iAnywhere®
A SYBASE COMPANY

# MobiLink Server-Initiated Synchronization User’s Guide

Part number: DC20056-01-0902-01
Last modified: October 2004

# Contents

# About This Manual

| | |
|---|---|
| Subject | This manual describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database. |
| Audience | This manual is for MobiLink users who wish to use this advanced feature. |
| Before you begin | ☞ For more information about MobiLink, see "Introducing MobiLink Synchronization" [*MobiLink Administration Guide,* page 3]. |

# SQL Anywhere Studio documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

**The SQL Anywhere Studio documentation**

The SQL Anywhere Studio documentation is available in a variety of forms: in an online form that combines all books in one large help file; as separate PDF files for each book; and as printed books that you can purchase. The documentation consists of the following books:

- **Introducing SQL Anywhere Studio** This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.
- **What's New in SQL Anywhere Studio** This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.
- **Adaptive Server Anywhere Database Administration Guide** This book covers material related to running, managing, and configuring databases and database servers.
- **Adaptive Server Anywhere SQL User's Guide** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **Adaptive Server Anywhere SQL Reference Manual** This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.
- **Adaptive Server Anywhere Programming Guide** This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools. It also describes the Adaptive Server Anywhere ADO.NET data provider.
- **Adaptive Server Anywhere SNMP Extension Agent User's Guide** This book describes how to configure the Adaptive Server Anywhere SNMP Extension Agent for use with SNMP management applications to manage Adaptive Server Anywhere databases.
- **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.

- **SQL Anywhere Studio Security Guide** This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.

- **MobiLink Administration Guide** This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.

- **MobiLink Clients** This book describes how to set up and synchronize Adaptive Server Anywhere and UltraLite remote databases.

- **MobiLink Server-Initiated Synchronization User's Guide** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization from the consolidated database.

- **MobiLink Tutorials** This book provides several tutorials that walk you through how to set up and run MobiLink applications.

- **QAnywhere User's Guide** This manual describes MobiLink QAnywhere, a messaging platform that enables the development and deployment of messaging applications for mobile and wireless clients, as well as traditional desktop and laptop clients.

- **iAnywhere Solutions ODBC Drivers** This book describes how to set up ODBC drivers to access consolidated databases other than Adaptive Server Anywhere from the MobiLink synchronization server and from Adaptive Server Anywhere remote data access.

- **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.

- **SQL Anywhere Studio Help** This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.

- **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.

- **UltraLite Interface Guides** A separate book is provided for each UltraLite programming interface. Some of these interfaces are provided as UltraLite components for rapid application development, and others are provided as static interfaces for C, C++, and Java development.

In addition to this documentation set, PowerDesigner and InfoMaker include their own online documentation.

Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere Studio documentation, including both the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on UNIX operating systems, see the HTML documentation under your SQL Anywhere installation.

- **PDF books** The SQL Anywhere books are provided as a set of PDF files, viewable with Adobe Acrobat Reader.

  The PDF books are accessible from the online books, or from the Windows Start menu.

- **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store, at *http://eshop.sybase.com/eshop/documentation*.

# Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

## Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in upper case, like the words ALTER TABLE in the following example:

  **ALTER TABLE** [ *owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  **ALTER TABLE** [ *owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  **ADD** *column-definition* [ *column-constraint*, . . . ]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  **RELEASE SAVEPOINT** [ *savepoint-name* ]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [ **ASC** | **DESC** ]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

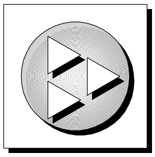
  [ **QUOTES** { **ON** | **OFF** } ]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

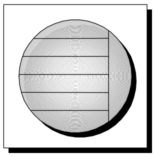
**Graphic icons**

The following icons are used in this documentation.

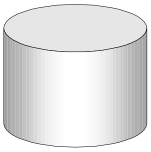
♦ A client application.



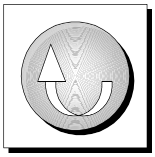
♦ A database server, such as Sybase Adaptive Server Anywhere.



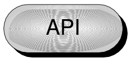
♦ A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



♦ Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink Synchronization Server and the SQL Remote Message Agent.



♦ A programming interface.



## Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at *http://www.ianywhere.com/developer/*.

If you have questions or need help, you can post messages to the iAnywhere Solutions newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere Studio. You can find this information by typing **dbeng9 -v** at a command prompt.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- sybase.public.sqlanywhere.qanywhere
- ianywhere.public.sqlanywhere.qanywhere

**Newsgroup disclaimer**

iAnywhere Solutions has no obligation to provide solutions, information or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.

iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can e-mail comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to e-mails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

CHAPTER 1

# Introducing Server-Initiated Synchronization

About this chapter

This chapter provides an overview of server-initiated synchronization.

Contents

## Introduction to server-initiated synchronization

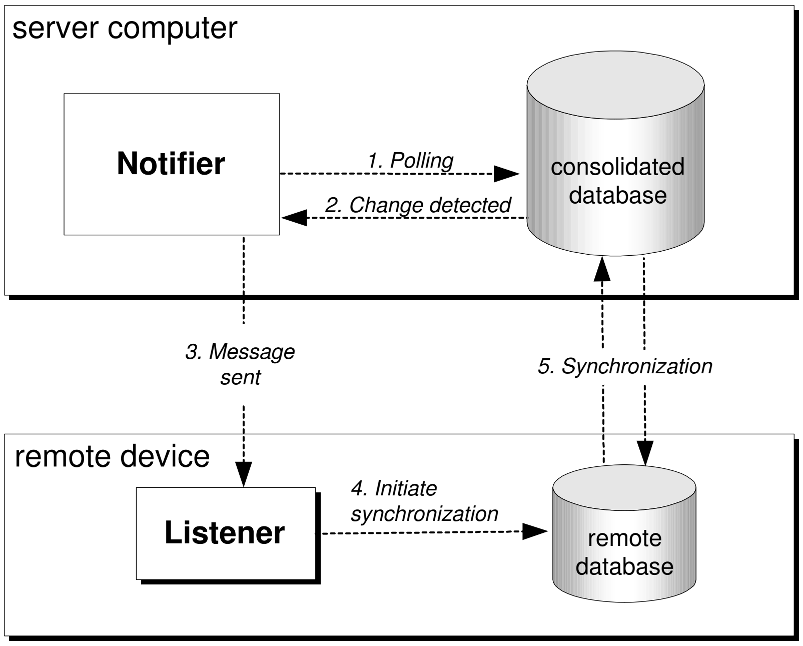
Server-initiated synchronization allows you to initiate MobiLink synchronization from the consolidated database. This means you can push data updates to remote databases, as well as cause remote databases to upload data to the consolidated database. This MobiLink component provides programmable options for determining what changes in the consolidated database initiate synchronization, how remotes are chosen to receive push messages, and how the remotes respond.

Example

For example, a fleet of truck drivers uses mobile databases to determine routes and delivery points. A driver synchronizes a report of a traffic disruption. A component called the Notifier detects the change in the consolidated database and automatically sends a message to the remote device of every driver whose route is affected, which causes the drivers' remote databases to synchronize so that the drivers will use an alternate route.

The notification process

In the following illustration, the Notifier polls a consolidated database and detects a change that it has been configured to look for. In this scenario, the Notifier sends a message to a single remote device, resulting in the remote database being updated via synchronization.



Following are the steps that occur in this example:

1. Using a query based on business logic, the Notifier polls the consolidated database to detect any change that needs to be synchronized to the remote.
2. When a change is detected, the Notifier prepares a message to send to the remote device.
3. The Notifier sends a message using UDP or SMTP.
4. The Listener checks the subject, content, and sender of the message against a filter.
5. If the message matches the filter, the Listener runs a program that has been associated with the filter. For example, the Listener runs dbmlsync or it launches an UltraLite application.

**Connection-initiated synchronization**

In addition to initiating synchronization on the server, you can also initiate synchronization using internal messages that are generated by the Listener on the remote device. These internal messages indicate a change in connectivity, such as when a device enters Wi-Fi coverage, the user makes a RAS connection, or the user puts the device in the cradle.

☞ For more information, see “Connection-initiated synchronization” on page 32.

## Components of server-initiated synchronization

MobiLink server-initiated synchronization uses the following components:

- **Push requests** cause synchronization to occur. A push request takes the form of some data that you insert into a table on the MobiLink consolidated database, or in some cases data inserted into a temporary table or even just a SQL result set. You can create push requests in any way that you cause data to be inserted into a table. For example, a push request could be created by a database trigger that is activated when a price changes. Any database application can create push requests, including the Notifier.

  ☞ For more information, see "Push requests" on page 10.

- **The Notifier** is a Java program running on the same computer as the MobiLink synchronization server. It polls the consolidated database on a regular basis, looking for push requests. You control how often the Notifier polls the database. You specify business logic that the Notifier uses to gather push requests, including which remote devices should be notified. When the Notifier detects a request, it sends the message associated with the request via SMTP or UDP to a Listener on one or more remote devices. You have the option to send repeatable messages with an expiry time.

  ☞ For more information about Notifiers, see "Notifiers" on page 18.

- **The Listener** is a program that is installed on each remote device. It receives messages from the Notifier and initiates action. The action is usually synchronization, but can be other things. You can configure the Listener to act only on messages from selected sources, or with specific content.

  On Windows or Windows CE, the Listener is an executable program configured by command line options. In order to receive a message, the remote device must be on and the Listener must be started.

  ☞ For more information, see "The Listener" on page 37.

  On the Palm OS, you first create a configuration file by running the Palm Listener Configuration utility on a Windows desktop. You then copy the configuration file to your Palm device and run the Palm Listener.

  ☞ For more information, see "Listeners for Palm Devices" on page 49.

- **Gateways** provide an interface to send messages from the Notifier to the Listener. You can send messages using an SMTP gateway or a UDP gateway. When you use an SMTP gateway, you send an e-mail message that your carrier converts into SMS before the Listener receives it. Most carriers provide an e-mail-to-SMS service.

**Device tracking gateways** provide a way to automatically track remote devices. Using device tracking functionality, you don't have to know the addresses of remote devices. You supply the gateway name of your device tracker gateway (by default, **Default-DeviceTracker**) and the MobiLink user name, and MobiLink routes the message through the appropriate gateway to the appropriate device.

☞ For more information, see "Gateways and carriers" on page 20.

## Supported platforms

In addition to MobiLink requirements, the computer must have JRE 1.4.1 or higher to use the Notifier.

☞ For more information about MobiLink requirements, see "SQL Anywhere Studio Supported Platforms" [*Introducing SQL Anywhere Studio*, page 95].

The Listener is not supported on Windows 95 or Windows NT 4.

If you are targeting Palm remotes, you must use the Palm Listener Configuration utility on a Windows desktop device to create a configuration file.

- ♦ **SMS messages** can be transmitted through an SMTP gateway and go through an e-mail-to-SMS conversion that is provided by wireless carriers. This has been tested on the following platforms:
  - Pocket PC 2002 with Sierra Wireless AirCard 510, 555, 710, or 750
  - Windows 2000 and XP with the Sierra Wireless AirCard 510, 555, 710, or 750
  - Palm 4.1 on the Kyocera 7135 and Palm 5.2 on the Treo 600
- ♦ **UDP messages** have been tested on the following platforms:
  - Pocket PC 2002
  - Windows 2000 and XP

The supported AirCards are supported for the following firmware and drivers. (710 is compatible with 750.)

| AirCard | Firmware version | Driver version |
|---|---|---|
| 510 | R1-3-4 | Not applicable |
| 555 | R1_1_2_10AC_GEN | R1_0_0_9ac_1xRTT |
| 750 | R1_1_2_10AC_GEN | R1_0_7_ac_gprs |
| 750 | R3_1_17ACAP | R1_0_9_ac_gprs |

## Deployment considerations

Following are some issues that you should consider before deploying server-initiated synchronization applications.

Limitations of Listeners when using UDP gateways

- On UDP gateways, the Listener keeps a socket open for listening, and so must be connected to an IP network to be able to listen.
- The IP address on the remote device needs to be reachable from the MobiLink synchronization server.

Limitations of Listeners on CE or PCs

- The current set of supported wireless modems require that the operating system is running, which could result in battery drain. Make sure that you have enough power for your usage pattern.

Palm Listeners can't automatically use device tracking

- On the Palm, device tracking does not work automatically. However, there is a way to enable it.

  ☞ For more information, see "Using device tracking with Listeners that don't support it" on page 25.

# Quick start

To set up server-initiated synchronization, you should perform the following steps. This assumes that MobiLink synchronization is already set up.

## ❖ Overview of setting up server-initiated synchronization

1. Create a table to store push requests on the consolidated database.

   ☞ See “Push requests” on page 10.

2. Set up the Notifier to create and manage push requests.

   ☞ See “Notifiers” on page 18.

3. Set up the Listener to filter and act on Notifier push requests.

   ☞ See “Listeners” on page 28.

4. If you are sending SMS notifications, configure gateways and carriers to identify the interface you are using to send messages. (If you are using UDP, you may be able to skip this step and send messages via the default settings.) To send SMS notifications, you must also specify SMTP listening libraries when you start the Listener.

   ☞ See “Gateways and carriers” on page 20 and “Listening libraries” on page 46.

**Other resources for getting started**

- “Tutorial: Server-Initiated Synchronization” on page 129
- Sample applications are installed to *Samples\MobiLink\SIS_** in your SQL Anywhere Studio installation directory.

CHAPTER 2

# Setting Up Server-Initiated Synchronization

About this chapter

This chapter describes how to set up and use server-initiated synchronization.

Contents

## Push requests

A push request takes the form of some data that you insert into a table on the MobiLink consolidated database, or in some cases data inserted into a temporary table or even just a SQL result set. You can create push requests in any way that you cause data to be inserted into a table.

The Notifier sends a message to a remote database when it detects a push request. The push request specifies the content of the message, along with when, how, and to whom the message should be sent.

### Creating the push request table

A push request is a row in a SQL result set on the consolidated database that contains the following columns in the following order. The first five columns are required and the last two columns are optional. The Notifier uses the request_cursor property to fetch push requests.

In a typical implementation, you add a table to your consolidated database with the following columns. However, push requests can also be stored in temporary tables and across multiple tables.

| Column | Description |
|---|---|
| request id | INTEGER. A unique ID for a push request. |
| gateway | VARCHAR. The gateway on which to send the message. This can be a predefined or user-defined gateway. Predefined gateways are **Default-DeviceTracker**, **Default-SMTP**, and **Default-UDP**. |
| subject | VARCHAR. The subject line of the message. |
| content | VARCHAR. The content of the message. |
| address | VARCHAR. The destination address. The format of the address is gateway-specific. For a DeviceTracker gateway (which might use either SMTP or UDP), it is the MobiLink user name of the Listener database, or other MobiLink user names that you register using dblsn -t+. For an SMTP gateway, it is an email address. For a UDP gateway, it is an IP address or host name, optionally followed by a colon and port number. |

| Column | Description |
|---|---|
| resend interval | VARCHAR. Optional. How often the message should be resent. The default unit is minutes. You can specify **S**, **M**, and **H** for units of seconds, minutes, and hours. You can also combine units, as in `1H 30M 10S`.<br><br>The resend interval is especially useful when the remote device is listening for UDP and the network is unreliable. The Notifier assumes that all attributes associated with a resendable notification request do not change: subsequent updates are ignored after the first poll of the request. The Notifier automatically adjusts the next polling interval if a resendable notification must be sent before the next polling time. You can stop a resendable notification using the request_cursor query or by deleting the request from the request table. The default is to send exactly once, with no resend. Delivery confirmation from the intended Listener may stop a subsequent resend. |
| time to live | VARCHAR. Optional. The time until the resend expires. The default unit is minutes. You can specify **S**, **M**, and **H** for units of seconds, minutes, and hours. You can also combine units, as in `1H 30M 10S`.<br><br>If this value is 0, NULL, or not specified, the default is to send exactly once, with no resend. |

☞ For more information about addressing notifications when you are using device tracking, see "Listener options for device tracking" on page 23.

Example

Following is an Adaptive Server Anywhere CREATE TABLE statement that creates a push request table.

```
create table PushRequest (
    req_id   integer default autoincrement primary key,
    gateway   varchar(128),
    subject   varchar(128),
    content   varchar(128),
    address   varchar(128),
    resend_minute varchar(30),
    minute_to_live varchar(30)
)
```

The following code uses the ml_add_property stored procedure to create a request_cursor property that creates the push request.

```
call ml_add_property( 'SIS', 'Notifier(Simple)', 'request_
         cursor',
  'select req_id,
          gateway,
            subject,
            content,
            address,
            resend_minute,
           minute_to_live
   from PushRequest' );
```

## Creating push requests

You can create push requests in any way that you cause data to be inserted into a table. Following is a list of common ways to create push requests:

♦ Specify SQL synchronization logic in Notifier properties. The most obvious property for creating push requests is the begin_poll property.

  A benefit of creating push requests inside the Notifier is that contention is minimized because only one database connection is used for push requests.

  ☞ For more information, see "begin_poll property" on page 58.

♦ Define a database trigger. For example, create a trigger that detects when a price changes and then inserts push request data into a table of push requests.

  ☞ For information about triggers, see "Introduction to triggers" [*ASA SQL User's Guide,* page 670].

♦ Use MobiLink synchronization logic to create push requests that notify other MobiLink users. For example, create an end_upload script that detects that a specific change has been uploaded and then creates a push request to update other users who should have the same data.

  ☞ For more information, see "end_upload table event" [*MobiLink Administration Guide,* page 404].

♦ Use a database client application that inserts data into a push request table directly.

♦ Manually insert push request data using an Interactive SQL utility.

## Sending push requests

The Notifier sends a set of push requests to remote devices by executing a SQL query that you provide in the request_cursor property.

☞ For more information about querying the consolidated database, see “request_cursor property” on page 64.

## Deleting push requests

You delete push requests to prevent resending old messages. Deleting requests in a timely manner can help minimize the number of messages sent and increase the efficiency of the application.

The most straightforward way to delete push requests is to use the Notifier property request_delete. This property is a SQL statement with a request ID as a parameter. Using this statement, the Notifier deletes requests that have been confirmed as delivered or that have expired.

☞ For more information, see “request_delete property” on page 65.

Built-in delivery confirmation is not available on Palm devices; on all devices, it can be disabled. You can optionally implement your own delivery-confirmation mechanism. For example, your synchronization logic can delete push requests from a request table when a specific synchronization occurs.

## Notifying the Listener with sa_send_udp

Adaptive Server Anywhere databases include a system stored procedure called sa_send_udp that can be used to send UDP notifications to the Listener.

If you use sa_send_udp as a way to notify the Listener, you should append a 1 to your UDP packet. This number is a server-initiated synchronization protocol number. In future versions of MobiLink, new protocol versions may cause the Listener to behave differently.

☞ For more information, see “sa_send_udp system procedure” [*ASA SQL Reference*, page 829].

Example

On a device, start the Listener as follows, where *path* is the location of your Internet Explorer program:

```
dblsn -v -l "message=TheMessage;action=start 'path\iexplore.exe'
        http://www.yahoo.com"
```

On a different device, start an Adaptive Server Anywhere database. Start Interactive SQL, and connect to the database. Execute the following SQL. (Note that the UDP packet has a 1 appended to it.)

```
call sa_send_udp('machine#1_ip_name',5001,'TheMessage1')
```

Internet Explorer will open, showing the Yahoo home page.

To make this example work with one device, use localhost as the first paramter to sa_send_udp.

## Setting properties

Notifiers, gateways, and carriers are configured via properties. These properties can be stored in the ml_property MobiLink system table or in a Notifier properties file.

**Storing properties in the database**

You have two options to configure properties in the MobiLink system table:

♦ Use the Notification folder in the MobiLink plug-in in Sybase Central. This stores property settings in the ml_property table on the consolidated database. You can also right-click the Notification folder in Sybase Central and choose to export settings to a Notifier properties file, or import settings from a Notifier properties file.

  ☞ For more information, click Help on the Sybase Central Notifier dialogs.

♦ Use the ml_add_property stored procedure. This also stores configuration information in the ml_property table on the consolidated database.

  ☞ For more information, see "ml_add_property" [*MobiLink Administration Guide,* page 486].

**Storing properties in a properties file**

Alternatively, you can store options in a Notifier properties file. This is a text file that you can edit with a text editor.

☞ For more information, see "Notifier properties file" on page 16.

**Properties**

For a detailed list of the properties you can set, see

♦ "MobiLink Notification Properties" on page 55

♦ "Device tracker gateway properties" on page 68

♦ "SMTP gateway properties" on page 70

♦ "UDP gateway properties" on page 72

♦ "Carrier properties" on page 74

**Setting properties in more than one place**

If you specify properties in both the ml_properties table and the Notifier properties file, the settings are determined as follows:

1. Server-initiated synchronization properties in the ml_property table in the consolidated database are loaded.

2. If a Notifier properties file is specified with the -notifier option, the settings in this file are loaded on top of the settings from the database.

   If a Notifier properties file is not specified, and if the default configuration file is found (*config.notifier*), the settings in the default file are loaded on top of the settings from the database.

**Changing properties** Properties are read at startup. When you change properties, you must shut down and restart the MobiLink synchronization server for them to take effect.

## Notifier properties file

Properties for Notifiers, gateways, and carriers can be stored in the ml_property MobiLink system table or in the Notifier properties file. For more information, see "Setting properties" on page 15.

The Notifier properties file is a text file. It can have any name. The easiest way to create this file is to alter the template, *%asany9%\samples\MobiLink\template.notifier*.

You can export the properties from the ml_property table into your Notifier properties file. To do this, connect to the MobiLink plug-in in Sybase Central, right-click the Notification folder, and choose Export Settings. The exported file may be copied to a different location and used to easily configure a Notifier there.

You can have several Notifier property files. To identify the properties file you want to use, specify the name and location when you start dbmlsrv9 with the -notifier option. Following is a partial dbmlsrv9 command line:

```
dbmlsrv9 ... -notifier "c:\samples\CarDealer.notifier"
```

☞ For information about how properties are read if you do specify a properties file at the command line, see "Setting properties in more than one place" on page 15.

A Notifier properties file can configure and start multiple Notifiers and multiple gateways. You provide a name for each Notifier and gateway that you want to define.

Notifier properties are normally entered on one line, but you can use the backslash ( \) as a line continuation character.

The backslash is also an escape character. You can use the following escape sequences in your property settings:

| Escape sequence | Description |
|---|---|
| \b | \u0008: backspace, BS |
| \t | \u0009: horizontal tab, HT |
| \n | \u000a: linefeed, LF |

| Escape sequence | Description |
|---|---|
| \f | \u000c: form feed, FF |
| \r | \u000d: carriage return, CR |
| \” | \u0022: double quote, “ |
| \’ | \u0027: single quote, ‘ |
| \\ | \u005c: backslash, \ |
| \uhhhh | Unicode character (hexadecimal) |
| \xhh | \xhh: ASCII character (hexadecimal) |
| \e | \u001b: Escape, ESC |

## Notifiers

The Notifier runs on the same computer as the MobiLink synchronization server. The Notifier polls the consolidated database on a regular basis, looking for push requests. When it detects a push request, it sends a message to a remote device. It also contains functionality for executing custom SQL scripts, handling delivery confirmation, deleting push requests, and reconnecting after lost database connections. You may use the custom SQL scripts to monitor your data and create push requests.

You can have more than one Notifier running within a single instance of the MobiLink synchronization server. Each Notifier keeps one database connection open all the time.

☞ For an example of multiple Notifiers, see the sample located in the *Samples\MobiLink\SIS_MultipleNotifier* subdirectory of your SQL Anywhere Studio installation.

### Starting the Notifier

You start Notifiers on the dbmlsrv9 command line. To start the Notifier, use the dbmlsrv9 option **-notifier**. Optionally, you can also specify the name of your Notifier properties file, if you have one.

Following is a partial dbmlsrv9 command line:

```
dbmlsrv9 ... -notifier c:\myfirst.notifier
```

For information about how properties are read if you specify a properties file in the command line, see "Setting properties in more than one place" on page 15.

☞ For information about how properties are applied, see "Setting properties" on page 15.

☞ For more information about the -notifier option, see "-notifier option" [*MobiLink Administration Guide,* page 202].

☞ When you use the -notifier option, you start every Notifier that you have enabled. For more information about enabling Notifiers, see "enable property" on page 61.

### Configuring Notifiers

The Notifier allows you to create custom SQL to program the server-initiated synchronization process. You do this by setting properties. For example, you would configure properties in order to perform tasks such as the following:

- Set a polling interval using the poll_every property.
- Create push requests in response to changes in the consolidated database. The begin_poll property is often used in this way.
- Use the request_cursor property to determine what information is sent in a message, to whom, where, and when.

  *Note*: The request_cursor property is the only required property. For more information, see "request_cursor property" on page 64.
- Delete push requests with the request_delete property.

☞ For a complete list of Notifier properties, see "MobiLink Notification Properties" on page 55.

☞ For information about how to set Notifier properties, see "Setting properties" on page 15.

**Notifier property sequence**

The following pseudo-code shows the sequence in which server-initiated synchronization properties are used. Note that except for request_cursor, all of these properties are optional.

```
connect_string
isolation
begin_connection
poll_every
For each poll (
  begin_poll
  shutdown_query
  request_cursor
  request_delete
  end_poll
)
end_connection
```

# Gateways and carriers

**Gateways** are the mechanisms for sending messages. You can define UDP gateways and SMTP gateways. In addition, you can use a device tracker gateway that automatically decides which UDP or SMTP gateway to use.

Use of the device tracker gateway is recommended. If you do not use device tracking, your request_cursor must include a UDP or SMTP gateway name and address, and for each push request, only that gateway will be tried. But if you use device tracking, you only need to supply a MobiLink user name, and there is a possibility of failover if a gateway fails.

☞ For more information, see "Device tracking" on page 22.

If you are using UDP, you may not need to make any changes to the default gateway settings. For SMTP, you need to configure an SMTP gateway and carrier.

You configure **carriers** to store information about the public wireless carriers that you want to use. Carrier information is used to create SMS e-mail addresses from device tracking information that is sent up from Listeners.

## Configuring gateways and carriers

☞ For information about how to set properties for gateways and carriers, see "Setting properties" on page 15.

For a detailed list of gateway and carrier properties, see

- "Device tracker gateway properties" on page 68
- "UDP gateway properties" on page 72
- "SMTP gateway properties" on page 70
- "Carrier properties" on page 74

Gateways

There are three default gateways. They are installed when you run the MobiLink setup scripts for your consolidated database. The default gateways are called:

- Default-DeviceTracker gateway
- Default-UDP gateway
- Default-SMTP gateway

A device tracker gateway can have up to two subordinate gateways: one SMTP and one UDP. The device tracker gateway automatically routes each message to one of its subordinate gateways based on device tracking information sent up from Listeners. For more information, see "Device tracking" on page 22.

Default-UDP and Default-SMTP are preconfigured with some settings that may work out of the box, especially UDP. In most cases, you should use the default gateways. You can customize their configuration, if required.

You should not delete the default gateways or change their names. You can create additional gateways and assign names to them.

Carriers

You only need to configure a carrier if you are using device tracking with an SMTP gateway. Carrier configuration allows you to specify information such as the name of a network provider, their email prefix, network provider id, and so on. This information is necessary for the Notifier to construct e-mail addresses for each public wireless carrier's e-mail-to-SMS service.

To configure a carrier, you can run the Listener on a device that has a modem and service provider working, and inspect the Listener console or log. If your Listener uses the -x option to connect to a running MobiLink synchronization server, you can also find carrier device tracking information in the ml_device_address MobiLink system table.

Once a carrier is configured, it requires no further attention. The carrier can be used to send SMS messages via SMTP to all devices using that public wireless carrier.

☞ For a list of carrier properties, see "Carrier properties" on page 74.

# Device tracking

Device tracking allows you to address a remote database by supplying only its MobiLink user name in a push request. When device tracking is enabled, MobiLink keeps track of how to reach users. For example, when a device's IP address changes, the Listener synchronizes with the consolidated database to update the device tracking information in the MobiLink system table ml_device_address. The device tracker gateway first attempts to use a UDP gateway (if one is assigned to it), and if the delivery fails it then attempts using an SMTP gateway (if one is assigned to it).

Device tracking is especially useful for UDP addresses that may change frequently; the Listener automatically sends the most recent address to the consolidated database when it changes.

In most cases, you should be able to use device tracking. It is recommended that you use it because it simplifies deployment.

Most 9.0.1 or later Listeners support device tracking. If you are using Listeners that don't support device tracking, you can still use a device tracker gateway by providing tracking information yourself.

☞ For more information, see "Using device tracking with Listeners that don't support it" on page 25.

If you do not use device tracking, your request_cursor must include a UDP or SMTP gateway name and address. For each push request, only that gateway will be used, and no other gateway will be attempted.

## Setting up device tracking

### ❖ To set up device tracking

1. Set up a UDP gateway and/or SMTP gateway, if necessary. *Note:* Typically, the UDP gateway is usable without further configuration, and you can skip this step. However, if you want to use e-mail-to-SMS notification, the default SMTP gateway requires configuration.

   ☞ See "Configuring gateways and carriers" on page 20.

2. Your request_cursor script should have the following settings:
   - The gateway name must be the name of a device tracker gateway. The default instance is called Default-DeviceTracker.
   - The address must be a MobiLink user name. By default, it can be the Listener user name. However, you can use the dblsn option -t+ to add the MobiLink user name of the remote database that you are synchronizing, and then directly address that database.

☞ See “request_cursor property” on page 64.

3. Add the Listener name to the MobiLink ml_user system table.

   The default Listener name is *device_name*-**dblsn**, where *device_name* is the name of your device. You can find the device name in your Listener console. Optionally, you can set the device name using the dblsn -e option. You can specify a different Listener name using the dblsn -u option.

   Whether or not you use the default name, you may need to add the *Listener_name* to the ml_user MobiLink system table on your consolidated database. This is because the *Listener_name* is a MobiLink user name. Like other MobiLink user names, it must be unique and it must be added to the ml_user MobiLink system table on your consolidated database.

   ☞ See “Creating MobiLink users” [*MobiLink Clients*, page 10].

4. Start the Listener with the required options.

   ☞ See “Listener options for device tracking” on page 23.

## Listener options for device tracking

The following dblsn options are used for device tracking.

Use -x, -u, and -w to specify how to connect to the MobiLink server. This is required if you are using device tracking so that the remote device can update the consolidated database if the address changes. These are also required if you wish to send delivery confirmations to the consolidated database.

The -t+ option is recommended. With it, you can register the MobiLink user name of your remote database and use it when you address notifications instead of addressing the MobiLink user name of the Listener database. You only need to do this once.

♦ **-t+ *ml_user***   Use this option to register the MobiLink user name of your remote database so that you can directly address that user name, instead of addressing the Listener database.

  This mapping is retained on the server (in the ml_listening table) once tracking information is uploaded successfully, so you only need to register a MobiLink user name once unless you change the MobiLink user name or location. However, using -t+ multiple times is not harmful.

♦ **-t- *ml_user_alias***   To disable a MobiLink user name created with -t+, use **-t-**.

- **-u *Listener_name*** Use -u to create a MobiLink user name for the Listener. The -u option is optional because there is a default *Listener_name*, which is *device_name*-**dblsn**, where *device_name* is the name of your device. You can find the device name in your Listener console. Optionally, you can set the device name using the -e option.

  Whether or not you use the default name, you may need to add the *Listener_name* to the ml_user MobiLink system table on your consolidated database. This is because the *Listener_name* is a MobiLink user name. Like other MobiLink user names, it must be unique and it must be in the ml_user MobiLink system table on your consolidated database.

  ☞ See "Creating MobiLink users" [*MobiLink Clients*, page 10].

- **-w *password*** This option sets the password for the Listener name.

- **-x *connection-parameters*** Use -x to specify how to connect to the MobiLink synchronization server. This is required if you are using device tracking because it lets the remote device update the consolidated database if the address changes. This option is also required if you wish to send delivery confirmations to the consolidated database.

- **-y** This option updates the password for the Listener name.

☞ For more information about Listener options, see "The Listener utility" on page 38.

Example

The following command starts the Listener with device tracking.

```
dblsn -x tcpip(host=MLSERVER_MACHINE) -t+ user1 -u remoteuser1
```

## Stopping device tracking

It might be useful to stop device tracking in situations such as the following:

- Your device listens only on UDP on a static IP address.

- Your device listens only on UDP and has dynamic IP with low latency DNS update, so you can use a static IP name to address your device directly.

To stop device tracking when you want to continue using delivery confirmation, use the dblsn option -g.

For more information about dblsn options, see "The Listener utility" on page 38.

## Using device tracking with Listeners that don’t support it

You cannot use the completely automatic form of device tracking if any of your Listeners have the following characteristics:

- are prior to Adaptive Server Anywhere 9.0.1 or are Palm Listeners

  ☞ For information about how to set up device tracking in these situations, see “Manually setting up device tracking” on page 25.

- are listening on UDP, and remote IP addresses are unreachable from the MobiLink server machine

  ☞ For information about how to deal with this situation, see “Unreachable addresses” on page 27.

**Manually setting up device tracking**

Several stored procedures are provided to help you manually set up device tracking for 9.0.0 Listeners or Palm Listeners. These stored procedures manipulate the MobiLink system tables ml_device, ml_device_address, and ml_listening on the consolidated database. With manual device tracking, you can address recipients by MobiLink user name—without providing network address information—but the information cannot be automatically updated by MobiLink if it changes: you must change it yourself.

This method is especially useful for SMTP gateways because e-mail addresses don’t tend to change. For UDP gateways, it is more difficult to rely on static entries if your IP address changes every time you reconnect. You may get around this problem by addressing by host name instead of IP address, but in that case slow updates to DNS server tables can cause misdirected messages. You can also deal with changing IP addresses by setting up the following stored procedures to update the MobiLink system tables programmatically.

### ❖ To manually set up device tracking

1. For each remote device, add a device record to the ml_device MobiLink system table. For example,

```
call ml_set_device(
   'myFirstTreo180',
   'MobiLink Listeners for Treo 180 - 9.0.1',
   '1',
   'not used',
   'y',
   'manually entered by administrator' );
```

   The first parameter, myFirstTreo180, is a user-defined unique device name. The second parameter contains optional remarks about the Listener version. The third parameter, set here to 1, specifies a Listener version;

use **0** for Listeners from SQL Anywhere Studio 9.0.0, **1** for post-9.0.0 Palm Listeners, and **2** for post-9.0.0 Windows Listeners. The fourth parameter specifies optional device information. The fifth parameter is set to **y** here, which specifies that device tracking should be ignored; if this were set to **n**, device tracking would overwrite this record. The final parameter contains optional remarks on the source of this record.

☞ For more information about using ml_set_device, see "ml_set_device" on page 81.

2. For each device that you just added, add an address record to the ml_device_address MobiLink system table. For example,

   ```
   call ml_set_device_address(
      'myFirstTreo180',
      'ROGERS AT&T',
      '3211234567',
      'y',
      'y',
      'manually entered by administrator' );
   ```

   The first parameter, myFristTreo180, is a user-defined unique device name. The second parameter is a network provider ID, and must match a carrier's network_provider_id property (for more information, see "network_provider_id property" on page 75). The third parameter is the phone number of your SMS-capable device. The fourth parameter, set here to **y**, activates this record for sending notifications. The fifth parameter, set here to **y**, specifies that device tracking should be ignored; if this were set to **n**, device tracking could overwrite this record. The final parameter contains optional remarks on the source of this record.

   ☞ For information about how to locate carrier information, see "Device tracking" on page 22.

   ☞ For more information about using ml_set_device_address, see "ml_set_device_address" on page 83.

3. For each remote database, add a recipient record to the ml_listening MobiLink system table for the device that was just added. This maps the device to the MobiLink user name. For example,

   ```
   call ml_set_listening(
      'myULDB',
      'myFirstTreo180',
      'y',
      'y',
      'manually entered by administrator' );
   ```

   The first parameter is a MobiLink user name. The second parameter is a user-defined unique device name. The third parameter, set here to **y**, activates this record for device tracking addressing. The fourth parameter, set here to **y**, specifies that device tracking should be ignored; if this were set to **n**, device tracking could overwrite this record. The final parameter contains optional remarks on the source of this record.

☞ For more information, see "ml_set_listening" on page 85.

## Troubleshooting gateways

This section describes some known problems and solutions connected with communication between remote devices and servers.

### Unreachable addresses

Symptom

The Notifier cannot reach the device with the tracked IP address.

Cause

Some or all devices cannot be addressed directly because they are private relative to the MobiLink server. For example, a remote device is on a private sub-network and its address is internal to that network.

Remedy

Try one of the following:

- ♦ If the IP address is assigned by a public wireless carrier or ISP, you may be able to upgrade your carrier plan so that you can obtain public IP addresses instead of private ones.
- ♦ If you are using Wi-Fi, the IP security policy in your organization may stop your device from being reachable. Contact your IT department for assistance.
- ♦ Use an SMS gateway.

  If the device's IP address is never reachable, you may want to stop device tracking on the Listener with the -g option. If you are using delivery confirmation, the first attempt to connect will be via UDP, and the lack of confirmation will prevent further UDP attempts.

### Tracked address is not correct

Symptom

Device tracking is not picking the best IP address for a device.

Cause

There may be a problem with the routing table on the device.

Remedy

Try one of the following:

- ♦ Fix the routing table.
- ♦ Use the ml_set_device_address stored procedure to ignore tracking for the device and set the address parameter to the correct address. Be sure to set the fourth parameter to **y**. In addition, use -g for the problematic Listener.

  For more information, see "ml_set_device_address" on page 83.

## Listeners

The Listener runs on remote devices. It receives messages from the Notifier and processes them into actions based on message handlers that you create. A typical message handler contains filters, actions, and options.

For example, for the following Listener command line, the Listener will only start dbmlsync if it receives a message with the subject **FullSync**:

```
dblsn -l "subject='FullSync';action='run dbmlsync.exe ...'"
```

Following are some of the actions that you can invoke. Typically, the desired action is synchronization initiated via either dbmlsync or an UltraLite application.

- ♦ Start a process.
- ♦ Run a process until it completes.
- ♦ Post a window message to a process that is already running.
- ♦ Perform text-based communication with local or remote applications via TCP/IP with optional confirmation.

Actions can be parameterized with variables derived from the message. This provides extra flexibility in implementing dynamic options.

Normally, you only need to start up one Listener on a device. One Listener can listen on multiple channels and it can serve multiple MobiLink users on the same device. A running Listener always listens on UDP (except for Palm Listeners).

Listeners can also synchronize device tracking information back to the consolidated database. For more information, see "Device tracking" on page 22.

See also

☞ For Listener syntax and options, see "The Listener utility" on page 38.

☞ For information about Palm devices, see "Listeners for Palm Devices" on page 49.

☞ For dbmlsync options, see "Adaptive Server Anywhere Client Synchronization Parameters" [*MobiLink Clients*, page 95].

☞ For more information about message handlers, see "Message handlers" on page 29

☞ Instead of typing dblsn options at a command prompt, it is often convenient to store them in a text file. For more information, see "Storing Listener options" on page 33.

Example

The following command starts the Listener utility. It must be typed on one line.

```
dblsn -v2 -m -ot dblsn.log -x "host=localhost"
 -l "subject=sync;action='start dbmlsync.exe
    -c eng=rem1;uid=dba;pwd=sql -ot dbmlsyncOut.txt -k';"
```

The options used in this example are:

| Option | Description |
|---|---|
| -v2 | Set verbosity to level 2 (log Listener DLL messages and action tracing). |
| -m | Log notification messages. |
| -ot | Truncate the log file and send output to it. In this case, the output file is dblsn.log. |
| -x | Specify a way to connect to the MobiLink synchronization server. This is required for device tracking and delivery confirmation. In this simple example, the only protocol options that are specified are "host=localhost". For a complete list of protocol options, see "-x option" [*MobiLink Clients,* page 151]. |
| -l | Specify a message handler. In this case the filter is that a message must contain the subject **sync**, and the action is to start dbmlsync. Three dbmlsync command line options are also provided: -c specifies a connection string to the MobiLink synchronization server for the synchronization; -ot names an output log file; and -k shuts down dbmlsync when the synchronization is complete. |

## Message handlers

Using the dblsn command line, you create **message handlers** to tell the Listener which messages to filter and what actions should result from each accepted message.

☞ For more information about dblsn, see "The Listener" on page 37.

### Message interpretation

Messages arrive as a single piece of text with the following structure:

```
message control_information
```

The *control_information* is for internal use only and is removed prior to message handling. The Listener substitutes non-printable characters with tildes, and then interprets the *message* portion with the following pattern:

```
message = sender subj-open subject subj-close content

subj-open = ( | [ | { | < | ' | "
```

The *subj-open* character is determined by the first possible character found by scanning from left to right. The value of *subj-open* determines the value of *subj-close*. The possible values of *subj-close* are ), ], }, >, ‘ and “.

The location of the first *subj-close* character marks the end of the *subject* and the beginning of the *content*.

The *sender* is empty when the message begins with a *subj-open*. In that case, the *sender* of the message is determined in a delivery path-dependent way. For example, messages going through UDP gateways arrive as `[`*`subject`*`]`*`content`*, and the *sender* is the IP address. SMTP gateways send an e-mail message that is converted by an e-mail to SMS service into a format that varies between different public wireless carriers.

☞ For more information about the Listener, see “The Listener” on page 37.

## Using subject and content filters

Use the filters **subject** and/or **content** to filter messages by subject and/or content as specified in your push requests. When you use these filters, the Listener automatically adjusts the filter to match the format received by the carrier. For example, you may want to filter a message with the subject Sync and the content Orders. You do not have to worry that in UDP, this would appear as `[Sync]Orders`, and on one e-mail to SMS conversion service, it would be `Bob@mail.com[Sync]Orders`.

Your subject cannot contain the closing character that is used to enclose the subject. In the previous example, UDP encloses the subject `Sync` in square brackets. This means that you cannot use a closing square bracket in subjects that might be received over UDP. For SMTP messages, your carrier determines the character used to enclose the subject. This might be one of ), ], }, >, ‘ or “.

**Note:**
For best results, only use alphanumeric characters in your subject when creating push requests.

The Listener trims leading and trailing spaces, as well as leading and trailing tilde (~) characters, from the sender name, subject, and content. Non-printable characters such as the new line character are converted to tildes by the Listener before filtering.

☞ For more information about the Listener, see “The Listener” on page 37.

## Using the filters message, message_start, and sender

The recommended filters are called **subject** and **content**. However, there are three other types of filter that you may also want to use.

The Listener translates non-printable characters in a message to a tilde (~) so if there are non-printable characters, the filter must also use tildes.

- **message** compares the entire message to text you specify. To match, this filter must also be the exact same length as the message. You can specify only one message per message handler.

  The format of messages is carrier-dependent, and you must account for this if you use the **message**, **message_start**, or **sender** filters. For example, you may want to match a message from a sender named Bob@mail.com with the subject Help and the message Me. In UDP, this would appear as `[Help]Me`. On Bell Mobility's e-mail to SMS conversion service, it would be `Bob@mail.com[Help]Me`. On Fido's e-mail to SMS conversion service, it would arrive as `Bob@mail.com\n(Help)\nMe`, but would be translated by the Listener to `Bob@mail.com~(Help)~Me`. You must test with your carrier to determine the appropriate format, using the dblsn options -v and -m.

- **message_start** compares a portion of the message (from the beginning) to text that you specify. When you specify message_start, the Listener creates the action variables $message_start and $message_end. For more information, see "Action variables" on page 44. There is a maximum of one message_start per message handler.

- **sender** is the sender of the message. You can only specify one sender per message handler. For UDP gateways, the sender is the IP address of the host of the gateway. For SMS e-mail, the sender is the e-mail address embedded in the beginning of the message if the SMS format is compatible with server-initiated synchronization. Otherwise, the sender information is not available.

Multiple message handlers may be required

Subject and content are the recommended filters when messages arrive in a compatible format. However, if your message format is incompatible, you need to use the message, message_start, and/or sender filters. In that case, if the delivery path can vary (sometimes through UDP and sometimes through SMTP), then you need multiple handlers with different filters.

For example, if you sent a message of the form `sub, content` through a UDP gateway, it would arrive as `[sub]content`. But if you sent it through an SMTP gateway, it might be rendered as `mySender@mySite.com \n sub \n content`. To catch a message that has the subject `sub`, you would need two message handlers, with the following filters:

```
-l "subject='sub';action=..."
```

```
-l "message_start='mySender@mySite.com ~ sub ~ ';action=..."
```

☞ For more information about the Listener, see "The Listener" on page 37.

## Connection-initiated synchronization

In addition to initiating synchronization from the server, on Windows devices you can also initiate synchronization when connectivity changes. This is possible because the Windows Listener generates an internal message with the content _IP_CHANGED_ whenever there is a change in connectivity, and it generates an internal message with the content _BEST_IP_CHANGED_ whenever there is a new "best" IP connection.

The internal messages _IP_CHANGED_ and _BEST_IP_CHANGED_ are generated only on Windows devices, including Windows CE.

**Identifying a change in the optimum path to a MobiLink server**

An IP connection is considered to be "best" if it is the best connection to use when connecting to the MobiLink synchronization server that is specified with the dblsn -x option. Although the "best" designation is defined by the path to the MobiLink synchronization server, in practice it tends to indicate the best IP connection to use in general.

To make use of a change in the best IP connection, use the keyword **_BEST_IP_CHANGED_** in your message filter. A MobiLink server is required as a destination for the network to determine which route is optimal, so you must also specify connection parameters for a MobiLink server using the -x option. The message filter should be of the form:

```
-l "message='_BEST_IP_CHANGED_';action=..."
```

The $best_ip action variable is very useful with the _BEST_IP_CHANGED_ filter. The value of $best_ip is the local IP address that represents the best IP connection. If there is no IP connection, $best_ip has the value 0.0.0.0.

You can only use _BEST_IP_CHANGED_ when the Listener is run on a separate machine from the MobiLink synchronization server.

In the following example, the _BEST_IP_CHANGED_ filter is used to initiate a synchronization when the best IP connection changes. If the connection is lost, an error is generated.

```
dblsn -x http(host=mlserver.company.com)
    -v2 -m -i 3 -ot dblsn.log
    -l "message=_BEST_IP_CHANGED_;
       action='start dbmlsync.exe -ra -c
         eng=remote;uid=dba;pwd=sql
               -n test_pub'"
```

Identifying any change in connectivity

To make use of a change in IP connectivity on your remote device, use the keyword **_IP_CHANGED_** in your message filter. _IP_CHANGED_ only indicates that there has been a change in IP connectivity. The message filter should be of the form:

```
-l "message='_IP_CHANGED_';action=..."
```

The following example shows a message handler that can be used in the dblsn command line. The filter captures messages that contain the content _IP_CHANGED_. The action makes use of the action variables $adapters and $network_names. If the connection is lost, an error is generated.

```
-l "message=_IP_CHANGED_;
    action='socket port=12345;
           sendText=IP changed: $adapters|$network_names;
       recvText=beeperAck;
       timeout=5';
    continue=yes;"
```

See also

- "The Listener" on page 37
- "Action variables" on page 44

### Multi-channel listening

To listen on multiple media, you can start the Listener with the -d option. A library for UDP listening is always loaded by default, but there are several others that you can load. For more information, see "The Listener utility" on page 38 and "Listening libraries" on page 46.

☞ For more information about the Listener, see "The Listener" on page 37.

## Storing Listener options

A convenient way to configure the Listener is to store the command line options in a text file and access it with the @ symbol. For example, store the settings in *mydblsn.txt* and start the Listener by typing

```
dblsn @mydblsn.txt
```

The path to the parameters file must be fully qualified.

☞ For more information about configuration files, see "Using configuration files” [*ASA Database Administration Guide*, page 495].

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

☞ See “Hiding the contents of files using the dbfhide command-line utility” [*ASA Database Administration Guide*, page 524].

You can also store command line options in an environment variable, and call it in the dblsn command line by typing @ and the environment variable name; for example, **dblsn @dblsnoptions**. If you have both a filename and an environment variable with the same name, the environment variable is used.

Default parameters file dblsn.txt

If you type dblsn without any parameters, dblsn will use *dblsn.txt* as the default argument file. This feature is particularly useful for CE devices.

Following is a sample parameters file.

```
#---- SIS_SimpleListener\dblsn.txt -----------------------------
         ---------------
#
# This is the default argument file for dblsn.exe
#

#----------------------------------------------------------------
         ---------------
# Device name
#
-e device1

#----------------------------------------------------------------
         ---------------
# MobiLink connection parameters
#
-x host=localhost
```

```
#---------------------------------------------------------------
         ---------------
# Verbosity level 2
#
-v2

#---------------------------------------------------------------
         ---------------
# Show notification messages in console and log
#
-m

#---------------------------------------------------------------
         ---------------
# Polling interval of 1 seconds
#
-i 1

#---------------------------------------------------------------
         ---------------
# Truncate, then write output to dblsn.log
#
-ot dblsn.log

#---------------------------------------------------------------
         ---------------
# First message handler
#   - No filter, so it applies to all messages
#   - Try to send the message to the beeper utility
#   - If that fails, start the beeper utility with the message
#   - Message handling continues with the next handler
#
-l "action='socket port=12345;
            sendText=$sender:$message;
            recvText=beeperAck;
            timeout=5';
    altaction='start java.exe Beeper 12345 $sender:$message';
    continue=yes;"
```

```
#-----------------------------------------------------------------
         ---------------
# Second message handler
#   - Only applies to messages with subject equals 'shutdown'
#   - The action is to send "shutdown" to the beeper utility
#   - Message handling continues with the next handler
#
-l "subject='shutdown';
    action='socket port=12345;
            sendText=shutdown;
            recvText=beeperAck;
            timeout=5';
    continue=yes;"

   #--------------------------------------------------------------
         ------------------
# Third handler
#   - Only applies to messages with subject equals 'shutdown'
#   - The action is to shut down the MobiLink Listener
#
-l "subject='shutdown';
    action='DBLSN FULL SHUTDOWN';"
```

CHAPTER 3

# The Listener

## About this chapter

This chapter is a detailed reference of the Listener utility. The Listener utility runs on Windows devices, including Windows CE.

☞ For usage information, see "Listeners" on page 28.

☞ For information about Palm devices, see "Listeners for Palm Devices" on page 49.

## Contents

| **Topic:** | **page** |
|---|---|
| The Listener utility | 38 |

# The Listener utility

The Listener utility, dblsn, configures and starts the Listener on Windows devices, including Windows CE.

☞ This section is a detailed reference of the Listener utility. For usage information, see "Listeners" on page 28.

☞ For information about Palm devices, see "Listeners for Palm Devices" on page 49.

Syntax

**dblsn** [ *options* ] **-l** *message-handler* [ **-l** *message-handler*... ]

*message-handler* :
 [ *filter* **;**... ]*action*
 [ **;continue = yes** ]
 [ **;maydial = no** ]
 [ **;confirm_delivery = no** ]

*filter* :
 [ **subject =** *string* ]
 [ **content =** *string* ]
 [ **message =** *string* | **message_start =** *string* ]
 [ **sender =** *string* ]

*action* :
  **action =** *command*[**;altaction =** *command* ]

*command* :
  **start** *program* [ *program-arguments* ]
  | **run** *program* [ *program-arguments* ]
  | **post** *window-message* **to** { *window-class-name* | *window-title* }
  | *tcpip-socket-action*
  | **DBLSN FULL SHUTDOWN**

*tcpip-socket-action* :
  **socket port=***app-port*
  [ **;host=***app-host* ]
  [ **;sendText=***text1* ]
  [ **;recvText=** *text2* [ **;timeout=***num-sec* ] ]

*window-message* : *string* | *message-id*

Parameters

**Options** The following options can be used to configure the Listener. They are all optional.

| dblsn options | Description |
|---|---|
| @*data* | Reads options from the specified environment variable or configuration file. If both exist, the environment variable is used. See "Storing Listener options" on page 33. |
| **-a** *option* | Specifies a Listener DLL option. If you specify multiple -d options, each -a is for the -d option it follows. To specify multiple options, repeat -a. For example, `-a port=2439 -a ShowSenderPort`. To see options for your dll, type `dblsn -d` *`filename.dll`* `-a ?` or see "Listening libraries" on page 46. |
| **-d** *filename* | Specifies the Listener dll that you want to use. The default dll is *lsn_udp.dll*. For SMTP gateways, there are several dll's that you can specify. For a list, see "Listening libraries" on page 46. You can also create a custom Listener library. See "MobiLink Listener SDKs" on page 87. To enable multi-channel listening, specify multiple dlls by repeating -d. After each -d option, specify the -a and -i options that relate to the dll. For example, `dblsn.exe -d lsn_udp.dll -i 10 -d maac750.dll -i 60` |
| **-e** *device-name* | Specifies the device name. By default, the device name is automatically extracted from the system. If you do not use -e, you must ensure that all devices have unique names. |
| **-f** *string* | Specifies extra information about the device. By default, this information is the operating system version. Using this option will override the default value. |
| **-g** | Stop tracking UDP addresses when -x is used. This is useful when you do not want device tracking but you do want delivery confirmation. |

| dblsn options | Description |
|---|---|
| **-i** *seconds* | Sets the polling interval in seconds for SMTP connections. This is the frequency at which the Listener checks for messages. If you use multiple -d options, each -i setting is for the -d it follows. The default for SMTP is 30 seconds. For UDP connections, the Listener attempts to connect immediately. |
| **-m** | Turns on message logging. The default is off. |
| **-o** *filename* | Logs output to a file. If -o is not used, output is logged to the console window. |
| **-os** *bytes* | Specifies a maximum size for the log file in bytes. The minimum size is 10 000. By default, there is no limit. |
| **-ot** *filename* | Logs output to file, but first truncates the file. |
| **-p** | Allows automatic idle power-off. This option has an effect only on CE devices. Use it to allow the device to shut down when idle. By default, the Listener prevents the device from shutting itself down so that Listening may continue. |
| **-q** | Runs in a minimized window. |
| **-t** {+\|-} *ml_user_-alias* | Register remote databases for notification so that you can address the remote database by name when using device tracking.<br>☞ See “Listener options for device tracking” on page 23. |
| **-u** *Listener_name* | Specifies a unique name for this Listener. This name is used for uploading tracking information and delivery confirmation, and can also be used as a notification address for the DeviceTracker gateway.<br>The *Listener_name* is a MobiLink user name. Like other MobiLink user names, it must be unique and you must add it to the ml_user MobiLink system table on your consolidated database. For more information, see “Creating MobiLink users” [*MobiLink Clients*, page 10].<br>The default Listener name is *device-name*-**dblsn**.<br>See “Listener options for device tracking” on page 23. |

| dblsn options | Description |
|---|---|
| **-v** [ *level* ] | Sets the verbosity level for the dblsn log and console. The *level* can be **0**, **1**, **2**, or **3**:<br>♦ **0** - show no informational messages (the default).<br>♦ **1** - show Listener dll messages and basic action tracing steps.<br>♦ **2** - show level 1 plus detailed action tracing steps.<br>♦ **3** - show level 2 plus polling and listening states.<br>To output notification messages, you must also use -m (see above). |
| **-w** *password* | Specifies a password for the *Listener_name.*<br>See "Listener options for device tracking" on page 23. |
| **-x** {**http**\|**tcpip**} [**(***keyword*=*value*;...**)**] | Specifies the network protocol and protocol options for the MobiLink synchronization server. For a list of protocol options, see "-x option" [*MobiLink Administration Guide*, page 214]. This information is required for the Listener to send device tracking information and delivery confirmation to the consolidated database.<br>See "Listener options for device tracking" on page 23. |
| **-y** *new_password* | Specifies a new MobiLink password for the Listener name. If your authentication system allows remote devices to change their passwords, this option lets them send up the new password.<br>See "Listener options for device tracking" on page 23. |

**Message-handlers** The -l option allows you to specify a message handler, which is a filter-action pair. The filter determines which messages should be handled, and the action is invoked when the filter matches a message.

You can specify multiple instances of -l. Each instance of -l specifies a different message handler for each incoming message. Message handlers are processed in the order they are specified.

You can also specify the following options for message handlers:

♦ **continue=yes** specifies that the Listener should continue after finding the first match. This is useful when you specify multiple -l clauses to cause one message to initiate multiple actions. The default is no.

♦ **maydial=no** specifies that the action cannot dial the modem. This provides information to the Listener to decide whether to release the modem or not before the action. This option is useful when the action or altaction need exclusive access to the modem used by the Listener. The default is yes.

- **confirm_delivery=no** specifies that this handler should not confirm delivery. A message requires confirmation if the gateway used to send it has its confirm_delivery property set to yes. Delivery can only be confirmed if the message requires confirmation and if the handler accepts the message. The default is yes.

  Normally, you do not need to specify this option. By default the first handler that accepts the message will send delivery confirmation, if required. This option can be used when multiple handlers might accept the same message to give you finer control over which handler should confirm the delivery.

**Filters** You specify a filter to compare to an incoming message. If the filter matches, the action you specify is invoked.

The filter is optional. If you do not specify a filter, the action is performed when any message is received. This is useful when debugging or when you want a catch-all message handler as the last message handler.

☞ For information about using the **subject** or **content** filters, see "Using subject and content filters" on page 30.

☞ For information about using the **message**, **message_start**, or **sender** filters, see "Using the filters message, message_start, and sender" on page 31.

**Action and altaction** Each filter is associated with an action and, optionally, an alternative action called the altaction. If a message meets the conditions of the filter, the action is invoked. You must specify an action. If you specify an altaction, the altaction is invoked only if the action fails.

For each action and altaction, there can be one command, and it can be one of **start**, **run**, **post**, **socket**, or **DBLSN FULL SHUTDOWN**.

- **start** spawns a process. When you start a program, the Listener continues listening for more messages.

  When you **start** a program, the Listener doesn't wait for a return code, so it can only tell that the action has failed if it cannot find or start the program.

  The following example starts dbmlsync with some command line options, parts of which are obtained from the message.

```
"action='start dbmlsync.exe @dbmlsync.txt -n
  $content -wc dbmlsync_$content -e sch=INFINITE';"
```

- **run** runs the program and waits for it to finish. The Listener resumes listening after the process is complete.

  When you **run** a program, the Listener determines that the program has failed if the Listener cannot find or start the program or if it returns a non-zero return code.

  The following example runs dbmlsync with some command line options, parts of which are obtained from the message.

  ```
  "action='run dbmlsync.exe @dbmlsync.txt -n $content';"
  ```

- **post** posts a window message to a window class. This is required by dbmlsync when scheduling is on. Post is also used when signaling applications that use Windows messages.

  You can identify the window message by message contents or by the ID of the window message, if one exists.

  You can identify the window class by its name or by the title of the window. If you identify the window class by name, you can use the dbmlsync -wc option to specify the window class name. If you identify the window class by its title, only the title of the top level window can be used to identify the window class.

  If there are non-alphanumeric characters such as spaces or punctuation marks in your window message or window class name, you can put the message or name in single quotes. In that case, to use a single quote in the string, use two single quotes in a row. For example, to post my'message to my'class, use the following syntax:

  ```
  ... -l "action='post my''''message to my''''class':"
  ```

  or

  ```
  ... -l "action='post ''my''''message'' to ''my''''class''':"
  ```

  The following example posts a Windows message registered as dbas_synchronize to a dbmlsync instance registered with the class name dbmlsync_FullSync.

  ```
  "action='post dbas_synchronize to dmblsync_FullSync';"
  ```

  ☞ For more information, see "-wc option" [*MobiLink Clients,* page 151].

- **socket** notifies an application by making a TCP/IP connection. This is especially useful for passing dynamic information to a running application. It is also useful for integrating with Java and Visual Basic applications, because Java and VB don't support custom window messaging, and eVB doesn't support command line parameters. You can connect to a local socket by specifying just a port, or you can connect to a remote socket by specifying the host along with the port. Using sendText, you can send a string. You can optionally verify that the response is as expected with recvText. When you use recvText, you can also specify a timeout to avoid hanging if the case of application or network problems.

When you perform a **socket** action, the Listener determines that the action has failed if it failed to connect, send, or receive expected acknowledgement before timeout.

The following example forwards the string in $sender=$message to a local application that is listening on port 12345, and expects the application to send back “beeperAck” as an acknowledgement within 5 seconds.

```
-l "action='socket port=12345;
   sendText=$sender=$message;
   recvText=beeperAck;
   timeout=5'"
```

- **DBLSN FULL SHUTDOWN** causes the Listener utility to shut down. After shutdown, the Listener stops handling inbound messages and stops synchronizing device tracking information. The remote user must restart the Listener in order to continue with server-initiated synchronization. This feature is mostly useful during testing.

  For example, `action='DBLSN FULL SHUTDOWN'`

You can only specify one action and one altaction in each instance of -l. If you want an action to perform multiple tasks, you can write a cover program or batch file that contains multiple actions, and run it as a single action.

Following is an example of altaction. In this example, $content is the protocol option for connecting to MobiLink. The primary action is to post the dbas_synchronize Windows message to the dbmlsync_FullSync window. The example uses altaction to start (not run) dbmlsync with the window class name dbmlsync_FullSync if the primary action fails. This is the standard way to make the Listener work with dbmlsync scheduling.

```
-l "subject=sync;
    action='post dbas_synchronize to dbmlsync_FullSync';
    altaction='start dbmlsync.exe
                    @dbmlsync.txt
                 -wc dbmlsync_FullSync
                 -e adr=$content;sch=INFINITE'"
```

See also “Listeners” on page 28

## Action variables

The following Listener action variables can be used anywhere in the action or altaction.

An action variable is substituted just before the action or altaction is performed.

Listener action variables start with a dollar sign ($). The escape character is also a dollar sign, so to specify a single dollar sign as plain text, type $$. For example, type $$message_start when you don't want $message_start to be substituted.

| Action variable | Description |
|---|---|
| $subject | The subject of the message. |
| $content | The content of the message. |
| $message | The entire message, including subject, content, and formatting that is specific to the delivery path. |
| $message_start | A portion of the text of the message from the beginning, as specified in -l message_start. This variable is only available if you have specified -l message_start. |
| $message_end | The part of the message that is left over after the part specified in -l message_start is removed. This variable is only available if you have specified -l message_start. |
| $sender | The sender of the message. |
| $type | The meaning of this variable is carrier library dependent. |
| $priority | The meaning of this variable is carrier library dependent. |
| $request_id | The request ID that was specified for the push request. For more information, see "Push requests" on page 10. |
| $year | The meaning of this variable is carrier library dependent. |
| $month | The meaning of this variable is carrier library dependent. Values can be from 1-12. |
| $day | The meaning of this variable is carrier library dependent. Values can be from 1-31. |
| $hour | The meaning of this variable is carrier library dependent. Values can be from 0-23. |
| $minute | The meaning of this variable is carrier library dependent. Values can be from 0-59. |
| $second | The meaning of this variable is carrier library dependent. Values can be from 0-59. |

| Action variable | Description |
|---|---|
| $best_adapter_-mac | The MAC address of the best NIC for reaching the MobiLink server that is specified in the dblsn command line with the -x option. If the best route does not go through a NIC, the value is an empty string. |
| $best_adapter_-name | The adapter name of the best NIC for reaching the MobiLink server that is specified in the dblsn command line with the -x option. If the best route does not go through a NIC, the value is an empty string. |
| $best_ip | The IP address of the best IP interface for reaching the MobiLink server that is specified in the dblsn command line with the -x option. If that server is unreachable, the value is 0.0.0.0. |
| $best_-network_name | The RAS or dialup profile name of the best profile for reaching the MobiLink server that is specified in the dblsn command line with the -x option. If the best route does not go through a RAS/dialup connection, the value is an empty string. |
| $adapters | A list of active network adapter names, each separated by a vertical bar ( \| ). |
| $network_-names | A list of connected RAS entry names, each separated by a vertical bar ( \| ). RAS entry names are sometimes referred to as dial-up entry names or Dial-Up Network (DUN). |

Example

For example, if a message arrives in the form `message_start` *`pub-name`*, you can use the following $message_end action variable to determine which publication to synchronize:

```
-l "message_start=message_start;action='dbmlsync.exe -c ... -n
         $message_end'"
```

## Listening libraries

When you run the Listener, by default the listening library *lsn_udp.dll* is used. If you are using SMTP, you need to specify an SMTP listening library.

You specify the listening library with the dblsn -d option, and specify options for the listening library with the -a option. To enable multi-channel listening, specify multiple dlls by repeating -d. After each -d option, specify the -a and -i options that relate to the dll. For example,

```
dblsn.exe -d lsn_udp.dll -i 10 -d maac750.dll -i 60
```

To specify multiple options, repeat -a. For example,

```
-d maac750.dll -a port=2439 -a ShowSenderPort
```

To see options for your dll, type `dblsn -d` *`filename.dll`* `-a ?`.

☞ You can also create a custom Listener library. For more information, see “MobiLink Listener SDKs” on page 87.

Following is a list of supported listening libraries and their options.

UDP (lsn_udp.dll)

| Option | Description |
|---|---|
| **Port=***port_number* | The default is 5001. |
| **Timeout=***seconds* | This value must be smaller than the polling interval of the UDP listening thread. The default is 0. |
| **ShowSenderPort** | Appends **:***port* to the sender. |
| **HideWSAError-Box** | Suppresses the error box showing errors on socket operations. |
| **CodePage=***number* | On CE, translates multi-byte characters into Unicode based on this code page number. |

SMS for AirCard510 (lsn_swi510.dll)

| Option | Description |
|---|---|
| **MessageStoreSize=***number* | This size affects how the library collapses redundant messages. If the message store is filled, the library stops collapsing identical messages until a message is consumed. The default is 20. |
| **NetworkProviderId=***name* | The matching Carrier(*name*).network_-provider_id. This information is sent up during a device tracking synchronization. This option is needed for device tracking. |
| **PhoneNumber=***number* | A 10-digit telephone number. This information is sent up during a device tracking synchronization. This option is needed for device tracking. |

SMS for AirCard555 (maac555.dll)

| Option | Description |
|---|---|
| **MessageStoreSize=***number* | This size affects how the library collapses redundant messages. If the message store is filled, the library stops collapsing identical messages until a message is consumed. The default is 20. |
| **PreserveMessage** | Specifies that messages should be left in the queue for other SMS applications. The default is for the Listener to consume messages as they are processed. |

SMS for AirCard710 and AirCard750 using firmware R2 (maac750.dll)

| Option | Description |
|---|---|
| **MessageStoreSize=***number* | This size affects how the library collapses redundant messages. If the message store is filled, the library stops collapsing identical messages until a message is consumed. The default is 20. |
| **PreserveMessage** | Specifies that messages should be left in the queue for other SMS applications. The default is for the Listener to consume messages as they are processed. |

SMS for AirCard710 and AirCard750 using firmware R3 (maac750r3.dll)

| Option | Description |
|---|---|
| **MessageStoreSize=***number* | This size affects how the library collapses redundant messages. If the message store is filled, the library stops collapsing identical messages until a message is consumed. The default is 20. |
| **PreserveMessage** | Specifies that messages should be left in the queue for other SMS applications. The default is for the Listener to consume messages as they are processed. |

CHAPTER 4

# Listeners for Palm Devices

About this chapter

This chapter describes how to set up and run server-initiated synchronization on Palm devices. Palm Listeners do not support UDP.

Contents

# Palm Listener utilities

To run server-initiated synchronization on Palm devices, you use two utilities:

- ♦ Palm Listener Configuration utility (dblsncfg)
- ♦ Palm Listener (lsnK7135.prc or lsnT600.prc)

First, run the Palm Listener Configuration utility on a Windows desktop to create a configuration file for the Palm. The configuration file must later be transferred to the Palm device via HotSync.

☞ For an overview of Listeners and message handlers, see "Listeners" on page 28.

## Palm Listener Configuration utility

The Palm Listener Configuration utility, running on a Windows desktop, creates a configuration file for the Palm Listener. For information about the Palm Listener, see "Palm Listener utility" on page 52.

Syntax

**dblsncfg -n** [ *filename* ] **-l** *message-handler* [ **-l** *message-handler*... ]

*message-handler* : [ *filter* **;**...] *action*

*filter* :
[ **subject =** *string* ]
[ **content =** *string* ]
[ **message =** *string* | **message_start =** *string* ]
[ **sender =** *string* ]

*action* : **action=run** *application-name* [ *arguments* ]

Options and parameters

**@data** Reads options from the specified environment variable or configuration file. If both exist, the environment variable is used. See "Storing Listener options" on page 33.

**-n [filename]** The -n option is used to create a configuration file for the Palm Listener. The *filename* should be *lsncfg.pdb*.

**-l message-handler** -l allows you to specify a message handler, which is a filter-action pair. The filter determines which message should be handled, and the action is invoked when the filter matches a message. You can specify multiple instances of -l. Each instance of -l specifies a different message handler.

**Filters** You specify a filter to compare to an incoming message. If the filter matches, the action you specify is invoked.

☞ For information about using the **subject** or **content** filters, see "Using subject and content filters" on page 30.

☞ For information about using the **message**, **message_start**, or **sender** filters, see "Using the filters message, message_start, and sender" on page 31.

The filter is optional. If you do not specify a filter, the action is performed when any message is received.

**Action** The action fully launches the specified application. The syntax is **run** *application-name* [ *arguments* ]. *arguments* is an application-dependent string; it may contain action variables. The PilotMain routine of the target application should take a string as the command block. For more information, see "Action variables" on page 51.

*Note:* When running the Palm Listener Configuration utility on a Windows desktop to generate a configuration file for the Palm, you must specify the **run** action. However, on the Palm device you can delete the run action using the Handler Editor in the Palm Listener. This way you can consume the message without causing an action.

## Action variables

The following action variables can be used in the arguments in the run clause.

An action variable is substituted just before the action is performed.

Listener action variables start with a dollar sign ($). The escape character is also a dollar sign, so to specify a dollar sign as plain text, type $$. For example, type $$message_start when you don't want $message_start to be substituted.

| Action variable | Description |
|---|---|
| $subject | The subject of the message. |
| $object | The object of the message. |
| $message | The full message string. |
| $message_start | A portion of the text of the message from the beginning, as specified in -l message_start. This variable is only available if you have specified -l message_start. |
| $message_end | The part of the message that is left over after the part specified in -l message_start is removed. This variable is only available if you have specified -l message_start. |
| $sender | The sender of the message. |
| $time | This is the current time in seconds since 12:00 AM, January 1, 1904. |

## Palm Listener utility

For Palm applications using server-initiated synchronization, each client must have a Palm Listener installed. The two supported Palm Listeners are for Kyocera 7135 and Treo 600. The Listener files are:

- **lsnK7135.prc** the Listener on Kyocera 7135
- **lsnT600.prc** the Listener on Treo 600

Currently, the two Palm Listeners only read from configuration file *lsncfg.pdb.*

The Palm Listener also allows you to set three options. These options remain until they are explicitly changed or until you perform a reset.

- **Listening** A way to stop the Listener from consuming messages.
- **Enable Actions** This is applicable only when Listening is on. When disabled, no action is invoked.
- **Prompt Before Actions** This is applicable only when actions are enabled. When this option is set, a confirmation dialog pops up before an action is invoked.

The device need not always be on if it turns on automatically when an SMS message is received. Kyocera and Treo devices do not need to be on for the Listener to work.

☞ A Listener SDK is provided that you can use to create support for other Palm devices. For more information, see "Listener SDK for Palm" on page 114.

CHAPTER 5

# MobiLink Notification Properties

About this chapter

This chapter describes the properties that you use to customize Notifiers, gateways, and carriers.

☞ For information about how to set properties, see "Setting properties" on page 15.

Contents

# Common properties

There is one common property, verbosity.

☞ For more information about setting properties, see "Setting properties" on page 15.

## verbosity property

The verbosity setting applies to all Notifiers, gateways, and carriers. You can set the verbosity to the following levels:

| Level | Description |
|---|---|
| 0 | No trace (the default) |
| 1 | Startup, shutdown, and property trace |
| 2 | Display notification messages |
| 3 | Poll-level trace |

See also "Setting properties" on page 15.

Examples Following are the ways you can set verbosity to level 2.

If you are using the MobiLink plug-in in Sybase Central to change the verbosity property, right-click the Notification folder and choose Properties.

If you are configuring properties using the Notifier properties file, include the line:

```
verbosity=2
```

If you are using the stored procedure ml_add_property to change the verbosity level, enter the following:

```
ml_add_property( 'SIS','global','verbosity','2' );
```

## Notifier properties

The following properties can be set in the Notifier properties file. The enable and request_cursor properties are required. All other Notifier properties are optional.

You can have multiple Notifiers running with one MobiLink server. To set up additional Notifiers, copy the properties for one Notifier and provide a different Notifier name and property values.

☞ For more information about the Notifier, see "Notifiers" on page 18.

☞ For more information about setting properties, see "Setting properties" on page 15.

### begin_connection property

This is a SQL statement that runs in a separate transaction after the Notifier connects to the database and before the first poll. For example, this property can be used to create temporary tables or variables.

If the Notifier loses its connection to the consolidated database, it will re-execute this transaction immediately after reconnecting.

You should not use this property to change isolation levels. To control isolation levels, use the isolation property.

See also

- "Setting properties" on page 15
- "isolation property" on page 62

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Open the Logic tab and select begin_connection from the dropdown list. Paste the following code into the box called Execute this SQL Statement.

```
set temporary option blocking = 'off'
```

If you are configuring properties using the Notifier properties file, begin_connection is defined for a Notifier called Car Dealer with the following line. The backslash is a line continuation character.

```
Notifier(Car Dealer).begin_connection = \
   set temporary option blocking = 'off'
```

If you are using the stored procedure ml_add_property to change the begin_connection property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property(
  'SIS',
  'Notifier(Car Dealer)',
  'begin_connection',
  'set temporary option blocking = ''off'''
);
```

## begin_poll property

This is a SQL statement that is executed before each Notifier poll. Typical uses are to detect data change in the database and create push requests that are later fetched with the request_cursor.

The statement is executed in a standalone transaction.

This property is optional. The default is NULL.

See also

"Setting properties" on page 15.

Examples

For example, the following SQL statement inserts rows into a table called PushRequest. Each row in this table represents a message to send to an address. The WHERE clause determines what push requests are inserted into the PushRequest table.

```
INSERT INTO PushRequest
( gateway, mluser, subject, content )
      SELECT 'MyGateway', DISTINCT mluser,
       'sync', stream_param
      FROM MLUserExtra, Dealer
      WHERE
    MLUserExtra.mluser.push_sync_status = "waiting for request"
      AND Dealer.last_modified > MLUserExtra.last_sync_time
```

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Open the Logic tab and select begin_poll from the dropdown list. Paste the code above into the box called Execute this SQL Statement.

If you are configuring properties using the Notifier properties file, include the following line for a Notifier called NotifierA. The backslash is a line continuation character.

```
Notifier(NotifierA).begin_poll = \
INSERT INTO PushRequest \
( gateway, mluser, subject, content ) \
      SELECT 'MyGateway', DISTINCT mluser, \
       'sync', stream_param \
      FROM MLUserExtra, Dealer \
      WHERE \
   MLUserExtra.mluser.push_sync_status = "waiting for request" \
      AND Dealer.last_modified > MLUserExtra.last_sync_time \
);
```

If you are using the stored procedure ml_add_property to change the begin_connection property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'Notifier(Car Dealer)',
  'begin_connection',
  'INSERT INTO PushRequest
     ( gateway, mluser, subject, content )
            SELECT ''MyGateway'', DISTINCT mluser,
         ''sync'', stream_param
            FROM MLUserExtra, mluser_union, Dealer
            WHERE
        MLUserExtra.mluser = mluser_union.name
          AND( push_sync_status = ''waiting for request''
               OR datediff( hour, last_status_change, now() ) >
         12 )
            AND ( mluser_union.publication_name is NULL
               OR mluser_union.publication_name =''FullSync'' )
             AND
               Dealer.last_modified > mluser_union.last_sync_
         time'
);
```

## connect_string property

By default, the Notifier uses ianywhere.ml.script.ServerContext to connect to the consolidated database. This means that it uses the connection string that was specified in the current dbmlsrv9 session's command line.

This is an optional property that can be used to override the default connection behavior. You can use it to connect to any database, including the consolidated database. It may be useful to connect to another database when you want notification logic and data to be separate from your synchronization data. Most deployments will not set this property.

☞ For more information, see "ServerContext interface" [*MobiLink Administration Guide*, page 311].

See also "Setting properties" on page 15.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. The connect string is set on the Connection tab.

If you are configuring properties using the Notifier properties file, configure a Notifier called Simple to use a DSN with the following line. The backslash is a line continuation character.

```
Notifier(Simple).connect_string = dsn=SIS_DB \
   ;uid=user;pwd=myPwd
```

If you are using the stored procedure ml_add_property to change the connect string, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'Notifier(Simple)',
  'connect_string',
  'dsn=SIS_DB;uid=user;pwd=myPwd' );
```

## gui property

This controls whether the Notifier dialog is displayed on the computer where the Notifier is running. This user interface allows users to temporarily change the polling interval, or poll immediately. It can also be used to shut down the Notifier without shutting down the MobiLink synchronization server. (Once stopped, the Notifier can only be restarted by shutting down and restarting the MobiLink synchronization server.)

This property is optional. The default is ON.

See also

"Setting properties" on page 15.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Open the General tab and change Display Control Window When Running.

If you are configuring properties using the Notifier properties file, disable the dialog for a Notifier called Simple with the following line:

```
Notifier(Simple).gui=off
```

If you are using the stored procedure ml_add_property to change this property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS','Notifier(Simple)','gui','off' );
```

## enable property

You can enable or disable existing Notifiers. If you have enabled multiple Notifiers, all are started when you start the MobiLink synchronization server with the -notifier option.

See also

"Setting properties" on page 15.

Examples

If you are using the MobiLink plug-in in Sybase Central, open the Notifiers folder and double-click Add Notifier to add a new Notifier. New Notifiers are automatically enabled. To disable an existing Notifier, right-click the Notifier and choose Properties; open the General tab and clear Enable This Notifier.

If you are configuring properties using the Notifier properties file, a Notifier called NotifierA is enabled with the following line:

```
Notifier(NotifierA).enable=yes
```

If you are using the stored procedure ml_add_property to enable NotifierA, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS','Notifier(NotifierA)','enable','yes' );
```

## end_connection property

This is a SQL statement that runs as a separate transaction just before a Notifier database connection is closed. For example, this property can be used to delete temporary storage such as SQL variables and temporary tables.

The statement is executed in a standalone transaction.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier called Simple and choose Properties. Open the Logic tab and select end_connection from the dropdown list. Paste the following code into the box called Execute this SQL Statement.

```
DELETE FROM NotifierShutdown WHERE name = 'Simple'
```

If you are configuring properties using the Notifier properties file, end_connection is defined for a Notifier called Simple with the following line. The backslash is a line continuation character.

```
Notifier(Simple).end_connection =   \
   DELETE FROM NotifierShutdown WHERE name = 'Simple'
```

If you are using the stored procedure ml_add_property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'Notifier(Simple)',
  'end_connection',
  'DELETE FROM NotifierShutdown WHERE name = ''Simple''');
```

## end_poll property

This is a SQL statement that is executed after each poll. Typical uses are to perform customized cleanup or track polling.

The statement is executed in a standalone transaction.

This property is optional. The default is NULL.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier called Simple and choose Properties. Open the Logic tab and select end_poll from the dropdown list. Paste the following code into the box called Execute this SQL Statement.

```
call reportAliveRequests( )
```

If you are configuring properties using the Notifier properties file, end_poll is defined for a Notifier called Simple with the following line.

```
Notifier(Simple).end_poll = call reportAliveRequests( )
```

If you are using the stored procedure ml_add_property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'Notifier(Simple)',
  'end_poll',
  'call reportAliveRequests( )');
```

## isolation property

Isolation is an optional property that controls the isolation level of the Notifier's database connection. The default value is 1. You can use the following values:

| Value | Isolation level |
|---|---|
| 0 | Read uncommitted |
| 1 | Read committed (the default) |
| 2 | Repeatable read |
| 3 | Serializable |

Description

Be aware of the consequences of setting the isolation level. Higher levels increase contention, and may adversely affect performance. Isolation level 0 allows reads of uncommitted data—data which may eventually be rolled back.

See also

"Setting properties" on page 15.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Isolation levels are set on the Connection tab.

If you are configuring properties using the Notifier properties file, the isolation level for a Notifier called NotifierA is set with the following line:

```
Notifier(NotifierA).isolation=2
```

If you are using the stored procedure ml_add_property to change the isolation level, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS','Notifier(NotifierA)','isolation','2' );
```

## poll_every property

This property specifies the Notifier polling interval. You can specify S, M, and H for units of seconds, minutes. and hours. You can also combine units, as in 1H 30M 10S. If no unit is specified, the interval is in seconds.

If the Notifier loses the database connection, it will recover automatically at the first polling interval after the database becomes available again.

This property is optional. The default is 30 seconds.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Open the Polling tab and choose a polling interval.

If you are configuring properties using the Notifier properties file, a Notifier called Simple is configured to poll every three hours with the following line:

```
Notifier(Simple).poll_every = 3H
```

If you are using the stored procedure ml_add_property to change the polling interval, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS','Notifier(Simple)','poll_every','3H' );
```

## request_cursor property

This property contains SQL used by the Notifier to fetch push requests. Each row is a push request that determines what information is sent in the message, who receives the information, when, and where. You must set this property.

The result set of this statement must contain at least five columns, and can optionally contain two other columns. These columns can have any name, but must be in the following order in the result set:

- ♦ request id
- ♦ gateway
- ♦ subject
- ♦ content
- ♦ address
- ♦ resend interval (optional)
- ♦ time to live (optional)

☞ For more information about these columns, see "Push requests" on page 10.

You might want to include a WHERE clause in your request_cursor to filter out requests that have been satisfied. For example, you can add a column to your push request table to track the time you inserted a request, and then use a WHERE clause to filter out requests that were inserted prior to the last time the user synchronized.

The statement is executed in a standalone transaction.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Open the Logic tab and select request_cursor from the dropdown list. Paste the following code into the box called Execute this SQL Statement.

```
SELECT req_id, gateway, subject, content, address
  FROM PushRequest
```

If you are configuring properties using the Notifier properties file, a request_cursor is defined for a Notifier called Simple with the following line. The backslash is a line continuation character.

```
Notifier(Simple).request_cursor = \
SELECT req_id, gateway, subject, content, address \
  FROM PushRequest
```

If you are using the stored procedure ml_add_property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property(
  'SIS',
  'Notifier(Simple)',
  'request_cursor',
  'SELECT req_id, gateway, subject, content, address
     FROM PushRequest'
);
```

## request_delete property

This is a SQL statement that specifies cleanup operations. The statement takes the request id as its only parameter. The placeholder for a parameter is a question mark (?).

Using the DELETE statement, the Notifier can automatically remove these forms of old request:

♦ **implicitly dropped requests** requests that appeared previously, but did not appear in the current set of requests obtained from the request_cursor.

♦ **confirmed requests** messages confirmed as delivered.

♦ **expired requests** requests that have expired based on their resend attributes and the current time. Requests without resend attributes are considered expired even if they appear in the next request.

The request_delete statement is executed per request ID in a standalone transaction when the need for deletion is detected. It is optional if you have provided another process to do the cleanup.

You can write the request_delete script in such a way to avoid eliminating expired or implicitly dropped requests. For example, the CarDealer sample uses request_delete to set the status field of the PushRequest table to 'processed'.

```
update PushRequest set status='processed' where req_id = ?
```

The sample's begin_poll script uses the last synchronization time to check that a remote device is up-to-date prior to eliminating processed requests.

☞ For more information, see the Car Dealer sample located in the *Samples\MobiLink\SIS_CarDealer* subdirectory of your SQL Anywhere Studio installation.

Examples

In the following examples, a Notifier called Simple is configured to substitute a req_id previously obtained from request_cursor for the question mark (?).

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier called Simple and choose Properties. Open the Logic tab and select request_delete from the dropdown list. Paste the following code into the box called Execute this SQL Statement.

```
DELETE FROM PushRequest WHERE req_id = ?
```

If you are configuring properties using the Notifier properties file, request_delete is defined for a Notifier called Simple with the following line. The backslash is a line continuation character.

```
Notifier(Simple).request_delete = \
   DELETE FROM PushRequest WHERE req_id = ?
```

If you are using the stored procedure ml_add_property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'Notifier(Simple)',
  'request_delete',
  'delete = DELETE FROM PushRequest WHERE req_id = ?');
```

## shutdown_query property

This is a SQL statement that is executed right after begin_poll. The result should contain only the value yes (or 1) or no (or 0). To shut down the Notifier, specify yes or 1. This statement is executed as a standalone transaction.

If you are storing the shutdown state in a table, then you can use the end_connection property to reset the state before the Notifier disconnects.

Examples

If you are using the MobiLink plug-in in Sybase Central, right-click the Notifier and choose Properties. Open the Logic tab and select shutdown_query from the dropdown list. Paste the following code into the box called Execute this SQL Statement.

```
SELECT COUNT(*) FROM NotifierShutdown
  WHERE name='Simple'
```

If you are configuring properties using the Notifier properties file, shutdown_query is defined for a Notifier called Simple with the following line. The backslash is a line continuation character.

```
Notifier(Simple).shutdown_query = \
  SELECT COUNT(*) FROM NotifierShutdown \
  WHERE name='Simple'
```

If you are using the stored procedure ml_add_property, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS','Notifier(Simple)','shutdown_query',
 'SELECT COUNT(*) FROM NotifierShutdown
  WHERE name=''Simple'''

);
```

# Device tracker gateway properties

To use the default device tracker gateway, include the name Default-DeviceTracker in the second column of the result set of the request_cursor.

A device tracker gateway utilizes automatically-tracked IP addresses, phone numbers, and public wireless network provider IDs to deliver messages through either a UDP or SMTP gateway. Your configuration defines which UDP gateway and which SMTP gateway are to be used by your device tracker gateway. You can also control tracking requirements for messages sent through this gateway.

☞ For more information about device tracking, see "Device tracking" on page 22.

☞ For more information about setting properties, see "Setting properties" on page 15.

## confirm_delivery property

Specifies whether the Listener should confirm with the consolidated database that the message was received. To be able to do this, you must start the Listener with the -x option. The default setting for confirm_delivery is yes.

For example,

```
DeviceTracker(Default-DeviceTracker).confirm_delivery = yes
```

Examples

If you are using the MobiLink plug-in in Sybase Central, open the Gateways folder, right-click Default-DeviceTracker, and choose Properties. Open the Delivery tab and select Confirm Delivery.

If you are configuring properties using the Notifier properties file, a DeviceTracker gateway called Default-DeviceTracker has delivery confirmation set with the following line:

```
DeviceTracker(Default-DeviceTracker).confirm_deivery=yes
```

If you are using the stored procedure ml_add_property to set confirm_delivery for Default-DeviceTracker, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'DeviceTracker(Default-DeviceTracker)',
  'confirm_delivery',
  'yes' );
```

## enable property

Specify **enable=yes** to use a device tracker gateway. Specify **enable=no** to disable a device tracker gateway. You can define and use multiple device tracker gateways.

Examples

If you are using the MobiLink plug-in in Sybase Central, open the Carriers folder and double-click Add Gateway. New gateways are automatically enabled.

If you are configuring properties using the Notifier properties file, a DeviceTracker gateway called Default-DeviceTracker is enabled with the following line:

```
DeviceTracker(Default-DeviceTracker).enable=yes
```

If you are using the stored procedure ml_add_property to enable Default-DeviceTracker, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( ’SIS’,
  ’DeviceTracker(Default-DeviceTracker)’,
  ’enable’,
  ’yes’ );
```

## smtp_gateway property

This names an SMTP gateway that the device tracker can use. The gateway must be enabled. A device tracker gateway can only use one SMTP gateway. The default is Default-SMTP.

## udp_gateway property

This identifies a UDP gateway that the device tracker can use. The gateway must be enabled. A device tracker gateway can only use one UDP gateway. The default is Default-UDP.

# SMTP gateway properties

SMTP gateway configuration is required only if you need to send SMS messages via SMTP.

SMTP gateways can be used to send e-mail messages. In particular, they can send SMS messages to SMS listeners via a wireless carrier's e-mail-to-SMS service.

In the following list of properties, the enable and server properties are required. The server and sender properties are often required. The user and password properties may be required, depending on your SMTP server setup. All other SMTP gateway properties are optional.

You can have multiple SMTP gateways. To set up additional SMTP gateways, copy the properties for one gateway and provide a different gateway name and property values.

☞ For more information about gateways, see "Gateways and carriers" on page 20.

☞ For more information about setting properties, see "Setting properties" on page 15.

## confirm_delivery property

Specify yes to confirm delivery. The default is no. This property has an effect only when sending directly through this gateway (not indirectly via a device tracking gateway).

## confirm_timeout property

Specify the amount of time before a confirmation should time out. Specify s, m, or h for seconds, minutes, or hours. If you do not specify s, m, or h, the default is seconds. The default confirmation timeout is 10m.

## enable property

Specify enable=yes to use an SMTP gateway. You can define and use multiple SMTP gateways.

Examples

If you are using the MobiLink plug-in in Sybase Central, open the Gateways folder and double-click Add Gateway. New gateways are automatically enabled.

If you are configuring properties using the Notifier properties file, an SMTP gateway called Gate3 is enabled with the following line:

```
SMTP(Gate3).enable=yes
```

If you are using the stored procedure ml_add_property to enable the gateway Gate3, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS','SMTP(Gate3)','enable','yes' );
```

## listeners_are_900 property

Specify yes if all Listeners are Adaptive Server Anywhere version 9.0.0 clients. Specify no if they are version 9.0.1 or later. The default is no.

## password property

This is the password for your SMTP service. Your SMTP service may not require a password.

## sender property

This is the sender address of the e-mails (SMTP requests). The default is anonymous.

The sender may not be available as an action variable to the Listener if the arriving message format is not compatible with MobiLink's message interpretation.

## server property

This is the IP address or host name of the SMTP server used to send the message to the Listener. The default is **mail**.

## user property

This is the user name for your SMTP service. Your SMTP service may not require a user name.

## UDP gateway properties

UDP gateway configuration is required only if you need to send UDP messages.

The format of the UDP message is [ *subject* ] *content* where *subject* and *content* come from the subject and content columns of the request_cursor Notifier property.

In the following list of properties, only the enable property is required. All other UDP gateway properties are optional.

You can have multiple UDP gateways. To set up additional UDP gateways, copy the properties for one gateway and provide a different gateway name and property values.

☞ For more information about gateways, see "Gateways and carriers" on page 20.

☞ For more information about setting properties, see "Setting properties" on page 15.

### confirm_delivery property

Specify yes to confirm delivery. The default is yes. This property has an effect only when sending directly through this gateway (not indirectly via a device tracking gateway).

### confirm_timeout property

Specify the amount of time before a confirmation should time out. Specify s, m, or h for seconds, minutes, or hours. If you do not specify s, m, or h, the default is seconds. The default confirmation timeout is 1m.

### enable property

Specify **enable=yes** to use a UDP gateway. You can define and use multiple UDP gateways.

Examples

If you are using the MobiLink plug-in in Sybase Central, open the Gateways folder and double-click Add Gateway. New gateways are automatically enabled.

If you are configuring properties using the Notifier properties file, a UDP gateway called Gate3 is enabled with the following line:

```
UDP(Gate3).enable=yes
```

If you are using the stored procedure ml_add_property to enable the gateway Gate3, type the following:

```
ml_add_property( ’SIS’,’UDP(Gate3)’,’enable’,’yes’ );
```

## listeners_are_900 property

Specify yes if all Listeners are Adaptive Server Anywhere version 9.0.0 clients. Specify no if they are version 9.0.1 or later. The default is no.

## listener_port property

This is the port on the remote device where the gateway sends the UDP packet. This property is optional. The default is the default listening port of the UDP Listener (5001).

## sender property

This is the IP address or host name of the sender. This property is optional, and is only useful for multi-homed hosts. The default is localhost.

## sender_port property

This is the port to use for sending the UDP packet. This property is optional; you may need to set it if your firewall restricts outgoing traffic. If not set, your operating system will assign a free port.

# Carrier properties

Carrier properties set up public wireless carrier configuration, which provides carrier-specific information such as how to map automatically-tracked phone numbers and network providers to SMS e-mail addresses.

Carrier information is used when the device tracker gateway needs an SMS e-mail address to be generated from an automatically-tracked device address. Addresses are generated in the following form:

```
email-address =
  sms_email_user_prefixphone-number@sms_email_domain
```

where:

- *sms_email_user_prefix* is the value of the sms_email_user_prefix property
- the phone number comes from the ml_device_address.address column
- *sms_email_domain* is the value of the sms_email_domain property

See also

- "sms_email_domain property" on page 75
- "sms_email_user_prefix property" on page 75
- "ml_device_address" [*MobiLink Administration Guide*, page 507]

☞ For more information about carriers, see "Gateways and carriers" on page 20.

☞ For more information about setting properties, see "Setting properties" on page 15.

## enable property

Specify **enable=yes** to use a Carrier mapping. You can define and use multiple Carrier mappings in one file.

Examples

If you are using the MobiLink plug-in in Sybase Central, open the Carriers folder and double-click Add Carrier Mapping. New carrier mappings are automatically enabled.

If you are configuring properties using the Notifier properties file, a carrier mapping called Bell Mobility 1x is enabled with the following line:

```
Carrier(Bell Mobility 1x).enable=yes
```

If you are using the stored procedure ml_add_property to enable the carrier Bell Mobility 1x, type the following (this assumes an Adaptive Server Anywhere consolidated database):

```
ml_add_property( 'SIS',
  'Carrier(Bell Mobility 1x)',
  'enable',
  'yes' );
```

## network_provider_id property

Specifies the network provider ID.

## sms_email_domain property

Specifies the domain name of the carrier.

Carrier information is used when the device tracker gateway needs an SMS e-mail address to be generated from an automatically-tracked device address. Addresses are generated in the following form:

```
email-address =
  sms_email_user_prefixphone-number@sms_email_domain
```

where:

- *sms_email_user_prefix* is the value of the sms_email_user_prefix property
- the phone number comes from the ml_device_address.address column
- *sms_email_domain* is the value of the sms_email_domain property

See also

- "sms_email_user_prefix property" on page 75
- "ml_device_address" [*MobiLink Administration Guide*, page 507]

## sms_email_user_prefix property

Specifies the prefix used in e-mail addresses.

Carrier information is used when the device tracker gateway needs an SMS e-mail address to be generated from an automatically-tracked device address. Addresses are generated in the following form:

```
email-address =
  sms_email_user_prefixphone-number@sms_email_domain
```

where:

- *sms_email_user_prefix* is the value of the sms_email_user_prefix property
- the phone number comes from the ml_device_address.address column

- *sms_email_domain* is the value of the sms_email_domain property

See also

- “sms_email_domain property” on page 75
- “ml_device_address” [*MobiLink Administration Guide,* page 507]

CHAPTER 6

# Server-Initiated Synchronization Stored Procedures

About this chapter

This chapter describes the stored procedures that are provided for server-initiated synchronization. These stored procedures add and delete rows in MobiLink system tables.

Note: These stored procedures are used for device tracking. If you use remote devices that support automatic device tracking, you do not need to use these stored procedures. If you use remote devices that do not support automatic device tracking, you can configure manual device tracking using these stored procedures.

☞ For more information, see "Device tracking" on page 22 and "Using device tracking with Listeners that don't support it" on page 25.

☞ For more information about MobiLink system tables, see "MobiLink System Tables" [*MobiLink Administration Guide,* page 501].

☞ For information about other MobiLink stored procedures, see "Stored Procedures" [*MobiLink Administration Guide,* page 479].

Contents

## ml_delete_device

Function

Use this stored procedure to delete all information about a remote device when you are manually setting up device tracking.

Parameters

| Item | Parameter | Description |
|---|---|---|
| 1 | device | VARCHAR(255). Device name. |

Description

Example

Delete a device record and all associated records that reference this device record:

```
call ml_delete_device( 'myOldDevice' );
```

## ml_delete_device_address

Function

Use this stored procedure to delete a device address when you are manually setting up device tracking.

Parameters

| Item | Parameter | Description |
|---|---|---|
| 1 | device | VARCHAR(255) |
| 2 | medium | VARCHAR(255) |

Description

☞ For more information, see "Using device tracking with Listeners that don't support it" on page 25.

Example

Delete an address record:

```
call ml_delete_device_address( 'myFirstTreo180', 'ROGERS AT&T'
         );
```

## ml_delete_listening

Function

Use this stored procedure to delete mappings between a MobiLink user and remote devices when you are manually setting up device tracking.

Parameters

| Item | Parameter | Description |
|---|---|---|
| 1 | ml_user | VARCHAR(128) |

Description

☞ For more information, see "Using device tracking with Listeners that don't support it" on page 25.

Example

Delete a recipient record:

```
call ml_delete_listening( 'myULDB' );
```

## ml_set_device

Function

Use this stored procedure to add or alter information about remote devices when you are manually setting up device tracking. It adds or updates a row in the ml_device table.

Parameters

| Item | Parameter | Description |
|---|---|---|
| 1 | device | VARCHAR(255). User-defined unique device name. |
| 2 | listener_version | VARCHAR(128). Optional remarks on listener version. |
| 3 | listener_protocol | INTEGER. Use **0** for version 9.0.0, **1** for post-9.0.0 Palm Listeners, **2** for post-9.0.0 Windows Listeners. |
| 4 | info | VARCHAR(255). Optional device information. |
| 5 | ignore_tracking | CHAR(1). Set to **y** to ignore tracking and stop it from overwriting manually entered data. |
| 6 | source | VARCHAR(255). Optional remarks on the source of this record. |

Description

The stored procedures ml_set_device, ml_set_device_address, and ml_set_listening are used to override automatic device tracking by changing information in the MobiLink system tables ml_device, ml_device_address, and ml_listening. For example, if some of your remote devices are Palm devices you may want to use automatic device tracking but manually insert data for the Palm devices.

☞ For more information, see “Using device tracking with Listeners that don’t support it” on page 25.

See also

- “ml_set_device_address” on page 83
- “ml_set_listening” on page 85
- “ml_device” [*MobiLink Administration Guide,* page 505]
- “ml_device_address” [*MobiLink Administration Guide,* page 507]
- “ml_listening” [*MobiLink Administration Guide,* page 509]

Example

For each device, add a device record:

```
call ml_set_device(
   'myFirstTreo180',
   'MobiLink Listeners for Treo 180 - 9.0.1',
   '1',
   'not used',
   'y',
   'manually entered by administrator' );
```

## ml_set_device_address

Function

Use this stored procedure to add or alter information about remote device addresses when you are manually setting up device tracking. It adds or updates a row in the ml_device_address table.

Parameters

| Item | Parameter | Description |
|---|---|---|
| 1 | device | VARCHAR(255). Existing device name. |
| 2 | medium | VARCHAR(255). Network provider ID (must match a carrier's network_provider_-id property). |
| 3 | address | VARCHAR(255). Phone number of an SMS-capable device. |
| 4 | active | CHAR(1).Set to **y** to activate this record to be used for sending notification. |
| 5 | ignore_tracking | CHAR(1). Set to **y** to ignore tracking and stop it from overwriting manually entered data. |
| 6 | source | VARCHAR(255). Optional remarks on the source of this record. |

Description

The stored procedures ml_set_device, ml_set_device_address, and ml_set_listening are used to override automatic device tracking by changing information in the MobiLink system tables ml_device, ml_device_address, and ml_listening. For example, if some of your remote devices are Palms you may want to use automatic device tracking but manually insert data for the Palm devices.

☞ For more information, see "Using device tracking with Listeners that don't support it" on page 25.

See also

- "ml_set_device" on page 81
- "ml_set_listening" on page 85
- "ml_device" [*MobiLink Administration Guide,* page 505]
- "ml_device_address" [*MobiLink Administration Guide,* page 507]
- "ml_listening" [*MobiLink Administration Guide,* page 509]

Example

For each device, add an address record for a device:

```
call ml_set_device_address(
      'myFirstTreo180',
      'ROGERS AT&T',
      '3211234567',
      'y',
  'y',
      'manually entered by administrator' );
```

## ml_set_listening

Function

Use this stored procedure to add or alter mappings between MobiLink users and remote devices when you are manually setting up device tracking. It adds or updates a row in the ml_listening table.

Parameters

| Item | Parameter | Description |
|---|---|---|
| 1 | ml_user | VARCHAR(128). MobiLink user name. |
| 2 | device | VARCHAR(255). Existing device name. |
| 3 | listening | CHAR(1). Set to **y** to activate this record to be used for DeviceTracker addressing. |
| 5 | ignore_tracking | CHAR(1). Set to **y** to ignore tracking and stop it from overwriting manually entered data. |
| 6 | source | VARCHAR(255). Optional remarks on the source of this record. |

Description

The stored procedures ml_set_device, ml_set_device_address, and ml_set_listening are used to override automatic device tracking by changing information in the MobiLink system tables ml_device, ml_device_address, and ml_listening. For example, if some of your remote devices are Palms you may want to use automatic device tracking but manually insert data for the Palm devices.

☞ For more information, see "Using device tracking with Listeners that don't support it" on page 25.

See also

- "ml_set_device" on page 81
- "ml_set_device_address" on page 83
- "ml_device" [*MobiLink Administration Guide*, page 505]
- "ml_device_address" [*MobiLink Administration Guide*, page 507]
- "ml_listening" [*MobiLink Administration Guide*, page 509]

Example

For each remote database, add a recipient record for a device. This maps the device to the MobiLink user name.

```
call ml_set_listening(
      'myULDB',
      'myFirstTreo180',
      'y',
      'y',
      'manually entered by administrator' );
```

CHAPTER 7

# MobiLink Listener SDKs

About this chapter

This chapter describes the Listener Software Development Kit, which is provided to help you create support for remote devices that are not supported.

Contents

## Introduction

If you want to use remote devices that are not currently supported by MobiLink server-initiated synchronization, you can use the Listener Software Development Kit to create Listeners for those devices. The Listener SDK is a simple program API that is provided to help you extend the Listener utility.

For example, you can use the Listener SDK to create Listeners for new Palm devices or new wireless network adapters. The SDK provides development material for both Windows (32-bit and CE) and Palm operating systems.

☞ For more information about the Listener SDK for Palm, see "Listener SDK for Palm" on page 114.

☞ For more information about the Listener SDK for Windows and Windows CE, see "Listener SDK for Windows" on page 90.

The MobiLink Listener SDK and sample implementations are located in the following files. All are located in the *MobiLink\ListenerSDK* subdirectory of your SQL Anywhere Studio installation.

Windows Listener SDK files

| Windows Files | Description |
|---|---|
| *\Win32andCE\Win32_-VC\lsn.def* | Visual C++ module definition for the Listener library. |
| *\Win32andCE\CE_EVC\lsn.-def* | Embedded Visual C module definition for the Listener library. |
| *\Win32andCE\src\lsn.h* | Win32 and CE Listener library API. |
| *\Win32andCE\src\swi510.c* | Sierra Wireless AirCard 510 implementation. |
| *Win32andCE\src\udp.c* | UDP implementation. |

Palm Listener SDK files

| Palm Files | Description |
|---|---|
| *\Palm\68k\cw\lib\PalmLsn.-lib* | Runtime library for Palm Listeners. This provides a message handling routine, Listener controls, and a handler editor. |
| *\Palm\68k\cw\rsc\* | Contains UI resources for the Palm Listener. |
| *\Palm\src\PalmLsn.h* | Runtime library header and Palm Listener API. |
| *\Palm\src\Kyocera7135.c* | Kyocera 7135 implementation. |
| *\Palm\src\Treo600.c* | Treo 600 implementation. |

# Listener SDK for Windows

You can use the Listener SDK for Windows (32-bit and CE) to create shared Listener libraries for new wireless network adapters or communication protocols.

The library, conforming to a common interface used for retrieving server initiated synchronization messages, includes:

- A class representing Listener message data.
- Enumerations defining return values and version numbers for the Listener SDK.
- Structures storing Listener state.
- Functions controlling Listener message or state information.

☞ For more information about Listener libraries, see "Listening libraries" on page 46.

The Listener utility (dblsn.exe) loads your library using the -d command line option.

☞ For more information about dblsn options, see "The Listener utility" on page 38.

## Data types

The following data types are defined in the Listener SDK for Windows. They map common value types to Listener types used in the SDK interface.

| Name | Description |
|---|---|
| lsn_bool | unsigned char |
| lsn_ulong | unsigned long |
| lsn_long | long |

## Message class

The message class (a_msg) provides an interface to manage Listener messages, including:

- Public fields to store message data.
- Functions for message allocation, comparison, and basic manipulation.

## a_msg public fields

Function

The message class contains fields you can use to represent the sender, the message contents, and optionally the time, type, and priority of a message.

Prototype

```
class a_msg {

// message class functions...
...

// message class fields

   public:
    lsn_long year;
    lsn_long month;    // 1 to 12
    lsn_long day;   // 1 to 31
    lsn_long hour;   // 0 to 23
    lsn_long minute;   // 0 to 59
    lsn_long second;   // 0 to 59
    lsn_long type;
    lsn_long priority;
    TCHAR * sender;
    TCHAR * message;
...

};
```

Parameters

| Name | Description |
|---|---|
| **sender** | A string used for the sender of a message. |
| **message** | A string used for the message contents. |
| **year** | An integer used for the year field of a message. |
| **month** | An integer used for the month field of a message. |
| **day** | An integer used for the day field of a message. |
| **hour** | An integer used for the hour field of a message. |
| **minute** | An integer used for the minute field of a message. |
| **second** | An integer used for the second field of a message. |
| **type** | An integer used for the type of a message. |
| **priority** | An integer used for the priority of a message. |

Remarks

In your implementation, you give meaning to each field listed above.

Use the lsn_info structure to specify if the time, type, and priority fields are applicable. For example, you can set the lsn_info isMsgPriorityApplicable field to FALSE to disable the a_msg priority field.

☞ For more information about the lsn_info structure, see "lsn_info structure" on page 97.

Example

The following example declares an a_msg class instance and determines if the priority field is set to 1.

```
a_msg * msgA;

//...

if( msgA -> priority == 1)
 {
      MessageBox( NULL,
      TEXT("Message Priority is 1."),
        TEXT( "Message from lsn_udp.dll" ), MB_ICONEXCLAMATION
         );
}
```

## a_msg allocateSize function

Function

Returns a new a_msg instance with the specified size for sender and message fields.

Prototype

static a_msg * **allocateSize (**
 long *senderTChars*,
 long *messageTChars*
**)**

Parameters

- **senderTChars** The number of characters used for the sender field.
- **messageTChars** The number of characters used for the message field.

Return value

A new message class instance instance with all fields initialized to zero.

See also

- "Message class" on page 90

Example

The following example uses the allocateSize function to create a new a_msg instance. In this implementation the sender field is 16 characters long and the message field is 200 characters long.

```
#define MAX_SENDER_TCHARS      16
#define MAX_MESSAGE_TCHARS      200

//...

a_msg* msgA;
msgA = a_msg::allocateSize( MAX_SENDER_TCHARS, MAX_MESSAGE_
         TCHARS );
```

## a_msg copy function

| | |
|---|---|
| Function | Overwrites the fields of the calling instance using an a_msg parameter. |
| Prototype | void **copy (** a_msg * *source-msg* **)** |
| Parameters | ♦ **source-msg** An a_msg instance used to replace the information contained in the calling instance. |
| See also | ♦ "Message class" on page 90 |

Example

The following example uses the copy function to initialize msgA. In this case, the source is the copyOfLastReceivedMsg field of stateB, an lsn_state instance.

For more information about the lsn_state structure, see "lsn_state structure" on page 98.

```
a_msg* msgA;
msgA -> copy( stateB -> copyOfLastReceivedMsg );
```

## a_msg equals function

| | |
|---|---|
| Function | Determines if the public fields of the calling instance equal all fields in the a_msg parameter. |
| Prototype | bool **equals (** a_msg * *source-msg* **)** |
| Parameters | ♦ **source-msg** An a_msg instance used to compare to the calling instance. |
| Return value | TRUE if the public fields of the calling instance equal all fields in the a_msg parameter. FALSE otherwise. |
| See also | ♦ "Message class" on page 90 |

Example

The following example uses the equals function to determine if msgA equals the copyOfLastReceivedMsg field of stateB, an lsn_state instance.

For more information about the lsn_state structure, see "lsn_state structure" on page 98.

```
a_msg* msgA;

//...

if(msgA -> equals( stateB -> copyOfLastReceivedMsg ))
 {
      MessageBox( NULL,
      TEXT("msgA equals stateB->copyOfLastReceivedMsg."),
        TEXT( "Message from lsn_sw786.dll" ), MB_ICONEXCLAMATION
         );
}
```

## a_msg makeEmpty function

| | |
|---|---|
| Function | Clears (zeros) the fields of an a_msg instance. |
| Prototype | void **makeEmpty ( )** |
| See also | ♦ “Message class” on page 90 |
| Example | The following example uses the makeEmpty function to clear the contents of an a_msg instance. |

```
a_msg* msgA;
msgA -> makeEmpty();
```

## a_msg reallocBuffers function

Function

Increases the size of the sender and message buffers for an a_msg instance.

Prototype

unsigned char **reallocBuffers (**
 long *senderTChars*,
 long *messageTChars*
**)**

Parameters

| | |
|---|---|
| ♦ **senderTChars** | The number of characters used for the sender field. |
| ♦ **messageTChars** | The number of characters used for the message field. |

Return value

TRUE if the sender and message fields are valid or non-NULL. FALSE otherwise.

Remarks

If the values specified at the input are less than the current length of the sender or message fields, the field size does not change.

See also

♦ “Message class” on page 90

Example

The following example uses the reallocBuffers function to increase the sender and message fields of an a_msg instance. The LSN_RET_OUT_OF_MEMORY return code is used if the reallocation fails.

For more information about LSN_RET_OUT_OF_MEMORY, see "LSN_RET enumeration" on page 95.

```
#define MAX_SENDER_TCHARS      16
#define MAX_MESSAGE_TCHARS      200
//...

a_msg* msgA;
//...

if( msgA.reallocBuffers(MAX_SENDER_TCHARS, MAX_MESSAGE_TCHARS))
{
   ret = LSN_RET_OUT_OF_MEMORY;
   goto failed;
}
```

## LSN_RET enumeration

Function

Defines the set of return values for Windows Listener SDK functions.

Prototype

```
typedef enum {
   LSN_RET_OK = 0,
   LSN_RET_NOT_SUP = 1,
   LSN_RET_NO_RESP = 2,
   LSN_RET_FAILED = 3,
   LSN_RET_MSG_NOT_READ = 4,
   LSN_RET_NO_MORE_MSG = 5,
   LSN_RET_OUT_OF_MEMORY = 6,
   LSN_RET_BAD_ARG = 7,
   LSN_RET_NOT_ENOUGH_ARG = 8,
   LSN_RET_TOO_MANY_ARGS = 9,
   LSN_RET_SERVICE_NOT_ACTIVATED = 10
} LSN_RET;
```

Parameters

| Value | Description |
|---|---|
| **LSN_RET_OK** | The function call is successful. |
| **LSN_RET_NOT_SUP** | The function call is invalid or not applicable. For example, the LsnSuspendListening function is not supported in the UDP Listener implementation. For more information, see "LsnSuspendListening function" on page 105. |
| **LSN_RET_NO_RESP** | The request timed out. |

| Value | Description |
|---|---|
| **LSN_RET_FAILED** | General failure. |
| **LSN_RET_MSG_NOT_READ** | Indicates unread messages in the Listener storage. The request to receive new messages has been ignored. |
| **LSN_RET_NO_MORE_MSG** | No unread messages remain. An attempt to read a message failed. |
| **LSN_RET_OUT_OF_MEMORY** | The function could not continue due to insufficient memory resources. |
| **LSN_RET_BAD_ARG** | A problem occurred with one or more input parameters. |
| **LSN_RET_NOT_ENOUGH_-ARG** | More input arguments are required. This return value indicates an insufficient number of string arguments contained in the input array. |
| **LSN_RET_TOO_MANY_ARGS** | Fewer input arguments are required. This return value indicates an excess number of string arguments contained in the input array. |

## LSN_VERSION enumeration

Function

Defines the Listener SDK version number.

Prototype

```
typedef enum {
   LSN_VERSION_1 = 1,
   LSN_VERSION_2 = 2,
   LSN_CURRENT_VERSION = LSN_VERSION_2
} LSN_VERSION;
```

Parameters

| Value | Description |
|---|---|
| **LSN_VERSION_1** | Specifies Listener SDK version 1. |
| **LSN_VERSION_2** | Specifies Listener SDK version 2. |
| **LSN_CURRENT_VERSION** | Specifies the current Listener version. |

Remarks

You should use this enumeration to specify the Listener SDK version that your application was developed against. To specify the current version, use

LSN_CURRENT_VERSION.

## lsn_info structure

Function

Stores Listener static information.

Prototype

```
typedef struct lsn_info {
   lsn_ulong version;
   lsn_ulong maxSenderTChars;
   lsn_ulong maxMessageTChars;
   lsn_bool isExtDialNeedSuspendListening;
   lsn_bool isMsgTimeApplicable;
   lsn_bool isMsgTypeApplicable;
   lsn_bool isMsgPriorityApplicable;
   lsn_bool isMsgTimeInUTC;

   // LSN_VERSION_1 fields ends here

} lsn_info;
```

Parameters

| Value | Description |
|---|---|
| **version** | Version of the Listener interface or SDK. |
| **maxSenderTChars** | Maximum number of characters in the sender name field. |
| **maxMessageTChars** | Maximum number of characters in the message field. |
| **isExtDialNeedSuspendListening** | If a Listener library needs to be suspended from receiving messages before another application can dial using a modem set this field to TRUE. |
| **isMsgTimeApplicable** | If the message class time fields (year, month, day, hour, minute, and second) are applicable set this field to TRUE. For more information, see "a_msg public fields" on page 91. |
| **isMsgTypeApplicable** | If the message class type field is applicable set this field to TRUE. For more information, see "a_msg public fields" on page 91. |

| Value | Description |
|---|---|
| **isMsgPriorityApplicable** | If the message class priority field is applicable set this field to TRUE. For more information, see "a_msg public fields" on page 91. |
| **isMsgTimeInUTC** | If the message class time fields are in coordinated universal time (UTC) set this field to TRUE. For more information, see "a_msg public fields" on page 91. |

See Also

- "Message class" on page 90

Example

The following example initializes lsn_info fields for a UDP Listener.

```
lsn_info * info;

info->version = LSN_VERSION_2;
info->maxSenderTChars = MAX_SENDER_TCHARS;
info->maxMessageTChars = MAX_MESSAGE_TCHARS;
info->isExtDialNeedSuspendListening = false;
info->isMsgTimeApplicable = false;
info->isMsgTypeApplicable = false;
info->isMsgPriorityApplicable = false;
info->isMsgTimeInUTC = false;
```

## lsn_state structure

Function

The Listener state structure used for dynamic information.

Prototype

```
struct lsn_state;
// Add your Listener state fields here
```

Remarks

You add fields to this structure to store dynamic Listener information such as a message field, receive buffer, or a flag to specify unread messages.

Example

The following example shows the lsn_state structure used for the UDP Listener implementation.

```
typedef struct lsn_state {
    WSADATA wsaData;
    SOCKET socket;
    bool hideWSAErrorBox;
    bool unread;
    bool showSenderPort;
    a_msg * msg;
    struct timeval timeout;
    char * recvBuff;
    UINT codePage;
    TCHAR port[6];
} lsn_state;
```

The following example shows the lsn_state structure used for the Sierra Wireless AirCard 510 Listener implementation.

```
typedef struct lsn_state {
    bool apiOpened;
    int nextMsg;
    int numReceivedMsg;
    int messageStoreSize;
    a_msg copyOfLastReceivedMsg;
    a_msg ** messages;
    TCHAR * phone;
    TCHAR * carrier;
} lsn_state;
```

## LsnInit function

Function

Allocates and initializes a Listener state structure.

Prototype

LSN_RET **LsnInit (**
 lsn_info* *info*,
 lsn_state** *state*,
 lsn_long *argc*,
 char* *argv[]*,
 lsn_long* *badArg*
**)**

Parameters

- **info** This is an output parameter containing an lsn_info structure with static Listener information.

  ☞ For more information, see "lsn_info structure" on page 97.

- **state** This is an output parameter pointing to the allocated lsn_state structure.

  ☞ For more information, see "lsn_state structure" on page 98.

- **argc** The number of string arguments contained in the argv[] array.

- ♦ **argv[]** An array of string arguments used to initialize the Listener state structure.

  You specify the arguments passed to argv[] using the dblsn.exe -a command line option. For example,

  ```
  dblsn -d my_lsn.dll -a prop1=v1 -a prop2=v2 ...
  ```

  results in the following string arguments:

  ```
  argv[] = { "prop1=v1", "prop2=v2", ... }
  ```

  For more information about the -a option, see "The Listener utility" on page 38.

- ♦ **badArg** This is an output parameter providing argument error information. The meaning of this parameter depends on the LsnInit return value:
  - If the return value is LSN_RET_BAD_ARG, badArg is the index of the problematic argument in argv.
  - If the return is LSN_RET_NOT_ENOUGH_ARG, badArg is the minimum number of arguments required.
  - If the return value is LSN_TOO_MANY_ARGS, badArg is the maximum number of allowed arguments.

Return value

Use the following return values defined in the LSN_RET enumeration for this function.

| Value | Description |
|---|---|
| **LSN_RET_OK** | The function call is successful. |
| **LSN_RET_FAILED** | General failure. |
| **LSN_RET_OUT_OF_MEMORY** | The function could not continue due to insufficient memory resources. |
| **LSN_RET_BAD_ARG** | A problem occurred with one or more input parameters. |
| **LSN_RET_NOT_ENOUGH_-ARG** | More input arguments are required. This return value indicates an insufficient number of string arguments contained in the input array. **[Am I right on this?]** |
| **LSN_RET_TOO_MANY_ARGS** | Fewer input arguments are required. This return value indicates an excess number of string arguments contained in the input array. |

For more information about the LSN_RET enumeration, see "LSN_RET enumeration" on page 95.

Remarks

Use this function to initialize your Listener and acquire necessary resources for the Listener to operate. If the function call is successful, your LsnInit implementation returns the filled lsn_info and allocated lsn_state structures.

The number and type of input parameters in the argv[] array depend on your implementation.

See also

- "LSN_RET enumeration" on page 95
- "lsn_state structure" on page 98
- "lsn_info structure" on page 97
- "Message class" on page 90

Example

The following example shows the LsnInit function used for the UDP Listener implementation.

```
LsnExport LSN_RET LsnExportFunc LsnInit(
    lsn_info * info, lsn_state** state, lsn_long argc,
    char* argv[], lsn_long* badArg )
/*******************************************************/
{
    // Argument names
    #define ARGN_RECEIVER_PORT "Port"
    #define ARGN_HIDE_WSA_ERROR_BOX "HideWSAErrorBox"
    #define ARGN_TIMEOUT_SEC "Timeout"
    #define ARGN_SHOW_SENDER_PORT "ShowSenderPort"
    #define ARGN_CODEPAGE "CodePage"

      // Helper macros
    #define IS_ARG( argn, x ) _strnicmp( argn "=", x,
         sizeof(argn) ) == 0
    #define ARG_VAL( argn, x ) &(x)[sizeof(argn)]

   LSN_RET ret = LSN_RET_FAILED;
    lsn_state * s;
    struct sockaddr_in receiverAddr;
    unsigned short port;
    int i;
    bool showUsageBox = true;
    *state = NULL;

      // Fill static info structure
    info->version = LSN_VERSION_2;
    info->maxSenderTChars = MAX_SENDER_TCHARS;
    info->maxMessageTChars = MAX_MESSAGE_TCHARS;
    info->isExtDialNeedSuspendListening = false;
    info->isMsgTimeApplicable = false;
    info->isMsgTypeApplicable = false;
    info->isMsgPriorityApplicable = false;
    info->isMsgTimeInUTC = false;
```

```
 // Check argc
    if( argc < 0 ) {
   *badArg = 0;
   ret = LSN_RET_NOT_ENOUGH_ARG;
   goto failed;
    }
    if( argc > 4 ) {
   *badArg = 4;
   ret = LSN_RET_TOO_MANY_ARGS;
   goto failed;
    }

 // Allocate state structure
    s = (lsn_state*)calloc( 1, sizeof(lsn_state) );
    if( s == NULL ) {
   showUsageBox = false;
   ret = LSN_RET_OUT_OF_MEMORY;
   goto failed;
    }

 // Initialize state structure
    memset( s, 0, sizeof(lsn_state) );
    s->socket = INVALID_SOCKET;
    s->hideWSAErrorBox = false;
    s->unread = false;
    s->showSenderPort = false;
    s->timeout.tv_sec = 0;
    s->timeout.tv_usec = 0;

// Allocate message in state structure
    s->msg = a_msg::allocateSize( MAX_SENDER_TCHARS, MAX_
         MESSAGE_TCHARS );
    if( s->msg == NULL ) {
   showUsageBox = false;
      ret = LSN_RET_OUT_OF_MEMORY;
   goto failed;
    }
#if defined( UNICODE )
    // Allocate recvBuff
    s->recvBuff = (char*)malloc( MAX_MESSAGE_BYTES + 1 );
    if( s->recvBuff == NULL ) {
   ret = LSN_RET_OUT_OF_MEMORY;
   goto failed;
    }
#endif
    port = 5001;
 // Read args
    for( i = 0; i < argc; i++ ) {
  *badArg = i;
      if( IS_ARG( ARGN_RECEIVER_PORT, argv[i] ) ) {
       port = atoi( ARG_VAL( ARGN_RECEIVER_PORT, argv[i] ) );
  } else if( _stricmp( ARGN_HIDE_WSA_ERROR_BOX, argv[i] ) == 0
         ) {
       s->hideWSAErrorBox = true;
  } else if( _stricmp( ARGN_SHOW_SENDER_PORT, argv[i] ) == 0 )
         {
       s->showSenderPort = true;
  } else if( IS_ARG( ARGN_TIMEOUT_SEC, argv[i] ) ) {
       s->timeout.tv_sec = atol( ARG_VAL( ARGN_TIMEOUT_SEC,
         argv[i] ) );
  } else if( IS_ARG( ARGN_CODEPAGE, argv[i] ) ) {
       s->codePage = atol( ARG_VAL( ARGN_CODEPAGE, argv[i] ) );
  } else {
       ret = LSN_RET_BAD_ARG;
       goto failed;
  }
   }

 // Fill the port buffer
    _stprintf( s->port, TEXT("%d"), port );
    // At this point, all args are valid
    showUsageBox = false;
    *badArg = -1;
    // WSAStartup
    if( WSAStartup( 0x202, &s->wsaData ) != 0 ) {
  foundWSAError( s );
  goto failed;
    }

// Setup receiver address
    memset( &receiverAddr, 0, sizeof(receiverAddr) );
    receiverAddr.sin_family = AF_INET;
    receiverAddr.sin_addr.s_addr = INADDR_ANY;
    receiverAddr.sin_port = htons( port );
    // Open socket
    s->socket = socket( AF_INET, SOCK_DGRAM, 0 );
    if( s->socket == INVALID_SOCKET ) {
  foundWSAError( s );
  goto failed;
    }
 // Bind the socket to the receiver address
    if( bind( s->socket, (struct sockaddr*)&receiverAddr,
         sizeof( receiverAddr ) )
        == SOCKET_ERROR ) {
   foundWSAError( s );
   goto failed;
    }
    *state = s;
    return( LSN_RET_OK );
  failed:
    if( showUsageBox ) {
   MessageBox( NULL, USAGE, USAGE_TITLE, MB_ICONINFORMATION );
    }
    LsnFini( s );
        return( ret );
}
```

## LsnFini function

Function

Shuts down the Listener and frees resources.

Prototype

void **LsnFini (** lsn_state* *state* **)**

Parameters

- **state** Points to an lsn_state structure instance.

  For more information, see "lsn_state structure" on page 98.

Remarks

Use this function to shutdown your Listener and free allocated resources, including the Listener state structure.

See also

- "lsn_state structure" on page 98
- "Message class" on page 90

Example

The following example shows the LsnFini function for the UDP Listener implementation.

```
LsnExport void LsnExportFunc LsnFini( lsn_state* state )
/*********************************************************/
{

 if( state != NULL ) {
    if( state->msg != NULL ) {
        delete( state->msg );
    }
    if( state->recvBuff != NULL ) {
        free( state->recvBuff );
    }
    if( state->socket != INVALID_SOCKET ) {
             closesocket( state->socket );
    }
    WSACleanup();
    free( state );
 }
}
```

## LsnIsListening

Function

Determines if the Listener is ready to receive messages.

Prototype

bool **LsnIsListening (** const lsn_state* *state* **)**

Parameters

- **state** Points to an lsn_state structure instance.

  For more information about lsn_state, see "lsn_state structure" on page 98.

Return value

Returns TRUE if the Listener is ready to receive messages and FALSE otherwise.

Remarks

Use this function to define when your Listener is ready to receive messages.

See also

- "lsn_state structure" on page 98
- "Message class" on page 90

Example

The following example shows the LsnIsListening implementation for UDP Listeners.

```
LsnExport bool LsnExportFunc LsnIsListening(
    const lsn_state* state)
/*********************************************/
{
    return( state->socket != INVALID_SOCKET );
}
```

## LsnSuspendListening function

Function

Suspends the Listener.

Prototype

LSN_RET **LsnSuspendListening (** lsn_state* *state* **)**

Parameters

- **state** Points to a Listener state structure instance.

  For more information about lsn_state, see "lsn_state structure" on page 98.

Return value

Use the following return values defined in the LSN_RET enumeration for this function.

| Value | Description |
|---|---|
| **LSN_RET_OK** | The function call is successful. |
| **LSN_RET_NOT_SUP** | The function call is invalid or not applicable. For example, the LsnSuspendListening function is not supported in the UDP Listener implementation. For more information, see "LsnSuspendListening function" on page 105. |
| **LSN_RET_NO_RESP** | The request timed out. |
| **LSN_RET_FAILED** | General failure. |

For more information about the LSN_RET enumeration, see "LSN_RET enumeration" on page 95.

Remarks

Use this function to prevent your Listener from receiving messages.

See also

- "LSN_RET enumeration" on page 95
- "lsn_state structure" on page 98
- "Message class" on page 90

Example

The following example shows the LsnSuspendListening function for the Sierra Wireless AirCard 510 implementation. The `SwiApiClose()` function call suspends the modem from listening on SMS.

```
LsnExport LSN_RET LsnExportFunc LsnSuspendListening(
    lsn_state* state )
/*******************************************************/
{
    if( !state->apiOpened ) {
        return( LSN_RET_OK );
    }
    state->apiOpened = FALSE;
    return( transformRcode( SwiApiClose() ) );
}
```

The following example shows the LsnSuspendListening implementation for UDP Listeners. Since UDP Listeners do not support this function, it uses the return value LSN_RET_NOT_SUP.

☞ For more information about LSN_RET_NOT_SUP, see "LSN_RET enumeration" on page 95.

[ what is _unused ?]

```
LsnExport LSN_RET LsnExportFunc LsnSuspendListening(
    lsn_state* state )
/*****************************************************/
{
    _unused( state );
    return( LSN_RET_NOT_SUP );
}
```

## LsnResumeListening function

Function

Resumes a suspended listener.

Prototype

LSN_RET **LsnResumeListening (** lsn_state* *state* **)**

Parameters

♦ **state** Points to a Listener state structure instance.

For more information about lsn_state, see "lsn_state structure" on page 98.

Return value

Use the following return values defined in the LSN_RET enumeration for this function.

| Value | Description |
|---|---|
| **LSN_RET_OK** | The function call is successful. |
| **LSN_RET_NOT_SUP** | The function call is invalid or not applicable. For example, the LsnSuspendListening function is not supported in the UDP Listener implementation. For more information, see "LsnSuspendListening function" on page 105. |
| **LSN_RET_NO_RESP** | The request timed out. |
| **LSN_RET_FAILED** | General failure. |

For more information about the LSN_RET enumeration, see "LSN_RET enumeration" on page 95.

Remarks

Use this function to resume a suspended Listener.

See also

♦ "LSN_RET enumeration" on page 95
♦ "lsn_state structure" on page 98
♦ "Message class" on page 90

Example

The following example shows the LsnSuspendListening function for the Sierra Wireless AirCard 510 implementation. The `SwiApiOpen( SESSION_NAME )` function call triggers the modem to resume listening on

SMS.

[Can we provide a few words to describe this?]

```
LsnExport LSN_RET LsnExportFunc LsnResumeListening(
    lsn_state* state )
/****************************************************/
{
    LSN_RET ret;
    if( state->apiOpened ) {
        return( LSN_RET_OK );
    }
    ret = transformRcode( SwiApiOpen( SESSION_NAME ) );
    state->apiOpened = ( ret == LSN_RET_OK );
    return( ret );
}
```

The following example shows the LsnSuspendListening implementation for UDP Listeners. Since UDP Listeners do not support this function, it uses the return value LSN_RET_NOT_SUP.

☞ For more information about LSN_RET_NOT_SUP, see "LSN_RET enumeration" on page 95.

```
LsnExport LSN_RET LsnExportFunc LsnResumeListening(
    lsn_state* state
)
/******************************************************/
{
    _unused( state );
    return( LSN_RET_NOT_SUP );
}
```

## LsnReceiveAll function

| | |
|---|---|
| Function | Receives pending messages. |
| Prototype | LSN_RET **LsnReceiveAll (**<br>lsn_state* *state*,<br>lsn_long* *nMsg*<br>**)** |
| Parameters | ♦ **state** Points to a lsn_state structure instance.<br>For more information about lsn_state, see "lsn_state structure" on page 98.<br>♦ **nMsg** Points to the number of received messages. |
| Return value | Use the following return values defined in the LSN_RET enumeration for this function. |

| Value | Description |
|---|---|
| **LSN_RET_OK** | The function call is successful. |
| **LSN_RET_NO_RESP** | The request timed out. |
| **LSN_RET_FAILED** | General failure. |
| **LSN_RET_MSG_NOT_READ** | Indicates unread messages in the Listener storage. The request to receive new messages has been ignored. |

For more information about the LSN_RET enumeration, see "LSN_RET enumeration" on page 95.

Remarks

Use this function to receive pending messages into your Listener storage. This function should terminate when all pending messages are received or the store is full.

The nMsg parameter points to the number of messages that can be read using the LsnReadNext function.

For more information about the LsnReadNext function, see "LsnReadNext function" on page 111.

See also

- "LsnReadNext function" on page 111
- "LSN_RET enumeration" on page 95
- "lsn_state structure" on page 98
- "Message class" on page 90

Example

The following example shows the LsnReceiveAll function for the UDP Listener implementation.

```
LsnExport LSN_RET LsnExportFunc LsnReceiveAll(
    lsn_state* state, lsn_long* nMsg )
/**************************************************/
{
    LSN_RET ret = LSN_RET_FAILED;
    int wsaRet;
    struct sockaddr_in from;
    int fromLen = sizeof(from);
    fd_set readableSockets;
    char* senderIpA;
#if defined( UNICODE )
    int mbwcRet;
    WCHAR senderIpW[ MAX_SENDER_TCHARS + 1 ];
    #define RECVBUFF   (state->recvBuff)
    #define SENDERIP   senderIpW
#else
    #define RECVBUFF (state->msg->message)
    #define SENDERIP   senderIpA
#endif

    if( state->unread ) {
   return( LSN_RET_MSG_NOT_READ );
    }
    if( nMsg != NULL ) {
   *nMsg = 0;
    }
    state->unread = false;

 // Select readable socket
    FD_ZERO( &readableSockets );
    FD_SET( state->socket, &readableSockets );
    switch( select( 0, &readableSockets, NULL, NULL, &state-
         >timeout ) ) {
    case 0:
        // Nothing to read
        ret = LSN_RET_OK;
   break;
    case 1:
        // Receive the message
   wsaRet = recvfrom( state->socket, RECVBUFF,
          MAX_MESSAGE_BYTES, 0,
          (struct sockaddr *)&from, &fromLen );
   if( wsaRet == SOCKET_ERROR ) {
       foundWSAError( state );
       break;
   }
   // Null terminate the message
   RECVBUFF[ wsaRet ] = '\0';
#if defined( UNICODE )
   mbwcRet = MultiByteToWideChar( state->codePage, 0, state-
         >recvBuff,
          -1, state->msg->message, MAX_MESSAGE_TCHARS );
   if( mbwcRet == 0 ) {
       MessageBox( NULL,
          TEXT("Failed to translate an incoming message"),
      TEXT( "Message from lsn_udp.dll" ), MB_ICONEXCLAMATION );
       break;
   }
#endif

   // Retrieve the sender name and port
   senderIpA = inet_ntoa( from.sin_addr );
   if( senderIpA == NULL ) {
       break;
   }
#if defined( UNICODE )
   size_t i;
   for( i = 0; i <= strlen( senderIpA ); i++ ) {
       senderIpW[i] = senderIpA[i];
   }
#endif

if( state->showSenderPort ) {
       _stprintf( state->msg->sender, TEXT("%s:%d"),
          SENDERIP, ntohs( from.sin_port ) );
   } else {
       _tcscpy( state->msg->sender, SENDERIP );
   }
        if( nMsg != NULL ) {
       *nMsg = 1;
   }

   state->unread = true;
   ret = LSN_RET_OK;
   break;
    case SOCKET_ERROR:
   foundWSAError( state );
   break;
    }
    return( ret );
}
```

## LsnReadNext function

Function

Used to read the next message into a message buffer.

Prototype

LSN_RET **LsnReadNext (**
 lsn_state* *state*,
 a_msg* *msg*
**)**

Parameters

- **state** Points to a Listener state structure instance.

  For more information about lsn_state, see "lsn_state structure" on page 98.

- **msg** Points to the destination message buffer.

Return value

Use the following return values defined in the LSN_RET enumeration for this function.

| Value | Description |
|---|---|
| **LSN_RET_OK** | The function call is successful. |
| **LSN_RET_NO_MORE_MSG** | No unread messages remain. An attempt to read a message failed. |

For more information about the LSN_RET enumeration, see "LSN_RET enumeration" on page 95.

Remarks

Use this function to read the next message from your Listener storage.

See also

- "LSN_RET enumeration" on page 95
- "lsn_state structure" on page 98
- "Message class" on page 90

Example

The following example shows the LsnReadNext function for the UDP Listener implementation.

```
LsnExport LSN_RET LsnExportFunc LsnReadNext(
lsn_state* state, a_msg* msg )
/***********************************************/
{
    if( !state->unread ) {
   return( LSN_RET_NO_MORE_MSG );
    }
    msg->copy( state->msg );
    state->unread = false;
    return( LSN_RET_OK );
}
```

## LsnGetAddress

Function

Returns the Listener address.

Prototype

const TCHAR * **LsnGetAddress (** lsn_state* *state* **)**

Parameters

- **state** Points to a Listener state structure instance.

  For more information about lsn_state, see "lsn_state structure" on page 98.

Return value

A character array containing the Listener address.

Remarks

Use this function to return the address used for your Listener.

The address, for example, can be specified as a phone number.

See also

- ♦ "lsn_state structure" on page 98
- ♦ "Message class" on page 90

Example

The following example shows the LsnGetAddress function for the Sierra Wireless AirCard 510 implementation.

```
LsnExport const TCHAR * LsnExportFunc LsnGetAddress(
  lsn_state* state )
/***************************************************/
{
    if( state->phone == NULL ) {
   return( TEXT("Unspecified") );
    }
    return( state->phone );
}
```

## LsnGetMedium function

Function

Returns the carrier or medium used for Listener communication.

Prototype

TCHAR * **LsnGetMedium (** lsn_state* *state* **);**

Parameters

- ♦ **state** Points to a Listener state structure instance.

  For more information about lsn_state, see "lsn_state structure" on page 98.

Return value

A character array storing the medium or carrier name.

Remarks

Use this function to return the carrier or medium used for your Listener.

See also

- ♦ "lsn_state structure" on page 98
- ♦ "Message class" on page 90

Example

The following example shows the LsnGetMedium function for the Sierra Wireless AirCard 510 implementation.

```
LsnExport const TCHAR * LsnExportFunc LsnGetMedium(
  lsn_state* state )
/**************************************************/
{
   if( state->carrier == NULL ) {
 return( TEXT("Unspecified") );
    }
   return( state->carrier );
}
```

# Listener SDK for Palm

You can use the Listener SDK for Palm to create Listeners for new Palm devices. The programming interface includes a message processing interface and device dependent functions.

## Message processing interface

The message processing interface is contained in *PalmLsn.lib*, the Palm Listener Library.

☞ For more information about *PalmLsn.lib*, see "Palm Listener SDK files" on page 88.

a_palm_msg structure

The Palm Listener SDK uses the a_palm_msg structure to represent Palm Listener messages. The SDK's message processing interface includes functions to allocate and process a_palm_msg instances.

### Overview

The following functions can be used for a_palm_msg allocation, message field initialization, and message processing.

- **Message allocation** You can use the following functions for message allocation and deallocation:

  "Overview" on page 114

  "PalmLsnFree function" on page 115

- **Message field initialization** You can use the following functions to assign values to the message, sender, and time fields of an a_palm_msg instance.

  "PalmLsnDupMessage function" on page 115

  "PalmLsnDupSender function" on page 116

  "PalmLsnDupTime function" on page 117

- **Message processing** You can use the PalmLsnProcess function to process a message's fields and launch an application.

  For more information, see "PalmLsnProcess function" on page 117.

### PalmLsnAllocate function

| | |
|---|---|
| Function | Returns a new a_palm_msg instance. |
| Prototype | struct a_palm_msg * **PalmLsnAllocate( )** |
| Return value | A new a_palm_msg instance with all fields initialized to zero. |

See Also

- “PalmLsnFree function” on page 115

Example

The following example uses PalmLsnAllocate to allocate an a_palm_msg instance.

```
a_palm_msg *   ulMsg;

// Allocate a message structure
ulMsg = PalmLsnAllocate();
```

## PalmLsnFree function

Function

Frees message memory resources.

Prototype

void **PalmLsnFree(** struct a_palm_msg * const *msg* **)**

Parameters

- **msg** The a_palm_msg instance to be freed.

See Also

- “Overview” on page 114

Example

The following example shows a partial listing for allocating the message structure, processing the message, and using PalmLsnFree to free resources.

```
a_palm_msg *   ulMsg;
...

// Allocate the message structure
ulMsg = PalmLsnAllocate();
...

// Fill the message fields
ret = PalmLsnDupMessage( ulMsg, msgBody );
...

// Process the message
ret = PalmLsnProcess( ulMsg, configDb, NULL, handled );
...

// Free the message
PalmLsnFree( ulMsg );
```

## PalmLsnDupMessage function

Function

Initializes the message field values of an a_palm_msg instance.

Prototype

Err **PalmLsnDupMessage(**
  struct a_palm_msg * const *msg*,
  Char const * *message*
**)**

Parameters

- **msg** A pointer to an a_palm_msg instance.
- **message** An input parameter containing the source message text.

| | |
|---|---|
| Return Value | A Palm OS error code. errNone indicates success. |

Remarks

The PalmLsnDupMessage function duplicates a text message, extracts the subject, content, and sender fields, and assigns these values to an a_palm_msg instance.

The sender field is not extracted if it does not appear in the message. If you use PalmLsnDupSender it overrides the sender field extracted from PalmLsnDupMessage (if any).

See Also

- ♦ "PalmLsnDupSender function" on page 116
- ♦ "PalmLsnDupTime function" on page 117
- ♦ "a_palm_msg structure" on page 114

Example

The following example, used for the Treo 600 smartphone implementation, retrieves a text message and calls PalmLsnDupMessage to initialize the appropriate fields in an a_palm_msg instance.

```
//
// Retrieve the entire message body
//
ret = PhnLibGetText( libRef, id, &msgBodyH );
if( ret != errNone ) {
    // handle error
    goto done;
}
msgBody = (Char *)MemHandleLock( msgBodyH );
ret = PalmLsnDupMessage( ulMsg, msgBody );
//
// msgBodyH must be disposed of by the caller
//
MemHandleUnlock( msgBodyH );
MemHandleFree( msgBodyH );
if( ret != errNone ) {
  // handle error
  goto done;
}
```

## PalmLsnDupSender function

Function

Initializes the sender field of an a_palm_msg instance.

Prototype

Err **PalmLsnDupSender(**
  struct a_palm_msg * const *msg*,
  Char const * *sender*
**)**

Parameters

- ♦ **msg** A pointer to an a_palm_msg instance.
- ♦ **sender** An input parameter containing the source sender field.

Return Value

A Palm OS error code. errNone indicates success.

Remarks

The PalmLsnDupSender function duplicates the sender input parameter and assigns the value to an a_palm_msg instance.

See Also

- "PalmLsnDupMessage function" on page 115
- "PalmLsnDupTime function" on page 117
- "a_palm_msg structure" on page 114

## PalmLsnDupTime function

Function

Initializes the time field of an a_palm_msg instance.

Prototype

Err **PalmLsnDupTime(**
    struct a_palm_msg * const *msg*,
    UInt32 const *time*
**)**

Parameters

- **msg** A pointer to an a_palm_msg instance.
- **time** An input parameter containing the source time field.

Return Value

A Palm OS error code. errNone indicates success.

Remarks

The PalmLsnDupTime function duplicates the time input parameter and assigns the value to an a_palm_msg instance.

See Also

- "PalmLsnDupMessage function" on page 115
- "PalmLsnDupSender function" on page 116
- "a_palm_msg structure" on page 114

## PalmLsnProcess function

Function

Processes a message according to the records in a configuration database.

Prototype

palm_lsn_ret **PalmLsnProcess(**
    struct a_palm_msg * *msg*,
    Char const * *configPDBName*,
    UInt16 * const *problematicRecNum*,
    Boolean * *handled*
**)**

Parameters

- **msg** A pointer to an a_palm_msg instance.
- **configPDBName** A character array containing the name of the configuration database. You can obtain the configuration database name using the PalmLsnGetConfigFileName function.

  ☞ See "PalmLsnGetConfigFileName" on page 124.
- **problematicRecNum** An output parameter identifying the index of a problematic or malformed record in the configuration database.

- **handled** An output parameter indicating if PalmLsnProcess successfully processed the message.

Return Value

Return codes defined in the palm_lsn_ret enumeration.

☞ See “palm_lsn_ret enumeration” on page 120.

Remarks

PalmLsnProcess determines the appropriate action to take in response to an incoming message. It compares the message’s fields to filters stored in a configuration database.

☞ For more information about creating the Palm Listener configuration database, see “Palm Listener Configuration utility” on page 50.

The records contained in the configuration database store information about message filters and what actions should result from an accepted message.

A configuration record has the following format:

```
[subject=<string>;] [content=<string>;]
[message|message_start=<string>;] [sender=<string>;]
action=run <app name> [arguments]
```

*arguments* is an application dependent string which may contain action variables.

☞ For more information about action variables, see “Action variables” on page 44.

See Also

- “Palm Listener Configuration utility” on page 50
- “Message handlers” on page 29
- “Action variables” on page 44
- “PalmLsnCheckConfigDB function” on page 119
- “a_palm_msg structure” on page 114

Example

The following is a partial listing used to handle a message. The example allocates the message structure, initializes fields, and processes the message using PalmLsnProcess.

```
a_palm_msg * ulMsg;
Boolean * handled
Char configDb[ dmDBNameLength ];
...

// Allocate the message structure
ulMsg = PalmLsnAllocate();
...

// Fill the message fields
ret = PalmLsnDupMessage( ulMsg, msgBody );
...

// Get the configuration database name
PalmLsnGetConfigFileName( configDb );

// Process the message
ret = PalmLsnProcess( ulMsg, configDb, NULL, handled );
...

// Free the message
PalmLsnFree( ulMsg );
```

## PalmLsnCheckConfigDB function

Function

Reports errors in a Palm Listener configuration database.

Prototype

palm_lsn_ret **PalmLsnCheckConfigDB(**
Char const * *cfg*,
UInt16 * const *rec*
**)**

Parameters

- **cfg** A character array containing the name of the configuration database. You can obtain the configuration database name using the PalmLsnGetConfigFileName function.

  ☞ See “PalmLsnGetConfigFileName” on page 124.

- **rec** An output parameter identifying the index of a problematic or malformed record in the configuration database.

Return Value

Return codes defined in the palm_lsn_ret enumeration.

☞ See “palm_lsn_ret enumeration” on page 120.

Remarks

You can use this function to detect errors opening a configuration database or reading its records.

See Also

- “PalmLsnProcess function” on page 117

Example

The following example uses PalmLsnCheckConfigDB to detect problematic or malformed records in a configuration database.

```
Err ret;
UInt16 badRec;
Char configDb[ dmDBNameLength ];

// Get configuration database name
PalmLsnGetConfigFileName( configDb );

// check for errors in the configuration database
ret = PalmLsnCheckConfigDB(configDb, &badRec);
if(ret!=errNone)
{
  // handle error
}
```

## palm_lsn_ret enumeration

Function

The palm_lsn_ret enumeration specifies the possible message processing return codes.

Prototype

```
typedef enum {
   PalmLsnOk = errNone,
   PalmLsnMissingConfig = appErrorClass,
   PalmLsnProblemReadingConfig,
   PalmLsnProblemParsingCmd,
   PalmLsnOutOfMemory,
   PalmLsnUnrecognizedAction,
   PalmLsnRunMissingApp
} palm_lsn_ret;
```

Parameters

| Value | Description |
|---|---|
| **PalmLsnOk** | The function call is successful. This value contains the same value as errNone, a Palm error code indicating no error. |
| **PalmLsnMissingConfig** | Indicates a missing Palm Listener configuration database. This field contains the same value as the Palm error code appErrorClass, indicating an application-defined error. |
| **PalmLsnProblemReadingConfig** | Indicates an error reading the Palm Listener configuration database. |
| **PalmLsnProblemParsingCmd** | Indicates an inability to process the command stored in the Palm Listener configuration database. |
| **PalmLsnOutOfMemory** | The function does not run to completion due to an error while allocating memory for message processing. |
| **PalmLsnUnrecognizedAction** | The Listener does not support an action specified in the Palm Listener configuration database. |
| **PalmLsnRunMissingApp** | The Listener cannot launch the application specified in the run action. |

See Also

- "PalmLsnProcess function" on page 117

## LsnMain function

Function

Provides the main entry point to *PalmLsn.lib*, the Palm Listener library.

Prototype

```
UInt32 LsnMain(
   UInt16 cmd,
   MemPtr cmdPBP,
   UInt16 launchFlags
)
```

Parameters

- **cmd** A Palm OS application launch code.
- **cmdPBP** A pointer to a structure containing launch code parameters. If

your application does not have any launch-command-specific parameters, this value is NULL.

- **launchFlags** Flags that provide extra information about the launch.

Return Value

A Palm OS error code. If the Palm listener library successfully processed the launch code, the function returns errNone.

Remarks

The values passed to LsnMain are analogous to the launch code parameters passed to PilotMain, the main entry point of a Palm OS application.

For more information about these parameters, consult your Palm OS Reference.

See Also

- "PalmLsnProcess function" on page 117
- "Palm Listener SDK files" on page 88

Example

The following example, used in theTreo 600 smartphone implementation, passes launch code parameters to LsnMain in the main entry point of the Listener application.

```
UInt32 PilotMain(
/**************/
    UInt16 cmd,
    MemPtr cmdPBP,
    UInt16 launchFlags )
{
    return( LsnMain( cmd, cmdPBP, launchFlags ) );
}
```

## Device dependent functions

You specify device dependent features using a group of functions defined in the Palm Listener SDK. These functions provide:

- **Identification** You can use the following functions to provide identification information for the Listener and the configuration database:

  "PalmLsnTargetCompanyID" on page 123

  "PalmLsnTargetDeviceID" on page 123

  "PalmLsnGetConfigFileName" on page 124

- **Registration or initialization** You can use the following functions to register or unregister the Listener.

  "PalmLsnNormalStart" on page 124

  "PalmLsnNormalStop" on page 124

- **Event handling** You can use the following function to handle application events:

"PalmLsnNormalHandleEvent" on page 125

You can use the following function to respond to launch codes which may be device dependent.

"PalmLsnSpecialLaunch" on page 125

## PalmLsnTargetCompanyID

|  |  |
|---|---|
| Function | Returns a device's company ID. |
| Prototype | UInt32 **PalmLsnTargetCompanyID( )** |
| Return Value | A value containing the ID of the device's company or manufacturer. |
| Remarks | You can use PalmLsnTargetCompanyID and PalmLsnTargetDeviceID to check for device compatibility. |
| See Also | ♦ "PalmLsnTargetDeviceID" on page 123 |
| Example | The following example, used in the Treo 600 smartphone implementation, returns 'hspr', a company ID for Handspring. |

```
UInt32 PalmLsnTargetCompanyID( void )
/***********************************/
{
    return( 'hspr' );
}
```

## PalmLsnTargetDeviceID

|  |  |
|---|---|
| Function | Returns the target device ID. |
| Prototype | UInt32 **PalmLsnTargetDeviceID( )** |
| Return Value | A positive integer containing the device ID. |
| Remarks | You can use PalmLsnTargetCompanyID and PalmLsnTargetDeviceID to check for device compatibility. |
| See Also | ♦ "PalmLsnTargetCompanyID" on page 123 |
| Example | The following example returns the device ID for the Treo 600 simulator. |

```
UInt32 PalmLsnTargetDeviceID( void )
/**********************************/
{
    // Simulator device ID is hsDeviceIDOs5Device1Sim
    return( hsDeviceIDOs5Device1 );
}
```

## PalmLsnGetConfigFileName

| | |
|---|---|
| Function | Returns a string containing the name of your Palm Listener configuration database. |
| Prototype | void **PalmLsnGetConfigFileName(** Char * *configPDBName* **)** |
| Parameters | ♦ **configPDBName** An output parameter containing the name of your Palm Listener configuration database. |
| Remarks | You can use this function to obtain the configuration database file name to pass into PalmLsnProcess.<br><br>To use the default configuration database file name *lsncfg* copy PalmLsnDefaultConfigDB (defined in *PalmLsn.h*) into the output parameter. |
| See Also | ♦ "PalmLsnProcess function" on page 117<br>♦ "Palm Listener SDK files" on page 88 |
| Example | The following example, used for the Treo 600 smartphone implementation, returns the default configuration database name in the output parameter. |

```
void PalmLsnGetConfigFileName( Char * configPDBName )
{
    StrCopy( configPDBName, PalmLsnDefaultConfigDB );
}
```

## PalmLsnNormalStart

| | |
|---|---|
| Function | Provides custom actions when your Listener application starts. |
| Prototype | Err **PalmLsnNormalStart( )** |
| Return Value | A Palm OS error code. errNone indicates success. |
| Remarks | PalmLsnNormalStart provides a means to register your Listener device. |
| See Also | ♦ "PalmLsnNormalStop" on page 124<br>♦ "PalmLsnSpecialLaunch" on page 125 |

## PalmLsnNormalStop

| | |
|---|---|
| Function | Provides custom actions when your Listener application exits from the event loop. |
| Prototype | void **PalmLsnNormalStop( )** |
| Remarks | If you want to continue listening, do not unregister your device in PalmLsnNormalStop. You can also use this function to get and set the current application preferences. |

See Also

- ♦ “PalmLsnNormalStart” on page 124

## PalmLsnNormalHandleEvent

Function — Handles application events.

Prototype — Boolean **PalmLsnNormalHandleEvent(** EventPtr *eventP* **)**

Parameters

- ♦ **eventP** A pointer to an application event.

Return Value — Returns true if the event was handled.

Remarks — You can use this function to handle application events.

## PalmLsnSpecialLaunch

Function — Responds to launch codes which may be device dependent.

Prototype — Err **PalmLsnSpecialLaunch(**
UInt16 *cmd*,
MemPtr *cmdPBP*,
UInt16 *launchFlags*
**)**

Parameters

- ♦ **cmd** The Palm OS application launch code.
- ♦ **cmdPBP** A pointer to a structure containing launch code parameters. If your application does not have any launch-command-specific parameters, this value is NULL.
- ♦ **launchFlags** Flags that indicate status information about your application.

Return Value — A Palm OS error code. errNone indicates success.

Remarks — This function responds to device dependent or standard launch codes not defined as sysAppLaunchCmdNormalLaunch.

Example — The following example, used for the Treo 600 smartphone implementation, uses PalmLsnSpecialLaunch to handle Listener events.

```
Err PalmLsnSpecialLaunch(
/**********************/
    UInt16 cmd,
    MemPtr cmdPBP,
    UInt16 /*launchFlags*/ )
{
switch( cmd ) {

  case sysAppLaunchCmdSystemReset:
    // Fall through
    case phnLibLaunchCmdRegister:
     break;

  case phnLibLaunchCmdEvent: {
     if( !IsFeatureOn( PalmLsnGetFeature(), Listening ) ) {
        return( errNone );
     }

   PhnEventPtr phoneEventP = (PhnEventPtr)cmdPBP;

   if( phoneEventP->eventType == phnEvtMessageInd ) {
        // handle the message
        return( handleMessage( phoneEventP->data.params.id,
           &phoneEventP->acknowledge ) );
      }
    }
   default:
     break;
  }
  return( errNone );
  }
```

If a message is detected, handleMessage is used to process the message into the appropriate action.

```
static Err handleMessage( PhnDatabaseID id, Boolean * handled )
/***************************************************************/
// This routine will construct a_palm_msg and then call
// PalmLsnProcess to process it.
{

  a_palm_msg *      ulMsg;
  Err               ret;
  Boolean           newlyLoaded;
  PhnAddressList    addrList;
  PhnAddressHandle  addrH;
  MemHandle         msgBodyH;
  Char *            msgSender;
  Char *            msgBody;
  UInt32            msgTime;
  Char              configDb[ dmDBNameLength ];
  UInt16            libRef   = 0;
  // CDMA workaround recommended by Handspring
  DmOpenRef         openRef   = 0;
  *handled = false;

  // Allocate a message structure for passing over
  // to PalmLsnProcess later

  ulMsg = PalmLsnAllocate();
  if( ulMsg == NULL ) {
    return( sysErrNoFreeRAM );
   }
```

```
 // Load the phone library

 ret = findOrLoadPhoneLibrary( &libRef, &newlyLoaded );
 if( ret != errNone ) {
  goto done;
 }
 openRef = PhnLibGetDBRef( libRef );
 // Retrieve sender of the message

 ret = PhnLibGetAddresses( libRef, id, &addrList );
 if( ret != errNone ) {
     goto done;
 }
 ret = PhnLibGetNth( libRef, addrList, 1, &addrH );
if( ret != errNone ) {
   PhnLibDisposeAddressList( libRef, addrList );
   goto done;
 }

 msgSender = PhnLibGetField( libRef, addrH, phnAddrFldPhone );
 if( msgSender != NULL ) {
   ret = PalmLsnDupSender( ulMsg, msgSender );
   MemPtrFree( msgSender );
 }
 PhnLibDisposeAddressList( libRef, addrList );
 if( ret != errNone ) {
   goto done;
 }

 // Retrieve message time

 ret = PhnLibGetDate( libRef, id, &msgTime );
 if( ret != errNone ) {
  goto done;
 }
 ret = PalmLsnDupTime( ulMsg, msgTime );
 if( ret != errNone ) {
   goto done;
 }
 // Retrieve the entire message body

 ret = PhnLibGetText( libRef, id, &msgBodyH );
 if( ret != errNone ) {
     goto done;
 }
 msgBody = (Char *)MemHandleLock( msgBodyH );
 ret = PalmLsnDupMessage( ulMsg, msgBody );
```

```
  // msgBodyH must be disposed of by the caller

  MemHandleUnlock( msgBodyH );
  MemHandleFree( msgBodyH );
  if( ret != errNone ) {
    goto done;
  }

  // Get the configuration database name

  PalmLsnGetConfigFileName( configDb );

  // Call PalmLsnProcess to process the message

  ret = PalmLsnProcess( ulMsg, configDb, NULL, handled );
done:
  if( ulMsg != NULL ) {
    PalmLsnFree( ulMsg );
  }
  PhnLibReleaseDBRef( libRef, openRef );

  // Unload the phone library before any possible application
         switch

      if( newlyLoaded ) {
    unloadPhoneLibrary( libRef );
    newlyLoaded = false;
  }
  return( ret );
}
```

CHAPTER 8

# Tutorial: Server-Initiated Synchronization

About this chapter

This chapter demonstrates how to configure a consolidated and remote database for server-initiated synchronization. It shows how to set up a simple synchronization from scratch using an Adaptive Server Anywhere consolidated database.

Several sample implementations of server-initiated synchronization are included in the SQL Anywhere Studio installation. They are fully documented in readmes and code comments. To locate the sample applications, navigate to the *Samples\MobiLink* directory in your SQL Anywhere Studio install path. All server-initiated synchronization sample directories start with the prefix SIS_.

Contents

# Server-initiated synchronization using the car dealer sample

Car dealer sample

This tutorial is based on the sample located in the *Samples\MobiLink\SIS_CarDealer* subdirectory in your SQL Anywhere installation. This tutorial uses a single client located on a desktop computer. However, server initiated synchronization Listeners can be installed on other platforms, such as Windows CE and Palm OS. For more information, consult the SIS_CarDealer sample and MobiLink server-initiated synchronization documentation.

## Tutorial overview

Following is an overview of the steps required to set up MobiLink server-initiated synchronization using the Car Dealer sample. The purpose of this lesson is to provide an overview of what's to come; later steps will describe how to do each of these steps.

### ❖ Set up the consolidated database and Notifier utility

1. Create an Adaptive Server Anywhere database with the following schema:
   - A table used for synchronization.

     For this tutorial, the table is called Dealer and stores the name and rating of automobile manufacturers.
   - Synchronization logic for download-only synchronization.

     For this tutorial, you will implement the download_cursor synchronization script.
   - A table to store push requests.

     Populating this table triggers remote notification.

2. Configure the Notifier utility.

   Provide logic to handle the begin_poll, request_cursor, and request_delete events.

3. Configure gateways and carriers.

   This tutorial uses the default gateway settings for a UDP Listener.

4. Start the MobiLink synchronization server with the -notifier option.

### ❖ Set up the remote database and Listener utility

1. Create an Adaptive Server Anywhere database with a table used for synchronization.

   This table will be synchronized with the consolidated database Dealer table.

2. Create a remote synchronization publication, synchronization user, and synchronization subscription.

3. Create *dblsn.txt*, the command-line file for the MobiLink Listener.

4. Start a local UDP Listener.

### ❖ Issue push requests

1. To issue push requests, you can insert data directly into the push request table or make other changes that will cause the Notifier begin_poll event to populate the push request table. Each request will cause the Notifier to send a message using the request_cursor event.

   If the message of a push request matches a message defined in the Listener command file, for example ‘sync’, the remote database will synchronize (or take the corresponding action).

# Lesson 1: Set up the consolidated database

In this lesson, you create a consolidated database with the scripts required for synchronization.

One way to create an Adaptive Server Anywhere database is to use the dbinit command-line utility. In this tutorial, the consolidated database is called **cons**.

### ❖ To create and start a new Adaptive Server Anywhere consolidated database

1. At a command prompt, navigate to the directory where you would like to create the database.

2. Type the following command to create the database:

```
dbinit cons.db
```

3. Now, to start the database, type:

```
dbeng9 cons.db
```

Generate the consolidated database schema

The consolidated database schema includes a Dealer table, a download_cursor synchronization script, and a table and stored procedure to generate server-initiated synchronization push requests.

### ❖ To add the Dealer table and download_cursor synchronization script

1. Connect to the consolidated database:
   - In Sybase Central, select the Adaptive Server Anywhere 9 plug-in. From the File menu, choose Connect.

     The Connect dialog appears.
   - On the Identification tab, type **DBA** as the User ID and **SQL** as the Password. On the Database tab, type **cons** as the Server Name.
   - Click OK to connect.

2. Start Interactive SQL:
   - In Sybase Central, select the cons database. From the File menu, choose Open Interactive SQL.

3. Install the Dealer table and download_cursor synchronization script.
   - Execute the following commands in Interactive SQL:

```
/* the dealer table */
create table Dealer (
    name       varchar(10) not null primary key,
    rating       varchar(5),
    last_modified   timestamp default timestamp
)
go
insert into Dealer(name, rating) values ( 'Audi', 'a');
insert into Dealer(name, rating) values ( 'Buick', 'b');
insert into Dealer(name, rating) values ( 'Chrysler',
         'c' );
insert into Dealer(name, rating) values ( 'Dodge', 'd');
insert into Dealer(name, rating) values ( 'Eagle', 'e');
insert into Dealer(name, rating) values ( 'Ford', 'f');
insert into Dealer(name, rating) values ( 'Geo', 'g');
insert into Dealer(name, rating) values ( 'Honda', 'h');
insert into Dealer(name, rating) values ( 'Isuzu', 'i');
go

/* the download_cursor synchronization script */
call ml_add_table_script( 'sis_ver1', 'Dealer',
         'download_cursor',
'SELECT * FROM Dealer WHERE last_modified >= ?' )
go
```

Further reading

☞ For more information about the topics in this lesson, see:

- "The Initialization utility" [*ASA Database Administration Guide*, page 530]
- "The database server" [*ASA Database Administration Guide*, page 116]
- "Using Interactive SQL" [*Introducing SQL Anywhere Studio*, page 217]
- "CREATE TABLE statement" [*ASA SQL Reference*, page 407]
- "Writing Synchronization Scripts" [*MobiLink Administration Guide*, page 227]
- "download_cursor table event" [*MobiLink Administration Guide*, page 371]

# Lesson 2: Create a push request table

The Notifier sends a message to a remote database when it detects a push request. In a typical implementation, you add a push request table to your consolidated database.

### ❖ To create a table for push requests

1. Execute the following command in Interactive SQL:

```
CREATE TABLE PushRequest (
    req_id   INTEGER DEFAULT AUTOINCREMENT PRIMARY KEY,
    mluser   VARCHAR(128),
    subject   VARCHAR(128),
    content VARCHAR(128),
    resend_interval VARCHAR(30) DEFAULT '20s',
    time_to_live VARCHAR(30) DEFAULT '1m',
    status   VARCHAR(128) DEFAULT 'created'
)
go
```

2. Close Interactive SQL.

Further reading

☞ For more information about the topics in this lesson, see:

- “Push requests” on page 10
- “Introducing Server-Initiated Synchronization” on page 1
- “Components of server-initiated synchronization” on page 4

## Lesson 3: Configure the Notifier

In this lesson you configure three Notifier properties to influence how the Notifier creates push requests, transmits the requests to remote Listeners, and cleans up expired requests.

### ❖ To configure the Notifier utility

1. Connect to the consolidated database using the MobiLink Synchronization plug-in:
   - ♦ Open Sybase Central.
   - ♦ In the left pane, select the MobiLink Synchronization 9 plug-in. From the File menu, choose Connect.

     The Connect dialog appears.
   - ♦ On the Identification tab, type **DBA** as the User ID and **SQL** as the Password. On the Database tab, type **cons** as the Server Name.
   - ♦ Click OK to connect.

2. Add a new Notifier.

   In the left-pane, open the Notification folder and select the Notifiers folder. In the right-pane double-click Add Notifier.

   The Add a New Notifier dialog appears.

3. Name the Notifier **CarDealerNotifier**. Click Finish.

4. Enter the begin_poll event script.

   The Notifier detects changes in the consolidated database and creates push requests using the begin_poll event. In this case, the begin_poll script will populate the PushRequest table if changes occur in the Dealer table and when a remote database is not up-to-date.
   - ♦ In the right pane, select the CarDealerNotifier. From the File menu, choose Properties.

     The CarDealerNotifier Notifier Properties dialog appears.
   - ♦ Click the Logic tab. From the dropdown menu, choose begin_poll.
   - ♦ Enter the following for the begin_poll script:

```
--
-- Insert the last consolidated database
 -- modification date into @last_modified
 --
declare @last_modified timestamp;
select max( last_modified ) into @last_modified from
         Dealer;
```

```
--
-- Delete processed requests if the mluser is up-to-date
--
delete from PushRequest
  from PushRequest as p, ml_user as u, ml_subscription
         as s
where
      p.status = 'processed'
and
  u.name = p.mluser
and
  u.user_id = s.user_id
and
  @last_modified <= greater( s.last_upload_time, s.last_
         download_time );
--
-- Insert new requests when a device is not up-to-date
--
insert into PushRequest( mluser, subject, content )
 select u.name, 'sync', 'ignored'
 from ml_user as u, ml_subscription as s
where
 u.name in ( select ml_user from ml_listening where
         listening = 'y' )
and
 u.user_id = s.user_id
and
 @last_modified > greater( s.last_upload_time, s.last_
         download_time )
and
 u.name not like '%-dblsn'
and
 not exists( select * from PushRequest
   where PushRequest.mluser = u.name
    and PushRequest.subject = 'sync')
```

In the first major section of the begin_poll script, processed requests from the PushRequest table are eliminated if a device is up to date:

```
@last_modified <= greater( s.last_upload_time, s.last_
         download_time)
```

@last_modified is the maximum modification date in the consolidated database Dealer table. The expression greater( s.last_upload_time, s.last_download_time) represents the last synchronization time for a remote database.

You can also delete push requests directly using the request_delete event. However, the begin_poll event, in this case, ensures that expired or implicitly dropped requests are not eliminated before a remote database synchronizes.

The next section of code checks for changes in the last_modified column of the Dealer table and issues push requests for all active listeners (listed in the ml_listening table) that are not up to date:

```
@last_modified > greater( s.last_upload_time, s.last_
         download_time)
```

When populating the PushRequest table, the begin_poll script sets the subject to ‘sync’.

5. Enter the request_cursor script.

   The request_cursor script fetches push requests. Each push request determines what information is sent in the message, and which remote databases receive the information.

   ♦ From the dropdown menu, choose request_cursor.

   ♦ Enter the following code for the request_cursor script:

   ```
   select
      p.req_id,
      'Default-DeviceTracker',
      p.subject,
      p.content,
      p.mluser,
      p.resend_interval,
      p.time_to_live
   from PushRequest as p
   ```

   The PushRequest table supplies rows to the request_cursor script.

   The order and values in the request_cursor result set is significant. The second parameter, for example, defines the default gateway Default-DeviceTracker. A device tracker gateway keeps track of how to reach users and automatically selects UDP or SMTP to connect to remote devices.

6. Enter the request_delete script.

   The request_delete notifier event specifies cleanup operations. Using this script, the Notifier can automatically remove implicitly dropped and expired requests.

   ♦ From the dropdown menu, choose request_delete.

   ♦ Enter the following for the request_delete script:

   ```
   update PushRequest set status='processed' where req_id =
            ?
   ```

   Instead of deleting the row, this request_delete script updates the status of a row in the PushRequest table to ‘processed’.

7. Click OK to save the Notifier properties.

Further reading

☞ For more information about the topics in this lesson, see:

- “ Managing Databases with Sybase Central” [*Introducing SQL Anywhere Studio*, page 241]
- “request_delete property” on page 65
- “begin_poll property” on page 58
- “ml_listening” [*MobiLink Administration Guide*, page 509]
- “Device tracking” on page 22
- “Listener options for device tracking” on page 23
- “request_cursor property” on page 64
- “request_delete property” on page 65

## Lesson 4: Configure gateways and carriers

Gateways are the mechanisms for sending messages. You can define UDP gateways and SMTP gateways. Alternatively, you can use a device tracker gateway. With device tracking, MobiLink keeps track of how to reach users, and automatically decides to use a UDP or SMTP gateway.

In this tutorial you use a default device tracker gateway, so no confirguration is necessary.

Further reading

☞ For more information about the topics in this lesson, see:

- ♦ "Device tracker gateway properties" on page 68
- ♦ "Gateways and carriers" on page 20

# Lesson 5: Define an ODBC data source

Use the Adaptive Server Anywhere 9.0 driver to define an ODBC data source for the database.

## ❖ To define an ODBC data source for the consolidated database

1. Start the ODBC Administrator:

   From the Start menu, choose Programs ➤ SQL Anywhere 9 ➤ Adaptive Server Anywhere ➤ ODBC Administrator.

   The ODBC Data Source Administrator appears.

2. On the User DSN tab, click Add.

   The Create New Data Source dialog appears.

3. Select Adaptive Server Anywhere 9.0 and click Finish.

   The ODBC Configuration for Adaptive Server Anywhere 9 dialog appears.

4. On the ODBC tab, type the Data source name **sis_cons**. On the Logic tab, type **DBA** for the User ID and **SQL** for the Password. On the Database tab, type **cons** for the Server Name.

5. Click OK.

Further reading

☞ For more information about the topics in this lesson, see:

- ♦ "Working with ODBC data sources" [*ASA Database Administration Guide,* page 53]

☞ If you are using a consolidated database other than Adaptive Server Anywhere, see "Introduction to iAnywhere Solutions ODBC Drivers" [*ODBC Drivers for MobiLink and Remote Data Access,* page 1].

## Lesson 6: Start the MobiLink server

### ❖ To run the MobiLink synchronization server (dbmlsrv9)

1. At a command prompt, navigate to the directory of your consolidated database. Type the following on a single line:

```
dbmlsrv9
   -notifier
   -c "dsn=sis_cons"
   -o ml.log
   -fr
   -v+
   -zu+
   -x tcpip
```

The following table describes each option used with the dbmlsrv9 utility. The options -o and -v provide debugging and troubleshooting information. Using these logging options is appropriate in a development environment. For performance reasons, -v is typically not used in production.

| Option | Description |
|---|---|
| -notifier | Starts the Notifier for server-initiated synchronization.<br>☞ See “-notifier option” [*MobiLink Administration Guide*, page 202]. |
| -c | Specifies a connection string.<br>☞ See “-c option” [*MobiLink Administration Guide*, page 196]. |
| -o | Specifies the message log file *ml.log*.<br>☞ See “-o option” [*MobiLink Administration Guide*, page 203]. |
| -fr | Prevents the MobiLink synchronization server from aborting if a synchronization does not include at least one script that uploads and one script that downloads data. This option is required for this tutorial which uses download-only synchronization.<br>☞ See “-fr option” [*MobiLink Administration Guide*, page 200]. |
| -v+ | The -v option specifies what information is logged. Using -v+ sets maximum verbose logging.<br>☞ See “-v option” [*MobiLink Administration Guide*, page 211]. |

| Option | Description |
|---|---|
| -zu+ | Adds new users automatically.<br>☞ See “-zu option” [*MobiLink Administration Guide,* page 222]. |
| -x | Sets the communications protocol and protocol options for MobiLink clients.<br>☞ See “-x option” [*MobiLink Administration Guide,* page 214]. |

A dialog appears to indicate the MobiLink synchronization server is ready to handle requests. The Notifier utility also appears.

Further reading

☞ For more information about the topics in this lesson, see:

- “Running the MobiLink synchronization server” [*MobiLink Administration Guide,* page 11]
- “MobiLink Synchronization Server Options” [*MobiLink Administration Guide,* page 189]

## Lesson 7: Set up a remote database

MobiLink server-initiated synchronization is designed for synchronization involving a consolidated database server and a large number of mobile databases. In this lesson, you create an Adaptive Server Anywhere remote database, create a synchronization publication, user, and subscription. Then, you create the command file for the Listener utility and start the Listener.

### ❖ To create and start a new Adaptive Server Anywhere remote database

1. At a command prompt, navigate to the directory where you would like to create the database.
2. Type the following command to create the database:

   ```
   dbinit rem1.db
   ```

3. Now, to start the database, type:

   ```
   dbeng9 rem1.db
   ```

### ❖ To generate the remote database schema

1. Connect to the database:
   - In Sybase Central, select the Adaptive Server Anywhere 9 plug-in. From the File menu, choose Connect.

     The Connect dialog appears.
   - On the Identification tab, type **DBA** as the User ID and **SQL** as the Password. On the Database tab, type **rem1** as the Server Name.
   - Click OK to connect.
2. Start Interactive SQL:
   - In Sybase Central, select the rem1 database. From the File menu, choose Open Interactive SQL.
3. Create the Dealer table.
   - Execute the following command in Interactive SQL:

     ```
     create table Dealer (
         name       varchar(10) not null primary key,
         rating      varchar(5),
         last_modified   timestamp default timestamp
     )
     go
     ```

4. Create a synchronization publication, user, and subscription.

♦ Execute the following commands in Interactive SQL:

```
CREATE PUBLICATION car_dealer_pub (table Dealer);
CREATE SYNCHRONIZATION USER sis_user1;
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO car_dealer_pub
     FOR sis_user1
     OPTION scriptversion='sis_ver1';
```

Further reading

☞ For more information about the topics in this lesson, see:

♦ "Introducing MobiLink Clients" [*MobiLink Clients*, page 3]
♦ "The Initialization utility" [*ASA Database Administration Guide*, page 530]
♦ "CREATE TABLE statement" [*ASA SQL Reference*, page 407]
♦ "Publishing data" [*MobiLink Clients*, page 64]
♦ "CREATE PUBLICATION statement" [*ASA SQL Reference*, page 385]
♦ "CREATE SYNCHRONIZATION USER statement [MobiLink]" [*ASA SQL Reference*, page 404]
♦ "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" [*ASA SQL Reference*, page 402]
♦ "Script versions" [*MobiLink Administration Guide*, page 239]

## Lesson 8: Configure the Listener

The Listener runs on remote devices. It receives messages from the Notifier and processes them into actions. For example, the Listener will start dbmlsync if it receives the message “sync” when the following dblsn option is specified:

```
-l "subject=sync;action='run dbmlsync.exe...'
```

A convenient way to configure the Listener is to store command line options in a text file. For example, if you store the settings in *mydblsn.txt*, you can start the Listener by typing:

```
dblsn @mydblsn.txt
```

Alternatively, if you type dblsn without any parameters, dblsn will use *dblsn.txt* as the default argument file.

### ❖ To create and start the MobiLink Listener

1. Create a text file *mydblsn.txt* with the following contents.

```
#----------------------------------
# Verbosity level
-v2

# Show notification messages in console and log
-m

# Polling interval, in seconds
-i 3

# Truncate, then write output to dblsn.log
-ot dblsn.log
# Mobilink address and connect parameter for dblsn
-x "host=localhost"

# Enable device tracking and specify the MobiLink user name.
 -t+ sis_user1

# Message handlers
# Synchronize using dbmlsync
-l "subject=sync;
action='start dbmlsync.exe
  -c eng=rem1;uid=dba;pwd=sql
 -ot dbmlsyncOut.txt -k';"
```

   ♦ Save the file as *mydblsn.txt.*

2. Start the Listener.

   At a command prompt, navigate to the directory of your Listener command file.

Start the listener by typing:

```
dblsn @mydblsn.txt
```

A dialog appears indicating the Listener is running and has uploaded device tracking information to the MobiLink synchronization server.

When tracking information is uploaded to the consolidated database, you should notice a new entry in the MobiLink synchronization server window. This information relays the successful initial communication between the Listener and the MobiLink synchronization server.

**Further reading**

☞ For more information about the topics in this lesson, see:

- “Listeners” on page 28
- “The Listener utility” on page 38
- “Storing Listener options” on page 33

## Lesson 9: Issue push requests

For server-initiated synchronization, you can issue push requests by populating the PushRequest table directly, or making a change in the Dealer table. In the latter case, the Notifier begin_poll script will detect the change in the Dealer table and populate the PushRequest table.

In either case, the PushRequest table supplies rows to the Notifier request_cursor script, which determines how remote devices receive messages.

### ❖ To insert a push request directly into the PushRequest table prompting server-initiated synchronization

1. From Interactive SQL, connect to the cons.db database and enter the following:

   ```
   INSERT INTO pushrequest(mluser,subject,content)
     VALUES ('sis_user1','sync','not used');
     COMMIT;
   ```

2. Wait a few seconds for the synchronization to occur.

When populated, the PushRequest table supplies rows to the Notifier's request_cursor script. The request_cursor script determines what information is sent in the message, and which remote devices receive the information.

### ❖ To make a change in the consolidated database Dealer prompting server-initiated synchronization

1. From Interactive SQL, enter the following:

   ```
   UPDATE Dealer SET RATING = 'B'
      WHERE name = 'Geo'; commit;
   ```

2. Wait a few seconds for the synchronization to occur.

In this case, the Notifier begin_poll script detects changes in the dealer table and will populate the push request table appropriately. As before, once the PushRequest table is populated, the Notifier request_cursor script determines what information is sent in the message, and which remote devices receive the information.

Further reading

☞ For more information about the topics in this lesson, see:

♦ "Creating push requests" on page 12
♦ "INSERT statement" [*ASA SQL Reference*, page 528]
♦ "UPDATE statement" [*ASA SQL Reference*, page 650]

# Index