

See Also `ExgDisconnect()`, `ExgSend()`, `ExgConnect()`

ExgReceive Function

Purpose Receives data from a remote device.

Declared In `ExgMgr.h`

Prototype
```
uint32_t ExgReceive (ExgSocketPtr socketP,
   void *bufP, uint32_t bufLen, status_t *err)
```

Parameters

→ *socketP*
A pointer to the socket structure (see `ExgSocketType`).

← *bufP*
A pointer to the buffer in which to receive the data.

→ *`bufLen`*
The number of bytes to receive.

← *`err`*
A pointer to an error code result.

Returns Returns the number of bytes actually received. A zero result indicates the end of the transmission.

An error code is returned in the address indicated by `err`. The error code `exgErrUserCancel` is returned if the user cancels the operation. The error code `exgErrNotSupported` is returned if the application calls this function during a preview and the exchange library does not have any more data available or does not support preview.

Comments Applications call this function in the following circumstances:

- In response to the `sysAppLaunchCmdExgReceiveData` launch code, following a successful call to `ExgAccept()`.
- In response to the `sysAppLaunchCmdExgPreview` launch code, following a successful call to `ExgAccept()`.
- To receive requested data following a successful call to `ExgGet()`.

After receiving the data, applications call `ExgDisconnect()` to terminate the connection.

This function blocks the application until the end of the transmission or until the requested number of bytes has been received. However, exchange libraries can provide their own user interface that is shown during this call, is updated as necessary, and allows the user to cancel the operation in progress.

See Also ExgNotifyReceive()

ExgRegisterDatatype Function

Purpose Registers an application to receive a specific type of data, or registers an exchange library to handle specific URL schemes.

Declared In `ExgMgr.h`

Prototype
```
status_t ExgRegisterDatatype (uint32_t creatorID,
   uint16_t id, const char *dataTypesP,
   const char *descriptionsP, uint16_t flags)
```

Parameters

→ *`creatorID`*
The creator ID of the registering application or exchange library.

→ *`id`*
A registry ID constant identifying the type of the items being registered. See Registry ID Constants.

→ *`dataTypesP`*
Pointer to a tab-delimited, null-terminated string listing the items to register. (Use `"\t"` for the tab character.) To unregister, pass a `NULL` value. Each item in the string must be no more than `exgMaxTypeLength` characters. There can be no more than 16 types total.

NOTE: If specifying file extensions, do not include the period (.) that precedes the extension. If specifying URL prefixes, do not include the colon (:) at the end of the prefix.

→ *`descriptionsP`*
Pointer to a tab-delimited, null-terminated string that lists descriptions for the items in the `dataTypesP` parameter. (Use `"\t"` for the tab character.) Each description must be no longer than `exgMaxDescriptionLength`. Pass `NULL` to leave out the descriptions.

There must either be one description for all types or the number of descriptions must match the number of types.

The descriptions are used in dialogs displayed by Exchange Manager to identify applications or libraries, so they should be user-friendly. Use information that describes the type of information handled, such as pictures, sounds, contact information, etc. Don’t use MIME types or file extensions because they are not meaningful to the average user.

→ *`flags`*
Always pass zero for this parameter. (This was formerly used for the `exgUnwrap` flag, but that is now deprecated.)

Returns Returns `errNone` if successful, `exgMemError` if there is not enough memory to save the registration info, or one of the data manager error codes (`dmErr`...).

Comments Both applications and exchange libraries use this function to register with the Exchange Manager to receive certain types of data.

Applications must register with the Exchange Manager to receive data objects that do not specifically target that application using the creator ID in the `target` field.

Exchange libraries register to receive data with certain URL schemes. Otherwise, the IR Library handles all incoming data for which a library could not be found.

Both applications and libraries should register to receive data as soon as possible after they are installed and as soon as possible after a hard reset. For example, applications can call `ExgRegisterDatatype()` in response to the `sysAppLaunchCmdSyncNotify` launch code, which they receive immediately after install. Exchange libraries implemented as applications can also use this strategy. Exchange libraries implemented as shared libraries should call `ExgRegisterDatatype()` in their startup functions.

Make only one call to `ExgRegisterDatatype()` per registry type. If you want to register to receive multiple items, use a tab character (`\t`) to separate the items. If you were to, for example, make one call to register for the DOC file extension and one call to register for the TXT extension, the second call overwrites the first.

Specify `exgRegExtensionID` to register to receive data that has a filename with a particular extension. For example, if your application wants to receive files with a TXT extension, it could register like this:

```
ExgRegisterDatatype(myCreator,
  exgRegExtensionID, "TXT", NULL, 0);
```

If the application wants to receive files with a TXT extension or with a DOC extension, it could register like this:

```
ExgRegisterDatatype(myCreator,
  exgRegExtensionID, "TXT\tDOC", NULL, 0);
```

Specify `exgRegTypeID` to register to receive data with a specific MIME type. For example, if your application wants to receive “setext” text files, it could register like this:

```
ExgRegisterDatatype(myCreator, exgRegTypeID,
"text/x-setext", NULL, 0);
```

Specify `exgRegCreatorID` to register to receive data targeted for a particular creator ID. For example, if your application wants to handle all data intended for the ToDo application, it could register like this:

```
char toDoCreatorStr[5];
MemMove(toDoCreatorStr, sysFileCToDo, 4);
toDoCreatorStr[4] = chrNull;
ExgRegisterDatatype(myCreator, exgRegCreatorID,
  toDoCreatorStr, NULL, 0);
```

NOTE: To override one application’s receipt of data, you need to also set your application as the default for this creator ID. See `ExgSetDefaultApplication()`.

Most exchange libraries will want to register for a unique URL scheme that identifies only that library, plus they should register for a more general scheme, such as the send scheme (`exgSendScheme`), which causes the library to be listed in the Send With dialog when the user performs the Send command. The registry ID constant for URL prefixes is `exgRegSchemeID`.

```
ExgRegisterDatatype(myLibCreator,
  exgRegSchemeID, myScheme "\t" exgSendScheme,
  NULL, 0);
```

Registrations are active until a hard reset or until the application or library is removed. The registration information is preserved across a soft reset. When an application is removed, its registry information is also automatically removed from the registry, so there is not normally a need to unregister. If you want to unregister, you can call `ExgRegisterDatatype()` with a `NULL` value for the `dataTypesP` parameter.

Multiple applications can be registered to receive the same type of data. If this is the case, the application that is registered as the default (using `ExgSetDefaultApplication()`) is the one that receives the data unless the exchange socket explicitly specifies another application should receive it. If there is no default specified, the Exchange Manager determines a default.

Multiple libraries may also be registered to receive the same type of URL. In this case, if the URL begins with a question mark (?), the Exchange Manager displays a dialog so that the user can select which exchange library to use. If the URL does not begin with a question mark, the exchange library registered as the default is used. If there is no default specified, the Exchange Manager determines a default.

See Also `ExgRegisterData()`, `ExgGetTargetApplication()`, `ExgPut()`, `ExgGetDefaultApplication()`, `ExgGetRegisteredApplications()`, `ExgGetRegisteredTypes()`

ExgRegisterData Function

Purpose Registers an application to receive a specific type of data. This function is deprecated and replaced with `ExgRegisterDatatype()`.

Declared In `ExgMgr.h`

Prototype
```
status_t ExgRegisterData (uint32_t creatorID,
   uint16_t id, const char *dataTypesP)
```

Parameters

→ *`creatorID`*
Creator ID of the registering application.

→ *`id`*
Registry ID identifying the type of the items being registered. Specify `exgRegExtensionID` or `exgRegTypeID`.

→ *`dataTypesP`*
Pointer to a tab-delimited, null-terminated string listing the items to register. (Use `\t` for the tab character.) These include file extensions or MIME types. To unregister, pass a `NULL` value.

Returns Returns `errNone` if successful, otherwise, one of the data manager error codes (`dmErr`...).

Comments Applications that wish to receive data from anything other than another Palm Powered™ handheld running the same application must use this function to register for the kinds of data they can receive. Call this function when your application is loaded on the device.

ExgRequest Function

Purpose Requests some data from an exchange library or an application using a URL.

Declared In `ExgMgr.h`

Prototype `status_t ExgRequest (ExgSocketType *socketP)`

Parameters → *`socketP`*
Pointer to the socket structure (see `ExgSocketType`). Specify a URL in the `name` field.

Returns Returns one of the following error codes:

`errNone`	Success
`exgErrBadLibrary`	Couldn't find default exchange library
`exgErrNotEnoughPower`	The device does not have enough power to perform the operation
`sysErrLibNotFound`	Couldn't find library or application to respond to URL

Other error codes depend on the exchange library or application.

Comments The `ExgRequest()` function is similar to `ExgGet()` in that both are used to request data. The difference is that the application that calls `ExgGet()` is always the application that receives the data.

When you call `ExgRequest()`, the application that receives the data is the application that is registered to receive it. For example, using `ExgRequest()`, it is possible for one application to use the Exchange Manager to retrieve a vCard using any supported transport mechanism and have that data sent directly to the Address Book application instead of to the calling application.

The `socketP` passed to this function identifies the exchange library using a URL in the `name` field. The application must know beforehand the proper URL prefix for the exchange library with which it wants to connect. See Predefined URL Prefixes for a list of URL prefixes that the Exchange Manager provides.

If the provided URL begins with a question mark (?) and there are several exchange libraries registered for the specified URL scheme, the Exchange Manager displays a dialog from which the user selects the appropriate transport mechanism.

If the Exchange Manager cannot find a library that is registered for the specified URL, it assumes that an application is registered to receive the URL, and it launches that application with the `sysAppLaunchCmdGoToURL` launch code.

See Also `ExgGet()`, `ExgNotifyReceive()`

ExgSend Function

Purpose Sends data to the destination device.

Declared In `ExgMgr.h`

Prototype
```
uint32_t ExgSend (ExgSocketPtr socketP,
   const void *bufP, uint32_t bufLen,
   status_t  *err)
```

Parameters

→ *socketP*
A pointer to the socket structure (see `ExgSocketType`). A value must be provided for the `name` field. The structure should also contain values for the `target` or `type` fields.

→ *bufP*
A pointer to the data to send.

→ *bufLen*
The number of bytes to send.

← *`err`*
A pointer to an error code result.

Returns Returns either the same number of bytes as specified in `bufLen`, or 0 if nothing was sent. An error code is returned in the address indicated by `err`. The error code `exgErrUserCancel` is returned if the user cancels the operation.

Comments Call this function one or more times to send all the data, following a successful call to `ExgPut()`. After sending the data, call `ExgDisconnect()` to terminate the connection.

The exchange library may break large amounts of data into multiple packets or assemble small send commands together into larger packets, but the application will not be aware of these transport level details.

This function blocks the application until all the requested data is sent. However, the exchange library may provide its own user interface that is updated as necessary and allows the user to cancel the operation in progress.

See Also ExgReceive(), ExgGet()

ExgSetDefaultApplication Function

Purpose Sets the application that receives a specified type of data by default. This function also sets the default exchange library that handles particular URL schemes.

Declared In `ExgMgr.h`

Prototype
```
status_t ExgSetDefaultApplication
   (uint32_t creatorID, uint16_t id,
   const char *dataTypeP)
```

Parameters → *`creatorID`*
The creator ID of the application or library that should become the default for this type of data.

→ *`id`*
A registry ID constant identifying the type of data in `dataTypeP`. See Registry ID Constants.

→ *dataTypesP*
A pointer to a null-terminated string containing the desired type of data.

NOTE: If specifying a file extension, do not include the period (.) that precedes the extension. If specifying a URL prefix, do not include the colon (:) at the end of the prefix.

Returns Returns `errNone` upon success or `exgErrNoKnownTarget` if the specified application is not registered to receive the specified data type.

Comments This function sets the default application that receives data of a certain type when no target is specified; and it sets the default exchange library that handles URLs with a certain prefix.

We strongly recommend that applications allow the user to determine which application should become the default recipient for a data type. To do so, an application can use `ExgGetRegisteredApplications()` to get the list of applications registered for the same type of data as it is, and then display a dialog listing those applications and allow the user to select it. Then it should call `ExgSetDefaultApplication()` with the user-specified default.

If you call `ExgSetDefaultApplication()` with an application or library that is already the default, this function has no effect.

An application can become the default for its own creator ID even if it has not specifically registered to receive its own creator ID. That is, suppose several applications are registered to receive objects targeted for the ToDo application's creator ID. The ToDo application itself is not registered for its own creator ID, as it is not necessary to do so. However, an application can use code like the following to set the ToDo application as the default for its own creator ID.

```
char toDoCreatorStr[5];
MemMove(toDoCreatorStr, sysFileCToDo, 4);
toDoCreatorStr[4] = chrNull;
ExgSetDefaultApplication(sysFileCToDo,
  exgRegCreatorID, toDoCreatorStr);
```

See Also `ExgGetDefaultApplication()`, `ExgRegisterDatatype()`

Application-Defined Functions

ExgDBDeleteProcPtr Function

Purpose Handles the case where a database with an identical name already exists on the device.

Declared In `ExgMgr.h`

Prototype

```
Boolean (*ExgDBDeleteProcPtr) (const char *nameP,
   uint16_t version, DatabaseID dbID,
   void *userDataP)
```

Parameters

→ *`nameP`*
A pointer to the name of the identical database.

→ *`version`*
The version of the identical database.

→ *`dbID`*
The database ID of the identical database.

→ *`userDataP`*
The `userDataP` parameter you passed to `ExgDBRead()`. If used, this parameter contains any application-specific data you find necessary. If the `ExgDBReadProcPtr()` function is implemented using Exchange Manager calls, this often contains the `ExgSocketType` structure.

Returns Return `true` to have the `ExgDBRead()` function continue to read the database. Use this return value if you have deleted or moved the existing database or if you want the database to be overwritten. Return `false` to have `ExgDBRead()` exit without reading the database.

Comments This function is called if the Data Manager can't create the incoming database because a database with the same name already exists. You should delete the existing database or take some other action, such as changing the database name. It is appropriate to prompt the user before choosing to delete or move the database.

ExgDBReadProcPtr Function

Purpose Reads in the database and pass it to ExgDBRead().

Declared In `ExgMgr.h`

Prototype
```
status_t (*ExgDBReadProcPtr) (void *dataP,
   uint32_t *sizeP, void *userDataP)
```

Parameters

← *dataP*
A pointer to a buffer where this function should place the database data. This buffer is allocated in the dynamic heap by `ExgDBRead`; you don't need to use DmWrite() when filling it.

↔ *sizeP*
The size of `dataP`. This value is set by `ExgDBRead()` to the number of bytes it expects to receive in `dataP`. You must set this value to the number of bytes you return in `dataP` (if it's not the same).

→ *userDataP*
The `userDataP` parameter you passed to `ExgDBRead()`. Pass the ExgSocketType structure if you implement this function using Exchange Manager calls.

Returns Return an error number, or `errNone` if there is no error. If this function returns an error, `ExgDBRead()` deletes the database it was creating, cleans up any memory it allocated, then exits, returning the error passed back from this function.

Comments `ExgDBRead()` is commonly used to receive a database from a beam or from some other transport mechanism. In this case, an appropriate implementation of this callback function is to call ExgReceive() as shown here:

```
status_t MyReadDBProc (void *dataP, uint32_t *sizeP,
  void *userDataP)
{
  status_t err = errNone;
  //userDataP contains ExgSocketType pointer.
  *sizeP =
    ExgReceive((ExgSocketType *)userDataP,
      dataP, *sizeP, &err);
  return err;
}
```

ExgDBWriteProcPtr Function

Purpose Writes out the database.

Declared In `ExgMgr.h`

Prototype
```
status_t (*ExgDBWriteProcPtr) (const void *dataP,
    uint32_t *sizeP, void *userDataP)
```

Parameters

→ *dataP*
A pointer to a buffer containing the database data, placed there by `ExgDBWrite()`.

↔ *sizeP*
The number of bytes placed in `dataP` by `ExgDBWrite()`. If you were unable to write out or send all of the data in this chunk, on exit, set `sizeP` to the number of bytes you did write.

→ *userDataP*
The `userDataP` parameter you passed to `ExgDBWrite()`. You can use it for application-specific data. Pass the `ExgSocketType` structure if you implement this function using Exchange Manager calls.

Returns Return an error number, or `errNone` if there is no error. If this function returns an error, `ExgDBWrite` closes the database it was reading, cleans up any memory it allocated, then exits, returning the error passed back from this function.

Comments `ExgDBWrite()` is commonly used to write a database that is going to be beamed to another device (or sent through some other transport mechanism). In this case, an appropriate implementation of this callback function is to call `ExgSend()` as shown here:

```
status_t MyWriteDBProc (void *dataP, uint32_t *sizeP,
  void *userDataP)
{
  status_t err = errNone;
  //userDataP contains ExgSocketType pointer.
  *sizeP =
    ExgSend((ExgSocketType *)userDataP,
      dataP, *sizeP, &err);
  return err;
}
```

Part III Personal Data Interchange

The Palm OS® provides the PDI library API for exchanging Personal Data Interchange (PDI) information with other devices and media. The PDI reader and writer objects make use of the United Data Access (UDA) Manager to manage input and output data streams.

6

Personal Data Interchange

The Palm OS® provides the PDI library API for exchanging Personal Data Interchange (PDI) information with other devices and media. This chapter contains the following sections that describe how to use the Palm OS PDI library:

- About Personal Data Interchange briefly introduces the PDI standard and provides links to sources of more complete information.
- About the PDI Library describes how the Palm OS PDI library implements PDI reader and writer objects for exchanging information.
- Using the PDI Library describes how to use the functions in the PDI library.
- Using UDA for Different Media describes how you can use the Unified Data Access (UDA) Manager to access data from different media in your PDI reader or writer.
- Using a PDI Reader - An Example provides a detailed walk-through of a code segment that creates a PDI reader and then uses it to parse vCard information.
- Using a PDI Writer - An Example provides a detailed walk-through of a code segment that creates a PDI writer and then uses it to generate vCal information.

For detailed information about the PDI library data types, constants, and functions, see Chapter 7, "Personal Data Interchange Reference," on page 247.

The PDI reader and writer objects make use of the United Data Access (UDA) Manager to manage input and output data streams. "Using UDA for Different Media" on page 236 provides an overview of using the UDA Manager. The reference information for UDA functions is in Chapter 8, "Unified Data Access Manager Reference," on page 285.

About Personal Data Interchange

Personal data interchange involves the exchange of information using a communications medium. The Palm OS PDI Library facilitates the exchange of information using standard **vObjects**, including data formatted according to **vCard** and **vCal** standards.

The vObject standards are maintained by a group known as the versit consortium, which consists of individuals from a number of companies and institutions. The best information about the PDI standards can be found at the consortium's web site:

`http://www.imc.org/pdi/`

These standards are finding increased use in a number of computers and handheld devices that wish to exchange personal data such as business card and calendar information.

The PDI Library provides a `PdiReaderType` object for reading vObjects from an input stream, and a `PdiWriterType` object for writing vObjects to an output stream. The input streams and output streams can be connected to various data sources.

About vObjects

This section provides a brief overview of vObject standards. Two common vObject types are vCards and vCals:

- vCards are used to exchange virtual business card information electronically. Each vCard can include a large variety of personal and business information about an individual, including name, address, and telecommunications numbers.
- vCals are used to exchange virtual calendaring and scheduling information electronically. Each vCal can include:
  - vEvent objects, each of which represents a scheduled amount of time on a calendar
  - vTodo objects, each of which defines an action item or assignment

Overview of vObject Structure

This section provides a brief overview of vObject standards, including the vCard and vCal standards. Each vObject standard provides the same, basic organizational structure:

- Each vObject is a collection of one or more property definitions.
- Each property definition contains a name, a value, and an optional collection of property parameter definitions.
- Each property parameter definition contains a name and a value. Each parameter value qualifies the property definition with additional information.
- A property value can be structured to contain multiple values. The values are typically separated with commas or semicolons.

The vObject standards also allow developers to add custom extensions. All vObject readers that conform to the standard, including the `PdiReaderType` object, can read these extensions, though not all readers will act upon the information contained in them.

Each property has the following syntax:

```
PropertyName [';' Parameters] ':' PropertyValue
```

Note that property and parameter names are case insensitive.

Listing 6.1 shows a typical vCard definition.

Listing 6.1 Example of a vCard definition

```
BEGIN:VCARD
VERSION:2.1
N:Smith, John;M.;Mr.; Esq.
TEL;WORK;VOICE; MSG:+1 (408) 555-1234
TEL;CELL:+1 (408) 555-4321
TEL;WORK;FAX:+1 (408) 555-9876
ADR;WORK;PARCEL;POSTAL;DOM:Suite 101;1 Central  St.;Any
 Town;NC;28654
END:VCARD
```

Each line in Listing 6.1 is a property definition, with the exception of the next to last line, which is a continuation of the `ADR` property definition, and begins with white space. Each property definition is delimited by a CR/LF sequence.

The `BEGIN`, `VERSION`, and `END` lines are examples of simple property definitions.

The `N` (Name) property has a structured value. The components of the name are separated by semicolons.

Each `TEL` (Telephone) property has parameters that qualify the kind of telephone number that is being specified.

The `ADR` (Address) property has parameters and a structured value.

NOTE: The vObject specifications also allow long lines of text to be **folded**. This means that wherever you can have white space in a property definition, you can insert a CR/LF followed by white space, as shown in the next to last line in Listing 6.1 When the vObject reader finds a CR/LF followed by white space, it **unfolds** the text back into one long line.

Grouping vObjects

You can specify multiple vObjects in a single vObject data stream. You can also specify a vObject as the value of a property; for example, you can include a vCard as the value of the `ADR` property of another vCard.

Grouping Properties

You can specify a name for a group of related properties within a vObject. The name is a single character that you use as a prefix to each property in the group.

One use of this facility is to group a comment that describes a property with the property to keep the two together. For example, the following creates a group named `G` that includes a vCard home telephone property with a comment property:

```
G.TEL;HOME:+1 (831) 555-1234
G.Note: This is my home office number.
```

Encodings

The default encoding for vObject properties is 7-bit. You can override this encoding for individual property values by using the `ENCODING` parameter. You can specify various encoding values, including `BASE64`, `QUOTED-PRINTABLE`, and `8-BIT`.

Character Sets

The default character set for vObject properties is ASCII. You can override the character set for individual property values by using the `CHARSET` parameter. You can specify any character set that has been registered with the Internet Assigned Numbers Authority (IANA). For example, to specify the Latin/Hebrew encoding, you would use the value `ISO-8859-8`.

The system automatically converts incoming property values from the vObject character set to the device character set by using the Text Manager.

Finding More Information

For a complete description of the vObject specifications, visit the versit consortium's web site:

`http://www.imc.org/pdi/`

About the PDI Library

The Palm OS PDI library is a shared library that provides objects and functions for:

- Reading vCard objects from an input data stream. The section Creating a PDI Reader describes how to create and use a PDI reader, and the section Using a PDI Reader - An Example provides an example of reading vCard data from an input stream.
- Writing vCard objects to an output data stream. The section Creating a PDI Writer describes how to create and use a PDI reader, and the section Using a PDI Writer - An Example provides an example of reading vCard data from an input stream.

The PDI library handles reading and writing objects in a number of different formats, and from or to a variety of media. For more information about specifying the media, see "Using UDA for Different Media" on page 236.

PDI Property and Parameter Types

The PDI library provides constants that you can use with the reader and writer objects to specify property information. These include the following types of constants that specify vObject standard entities:

- The **Property Name** constants represent the PDI property names. Each of the property name constants starts with the `kPdiPRN_` prefix. For example, the `kPdiPRN_ADR` constant represents the `ADR` property name. For more information, see the section "Property Name Constants" on page 255.
- The **Property Value Field** constants represent the position of property value fields for properties with structured field values. Each of the property value field constants starts with the `kPdiPVF_` prefix. For example, the `kPdiPVF_ADR_COUNTRY` constant represents the `COUNTRY` field of an `ADR` property value. For more information, see the section "Property Value Field Constants" on page 256.
- The **Parameter Name** constants represent the names of vObject property parameters. Each of the parameter name constants starts with the `kPdiPAN_` prefix. For example, the `kPdiPAN_Type` constant represents the `TYPE` parameter, and the `kPdiPAN_Encoding` constant represents the `ENCODING` parameter. For more information, see the section "Parameter Name Constants" on page 254.
- The **Parameter Value** constants represent the combined name and value of parameters. Each of the parameter value constants starts with the `kPdiPAV_` prefix. For example, `kPdiPAV_ENCODING_BASE64` constant represents the `Base64` encoding. For more information, see the section "Parameter Value Constants" on page 254.

For a complete list of all of these constants, see the `PdiConst.h` file.

The PDI Library Properties Dictionary

The PDI library features a dictionary that stores information about the properties that are considered “well-known.” A well-known property is one that is defined in one of the vObject standard specifications, including the vCard and vCal standards. Both of these standards can be found online at the PDI developer’s web page:

`http://www.imc.org/pdi/pdiproddev.html`

PDI readers and writers use information in the properties dictionary to determine how to read or write a certain property. Specifically, the dictionary stores information about the format of each property value; the reader uses this information to correctly parse the property value, and the writer uses this information to correctly format the written value. This information is important because some property values are structured with multiple fields, while others contain a single value field.

For example, the standard address (`ADR`) property has a structured value with seven required fields, and the fields are separated by semicolons. The dictionary stores this information, and the PDI reader then knows to read seven, semicolon-separated fields when parsing an ADR property.

By default, each PDI reader and writer uses a standard dictionary when parsing input and generating output. You can, however, override this behavior to parse or generate the value for a property in some other way. For more information, see “Reading Property Values” on page 230 and “Writing Property Values” on page 236.

You can also amend or replace the dictionary to add parsing and/or generation of customized PDI properties for your application. For more information, see “Adding Custom Extensions” on page 234.

PDI Readers

The PDI library provides the PDI reader object for reading and parsing vObject input. A PDI reader object is a structure that stores the current state of parsing through a PDI input stream.

The PDI reader parses the input stream one property at a time, starting with the Begin Object property and finishing with the End Object property.

The `PdiReaderType` structure stores a variety of information about the current state of parsing the input stream, including the following information about the current property:

- the encoding and character set
- the type of the current property, parameter, and property value
- the name of the current property and parameter
- the current property's value string
- a mask of the parsing events encountered for the current property

About Parsing Events

The PDI reader records each parsing event that it encounters while processing a property. For example, when it parses a `BEGIN:VCARD` property, the PDI reader records the `kPdiBeginObjectEventMask`, and when it parses a property name, the PDI reader records the `kPdiPropertyNameEventMask`.

Each event is represented by one of the reader event constants, described in "Reader Event Constants" on page 261. The PDI reader records the event by adding (`OR'`ing) the event constant into the `events` field of the `PdiReaderType` structure.

You can determine if a specific event has occurred while parsing the current property by testing that event's constant against the `events` field in the reader structure. For example, the following statement returns `false` if the end of the input stream was reached.

```
return((reader->events & kPdiEOFEventMask)==0);
```

PDI Writers

The PDI library provides the PDI writer object for writing vObject output. A PDI writer object is a structure that stores the current state of and manages the generation of PDI data.

The PDI writer sends data to the output stream one property at a time, starting with the Begin Object property and finishing with the End Object property.

The `PdiWriterType` structure stores information about the current state of writing the output stream, including the following:

- the encoding and character set of the current property
- the mode used to write the current property value, which specifies how the property value is structured
- the number of required fields for the current property value

Format Compatibility

The PDI library can read and write data streams in the following formats:

- vCard 3.0
- vCard 2.1
- vCal 1.0
- iCalendar
- Palm format

You can use the PDI library to convert an input data stream that uses one format into an output data stream in another format. For more information, see "Specifying PDI Versions" on page 236.

Compatibility with Earlier Versions of the Palm OS

The PDI library has been designed to maintain compatibility with versions of the Palm OS earlier than 4.0, which means that you can use the library functions to receive vObjects from or send vObjects to devices that use those earlier versions.

To take advantage of this compatibility, the PDI library has been built to send or receive data in different formats, one of which is the format supported by versions of the Palm OS earlier than 4.0 that included the `ImcUtils` implementation.

To include support for this compatibility in a PDI Reader, specify the `kPdiOpenParser` constant in your call to the `PdiReaderNew()` function.

To include support for this compatibility in a PDI Writer, specify the `kPdiPalmCompatibility` option when calling the `PdiWriterNew()` function.

International Considerations

The PDI library handles various character sets, including all those supported by the Text Manager. If you specify the `CHARSET` parameter in the input stream, the PDI reader will correctly read the property value.

If you specify an unknown character set, the current character set becomes unknown, as represented by the `charEncodingUnknown` constant.

The system automatically converts incoming property values from the vObject character set to the device character set by using the Text Manager.

Features Not Yet Supported

The PDI library included with Palm OS Cobalt does not handle the following features:

- Multi-part MIME messages are not handled.
- The XML version of vObjects is not supported.
- Applications ignore grouping. The PDI reader parses group identifiers, but ignores them. However, the name of the group most recently parsed is stored in the `groupName` field of the `PdiReaderType` object.

Using the PDI Library

This section describes how to use the functions in the PDI library to read or write PDI content. Figure 6.1 shows the typical sequences of calls that you make to read or write vObjects.

To read vObjects, you need to:

- create a PDI reader
- read each property in the input stream:
  - read the property name

  - read any parameters for the property
  - read the property value
- delete the PDI reader

To write vObjects, you need to:

- create a PDI writer
- write each property to the output stream:
  - write the property name
  - write any parameters for the property
  - write the property value
- delete the PDI writer

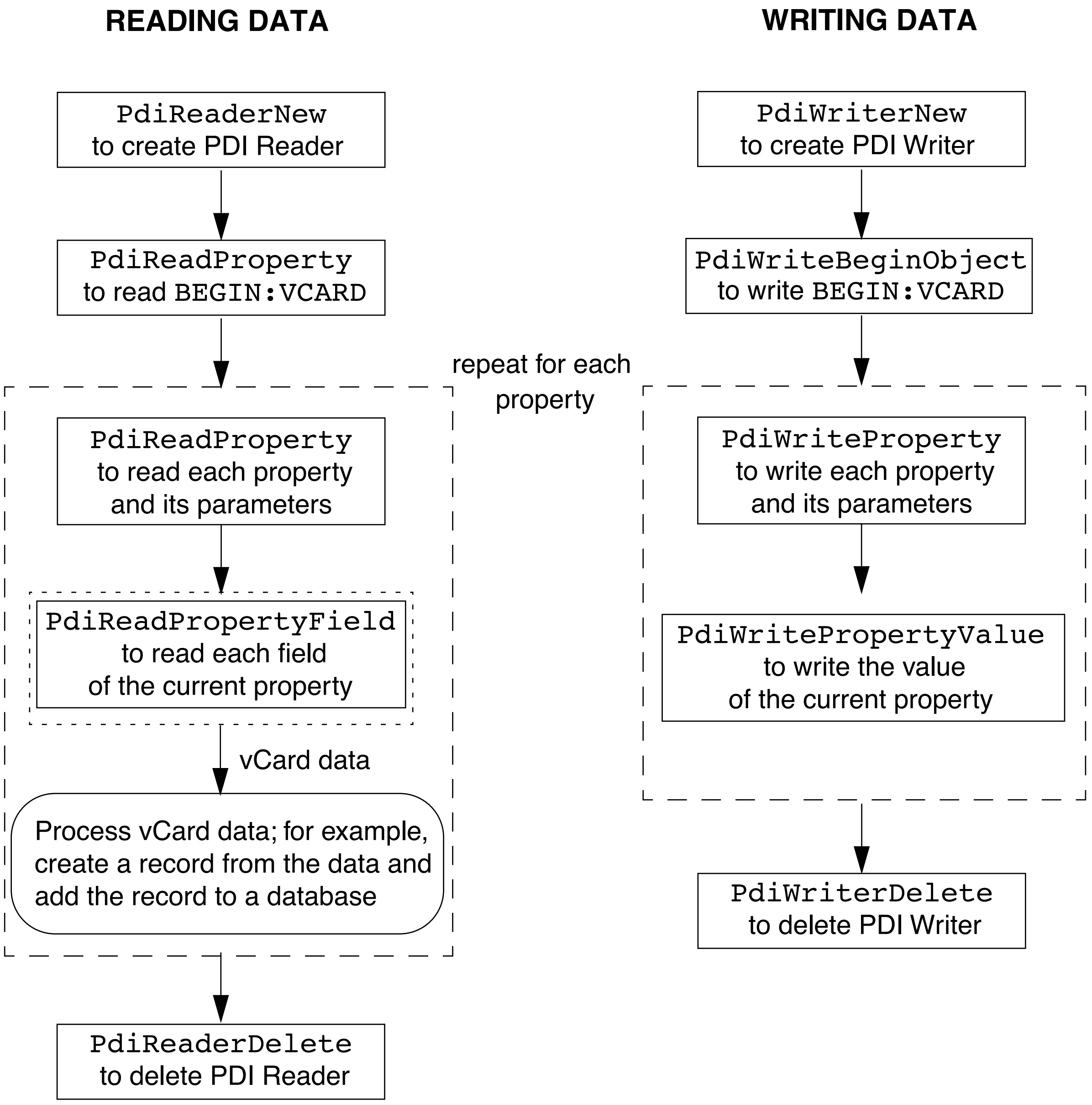
The remainder of this section describes the following operations:

- Creating a PDI Reader
- Reading Properties
- Creating a PDI Writer
- Writing Property Values
- Specifying PDI Versions
- Using UDA for Different Media

The section "Using a PDI Reader - An Example" on page 238 provides a detailed example of creating a PDI Reader and using it to import vCard data into a database.

The section "Using a PDI Writer - An Example" on page 241 provides a detailed example of creating a PDI Writer and using it to export data from a database in vCal format.

Figure 6.1 Using the PDI library


READING DATA
WRITING DATA
PdiReaderNew to create PDI Reader
PdiReadProperty to read BEGIN:VCARD
repeat for each property
PdiReadProperty to read each property and its parameters
PdiReadPropertyField to read each field of the current property
vCard data
Process vCard data; for example, create a record from the data and add the record to a database
PdiReaderDelete to delete PDI Reader
PdiWriterNew to create PDI Writer
PdiWriteBeginObject to write BEGIN:VCARD
PdiWriteProperty to write each property and its parameters
PdiWritePropertyValue to write the value of the current property
PdiWriterDelete to delete PDI Writer

Creating a PDI Reader

To create a PDI reader, you need to first access the library, and then call the `PdiReaderNew()` function, which is declared as follows:

```
PdiReaderType* PdiReaderNew(UDAReader *input,
uint16_t optionFlags)
```

The `PdiReaderNew()` parameters are:

- The Unified Data Access (UDA) input stream to use with the reader. The UDA Manager allows you to read input from various sources, including strings and the Exchange Manager. For more information, see "Using UDA for Different Media" on page 236.
- Option flags that control the parsing behavior of the reader, including its default encoding and compatibility settings. The option flags are described in "Reader and Writer Options Constants" on page 259.

Once you have created the reader, you can use it to parse properties from the input stream. The section "Using a PDI Reader - An Example" on page 238 provides an example of creating and using a PDI reader.

Reading Properties

To read PDI property data with a PDI reader, you need to call the data reading functions:

- `PdiReadProperty()` reads a property and all of its parameters from the input stream.
- `PdiReadPropertyName()` reads just the name of the next property from the input stream. You can call this function if you want to then handle the reading of the property's parameters individually.
- `PdiReadParameter()` reads a single parameter and its value from the input stream.
- `PdiReadPropertyField()` reads a property value field. A property value can a simple value, or it can be structured to contain multiple fields that are separated by commas or semicolons, as described in "Reading Property Values" on page 230.

The most common way to read input data is to follow these steps:

- Call `PdiReadProperty()` to read the vObject Begin property. For example, if you are reading vCards, you can call `PdiReadProperty` until it reads the `kPdiPRN_BEGIN_VCARD` property from the input stream.
- Once you have found the beginning of the object, repeatedly call `PdiReadProperty()` to read the next property and its parameters.
- For each property, call the `PdiReadPropertyField()` function as required to read the fields of the property.
- Continue reading properties until you read the vObject End property. For vCards, you process properties until `PdiReadProperty()` reads the `kPdiPRN_END_VCARD` property from the input stream.

Examining Property Information

After calling a property-reading function, you can access fields of the `PdiReaderType` object to determine information about the current property. The current property is the one that is currently being parsed, or which has just been parsed.

For example, you can examine the `property` field of the `PdiReaderType` object to determine which type of property has just been read, or you can call the `PdiParameterPairTest()` macro to determine if a certain parameter pair was present in the property definition.

Reading Property Values

Some properties have simple values and others have structured values. A structured property value has multiple fields that are separated by commas or semicolons.

For example, the following phone property definition has a simple value:

```
TEL;CELL:+1 (408) 555-4321
```

Note that the phone property contains a semicolon to separate the `CELL` parameter from the property name. Each property's value follows the colon in the definition.

The following name property definition has a structured value that contains five fields separated by semicolons:

```
N:Smith; John;M.;Mr.; Esq.
```

You must pass a parameter to the `PdiReadPropertyField()` function to tell it how to process a property value. To specify how the field is formatted, use one of the property value format constants described in "Property Value Format Constants" on page 257.

You can specify `kPdiDefaultFields` to allow the PDI reader to determine the property value format. The reader looks up the property name in the dictionary to determine its format.

- Specify `kPdiNoFields` to have the reader parse the entire value in one operation.
- Specify `kPdiCommaFields` or `kPdiSemicolonFields` to have the reader parse a single field from the value.
- Specify `kPdiConvertComma` or `kPdiConvertSemicolon` to have the reader parse all of the fields in a value into a single value.

You can usually specify `kPdiDefaultFields` and allow the PDI Reader to use the information in the dictionary to properly parse the value. However, this might not always meet your needs, especially if your input stream contains custom properties.

Table 6.1 shows the results of using the different format constants to read the same property from the input stream. The example property is a standard address (`ADR`) property that has a structured value with seven, semicolon-delimited fields:

```
ADR:postoffice;extended;street;locale;region;postal_code;country
```

Note that since the `ADR` property is defined in the vCard standard as a structured value with seven, semicolon-delimited field, the PDI library dictionary defines its default format as `kPdiSemicolon`.

Table 6.1 Parsing a structured value with different value format types

Value format type	Description of `PdiReadPropertyField()` results
`kPdiNoFields`	One call returns the entire value as a string: "postoffice;extended;street;locale;region;postal_code;co untry"
`kPdiSemicolon`	Each call returns a single, semicolon-delimited field from the value. For example: • the first call returns "postoffice" • the second call returns "extended" • the third call returns "street"
`kPdiComma`	Each call returns a single, comma-delimited field from the value. For example, if the input string is "postoffice,extended,street," then: • the first call returns "postoffice" • the second call returns "extended" • the third call returns "street"
`kPdiConvertSemicolon`	One call returns the entire value as a string that has newline characters wherever a semicolon appeared in the input: "postoffice extended street locale region postal_code country"

Table 6.1 Parsing a structured value with different value format types *(continued)*

Value format type	Description of `PdiReadPropertyField()` results
`kPdiConvertComma`	One call returns the entire value as a string that has newline characters wherever a comma appeared in the input: "postoffice extended street locale region postal_code country"
`kPdiDefaultFields`	Same as `kPdiSemicolon`, because the PDI library dictionary defines the property value format of the `ADR` field as `kPdiSemicolon`.

Reading Value Fields One At a Time

If you are reading the fields in a structured value one at a time, and you don't know the exact number of fields, you can call `PdiReadPropertyField()` repeatedly until it returns a nonzero result.

For example, the following code segment from the `DateTransfer.c` program parses each field of the `EXDATE` property value fields:

Listing 6.2 Reading an undetermined number of value fields

```
while (PdiReadPropertyField(reader, &tempP, kPdiResizableBuffer,
                             kPdiSemicolonFields) == 0)
   {
               // Resize handle to hold exception
      err = MemHandleResize(exceptionListH,
sizeof(ExceptionsListType) + sizeof(DateType) * exceptionCount);
      ErrFatalDisplayIf(err != 0, "Memory full");
              // Lock exception handle
      exceptionListP = MemHandleLock(exceptionListH);
              // Calc exception ptr
```

```
        exceptionP = (DateType*)((uint32_t)exceptionListP
                        + (uint32_t)sizeof(uint16_t)
                        + (uint32_t)(sizeof(DateType) * exceptionCount));
                // Store exception into exception handle
        MatchDateTimeToken(tempP, exceptionP, NULL);
                // Increase exception count
        exceptionCount++;
                // Unlock exceptions list handle
        MemHandleUnlock(exceptionListH);
    }
```

NOTE: If you leave fields in a structured value unread, the next call to `PdiReadProperty` will skip over them and correctly find the beginning of the next property.

Adding Custom Extensions

The vObject standards are extensible, which means that you can add custom properties to vCards and other vObjects. The PDI library handles these custom properties; however, you must either add an entry to the library's dictionary for each custom property, or specify a constant other than `kPdiDefaultFields` when parsing the property's value.

Each PDI reader object and each PDI writer object can have a custom dictionary associated with it. You can configure the custom dictionary to amend or to replace the standard, built-in dictionary.

To associate a custom dictionary with a reader or writer, you need to first create the dictionary with the You can then call the `PdiDefineReaderDictionary()` function to associate that dictionary with a reader object or call the `PdiDefineWriterDictionary()` function to associate the dictionary with a writer object.

Creating a PDI Writer

To create a PDI writer, you need to first access the library, and then call the `PdiWriterNew()` function, which is declared as follows:

```
PdiWriterType* PdiWriterNew(UDAWriter *output,
uint8_t optionFlags)
```

The `PdiWriterNew()` parameters are:

- The UDA output stream to use with the writer. For more information, see "Using UDA for Different Media" on page 236.
- Option flags that control the output generation behavior of the writer, including its default encoding and compatibility settings. The option flags are described in "Reader and Writer Options Constants" on page 259.

Once you have created the writer, you can use it to generate properties to the output stream. The section "Using a PDI Writer - An Example" on page 241 provides an example of creating and using a PDI writer.

Writing Properties

To write PDI data with a PDI writer, you need to call the data writing functions. The most commonly used functions are:

- `PdiWriteBeginObject()`, which writes a vObject Begin tag to the output stream.
- `PdiWriteEndObject()`, which writes a vObject End tag to the output stream.
- `PdiWriteProperty()`, which writes a property to the output stream.
- `PdiWritePropertyValue()`, which writes a property value to the output stream.

The most common way to write output data is to follow these steps:

- Call `PdiWriteBeginObject()` to write the vObject Begin property. For example, if you are writing vCards, you call `PdiWriteBeginObject()` to write the `kPdiPRN_BEGIN_VCARD` property to the output stream.
- For each property that you want to write, call `PdiWriteProperty()` to write the next property and its parameters, and then call the `PdiWritePropertyValue()` function to write the property's value.
- Call `PdiWriteEndObject()` to write the vObject End property. For example, if you are writing vCards, you call

`PdiWriteEndObject()` to write the `kPdiPRN_END_VCARD` property to the output stream.

Writing Property Values

In many cases, you can simply call the `PdiWritePropertyValue()` function to write a value to the output stream. If a value contains a variable number of fields, you can instead use the `PdiWritePropertyFields()` to write the fields from an array. Or you can use the `PdiWritePropertyStr()` to write multiple fields separated by commas or semicolons.

Specifying PDI Versions

The PDI library options constants control how the PDI reader and PDI writer operate. These options are described in "Reader and Writer Options Constants" on page 259.

Using UDA for Different Media

The PDI reader and writer objects use Unified Data Access (UDA) Manager objects for reading from and writing to a variety of media. The UDA data types, constants, and functions are documented in Chapter 8, "Unified Data Access Manager Reference," on page 285. This section provides an overview of using UDA objects with the PDI library.

About the UDA Library

The UDA Manager provides an abstract layer for reading, filtering, and writing data to and from different media. The UDA Manager provides three general purpose object types:

- `UDAReaderType` objects (UDA Readers) read data from an input stream.
- `UDAFilterType` objects (UDA Filters) take input from UDA Readers or UDA Filters, perform some encoding or decoding operations, and output the data to a memory buffer.

- `UDAWriterType` objects (UDA Writers) write data to a filter or an output stream.

The UDA Manager provides general purpose functions for creating these object types. In addition, the UDA Manager provides built-in object types for working with memory buffers and the Exchange Manager.

NOTE: The implementation of the UDA Manager in Palm OS Cobalt does not provide built-in filter objects.

Interfacing with the Exchange Manager

The UDA Manager provides two functions for interfacing with the Exchange Manager:

- The `UDAExchangeReaderNew()` function creates a UDA Reader object that reads data from an Exchange Manager socket.
- The `UDAExchangeWriterNew()` function creates a UDA Writer object that writes data to an Exchange Manager socket.

The Exchange Manager, which is described in Chapter 4, "Object Exchange," on page 105, provides a mechanism for reading typed data in a transport-independent manner.

When you use the UDA interface to the Exchange Manager, you add the benefits of a simple, uniform way to read and write data in a transport-independent manner. This allows you to create PDI readers and writers that can work on data that is stored on a variety of media types.

If you wish to parse PDI objects from memory, you can use an object created by the `UDAMemoryReaderNew()` function instead of an Exchange Manager reader object.

The PDI Reader example in the next section reads its data from an Exchange Manger socket, using the `UDAExchangeReaderNew()` function to create the reader object.

The PDI Writer example in "Using a PDI Writer - An Example" on page 241 writes its data to an Exchange Manager socket, using the `UDAExchangeWriterNew()` function to create the writer object.

Using a PDI Reader - An Example

This section provides an example of reading PDI data from an input stream and storing it in a database. This example is from the `AddressTransfer.c` file, which is located inside of the `Examples/Address/Src` folder.

Listing 6.3 shows the `TransferReceiveData()` function from the `AddressTransfer.c` sample program. This function controls the reading of vCard data into the address database by performing the following operations:

- Calls the `ExgAccept()` function to accept a connection from a remote device.
- Calls the `UDAExchangeReaderNew()` function to create an input data stream for connection with the Exchange Manager.
- Calls the `PdiReaderNew()` function to create a new PDI reader object that reads from the input stream.
- Repeatedly calls the local function `TransferImportVCard()` to read vCard data and store it into the address database. This function is described in the next section, Importing vCard Data Into a Database.
- Calls the `ExgDisconnect()` function to terminate the transfer and close the connection.
- Deletes the PDI reader and UDA input stream objects.

Listing 6.3 Reading a PDI input stream

```
extern Err TransferReceiveData(DmOpenRef dbP, ExgSocketPtr exgSocketP)
{
   volatile Err err;
   PdiReaderType* reader = NULL;
   UDAReader* stream = NULL;

   if ((err = ExgAccept(exgSocketP)) != 0)
      return err;
   if ((stream = UDAExchangeReaderNew(exgSocketP)) == NULL)
   {
      err = exgMemError;
      goto errorDisconnect;
   }
   if ((reader = PdiReaderNew(stream, kPdiOpenParser)) == NULL)
   {
      err = exgMemError;
      goto errorDisconnect;
   }
   reader->appData = exgSocketP;
   ErrTry
   {
      while(TransferImportVCard(dbP, reader, false, false)){};
   }
   ErrCatch(inErr)
   {
      err = inErr;
   } ErrEndCatch
   if (err == errNone && exgSocketP->goToParams.uniqueID == 0)
      err = exgErrBadData;
errorDisconnect:
   if (reader)
      PdiReaderDelete(&reader);
   if (stream)
      UDADelete(stream);
   ExgDisconnect(exgSocketP, err); // closes transfer dialog
   err = errNone; // error was reported, so don't return it
   return err;
}
```

Importing vCard Data Into a Database

The `TransferImportVCard()` function imports a vCard record from an input stream. Listing 6.4 shows the basic outline of the `TransferImportVCard()` function; you can review the entire function by viewing the `AddressTransfer.c` file, which is located inside of the `Examples/Address/Src` folder.

Listing 6.4 Importing vCard data into a database

```
Boolean TransferImportVCard(DmOpenRef dbP, PdiReaderType* reader,
                            Boolean obeyUniqueIDs, Boolean beginAlreadyRead)
{

...    // local declarations and initialization code

   ErrTry
   {
      phoneField = firstPhoneField;
      if (!beginAlreadyRead)
      {
         PdiReadProperty(reader);
         beginAlreadyRead = reader->property == kPdiPRN_BEGIN_VCARD;
      }
      if (!beginAlreadyRead)
         ErrThrow(exgErrBadData);
      PdiEnterObject(reader);
      PdiDefineResizing(reader, 16, tableMaxTextItemSize);
      while (PdiReadProperty(reader) == 0
            && (property = reader->property) != kPdiPRN_END_VCARD)
      {
         switch(property)
         {
         case kPdiPRN_N:
         PdiReadPropertyField(reader,(char **) &newRecord.fields[name],
                              kPdiResizableBuffer, kPdiDefaultFields);
         PdiReadPropertyField(reader, (char **) &newRecord.fields[firstName],
                              kPdiResizableBuffer, kPdiDefaultFields);
      break;
         case kPdiPRN_NOTE:
            PdiDefineResizing(reader, 16, noteViewMaxLength);
            PdiReadPropertyField(reader, char **) &newRecord.fields[note],
                                 kPdiResizableBuffer, kPdiNoFields);
            PdiDefineResizing(reader, 16, tableMaxTextItemSize);
             break;

,,,  // other cases here for other properties

      }
      } // end while
                if (newRecord.fields[name] != NULL
         && newRecord.fields[company] != NULL
         && newRecord.fields[firstName] != NULL
         && StrCompare(newRecord.fields[name],
                              newRecord.fields[company]) == 0)
      {     // if company & name fields are identical, assume company only
         MemPtrFree(newRecord.fields[name]);
         newRecord.fields[name] = NULL;
      }
AddRecord:
         err = AddrDBNewRecord(dbP, (AddrDBRecordType*) &newRecord,
                                 &indexNew);
         if (err)
            ErrThrow(exgMemError);

   ...      // handle category assignment here

   }   //end of ErrTry
   if (error == exgErrBadData)
```

```
      return false;
   if (error != errNone)
      ErrThrow(error);
   return ((reader->events & kPdiEOFEventMask) == 0);
}
```

The `TransferImportVCard()` function performs the following operations:

- Calls the `PdiReadProperty()` function to read the `BEGIN:VCard` property from the input stream.
- Calls the `PdiEnterObject()` function to notify the PDI library that it is reading a new object from the input stream.
- Calls the `PdiDefineResizing()` function to set the maximum buffer size for reading properties for the address card.
- Repeatedly calls the `PdiReadProperty()` function to read properties of the address card. This repeats until `PdiReadProperty()` reads the `END:VCard` property, which indicates the end of data for the address card.
- For each address card property, calls `PdiReadPropertyField()` as required to read the values associated with the property. For example, when it reads the `kPdiPRN_N` name property, `AddrImportVCard()` calls `PdiReadPropertyField()` twice: once to read the last name, and a second time to read the first name.
- Creates a new address record and adds it to the Address Book database.
- Deallocates memory that it has allocated and performs other cleanup operations.

Again, note that Listing 6.4 only shows the outline of this function. You can find the entire function in the `AddressTransfer.c` file.

Using a PDI Writer - An Example

This section provides an example of writing PDI data from a database record to an output stream. This example is from the `ToDoTransfer.c` file, which is located inside of the `Examples/ToDo/Src` folder.

Listing 6.5 shows an example of creating and using a PDI writer. The `ToDoSendRecordTryCatch()` function controls the writing of data from the To Do database to vCal objects by performing the following operations:

- Calls the `PdiWriterNew()` function to create a new PDI writer object that writes to the UDA output stream specified by the `media` parameter.
- Calls the `PdiWriteBeginObject()` function to write the `BEGIN:VCAL` property to the output stream.
- Calls the `PdiWriteProperty()` function to write the `VERSION` property, and then calls the `PdiWritePropertyValue()` function to write the version value.
- Calls the `ToDoExportVCal()` function to write the To Do record, as described in the next section, Exporting vCal Data From a Database.
- Calls the `PdiWriteEndObject()` function to write the `END:VCAL` property to the output stream.
- Deletes the PDI writer object.

Listing 6.5 Writing a PDI Output Stream

```
static Err ToDoSendRecordTryCatch (DmOpenRef dbP, int16_t recordNum,
       ToDoDBRecordPtr recordP, UDAWriter* media)
{
   volatile Err error = 0;
   PdiWriterType* writer;

   writer = PdiWriterNew(media, kPdiPalmCompatibility);
   if (writer)
      {
ErrTry
      {
         PdiWriteBeginObject(writer, kPdiPRN_BEGIN_VCALENDAR);
         PdiWriteProperty(writer, kPdiPRN_VERSION);
         PdiWritePropertyValue(writer, (char*)"1.0", kPdiWriteData);
         ToDoExportVCal(dbP, recordNum, recordP, writer, true);
         PdiWriteEndObject(writer, kPdiPRN_END_VCALENDAR);
          }
ErrCatch(inErr)
        {
          error = inErr;
        } ErrEndCatch
         PdiWriterDelete(&writer);
       }
   return error;
}
```

Exporting vCal Data From a Database

The `ToDoExportVCal()` function exports a vCal record from the To Do database to an output stream. Listing 6.6 shows the basic outline of the `ToDoExportVCal()` function; you can review the entire function by viewing the `ToDoTransfer.c` file, which is located inside of the `Examples/Address/Src` folder.

Listing 6.6 Exporting vCal data from a database

```
extern void ToDoExportVCal(DmOpenRef dbP, int16_t index,
ToDoDBRecordPtr recordP, uint16_t PdiWriterType* writer, Boolean
writeUniqueIDs)
{
char * note;
uint32_t uid;
char tempString[tempStringLengthMax];
uint16_t attr;
...

   PdiWriteBeginObject(writer, kPdiPRN_BEGIN_VTODO);
      // Emit the Category
   PdiWriteProperty(writer, kPdiPRN_CATEGORIES);
      // ...code to create the property string (tempString)
   PdiWritePropertyValue(writer, tempString, kPdiWriteText);

      // Code to emit the record information, including the:
      //   - due date
      //   - completed flag
      //   - priority value
      //   - description text
...


      // Emit the note
   if (*note != '\0')
   {
      PdiWriteProperty(writer, kPdiPRN_ATTACH);
      PdiWritePropertyValue(writer, note, kPdiWriteText);
   }
```

```
      // Emit an unique id
   if (writeUniqueIDs)
   {
      PdiWriteProperty(writer, kPdiPRN_UID);
      // Get the record's unique id and append to the string.
      DmRecordInfo(dbP, index, NULL, &uid, NULL);
      StrIToA(tempString, uid);
      PdiWritePropertyValue(writer, tempString, kPdiWriteData);
   }

   PdiWriteEndObject(writer, kPdiPRN_END_VTODO);
}
```

The `ToDoExportVCal()` function performs the following operations:

- Calls the `PdiWriteBeginObject()` function to write the `BEGIN:VTODO` property to the output stream.
- Calls the `PdiWriteProperty()` function to write the category information for the To Do record.
- Calls the `PdiWriteProperty()` function to write other information for the To Do record, including the due date, completed flag, priority value, and description text.
- Calls the `PdiWriteProperty()` function to write the note and again to write a unique ID for the note.
- Calls the `PdiWriteEndObject()` function to write the `END:VTODO` property to the output stream.

Again, note that Listing 6.6 only shows the outline of this function. You can find the entire function in the `ToDoTransfer.c` file.

Summary of Personal Data Interchange

PDI Library Functions

Object Creation and Deletion

`PdiReaderNew()`	`PdiWriterNew()`
`PdiReaderDelete()`	`PdiWriterDelete()`

Property Reading

PDI Library Functions	
PdiDefineResizing()	PdiReadProperty()
PdiEnterObject()	PdiReadPropertyField()
PdiParameterPairTest()	PdiReadPropertyName()
PdiReadParameter()	
Property Writing	
PdiSetCharset()	PdiWriteParameterStr()
PdiSetEncoding()	PdiWriteProperty()
PdiWriteBeginObject()	PdiWritePropertyBinaryValue()
PdiWriteEndObject()	PdiWritePropertyFields()
PdiWriteParameter()	PdiWritePropertyStr()
	PdiWritePropertyValue()
Property Dictionary	
PdiDefineReaderDictionary()	PdiDefineWriterDictionary()

Summary of Unified Data Access Manager

UDA Manager Functions	
UDAControl()	UDAMoreData()
UDADelete()	UDARead()
UDAEndOfReader()	UDAWriterFlush()
UDAFilterJoin()	UDAWriterJoin()
UDAInitiateWrite()	
Object Creation	
UDAExchangeReaderNew()	UDAMemoryReaderNew()
UDAExchangeWriterNew()	

7

Personal Data Interchange Reference

This chapter provides reference material for the Personal Data Interchange (PDI) library, which provides tools for reading and writing vObjects, including vCards and vCals. This chapter discusses the following topics:

- PDI Library Data Structures
- PDI Library Constants
- PDI Library Functions

The header file `PdiLib.h` declares the Personal Data Interchange library API. The header file `PdiConst.h` declares the constants that you use with the PDI library.

For information about how to use the functions and constants described in this chapter, see Chapter 6, "Personal Data Interchange," on page 217.

PDI Library Data Structures

This section describes the data structures used with the PDI library functions.

PdiDictionary Typedef

Purpose The `PdiDictionary` type is a simple typedef that represents an internal, binary object.

Prototype

```
typedef uint8_t PdiDictionary;
```

PdiReaderType Struct

Purpose The `PdiReaderType` data structure represents a PDI reader object, which you use to read data from an input stream.

Prototype

```
typedef struct PdiReaderTag {
   Err16 errorLowWord;
   uint16_t encoding;
   uint8_t fieldNum;
   CharEncodingType charset;
   uint16_t written;
   uint16_t property;
   uint16_t propertyValueType;
   uint16_t parameter;
   uint16_t padding1;
   uint32_t parameterPairs[8];
   uint16_t customFieldNumber;
   uint16_t padding2;
   void *appData;
   uint16_t pdiRefNum;
   uint16_t events;
   char *groupName;
   char *propertyName;
   char *parameterName;
   char *parameterValue;
   char *propertyValue;
} PdiReaderType
```

Fields

`errorLowWord`
: The most recent error.

`encoding`
: The type of encoding for the property value.

`fieldNum`
: The current field number.

`charset`
: The character set of the property value.

`written`
: The number of characters that have currently been written to the buffer.

`property`
: The ID of the current property.

`propertyValueType`
: The value type of the current property value.

`parameter`
: The ID of the most recently parsed parameter name.

`padding1`
: Padding; not used.

`parameterPairs`
: An integer array with bits set for each parameter value that has been parsed for the current property value.

NOTE: You must use the `PdiParameterPairTest` macro to access this field.

`customFieldNumber`
: The number of the custom field parsed by the reader for the current property. Custom fields are used in the Address Book.

`padding2`
: Padding; not used.

`appData`
: Application-dependent data field.

`pdiRefNum`
: The library reference number associated with this reader. This is no longer used for ARM code development, but is kept for compatibility with code written for the 68K environment.

`events`
: The mask of events handled by the reader in its most recent operation. This is a combination of some number of the event constants described in Reader Event Constants.

`groupName`
: The group name for the current property.

`propertyName`
: The name of the current property.

`parameterName`
: The name of the current parameter.

`parameterValue`
: The value of the current parameter.

`propertyValue`
: The current property value string.

PdiWriterType Struct

Purpose The `PdiWriterType` data structure represents a PDI writer object, which you use to write data to an output stream.

Prototype

```
typedef struct _PdiWriter {
   Err16 errorLowWord;
   uint16_t encoding;
   CharEncodingType charset;
   uint8_t padding1[2];
   uint32_t reserved;
   void *appData;
   uint16_t pdiRefNum;
   uint16_t padding2;
} PdiWriterType
```

Fields

`errorLowWord`
: The most recent error.

`encoding`
: The type of encoding for the property value.

`charset`
: The character set of the property value.

`padding1`
: Padding; not used.

`reserved`
: Reserved for future use.

`appData`
: Application-dependent data field.

`pdiRefNum`
: The library reference number associated with this reader. This is no longer used for native code development, but is kept for compatibility with code written for the 68K environment.

`padding2`
Padding; not used.

PDI Library Constants

This section describes the constants used in the PDI library, which include the following constant types:

- Buffer Management Constants
- Encoding Type Constants
- Error Code Constants
- Parameter Name Constants
- Parameter Value Constants
- Property Name Constants
- Property Type Constants
- Property Value Field Constants
- Property Value Format Constants
- Reader and Writer Options Constants
- Reader Event Constants
- Value Type Constants

Buffer Management Constants

You use the buffer management constants to determine how buffers are managed in the PDI reader.

Constant	Value	Description
`kPdiResizableBuffer`	`0xFFFF`	Indicates that the buffer can be automatically resized by the PDI library.

Constant	Value	Description
`kPdiDefaultBufferMaxSize`	`0x3FFF`	The default maximum buffer size, in bytes. You can change the maximum size of a reader's buffer by calling the `PdiDefineResizing()` function.
`kPdiDefaultBufferDeltaSize`	`0x0010`	The default number of bytes by which the input buffer is grown when the PDI library performs automatic resizing. You can change the delta amount of a reader's buffer by calling `PdiDefineResizing()` function.

Encoding Type Constants

You use the encoding type constants to specify the type of encoding used in a vObject reader or writer.

Constant	Value	Description
`kPdiASCIIEncoding`	`0`	The vObject is not encoded.
`kPdiQPEncoding`	`kPdiPAV_ENCODING_QUOTED_PRINTABLE`	The vObject uses the quoted printable encoding.
`kPdiB64Encoding`	`kPdiPAV_ENCODING_BASE64`	The vObject uses Base 64 encoding. The writer outputs "`ENCODING=BASE64`."
`kPdiBEncoding`	`kPdiPAV_ENCODING_B`	The vObject uses Base 64 encoding. This is the same as the `kPdiB64Encoding` value, except that the PDI writer outputs "`ENCODING=B`." This encoding is used with the vCard 3.0 standard.

Constant	Value	Description
`kPdiEscapeEncoding`	`0x8000`	The vObject uses escapes for special characters.
`kPdiNoEncoding`	`0x8001`	The PDI writer does not encode the vObject value.

Error Code Constants

The PDI library functions return the error code constants shown in the following table to indicate their status.

Constant	Description
`pdiErrRead`	An error occurred while reading from the input stream.
`pdiErrWrite`	An error occurred while writing to the output stream.
`pdiErrNoPropertyName`	An attempt was made to write a property value before the property name was written.
`pdiErrNoPropertyValue`	The application did not write the last property value.
`pdiErrMoreChars`	The buffer is full. Superfluous characters have been discarded.
`pdiErrNoMoreFields`	There are no more property fields to read.
`pdiErrOpenFailed`	The PDI library could not be opened.
`pdiErrCloseFailed`	The PDI library could not be closed. This can occur if another application is using the library.

Parameter Name Constants

The `PdiConst.h` file defines several parameter name constants that you can use to specify the name of a parameter in functions that use parameter names. The parameter name constants have the following format:

`kPdiPAN_`*`parameterName`*

where *parameterName* is replaced by a parameter name.

The following table shows examples of parameter name constants. For a complete list, see the `PdiConst.h` file.

Constant	Description
`kPdiPAN_TYPE`	The `TYPE` parameter.
`kPdiPAN_ENCODING`	The `ENCODING` parameter.
`kPdiPAN_STATUS`	The `STATUS` parameter.

Parameter Value Constants

The `PdiConst.h` file defines several parameter value constants that you can use to specify the name and value of a parameter in functions that use name and value pairs. The parameter value constants have the following format:

`kPdiPAV_`*`parameterName_parameterValue`*

where *parameterName* is replaced by a parameter name and *parameterValue* is replaced by a parameter value.

The following table shows examples of parameter value constants. For a complete list, see the `PdiConst.h` file.

Constant	Description
`kPdiPAV_TYPE_VIDEO`	The parameter name is `TYPE` and the parameter value is `VIDEO`.
`kPdiPAV_ENCODING_BASE64`	The parameter name is `ENCODING` and parameter value is `BASE64`.
`kPdiPAV_ENCODING_8BIT`	The parameter name is `ENCODING` and the parameter value is `8BIT`.
`kPdiPAV_STATUS_ACCEPTED`	The parameter name is `STATUS` and the parameter value is `ACCEPTED`.

Property Name Constants

The `PdiConst.h` file defines several property name constants that you can use to specify the name of a PDI property in functions that use property names. The property name constants have the following format:

`kPdiPRN_`*`propertyName`*

where *propertyName* is replaced by a property name.

The following table shows examples of property name constants. For a complete list, see the `PdiConst.h` file.

Constant	Description
`kPdiPRN_ADR`	The `ADR` property.
`kPdiPRN_BDAY`	The `BDAY` property.
`kPdiPRN_BEGIN`	The `BEGIN` property.
`kPdiPRN_BEGIN_VCARD`	The `BEGIN:VCARD` property.

Property Type Constants

You use the property type constants defined in `PdiConst.h` to specify the data type of a property.

Constant	Description
`kPdiType_URI`	The data is a uniform resource identifier.
`kPdiType_UTC_OFFSET`	The data is an offset from UTC to local time.
`kPdiType_RECUR`	The data is a specification of a recurrence rule.
`kPdiType_PERIOD`	The data is a precise period of time.
`kPdiType_CAL_ADDRESS`	Calendar user data.
`kPdiType_BINARY`	Binary data.
`kPdiType_TEXT`	Text data.
`kPdiType_FLOAT`	Floating-point data.
`kPdiType_DURATION`	Time duration data.
`kPdiType_DATE_TIME`	Calendar date and time data.
`kPdiType_BOOLEAN`	Boolean data.
`kPdiType_INTEGER`	Signed integer data.
`kPdiType_TIME`	Time-of-day data.
`kPdiType_VCARD`	vCard data.
`kPdiType_PHONE_NUMBER`	Phone number data.

Property Value Field Constants

The `PdiConst.h` file defines several property value field constants that you can use to specify the position of a PDI property value field in functions that use fields. The property value field constants have the following format:

```
kPdiPVF_propertyValueField
```

where *propertyValueField* is replaced by a property value field name.

The following table shows examples of property name constants. For a complete list, see the `PdiConst.h` file.

Constant	Description
`kPdiPVF_ADR_POST_OFFICE`	The property value field that stores the post office portion of the address.
`kPdiPVF_ADR_EXTENDED`	The property value field that stores the extended portion of the address.
`kPdiPAN_ADR_COUNTRY`	The property value field that stores the country portion of the address.

Property Value Format Constants

Some properties have **structured values**, which are values that contain multiple fields. These fields are typically separated by commas or semicolons in the vObject input or output stream. You use the property value format constants with the `PdiReadPropertyField()` and `PdiWritePropertyStr()` functions to specify how to handle fields in a structured value.

Constant	Value	Description
`kPdiNoFields`	0	There are no fields in the property value; `PdiReadPropertyField()` reads the entire value, or `PdiWritePropertyStr()` specifies that the entire value should be written.
`kPdiCommaFields`	1	Fields are separated with comma ("`,`") characters; `PdiReadPropertyField()` reads one field, or `PdiWritePropertyStr()` specifies that one field should be written.
`kPdiSemicolonFields`	2	Fields are separated with semicolon ("`;`") characters; `PdiReadPropertyField()` reads one field, or `PdiWritePropertyStr()` specifies that one field should be written.
`kPdiDefaultFields`	4	The parser decides the property value format, based on the property name.
`kPdiConvertComma`	8	Fields are separated with comma characters; `PdiReadPropertyField()` reads the entire value and converts each comma into a newline ("`\n`") character.
`kPdiConvertSemicolon`	16	Fields are separated with semicolon characters; `PdiReadPropertyField()` reads the entire value and converts each semicolon into a newline ("`\n`") character.

Reader and Writer Options Constants

You use the reader and writer option constants to control how the PDI reader (parser) reads values from the input stream or to control how the PDI writer (generator) writes values to the output stream.

Constant	Value	Description
`kPdiEnableFolding`	`1`	Enables folding of properties in the output stream. **Folding** is a mechanism for breaking long lines to allow them to be transmitted without change. If you specify this flag, the PDI library folds long lines. Note that folding is not compatible with versions of the Palm OS® earlier than 4.0. Also note that other encoding formats, including quoted-printable and Base 64, define their own mechanisms for splitting long lines.
`kPdiEnableQuotedPrintable`	`2`	Enables quoted-printable encoding in the output stream and makes it the default encoding. This is an encoding format for non-ASCII values. You must have this enabled for compatibility with versions of the Palm OS earlier than 4.0. If you do not specify this option, the default encoding is Base 64.

Constant	Value	Description
`kPdiEscapeMultiFieldValues`	4	For compatibility with versions of the Palm OS earlier than 4.0. You must enable this for compatibility with versions of the Palm OS earlier than 4.0. However, some non-Palm PDI software does not support this format. For more information about compatibility with earlier versions of the Palm OS, see "Format Compatibility" on page 225.
`kPdiPalmCompatibility`	6	This is a combination of `kPdiEscapeMultiFieldValues | kPdiEnableQuotedPrintable | kPdiBypassLocaleCharEncoding`. It forces the PDI writer to generate output that is compatible with versions of the Palm OS earlier than 4.0.
`kPdiEnableB`	8	Enables base 64 encoding in the output stream, and tells the PDI writer to output "`ENCODING=B`" instead of "`ENCODING=BASE64`" when encoding a value with Base 64. Note: the vCard 3.0 standard has replaced the earlier `ENCODING=BASE64` with `ENCODING=B`. The meaning is the same.
`kPdiOpenParser`	16	Specifies that the PDI reader is open to all formats, including Palm and others.
`kPdiBypassLocaleCharEncoding`	32	Bypasses the default character encoding for reading and writing.

Reader Event Constants

The PDI reader event constants specify the events that the reader has handled during the current read operation. The event values are combined together and stored in the `events` field of the PDI reader object. You can use them to test whether the reader handled a certain event.

Constant	Value	Description
`kPdiEOFEventMask`	1	End of file was reached.
`kPdiGroupNameEventMask`	2	A group name was found.
`kPdiPropertyNameEventMask`	4	A property name was found.
`kPdiParameterNameEventMask`	8	A parameter name was found.
`kPdiParameterValueEventMask`	16	A parameter value was found.
`kPdiPropertyDefinedEventMask`	32	A property definition was found; this implies that the ":" separator character was found.
`kPdiPropertyValueEventMask`	64	An entire property value was found
`kPdiPropertyValueFieldEventMask`	128	A value field was found; this implies that the ";" or CR/LF separator character was found.
`kPdiPropertyValueItemEventMask`	256	A value item was found; this implies that the "," or CR/LF separator character was found.

Constant	Value	Description
`kPdiPropertyValueMoreCharsEventMask`	`512`	The buffer that you provided was not large enough. The superfluous characters have been discarded.
`kPdiBeginObjectEventMask`	`1024`	The object begin indicator `BEGIN` was reached.
`kPdiEndObjectEventMask`	`2048`	The object end indicator `END` was reached.
`kPdiPropertyValueCRLFEventMask`	`4096`	Not used.

Value Type Constants

You can use the following constants to specify data typing information for the `PdiWritePropertyBinaryValue()`, `PdiWritePropertyFields()`, and `PdiWritePropertyValue()` functions.

Constant	Value	Description
`kPdiWriteData`	`0`	The value is data. The PDI writer does not compute a character set. You can use this for binary data or pure ASCII data.
`kPdiWriteText`	`8`	The value is text data. The PDI writer parses the data character by character to compute the correct charset and character encoding for the data.
`kPdiWriteMultiline`	`16`	Explicitly specifies that the value contains special characters, such as newlines, and must be encoded. If this flag is not specified, the encoding is determined by the applied character set.

PDI Library Functions

PdiDefineReaderDictionary Function

Purpose Installs a new custom dictionary for use with a PDI reader object.

Declared In `PdiLib.h`

Prototype

```
PdiDictionary *PdiDefineReaderDictionary
    (PdiReaderType *ioReader,
    PdiDictionary *dictionary,
    Boolean disableMainDictionary)
```

Parameters

→ *ioReader*
The PDI reader object with which to associate the dictionary. This object must have previously been created by a call to the `PdiReaderNew()` function.

→ *dictionary*
A pointer to a dictionary object that was created by the . The dictionary object is an array of binary data.

→ *disableMainDictionary*
If `true`, the main reader dictionary is disabled, and only this new dictionary is searched for terms; if `false`, the new dictionary supplements the main dictionary.

Returns Returns a pointer to the previously installed custom dictionary, or `NULL` if there was not a previously installed custom dictionary.

Comments This function installs a dictionary for use with the `ioReader` object. The dictionary contains the syntax for extensions or replacements of the PDI properties about which the PDI reader knows. The reader knows about properties specified in one of the vObject standards, including the vCard or vCal standards.

You can uninstall the current custom dictionary by specifying `NULL` as the value of the `dictionary` parameter,

PdiDefineResizing Function

Purpose Defines the sizing information to use when automatically resizing a buffer. PDI reader objects read information from the input stream into a buffer and automatically adjust the buffer size as required.

Declared In `PdiLib.h`

Prototype

```
status_t PdiDefineResizing
   (PdiReaderType *ioReader, uint16_t deltaSize,
   uint16_t maxSize)
```

Parameters

→ *`ioReader`*
The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

→ *`deltaSize`*
The number of bytes by which to grow the buffer when it needs resizing.

→ *`maxSize`*
The maximum allowable size for the buffer.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments This function redefines the values to use when resizing a buffer. It does not perform any other actions.

The resizing values are used if your reader runs out of buffer space when storing input data during the processing of a property value. If possible, the reader resizes its internal buffer, using the values that you supply in this function.

The default resizing values apply if you do not call this function. The default values are:

`kPdiDefaultBufferDeltaSize 0x0010`

`kPdiDefaultBufferMaxSize 0x3FFF`

PdiDefineWriterDictionary Function

Purpose Installs a new custom dictionary for use with a PDI writer object.

Declared In `PdiLib.h`

Prototype

```
PdiDictionary *PdiDefineWriterDictionary
    (PdiWriterType *ioWriter,
    PdiDictionary *dictionary,
    Boolean disableMainDictionary)
```

Parameters

→ *`ioWriter`*
The PDI writer object with which to associate the dictionary. This object must have previously been created by a call to the `PdiWriterNew()` function.

→ *`dictionary`*
A pointer to a dictionary object that was created by the . The dictionary object is an array of binary data.

→ *`disableMainDictionary`*
If `true`, the main dictionary is disabled, and only this new dictionary is searched for terms; if `false`, the new dictionary supplements the main dictionary.

Returns Returns a pointer to the previously installed custom dictionary, or `NULL` if there was not a previously installed custom dictionary.

Comments This function installs a dictionary for use with the `ioWriter` object. The dictionary contains the syntax for extensions or replacements of the PDI properties about which the PDI writer knows. The writer knows about properties specified in one of the vObject standards, including the vCard or vCal standards.

You can uninstall the current custom dictionary by specifying `NULL` as the value of the `dictionary` parameter,

PdiEnterObject Function

Purpose Tells the PDI library to enter into a recursively-defined object.

Declared In `PdiLib.h`

Prototype `status_t PdiEnterObject (PdiReaderType *ioReader)`

Parameters → *`ioReader`*
The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments Some vObjects recursively define other vObjects. Your application can choose whether or not to enter and parse the recursively defined objects.

If you want to parse the nested object definition, you need to call this function; otherwise, all of the properties of the nested object are skipped when the next call is made to the `PdiReadProperty()` or `PdiReadPropertyName()` functions.

Call this function after a `BEGIN_VObject` statement of the nested object has been parsed.

PdiLibClose Function

Purpose Decrements a reference count of `PdiLibOpen()` calls.

Declared In `PdiLib.h`

Prototype `status_t PdiLibClose (void)`

Parameters None.

Returns Returns 0 if no other application uses the library. Returns `pdiErrCloseFailed` if the library is in use by another application.

Comments This function is no longer needed for ARM code development, but is kept for compatibility with code written for the 68K environment.

See Also `PdiLibOpen()`

PdiLibOpen Function

Purpose Increments a reference count of `PdiLibOpen()` calls.

Declared In `PdiLib.h`

Prototype `status_t PdiLibOpen (`*`void`*`)`

Parameters None.

Returns Returns `errNone`.

Comments This function is no longer needed for ARM code development, but is kept for compatibility with code written for the 68K environment.

See Also `PdiLibClose()`

PdiParameterPairTest Macro

Purpose Determines if the reader has already parsed the specified parameter value or name-value pair.

Declared In `PdiLib.h`

Prototype `#define PdiParameterPairTest (`*`reader`*`, `*`pair`*`)`

Parameters → *`reader`*

The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

→ *`pair`*

The ID of the parameter. This must be one of the Parameter Value Constants.

Returns Returns `true` if the specified parameter name-value pair has been parsed for the current property, and `false` if not.

Comments Some vObject generators do not specify the parameter name if the name is considered evident from the context. This means that both of the following constructs are considered proper:

`Name=Value`

`Value`

The `PdiParameterPairTest` macro returns `true` if the value has been parsed in either format. For example,

```
PdiParameterPairTest(reader, kPdiPAV_TYPE_WORK)
```

returns true for either of the following:

`Type=WORK`

`WORK`

PdiReaderDelete Function

Purpose Delete a PDI reader object that is associated with the specified library number.

Declared In `PdiLib.h`

Prototype `void PdiReaderDelete (PdiReaderType **`*`ioReader`*`)`

Parameters ↔ *`ioReader`*

A pointer to the PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

Returns Returns nothing.

Comments This function deletes the `UDAReader` object associated with the reader object and frees the memory that was allocated for the reader object. The `ioReader` parameter is set to `NULL`.

See Also `PdiReaderNew()`

PdiReaderNew Function

Purpose Create and initialize a new PDI reader object for use with the specified PDI library number.

Declared In `PdiLib.h`

Prototype

```
PdiReaderType *PdiReaderNew
   (UDAReaderType *input, uint16_t version)
```

Parameters → *`input`*

The Unified Data Access (UDA) input object associated with the reader.

→ *`version`*

Options to control the parsing behavior of the reader. You can use a combination of the Reader and Writer Options Constants.

Returns Returns a pointer to the new PDI reader object. Returns `NULL` if the reader cannot be created.

Comments The current implementation of the `PdiReaderNew()` function does not make use of the `optionFlags` settings because the reader knows how to adapt itself to all of the supported formats. The options will be used in future versions.

The `input` value is a UDA object for reading data from an input stream that can be connected to various data sources. For example, you can use a `UDAExchangeReader` to read data from the Exchange Manager, and you can use a `UDAStringReader` to read data from a string. For more information about the UDA Manager, see Chapter 8, "Unified Data Access Manager Reference."

See Also `PdiReaderDelete()`, `PdiWriterNew()`

PdiReadParameter Function

Purpose Read a single parameter name and its value from an input stream.

Declared In `PdiLib.h`

Prototype

```
status_t PdiReadParameter
    (PdiReaderType *ioReader)
```

Parameters → *`ioReader`*
The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

Returns `0`
The parameter and its value were read successfully.

`kPdiReadError`
The parameter and its value could not be successfully read from the input stream.

Comments This function initializes the `parameterName` and `parameter` fields of the `ioReader` object.

This function sets the appropriate bits in the reader's `parameterValues` field if the parameter name is recognized.

If you are reading a property, it's parameters, and its values individually, you should call `PdiReadPropertyName()` before calling `PdiReadParameter()` one or more times to read parameters. Then call `PdiReadPropertyField()` to read the property value fields.

See Also `PdiReaderNew()`

PdiReadProperty Function

Purpose Read the next property and its parameters from the input stream.

Declared In `PdiLib.h`

Prototype
```
status_t PdiReadProperty
   (PdiReaderType *ioReader)
```

Parameters → *`ioReader`*
The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

Returns Returns `errNone` if successful. Returns `kPdiReadError` if an error occurs.

Comments The `PdiReadProperty()` function reads a property name and its parameters, by reading until it encounters the PDI "`:`" separator character.

This function looks each name up in the properties dictionary, and sets the appropriate bit in the `ioReader` object structure to indicate that property-parameter pair has been read. The properties dictionary stores information about properties that are considered well known, as described in "The PDI Library Properties Dictionary" on page 223.

To read a property, you call `PdiReadProperty()`, followed by a call or calls to the `PdiReadPropertyField()` function to read the property value. For more information, see "Reading Properties" on page 229.

See Also `PdiReaderNew()`, `PdiReadPropertyField()`, `PdiReadPropertyName()`, `PdiReadParameter()`

PdiReadPropertyField Function

Purpose Read one field of a property value. The property value can be structured to contain multiple fields that are separated by commas or semicolons.

Declared In `PdiLib.h`

Prototype
```
status_t PdiReadPropertyField
    (PdiReaderType *ioReader, char **bufferPP,
    uint16_t bufferSize, uint16_t readMode)
```

Parameters

→ *ioReader*
The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

↔ *bufferPP*
A pointer to a pointer to the buffer into which the field characters are stored. Set this value to `NULL` to allow the PDI library to manage it.

Note that the PDI library may need to resize the buffer; thus, the value of this parameter might change.

→ *bufferSize*
The size, in bytes, of the input buffer for reading the field.

You can use the `PdiResizableBuffer` constant to specify that the PDI Library can automatically resize the buffer as required.

If you do not specify the `PdiResizableBuffer` value, then the PDI library assumes that `buffer` cannot be moved, and that its size is fixed.

→ *readMode*
The format of the fields in the property value. Use one of the Property Value Format Constants.

Returns

0
The field was read successfully.

`kPdiNoMoreFieldsError`
There are no more fields to read because the entire value has already been read.

`kPdiMoreCharsError`
The buffer is not large enough to store the entire field.

Comments The value returned in the `buffer` is terminated with the "`\0`" character.

If the field is an empty string, the buffer is erased from memory, and the value of `buffer` is set to `NULL`.

If you specify `kPdiResizableBuffer` for the value of the `bufferSize` parameter, and the buffer needs more space, `PdiReadPropertyField()` resizes the buffer for you, which may cause the value of `buffer` to be modified.

This function initializes the `propertyValue` and `fieldNum` fields of the `ioReader` object.

To read a property, you usually call the `PdiReadProperty()` function, followed by a call or calls to `PdiReadPropertyField()` to read the property value. For more information, see "Reading Properties" on page 229.

See Also `PdiReaderNew()`, `PdiReadProperty()`, `PdiReadPropertyName()`, `PdiReadParameter()`

PdiReadPropertyName Function

Purpose Read a property name from an input stream. Use this function when you want to parse and process each parameter individually.

Declared In `PdiLib.h`

Prototype

```
status_t PdiReadPropertyName
    (PdiReaderType *ioReader)
```

Parameters

→ *`ioReader`*
The PDI reader object, which was created by a previous call to the `PdiReaderNew()` function.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments The `PdiReadProperty()` function reads a property name only, reading until it encounters the PDI "`:`" separator character, or until it encounters the first parameter "`,`" separator character.

To then read parameters, call `PdiReadParameter()` one or more times. And to read property value fields, call `PdiReadPropertyField()`.

This function initializes the `property` and `propertyName` fields of the `ioReader` object.

See Also `PdiReaderNew()`, `PdiReadProperty()`

PdiSetCharset Function

Purpose Force the character set of the next property value that is written by the specified PDI writer.

Declared In `PdiLib.h`

Prototype

```
status_t PdiSetCharset (PdiWriterType *ioWriter,
    CharEncodingType charset)
```

Parameters

→ *ioWriter*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *charset*
The character set to use for the property value. This must be a `CharEncodingType` value that is supported by the device as a valid value for the *dstEncoding* parameter passed to `TxtConvertEncoding()`. The valid values depend on the version and locale of the device’s ROM.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments This function tells `ioWriter` to use the specified `charSet` for the next property value that it writes, rather than computing a character set for that value. We strongly suggest, however, not to use this function and to let the PDI library compute and set the character set.

You can determine the current character setting by examining the `charset` field of your PDI writer object.

See Also `PdiSetEncoding()`

PdiSetEncoding Function

Purpose Force the encoding of the current property value.

Declared In `PdiLib.h`

Prototype
```
status_t PdiSetEncoding (PdiWriterType *ioWriter,
   uint16_t encoding)
```

Parameters

→ *`ioReader`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`encoding`*
The encoding to apply to the property value. This must be one of the Encoding Type Constants.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments This function changes the encoding for the property value to the specified `encoding` value

You can determine the current encoding setting by examining the `encoding` field of your PDI writer object.

See Also `PdiSetCharset()`

PdiWriteBeginObject Function

Purpose Writes a vObject begin tag to an output stream.

Declared In `PdiLib.h`

Prototype
```
status_t PdiWriteBeginObject
   (PdiWriterType *ioWriter,
   uint16_t objectNameID)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`objectNameID`*
The object name ID. This must be one of the Property Name Constants that begins an object, including the following:

`kPdiPRN_BEGIN_VCAL`

`kPdiPRN_BEGIN_VCAL`

`kPdiPRN_BEGIN_VCARD`

`kPdiPRN_BEGIN_VEVENT`

`kPdiPRN_BEGIN_VFREEBUSY`

`kPdiPRN_BEGIN_VJOURNAL`

`kPdiPRN_BEGIN_VTIMEZONE`

`kPdiPRN_BEGIN_VTODO`

Returns Returns `errNone` if successful, and an error code if not successful.

Comments Call this function to begin writing a vObject to the output stream. It writes a begin tag such as "`BEGIN:VCARD`" to the output stream.

See Also `PdiWriteEndObject()`, `PdiWriteProperty()`

PdiWriteEndObject Macro

Purpose Writes a vObject end tag to an output stream.

Declared In `PdiLib.h`

Prototype `#define PdiWriteEndObject PdiWriteBeginObject`

Parameters → *`ioWriter`*

The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`objectNameID`*

The object name ID. This must be one of the Property Name Constants that ends an object, including the following:

`kPdiPRN_END_VCAL`

`kPdiPRN_END_VCAL`

`kPdiPRN_END_VCARD`

`kPdiPRN_END_VEVENT`

`kPdiPRN_END_VFREEBUSY`

`kPdiPRN_END_VJOURNAL`

`kPdiPRN_END_VTIMEZONE`

`kPdiPRN_END_VTODO`

Returns Returns `errNone` if successful, and an error code if not successful.

Comments This macro is defined as `PdiWriteBeginObject()`. The only difference is that you should pass one of the properties that ends an object in the `objectNameID` parameter.

Use this macro to finish writing a vObject to the output stream. It writes an end tag such as "`END:VCARD`" to the output stream.

See Also `PdiWriteBeginObject()`, `PdiWriteProperty()`

PdiWriteParameter Function

Purpose Write a parameter, and optionally its name, to an output stream.

Declared In `PdiLib.h`

Prototype

```
status_t PdiWriteParameter
   (PdiWriterType *ioWriter, uint16_t parameter,
   Boolean parameterName)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`parameter`*
The ID of the parameter. This must be one of the Parameter Value Constants.

→ *`parameterName`*
If this is `true`, the parameter name, followed by the "=" symbol, followed by the parameter value is written to the output stream.

If this is `false`, only the parameter value is written to the output stream.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments Use this function to write a parameter to the output stream. To write a property, you usually call the `PdiWriteProperty()` function, followed by calls to `PdiWriteParameter()` to write any parameters, followed by a call to the `PdiWritePropertyValue()` function to write the property value. For more information, see "Writing Properties" on page 235.

You can use the `parameterName` argument to specify that you want the parameter name written as well as the parameter value.

For example, the following table shows what is written if the value of `parameter` is `kPdiPAV_TYPE_HOME`.

Value of parameterName	Data written to output stream
`true`	`TYPE=HOME`
`false`	`HOME`

See Also `PdiWriteProperty()`, `PdiWritePropertyValue()`, `PdiWritePropertyFields()`, `PdiWritePropertyStr()`, `PdiWriteParameterStr()`

PdiWriteParameterStr Function

Purpose Write a parameter name and the parameter value to an output stream.

Declared In `PdiLib.h`

Prototype

```
status_t PdiWriteParameterStr
    (PdiWriterType *ioWriter,
    const char *parameterName,
    const char *parameterValue)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`parameterName`*
The name of the parameter. If the value of this is the empty string or `NULL`, only the parameter value is written.

→ *`parameterValue`*
The parameter value string.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments This function writes the parameter name, followed by the "=" symbol, followed by the parameter value, to the output stream. If `parameterName` is `NULL`, or if its value is the empty string, just the parameter value is written.

This function is similar to the `PdiWriteParameter()` function. The difference is that `PdiWriteParameterStr()` takes the name and value of the parameter as strings, while `PdiWriteParameter()` takes them as ID constants.

See Also `PdiWriteProperty()`, `PdiWritePropertyValue()`, `PdiWritePropertyFields()`, `PdiWritePropertyStr()`, `PdiWriteParameter()`

PdiWriteProperty Function

Purpose Writes a property name to an output stream.

Declared In `PdiLib.h`

Prototype

```
status_t PdiWriteProperty
   (PdiWriterType *ioWriter,
   uint16_t propertyNameID)
```

Parameters

→ *ioWriter*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *propertyNameID*
The property name to write. This must be one of the Property Name Constants.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments To write a property, you usually call `PdiWriteProperty()`, followed by calls to the `PdiWriteParameter()` function to write any parameters, followed by a call to the `PdiWritePropertyValue()` function to write the property value. For more information, see "Writing Properties" on page 235.

See Also `PdiWritePropertyValue()`, `PdiWritePropertyFields()`, `PdiWritePropertyStr()`, `PdiWriteParameter()`

PdiWritePropertyBinaryValue Function

Purpose Write a binary property value to an output stream.

Declared In `PdiLib.h`

Prototype

```
status_t PdiWritePropertyBinaryValue
    (PdiWriterType *ioWriter, const char *buffer,
    uint16_t size, uint16_t options)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`buffer`*
A buffer that contains the binary data.

→ *`size`*
The number of bytes of data to write from the buffer.

→ *`options`*
The data type. This must be a combination of one or more of the Value Type Constants.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments Use this function to write a binary data property value, such as a sound or an image.

This function encodes the data when it is written. The character set that gets applied to the data is not computed by this function; however, you can call the `PdiSetCharset()` function to set the character set.

See Also `PdiWriteProperty()`, `PdiWritePropertyFields()`, `PdiWritePropertyValue()`

PdiWritePropertyFields Function

Purpose Write a structured property value with multiple fields to an output stream.

Declared In `PdiLib.h`

Prototype
```
status_t PdiWritePropertyFields
   (PdiWriterType *ioWriter, char *fields[],
   uint16_t fieldNumber, uint16_t options)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`fields`*
An array of strings, each of which is a field of the property value. Individual fields can be `NULL`.

→ *`fieldNumber`*
The number of fields in the `Fields` array.

→ *`options`*
The data type. This must be a combination of one or more of the Value Type Constants.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments Use this function to write a property value that contains multiple fields.

See Also `PdiWritePropertyValue()`, `PdiWritePropertyBinaryValue()`, `PdiReadPropertyField()`

PdiWritePropertyStr Function

Purpose Writes the name of a property to the output stream, and specifies the property value's structure for subsequent write operations.

Declared In `PdiLib.h`

Prototype
```
status_t PdiWritePropertyStr
   (PdiWriterType *ioWriter,
   const char *propertyName, uint8_t writeMode,
   uint8_t requiredFields)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`propertyName`*
The name of the property to write.

→ *`writeMode`*
The format of the fields in the property value. Use one of the following Property Value Format Constants:

`kPdiNoFields`

`kPdiCommaFields`

`kPdiSemicolonFields`

→ *`requiredFields`*
The number of required fields for the value. This is usually set to `1`.

Returns Returns `errNone` if successful, and an error code if not successful.

Comments Use this function when you are writing a property that is not in the dictionary, or when you are writing a property that uses value formatting that differs from the standard formatting stored in the dictionary for the property name.

This function writes the property name to the output stream, and then establishes the structure of the property's value, including the number of required fields and the separator character for those fields. After calling this function, the next call to the `PdiWritePropertyValue()` or `PdiWritePropertyFields()` functions correctly writes the property value.

See Also `PdiWriteProperty()`, `PdiWritePropertyValue()`, `PdiWritePropertyFields()`, `PdiWriteParameter()`

PdiWritePropertyValue Function

Purpose Write a string to the output stream as the entire value of a property.

Declared In `PdiLib.h`

Prototype

```
status_t PdiWritePropertyValue
   (PdiWriterType *ioWriter, char *buffer,
   uint16_t options)
```

Parameters

→ *`ioWriter`*
The PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

→ *`buffer`*
The input buffer that contains the string to be written.

→ *`options`*
The data type. This must be a combination of one or more of the Value Type Constants.

Returns Returns `errNone` if successful, and an error code if not successful.

See Also `PdiWriteProperty()`, `PdiWritePropertyFields()`, `PdiWriteParameter()`, `PdiWritePropertyBinaryValue()`

PdiWriterDelete Function

Purpose Delete a PDI output stream object.

Declared In `PdiLib.h`

Prototype

```
void PdiWriterDelete (PdiWriterType **ioWriter)
```

Parameters

↔ *`ioWriter`*
A pointer to the PDI writer object, which was created by a previous call to the `PdiWriterNew()` function.

Returns Returns nothing.

Comments This function frees the memory that was allocated for the writer object. The `ioWriter` parameter is set to `NULL`.

See Also `PdiWriterNew()`

PdiWriterNew Function

Purpose Initializes a new PDI writer object for use with the specified library number.

Declared In `PdiLib.h`

Prototype
```
PdiWriterType *PdiWriterNew
   (UDAWriterType *output, uint16_t version)
```

Parameters

→ *`output`*
The Unified Data Access (UDA) output object associated with the writer.

→ *`version`*
Options to control the behavior of the writer. You can use a combination of the Reader and Writer Options Constants.

Returns Returns a pointer to the new PDI writer object. Returns `NULL` if the reader cannot be created.

Comments The `media` pointer is copied into a field in the writer object; thus, you do not need to retain your copy.

See Also `PdiWriterDelete()`, `PdiReaderNew()`

8

Unified Data Access Manager Reference

This chapter provides reference material for the Unified Data Access (UDA) Manager, which provides a mechanism for abstracting read and write access to different kinds of source and destination media, including memory and the Exchange Manager.

The Personal Data Interchange (PDI) reader and writer objects use UDA objects, and you must create UDA objects to use the PDI functions.

This chapter discusses the following topics:

- UDA Manager Data Structures
- UDA Manager Constants
- UDA Manager Functions
- UDA Object Creation Functions

The header file `UDAMgr.h` declares the Unified Data Access Manager API.

You use the UDA Manager in conjunction with the PDI library. For more information about the PDI library, see Chapter 7, "Personal Data Interchange Reference," on page 247.

Chapter 6, "Personal Data Interchange," on page 217, provides examples of using the UDA functions with the PDI library.

UDA Manager Data Structures

UDABufferSize Typedef

Purpose The `UDABufferSize` type is a simple typedef that defines the size of buffers used with UDA read functions.

Prototype
```
typedef uint16_t UDABufferSize;
```

UDAObjectType Struct

Purpose The `UDAObjectType` is the base class for all UDA objects, and defines the common properties of all of the objects.

Prototype
```
typedef struct UDAObjectTag {
   uint16_t optionFlags;
   uint16_t padding;
   UDADeleteFunction deleteF;
   UDAControlFunction controlF;
} UDAObjectType;
```

Fields

`optionFlags`
: Options for the object. This is a combination of values described in Object Option Flags.

`padding`
: Padding; not used.

`deleteF`
: The delete function associated with this UDA object.

`controlF`
: The control function associated with this UDA object.

UDAFilterType Struct

Purpose The `UDAFilterType` represents UDA Filters, which take input from a UDA Reader or UDA Filter, perform some encoding or decoding operation, and output the data to a memory buffer.

Prototype
```
typedef struct UDAFilterTag {
   uint16_t optionFlags;
   uint16_t padding;
   UDADeleteFunction deleteF;
   UDAControlFunction controlF;
   UDAReadFunction readF;
   UDAReaderType* upperReader;
} UDAFilterType;
```

Fields

`optionFlags`
: Options for the object. This is a combination of values described in Object Option Flags.

`padding`
: Padding; not used.

`deleteF`
: The delete function associated with this UDA object.

`controlF`
: The control function associated with this UDA object.

`readF`
: The read function associated with this UDA object.

`upperReader`
: The `UDAReaderType` or `UDAFilterType` object that reads the data that this object outputs.

UDAReaderType Struct

Purpose The `UDAReaderType` represents UDA Readers, which read input from a medium.

Prototype

```
typedef struct UDAReaderTag {
   uint16_t optionFlags;
   uint16_t padding;
   UDADeleteFunction deleteF;
   UDAControlFunction controlF;
   UDAReadFunction readF;
} UDAReaderType;
```

Fields

`optionFlags`
: Options for the object. This is a combination of values described in Object Option Flags.

`padding`
: Padding; not used.

`deleteF`
: The delete function associated with this UDA object.

`controlF`
: The control function associated with this UDA object.

`readF`
: The read function associated with this UDA object.

UDAWriterType Struct

Purpose The `UDAWriterType` represents UDA Writers, which take data from a UDA Reader or UDA Filter and write the data to an output medium.

Prototype

```
typedef struct UDAWriterTag {
   uint16_t optionFlags;
   uint16_t padding;
   UDADeleteFunction deleteF;
   UDAControlFunction controlF;
   UDAWriteFunction initiateWriteF;
   UDAFlushFunction flushF;
   UDAReaderType* upperReader;
} UDAWriterType;
```

Fields

`optionFlags`
: Options for the object. This is a combination of values described in Object Option Flags.

`padding`
: Padding; not used.

`deleteF`
: The delete function associated with this UDA object.

`controlF`
: The control function associated with this UDA object.

`initiateWriteF`
: The write function associated with this UDA object.

`flushF`
: The flush function associated with this UDA object.

`upperReader`
: The `UDAReaderType` object that reads the data that this object writes.

UDA Manager Constants

This section describes the constants used with the UDA Manager, which include the following constant types:

- Control Flags
- Error Constants
- Object Option Flags
- Miscellaneous Constants

Control Flags

Use the control flag constants to control UDA objects with the `UDAControl()` macro.

Constant	Value	Description
`kUDAReinitialize`	1	Used with the `UDAControl()` macro to reinitialize the UDA object.

Error Constants

At the time of this writing, there is only one error constant associated with the UDA object API.

Constant	Description
`udaErrControl`	Returned by the `UDAControl()` macro when the control parameter is not valid for the UDA object.

Object Option Flags

You use the object option flag constants to determine information about the internal state of UDA objects. Note that the `UDAEndOfReader()` and `UDAMoreData()` macros provide you with a convenient means of accessing this same information.

Constant	Value	Description
`kUDAEndOfReader`	1	Indicates that the UDA reader has reached the end of its data.
`kUDAMoreData`	2	Indicates that the UDA reader needs more space in the read buffer to do its work.

Miscellaneous Constants

Constant	Value	Description
`kUDAZeroTerminatedBuffer`	`0xFFFF`	Indicates that the buffer is a null-terminated string. Use this value when creating or reinitializing a `UDAMemoryReader` object.

UDA Manager Functions

UDAControl Function

Purpose Applies controls to a UDA object.

Declared In `UDAMgr.h`

Prototype
```
status_t UDAControl (UDAObjectType *ioObject,
   uint16_t parameter, ...)
```

Parameters

→ *`ioObject`*
A pointer to the `UDAObjectType` object that you want to control. This can be a `UDAReaderType`, a `UDAFilterType`, or a `UDAWriterType` object.

→ *`parameter`*
The control action that you want applied to the object.

→ `...`
Additional parameters, as required for the control and object type.

Returns Returns `errNone` if no error, or `udaErrorClass` if the control parameter is not valid for the `ioObject`.

Comments The `UDAControl()` function applies a control action to a UDA object. You may need to supply additional parameters, depending on the object type and control parameter values.

The only control action defined is `kUDAReinitialize`. You can use it as shown in Table 8.1.

Table 8.1 UDAControl actions

Object Type	Usage	Action
`UDAExchangeReaderType`	`UDAControl(myExgRdr, kUDAReinitialize)`	Does nothing
`UDAExchangeWriterType`	`UDAControl(myExgWtr, kUDAReinitialize)`	Does nothing
`UDAMemoryReaderType`	`UDAControl(myMemRdr, kUDAReinitialize, bufferP, bufferSize)`	Reinstalls a new buffer for the memory reader. See `UDAMemoryReaderNew()` for more information about the parameters.

UDADelete Macro

Purpose Deletes a UDA object.

Declared In `UDAMgr.h`

Prototype `#define UDADelete (`*`ioObject`*`)`

Parameters → *`ioObject`*
A pointer to the `UDAObjectType` object that you want to delete. This can be a `UDAReaderType`, a `UDAFilterType`, or a `UDAWriterType` object.

Returns Returns nothing.

Comments The `ioObject` pointer is not valid after this macro completes.

UDAEndOfReader Macro

Purpose Tests if the end of the reader has been reached.

Declared In `UDAMgr.h`

Prototype `#define UDAEndOfReader (`*`ioReader`*`)`

Parameters → *`ioReader`*
A pointer to a `UDAReaderType` object.

Returns Returns `true` if the end of the reader referenced by `ioReader` has been reached, and `false` if not.

Comments The end of the reader has been reached.

UDAFilterJoin Macro

Purpose Joins a filter with a reader.

Declared In `UDAMgr.h`

Prototype `#define UDAFilterJoin (`*`ioFilter`*`, `*`newReader`*`)`

Parameters → *`ioFilter`*
A pointer to a `UDAFilterType` object.

→ *`newReader`*
A pointer to the `UDAReaderType` object with which you want the filter joined.

Returns Returns nothing.

Comments Each `UDAFilterType` object receives its data from the `UDAReaderType` object to which it is joined; this relationship is established when you create the filter object. You can use this macro to change the reader with which the filter is joined. Upon completion, the filter referenced by `ioFilter` is joined with the reader referenced by `newReader`.

UDAInitiateWrite Macro

Purpose Causes the `UDAWriterType` object to read data and then write that data to output.

Declared In `UDAMgr.h`

Prototype `#define UDAInitiateWrite (`*`ioWriter`*`)`

Parameters → *`ioWriter`*
A pointer to a `UDAWriterType` object.

Returns Returns `errNone` if successful, and an error code if not.

Comments When you use this macro, the `ioWriter` reads data from the reader to which it is joined. It reads data until the reader is empty, and then writes the data to the output medium.

UDAMoreData Macro

Purpose Tests if there is more data available to read, but not enough room in the buffer to read it in.

Declared In `UDAMgr.h`

Prototype `#define UDAMoreData (`*`ioReader`*`)`

Parameters → *`ioReader`*
A pointer to a `UDAReaderType` object.

Returns Returns `true` if there is more data available for the reader and `false` if there is no more data available.

Comments You can use this macro with `UDAReaderType` objects to determine if there is more data waiting to read. This can happen when the reader's buffer is full.

UDARead Macro

Purpose Uses the specified `UDAReaderType` object to read data from the input source and place that data into the specified buffer.

Declared In `UDAMgr.h`

Prototype
```
#define UDARead (ioReader, bufferToFillP,
   bufferSizeInBytes, error)
```

Parameters

→ *`ioReader`*
A pointer to a `UDAReaderType` object that performs the read.

→ *`bufferToFillP`*
A pointer to the buffer into which data is placed.

→ *`bufferSizeInBytes`*
The size of the buffer, in bytes.

→ *`error`*
A pointer to a `status_t` value that represents the result of the read operation; if the operation is successful, the value is set to `errNone`.

Returns Returns the number of bytes that were actually read. This value can be less than or equal to the value of `bufferSizeInBytes`.

Comments The reader reads from the input source associated with the reader object and places the data into the specified buffer, reading no more than `bufferSizeInBytes` bytes of data.

UDAWriterFlush Macro

Purpose Flushes the buffer used by the `UDAWriterType` object.

Declared In `UDAMgr.h`

Prototype
```
#define UDAWriterFlush (ioWriter)
```

Parameters

→ *`ioWriter`*
A pointer to a `UDAWriterType` object.

Returns Returns `errNone` if successful, and an error code if not.

Comments You can use this macro to flush any data remaining in the buffer of the writer object referenced by `ioWriter`. This causes any data in the buffer to be sent to the output medium.

UDAWriterJoin Macro

Purpose Joins a writer object to a different reader object.

Declared In `UDAMgr.h`

Prototype `#define UDAWriterJoin (`*`ioWriter`*`, `*`newReader`*`)`

Parameters

→ *`ioWriter`*
A pointer to a `UDAWriterType` object.

→ *`newReader`*
A pointer to the `UDAReaderType` object with which you want the writer joined.

Returns Returns nothing.

Comments Each `UDAWriterType` object receives its data from the `UDAReaderType` object to which it is joined; this relationship is established when you create the writer object. You can use this macro to change the reader with which the writer is joined. Upon completion, the writer referenced by `ioWriter` is joined with the reader referenced by `newReader`.

UDA Object Creation Functions

UDAExchangeReaderNew Function

Purpose Creates a new `UDAReaderType` object that you can use to read data from an Exchange Manager socket.

Declared In `UDAMgr.h`

Prototype

```
UDAReaderType* UDAExchangeReaderNew
   (ExgSocketType *socket)
```

Parameters

→ *`ExgSocketType`*
A pointer to an `ExgSocketType` structure that describes the connection.

Returns Returns a pointer to the newly created `UDAReaderType` object, or `NULL` if the reader could not be created.

Comments Use this function to create a UDA Reader object that takes input from an Exchange Manager socket.

See Also `ExgSocketType`

UDAExchangeWriterNew Function

Purpose Creates a new `UDAWriterType` object that you can use to write data to an Exchange Manager socket.

Declared In `UDAMgr.h`

Prototype
```
UDAWriterType* UDAExchangeWriterNew
    (ExgSocketType *socket,
    UDABufferSize bufferSize)
```

Parameters

→ *`ExgSocketType`*
A pointer to an `ExgSocketType` structure that describes the connection.

→ *`bufferSize`*
The size, in bytes, of the buffer for the new writer object.

Returns Returns a pointer to the newly created UDA Writer, or `NULL` if the writer could not be created.

Comments Use this function to create a UDA Writer object that sends output to an Exchange Manager socket.

See Also `ExgSocketType`

UDAMemoryReaderNew Function

Purpose Creates a new `UDAReaderType` object that you can use to read data from a memory buffer.

Declared In `UDAMgr.h`

Prototype
```
UDAReaderType* UDAMemoryReaderNew
    (const uint8_t *bufferP,
    UDABufferSize bufferSizeInBytes)
```

Parameters

→ *`bufferP`*
A pointer to a buffer in memory from which data is read.

→ *`bufferSize`*
The size of the buffer, in bytes. If this value is equal to `kUDAZeroTerminatedBuffer`, `bufferP` must point to a null-terminated string buffer.

Returns Returns a pointer to the newly created `UDAReaderType` object, or `NULL` if the reader could not be created.

Comments Use this function to create a reader that takes input from memory.

Index