MobiLink
Server-Initiated Synchronization

Published: October 2006

Copyright and trademarks

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 want to use this advanced feature.

Before you begin

☞ For more information about MobiLink, see MobiLink - Getting Started [*MobiLink - Getting Started*].

SQL Anywhere 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 documentation

The complete SQL Anywhere documentation is available in two forms: an online form that combines all books, and as separate PDF files for each book. Both forms of the documentation contain identical information and consist of the following books:

- **SQL Anywhere 10 - Introduction** This book introduces SQL Anywhere 10—a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 10 - Changes and Upgrading** This book describes new features in SQL Anywhere 10 and in previous versions of the software.
- **SQL Anywhere Server - Database Administration** This book covers material related to running, managing, and configuring SQL Anywhere databases. It describes database connections, the database server, database files, security, backup procedures, security, and replication with Replication Server, as well as administration utilities and options.
- **SQL Anywhere Server - SQL Usage** 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.
- **SQL Anywhere Server - SQL Reference** This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.
- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools.
- **SQL Anywhere 10 - Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.
- **MobiLink - Getting Started** This manual introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Server Administration** This manual describes how to set up and administer MobiLink applications.
- **MobiLink - Client Administration** This manual describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases.
- **MobiLink - Server-Initiated Synchronization** 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.

- **QAnywhere** This manual describes QAnywhere, which defines a messaging platform for mobile and wireless clients as well as traditional desktop and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **SQL Anywhere 10 - Context-Sensitive Help** This manual provides context-sensitive help for the Connect dialog, the Query Editor, the MobiLink Monitor, the SQL Anywhere Console utility, the Index Consultant, and Interactive SQL.
- **UltraLite - Database Management and Reference** This manual introduces the UltraLite database system for small devices.
- **UltraLite - AppForge Programming** This manual describes UltraLite for AppForge. With UltraLite for AppForge you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS, Symbian OS, or Windows CE.
- **UltraLite - .NET Programming** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This manual describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows CE, or Windows XP.
- **UltraLite - C and C++ Programming** This manual describes UltraLite C and C++ programming interfaces. With UltraLite you can develop and deploy database applications to handheld, mobile, or embedded devices.

Documentation formats

SQL Anywhere provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere documentation, including 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 10 ► 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 or on your installation CD.

- **PDF files** The complete set of SQL Anywhere books is provided as a set of Adobe Portable Document Format (pdf) files, viewable with Adobe Reader.

  On Windows, the PDF books are accessible from the online books via the PDF link at the top of each page, or from the Windows Start menu (Start ► Programs ► SQL Anywhere 10 ► Online Books - PDF Format).

On Unix, the PDF books are accessible on your installation CD.

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 uppercase, 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.

File name conventions

The documentation generally adopts Windows conventions when describing operating-system dependent tasks and features such as paths and file names. In most cases, there is a simple transformation to the syntax used on other operating systems.

- **Directories and path names** The documentation typically lists directory paths using Windows conventions, including colons for drives and backslashes as a directory separator. For example,

  ```
  MobiLink\redirector
  ```

  On Unix, Linux, and Mac OS X, you should use forward slashes instead. For example,

  ```
  MobiLink/redirector
  ```

- **Executable files** The documentation shows executable file names using Windows conventions, with the suffix *.exe*. On Unix, Linux, and Mac OS X, executable file names have no suffix. On NetWare, executable file names use the suffix *.nlm*.

  For example, on Windows, the network database server is *dbsrv10.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv10*. On NetWare, it is *dbsrv10.nlm*.

- **install-dir** The installation process allows you to choose where to install SQL Anywhere, and the documentation refers to this location using the convention *install-dir*.

  After installation is complete, the environment variable SQLANY10 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*). SQLANYSH10 specifies the location of the directory containing components shared by SQL Anywhere with other Sybase applications.

  For more information on the default location of *install-dir*, by operating system, see "File Locations and Installation Settings" [*SQL Anywhere Server - Database Administration*].

- **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

  After installation is complete, the environment variable SQLANYSAMP10 specifies the location of the directory containing the samples (*samples-dir*). From the Windows Start menu, choosing Programs ► SQL Anywhere 10 ► Sample Applications and Projects opens a Windows Explorer window in this directory.

  For more information on the default location of *samples-dir*, by operating system, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%envvar%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$envvar* or *${envvar}*.

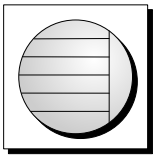
Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

Graphic icons

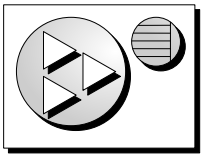
The following icons are used in this documentation.

- A client application.

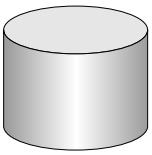
- A database server, such as SQL Anywhere.



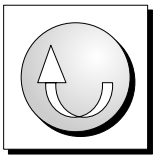
- An UltraLite application.



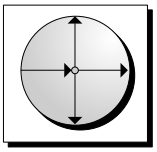
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



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



- A Sybase Replication Server



- 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. You can find this information by entering **dbeng10 -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
- 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 email comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails 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

Contents

About this chapter

This chapter provides an overview of server-initiated synchronization.

Introduction to server-initiated synchronization

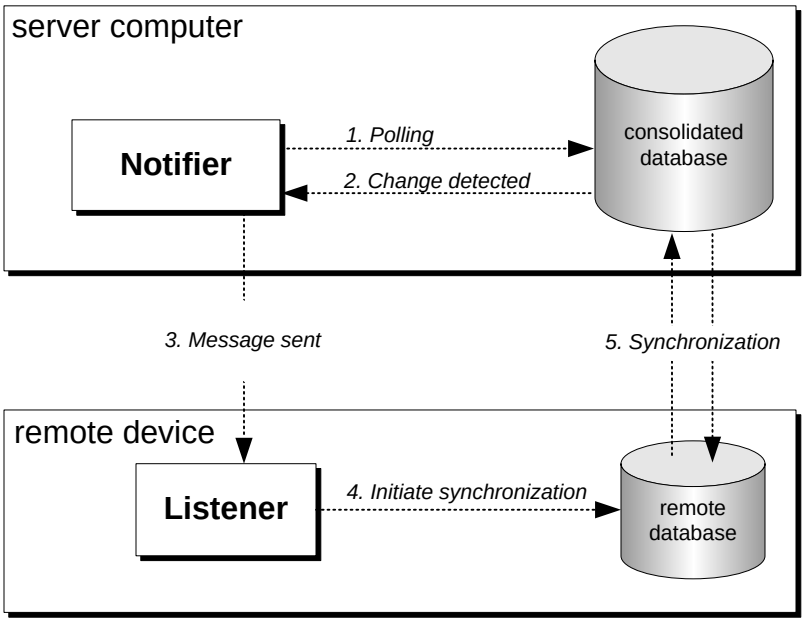
Server-initiated synchronization allows you to initiate MobiLink synchronization from the consolidated database. This means you can push data updates to remote databases, as well as cause remote databases to upload data to the consolidated database. This MobiLink component provides programmable options 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 the message. By default, it uses the same protocol as you use for synchronization. Alternatively, you can set up UDP or SMTP gateways for it to use.
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 31.

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 program running on the same computer as the MobiLink 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 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 17.

- **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. To receive a message, the remote device must be on and the Listener must be started.

  For more information, see "Listener Utility" on page 35.

  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 47.

- **Gateways** provide an interface to send messages from the Notifier to the Listener. You can send messages using a SYNC gateway, a UDP gateway, or an SMTP gateway. The SYNC gateway uses the same protocol as your MobiLink synchronizations.

  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 19.

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 the MobiLink table in SQL Anywhere 10.0.0 Components by Platform.

The Listener is supported on all supported Windows platforms and Palm OS.

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

- **SYNC gateway and UDP gateway** have been tested on the following platforms:
  - Pocket PC 2002
  - Windows 2000 and XP
- **SMTP gateway** transmits messages through an email-to-SMS conversion that is provided by wireless carriers. This has been tested on the following platforms:
  - Windows Mobile 2003 and up Phone Edition
  - 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 5.2 on the Treo 600
  - Palm 5.4 on the Treo 650

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

You can use the Windows or Palm Listener SDK to create Listeners for unsupported platforms or devices. For more information, see "MobiLink Listener SDK for Palm" on page 83.

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 receive notifications.
- The IP address on the remote device needs to be reachable from the MobiLink server.

Limitations of Listeners on Windows, including Windows CE

- 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 23.

Quick start

The following steps provide an overview of the tasks required to set up server-initiated synchronization, assuming that you already have MobiLink synchronization set up.

This is the manual method. To set up server-initiated synchronization in Sybase Central, see "Setting up server-initiated synchronization in Model mode" [*MobiLink - Getting Started*].

♦ 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 17.

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

   ☞ See "Listeners" on page 27.

4. You transmit via a gateway.

   - If you are using the default SYNC gateway, which uses the same protocol as you use for synchronization, you may not need to make any changes to the default gateway settings.
   - If you are using UDP, you may be able to send messages via the default settings.
   - If you are sending SMS notifications, you need to configure gateways and carriers and also specify SMS listening libraries.

   ☞ See "Gateways and carriers" on page 19 and "Listening libraries" on page 44.

Other resources for getting started

- Sample applications are installed to *samples-dir\MobiLink\SIS_**. For more information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

CHAPTER 2

Setting Up Server-Initiated Synchronization

Contents

About this chapter

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

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 push request table to your consolidated database. You populate the push request table when a change is detected on your consolidated database, and use the request_cursor Notifier event to send push requests to remote Listeners. The request_cursor Notifier event must receive the following columns in the following order:

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. For a SYNC gateway or DeviceTracker gateway, it is the MobiLink user name of the Listener, or other MobiLink user names that you register using dblsn -t+. For an SMTP gateway without device tracking, it is an email address. For a UDP gateway without device tracking, it is an IP address or host name, optionally followed by a colon and port number.
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`. 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.

Column	Description
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`. If this value is 0, NULL, or not specified, the default is to send exactly once, with no resend.

Note: push requests can also be stored in temporary tables and across multiple tables.

☞ For more information about addressing notifications when you are using device tracking, see "Listener options for device tracking" on page 22.

Example

Following is a SQL 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 57.

- 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" [*SQL Anywhere Server - SQL Usage*].

- 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 - Server Administration*].

- 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 59.

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 60.

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

SQL 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" [*SQL Anywhere Server - SQL Reference*].

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.sybase.com"
```

On a different device, start a SQL 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 opens, showing the Sybase home page.

To make this example work with one device, use localhost as the first parameter 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 can store property settings in the MobiLink system table ml_property. There are two ways to do this:

- Use the Notification folder in the MobiLink plug-in in Sybase Central. 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 server-initiated synchronization tab in Model mode in the MobiLink plug-in in Sybase Central.
- Use the stored procedure ml_add_property.

  For more information, see "ml_add_property" [*MobiLink - Server Administration*].

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. If you use the properties file, you cannot use the default SYNC gateway.

For more information, see:

- "Notifier properties file" on page 15
- "SYNC gateway properties" on page 67

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 server for them to take effect.

Properties

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

- "Common properties" on page 52
- "Notifier properties" on page 53
- "Gateway properties" on page 64
- "Carrier properties" on page 70

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 14.

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, *template.notifier*, located in *samples-dir\MobiLink*. For more information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

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 mlsrv10 with the -notifier option. Following is a partial mlsrv10 command line:

```
mlsrv10 ... -notifier "c:\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 14.

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
\f	\u000c: form feed, FF
\r	\u000d: carriage return, CR
\"	\u0022: double quote, "

Escape sequence	Description
\'	\u0027: single quote, '
\\	\u005c: backslash, \
\uhhhh	Unicode character (hexadecimal)
\xhh	\xhh: ASCII character (hexadecimal)
\e	\u001b: Escape, ESC

Example

For a fully commented sample Notifier properties file, see *samples-dir\MobiLink\template.notifier*. For more information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

Note
If you want to use the default SYNC gateway, you cannot store Notifier configuration settings in this properties file. You must store them in the database. See "Setting properties" on page 14.

Notifiers

The Notifier runs on the same computer as the MobiLink 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 server. Each Notifier keeps one database connection open all the time.

☞ For an example of multiple Notifiers, see the sample located in the *samples-dir\MobiLink \SIS_MultipleNotifier*. For more information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

Starting the Notifier

You start Notifiers on the mlsrv10 command line. To start the Notifier, use the mlsrv10 option **-notifier**. Optionally, you can also specify the name of your Notifier properties file, if you have one.

Following is a partial mlsrv10 command line:

```
mlsrv10 ... -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 14.

☞ For information about how properties are applied, see "Setting properties" on page 14.

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

☞ 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.

Another way to send notifications

As an alternative to the Notifier, you can also use the sa_send_udp stored procedure to send simple notifications if you are using a SQL Anywhere consolidated database.

☞ For more information about sa_send_udp, see "Notifying the Listener with sa_send_udp" on page 12.

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 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. The Notifier uses the result set returned by the request_cursor to send push requests to remote Listeners.

  Note: The request_cursor property is the only required property. For more information, see "request_cursor property" on page 59.
- Delete push requests with the request_delete property.

☞ For a complete list of Notifier properties, see "MobiLink Notification Properties" on page 51.

☞ For information about how to set Notifier properties, see "Setting properties" on page 14.

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
    For all requests expired before required confirmation (
         error_handler
    )
  request_delete
  end_poll
)
end_connection
```


Gateways and carriers

Gateways are the mechanisms for sending messages. You can define SYNC gateways, UDP gateways, and SMTP gateways. In most cases, you will also use a device tracker gateway that automatically decides which gateway to use.

- **Device tracker gateway** When you use the device tracker gateway, the address in your push request is the MobiLink user name of the Listener.

  Device tracking allows automatic management of remote device address changes. When using device tracking, the SYNC gateway is attempted first, with fallback to the UDP and then the SMTP gateways if they are enabled.

  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.

  ☞ See "Device tracking" on page 21 and "Device tracker gateway properties" on page 64.

- **SYNC gateway** The SYNC gateway can use the same protocol as you use for synchronization. The connection is persistent, and is used for all Listener communication (notifications, confirmation, and device tracking). If you are using SYNC, you may not need to make any changes to the default gateway settings.

  ☞ See "SYNC gateway properties" on page 67.

- **UDP gateway** This gateway allows you to send push requests to remote listeners using UDP. If you are using UDP, you may not need to make any changes to the default gateway settings.

  ☞ See "UDP gateway properties" on page 68.

- **SMTP gateway** You can use the SMTP gateway to send messages to SMS listeners via a wireless carrier's email-to-SMS service. For SMTP, you need to configure an SMTP gateway and carrier.

  ☞ See "SMTP gateway properties" on page 65.

When you use an SMTP gateway, you configure **carriers** to store information about the public wireless carriers that you want to use. Carrier information is used to create SMS email 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 14.

For a detailed list of gateway and carrier properties, see

- "Device tracker gateway properties" on page 64

- "UDP gateway properties" on page 68
- "SMTP gateway properties" on page 65
- "Carrier properties" on page 70

Gateways

There are four 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-SYNC gateway
- Default-UDP gateway
- Default-SMTP gateway

A device tracker gateway can have up to three subordinate gateways: one SYNC, 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 21.

Default-SYNC, Default-UDP, and Default-SMTP are preconfigured with some settings that may work out of the box, especially SYNC and 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 email addresses for each public wireless carrier's email-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 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 70.

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 with an SMTP gateway, 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 SYNC gateway, and if the delivery fails it then attempts using a UDP gateway or an SMTP gateway.

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 23.

If you do not use device tracking, your request_cursor must include a specific 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 do not have to set up a gateway or carrier for it. However, if you want to use email-to-SMS notification, the default SMTP gateway requires configuration.

   ☞ See "Configuring gateways and carriers" on page 19.

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 59.

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 and registering MobiLink users" [*MobiLink - Client Administration*].

4. Start the Listener with the required options.

☞ See "Listener options for device tracking" on page 22.

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 want 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 use that user name as the address in your push requests.

  You can register multiple MobiLink user names with -t+. This is useful if you need to address notifications to different applications on the remote device, such as multiple remote databases.

  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*** To disable a MobiLink user name that was registered for device tracking with -t+, use **-t-**.

- **-u *ML_user_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 and registering MobiLink users" [*MobiLink - Client Administration*].

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

- **-x *connection-parameters*** Use -x to specify how to connect to the MobiLink 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 want 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 "Listener syntax" on page 36.

Example

The following one-line command starts the Listener with device tracking.

```
dblsn -l "subject=sync;action='run dbmlsync.exe -c dsn=rem1'"
 -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 "Listener syntax" on page 36.

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 23.

- 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 25.

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 email 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 for 9.0.0 Listeners or Palm Listeners

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 Palm Listener',
    '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 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 77.

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 70). The third parameter is an IP address for UDP or 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 21.

   ☞ For more information about using ml_set_device_address, see "ml_set_device_address" on page 79.

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 81.

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 SMTP gateway.

  If the device's IP address is never reachable, you may want to stop device tracking on the Listener with the -g option. The -g option is useful when you do not want to use device tracking but you do want delivery confirmation. 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.

  For more information about delivery confirmation, see "confirmation_handler property" on page 54.

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 79.

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 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 21.

See also

- For Listener syntax and options, see "Listener syntax" on page 36.
- For information about Palm devices, see "Listeners for Palm Devices" on page 47.
- For dbmlsync options, see "MobiLink SQL Anywhere Client Utility [dbmlsync]" [*MobiLink - Client Administration*].
- For more information about message handlers, see "Message handlers" on page 28.
- Instead of entering 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 32.

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 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 - Client Administration*].
-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 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 "Listener Utility" on page 35.

Message interpretation

Messages (push requests) 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 email message that is converted by an email to SMS service into a format that varies between different public wireless carriers.

Filters and action variables

The Listener provides filters and action variables to determine how to process an incoming message (push request).

- **Filters** Filters allow the Listener to determine what action to take in response to the subject, content, and other parts of a message (push request). Filters are specified using the dblsn -l option.

  ☞ For information about using the **subject** or **content** filters, see "Using subject and content filters" on page 29.

  ☞ For information about using the **message**, **message_start**, or **sender** filters, see "Using the filters message, message_start, and sender" on page 30.

- **Action variables** Action variables allow you to incorporate parts of a message (push request) in the Listener action.

  ☞ For more information about action variables, see "Action variables" on page 42.

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 email 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.

For SMS messages, 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 deleted by the Listener before filtering.

Filtering by remote ID

For SQL Anywhere remote databases, the first time you synchronize, a **remote ID file** is created that contains the remote ID for the remote database. This file has the same name as the database, but with the extension *.rid*, and is stored in the same directory as the database.

For UltraLite databases, to specify the remote ID file you use the name of the database.

To use the remote ID in a filter, you must use the dblsn -r option and provide the name and location of the remote ID file. Then you can use the $remote_id variable in your filter. If you use -r more than once, the $remote_id refers to the file specified in the -r option just before it.

You can also use the remote ID directly in filters. However, by default the remote ID is a GUID, so unless you provide a more meaningful remote ID name, it is not easy to reference directly.

Example

For example, the following code is a partial dblsn command file. It assumes that you have two databases on the device, a SQL Anywhere database called *business.db* and an UltraLite database called *personal.udb*. In this example, ulpersonal is the window class name of the UltraLite application.

```
-r "c:\app\db\business.rid"
-l "subject=$remote_id;action='dbmlsync.exe -k -c dsn=business';"
-r "c:\ulapp\personal.udb"
-l "subject=$remote_id;action=post dbas_synchronize to ulpersonal;"
```

See also

- -r option in "Listener syntax" on page 36
- $remote_id variable in "Action variables" on page 42
- "Remote IDs" [*MobiLink - Client Administration*]

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 email to SMS conversion service, it would be `Bob@mail.com[Help]Me`. On Fido's email 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 42. 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 email, the sender is the email 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.

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 server that is specified with the dblsn -x option. Although the "best" designation is defined by the path to the MobiLink 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 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

- "Listener Utility" on page 35
- "Action variables" on page 42

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 "Listener syntax" on page 36 and "Listening libraries" on page 44.

☞ For more information about the Listener, see "Listener Utility" on page 35.

Storing Listener options

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

```
dblsn @mydblsn.txt
```

The path to the configuration file must be fully qualified.

☞ For more information about configuration files, see "Using configuration files" [*SQL Anywhere Server - Database Administration*].

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 "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].

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

Default parameters file dblsn.txt

If you enter dblsn without any parameters, dblsn uses *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

Listener Utility

Contents

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 27.

☞ For information about Palm devices, see "Listeners for Palm Devices" on page 47.

Listener syntax

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 27.

☞ For information about Palm devices, see "Listeners for Palm Devices" on page 47.

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 32.

dblsn options	Description
-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, enter `dblsn -d` *filename.dll* `-a ?` or see "Listening libraries" on page 44.
-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 44. 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 overrides the default value.
-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.
-ni	Stop tracking UDP addresses when -x is used. This is useful when you do not want device tracking but you do want delivery confirmation. For information about handling delivery confirmation on the server, see "confirmation_handler property" on page 54.
-ns	For Windows Mobile 2003 and up Phone Edition, the Listener listens for SMS as well as UDP. It uses a filtering mechanism that runs as a system process, so the filtering continues even when the Listener is not running. This option disables this behavior. When you use -ns, the Listener listens by default on UDP only, and you can use SMS listening by specifying a listening library with the -d option.
-nu	Disable default UDP listening.
-o *filename*	Logs output to a file. If -o is not used, output is logged to the console window.

dblsn options	Description
-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.
-pc {**+**|**-**}	Use -pc- to disable persistent connections for receiving notification but continue to use short-lived persistent connections for device tracking and confirmation. By default, persistent connections are enabled (-pc+). If a persistent connection is broken, the Listener attempts to reconnect continuously.
-q	Runs in a minimized window.
-r *remote-id-file*	Identifies a MobiLink remote database that is involved in the responding action of a message handler. When -r is used, the $remote_id variable can be used in message handlers to refer to the remote ID that is contained in *remote-id-file*. This option can simplify references to remote IDs, which by default are GUIDs. If you have multiple databases on the device, you can use this option multiple times. The *remote-id-file* is the full path/name of a file that contains the remote ID. This file is automatically created by dbmlsync after the first synchronization. It uses the same location and name as the database file, and has the extension .rid. For UltraLite databases, use the UltraLite database name as the remote ID file. See "Filtering by remote ID" on page 29.
-t {**+**|**-**} *ml-user-alias*	Register remote databases for notification so that you can address the remote database by name when using device tracking. You can also use the $remote_id variable to identify databases. ☞ See "Listener options for device tracking" on page 22 and "Action variables" on page 42.
-u *Listener-name*	Specifies a MobiLink user name. This name is used when the Listener needs to connect to the MobiLink server, which it does for device tracking, confirmations, and persistent connection. The default MobiLink user name is *device-name*-**dblsn**. MobiLink user names must be registered with the MobiLink server. See "Adding MobiLink user names to the consolidated database" [*MobiLink - Client Administration*].

dblsn options	Description
-v [*level*]	Sets the verbosity level for the dblsn log and console. The *level* can be **0**, **1**, **2**, or **3**: ♦ **0** - show no informational messages (the default). ♦ **1** - show Listener dll messages, basic action tracing steps, and command line options. ♦ **2** - show level 1 plus detailed action tracing steps. ♦ **3** - show level 2 plus polling and listening states. To output notification messages, you must also use -m (see above).
-w *password*	Specifies a password for the *Listener-name*. See "Listener options for device tracking" on page 22.
-x {**http**|**https**|**tcpip**} [**(***keyword***=***value***;...)**]	Specifies the network protocol and protocol options for the MobiLink server. For a list of protocol options, see "MobiLink Client Network Protocol Options" [*MobiLink - Client Administration*]. A connection to the MobiLink server is required for the Listener to send device tracking information and delivery confirmation to the consolidated database, and for the SYNC gateway. See "Listener options for device tracking" on page 22.
-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. See "Listener options for device tracking" on page 22.

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 sends 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.

☞ For information about handling delivery confirmation on the server, see "confirmation_handler property" on page 54.

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 29.

☞ For information about using the **message**, **message_start**, or **sender** filters, see "Using the filters message, message_start, and sender" on page 30.

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 using the $content action variable.

  ```
  "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 Windows 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 Windows message by message contents or by the ID of the Windows 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 Windows 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 - Client Administration*].

- **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 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 27

Action variables

The following Windows 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.

Action variable	Description
$ml_connect	The MobiLink connection parameters specified by the mlsrv10 -x option. The default is an empty string.
$ml_user	The MobiLink user name as specified by dblsn -u, or the default name (*device-name*-**dblsn**).
$ml_password	The MobiLink user name password as specified by dblsn -w, or the new MobiLink user name password if -y is used.
$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.
$remote_id	The remote ID. This variable can only be used when the dblsn -r option is specified. See "Filtering by remote ID" on page 29.
$sender	The sender of the message.
$type	The meaning of this variable is carrier library dependent.
$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.
$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.

Action variable	Description
$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 Windows 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 ?`.

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.
HideWSAErrorBox	Suppresses the error box showing errors on socket operations.
CodePage=*number*	On CE, translates multibyte 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

Contents

About this chapter

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

Palm Listener utilities

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

- Palm Listener Configuration utility (dblsncfg)
- Palm Listener (LsnT600.prc or LsnT650.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 27.

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 49.

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 32.

-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 29.

☞ For information about using the **message**, **message_start**, or **sender** filters, see "Using the filters message, message_start, and sender" on page 30.

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 "Palm action variables" on page 49.

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.

Palm action variables

The following Palm 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, enter **$$**. For example, enter $$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 Listener files are:

- **LsnT600.prc** the Listener on Treo 600
- **LsnT650.prc** the Listener on Treo 650

Currently, the 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. 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 "MobiLink Listener SDK for Palm" on page 83.

CHAPTER 5

MobiLink Notification Properties

Contents

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 14.

Common properties

There is one common property, verbosity.

☞ For more information about setting properties, see "Setting properties" on page 14.

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 14.

Notifier properties

Notifier properties can be set in the Notifier properties file or stored in the MobiLink system tables. The enable and request_cursor properties are required. All other Notifier properties are optional.

There are three types of Notifier property:

Notifier events

The confirmation_handler event occurs asynchronously from the other Notifier events. All of these events are optional.

- "begin_connection property" on page 54
- "confirmation_handler property" on page 54
- "end_connection property" on page 57

Notifier polling events

When the Notifier polls the consolidated database, it invokes the polling events in the following order. All of these events are optional except for the request_cursor.

```
For each poll (
  begin_poll
  shutdown_query
  request_cursor
    For all requests expired before required confirmation (
         error_handler
    )
  request_delete
  end_poll
)
```


- "begin_poll property" on page 57
- "end_poll property" on page 58
- "error_handler property" on page 58
- "request_cursor property" on page 59
- "request_delete property" on page 60
- "shutdown_query property" on page 61

Notifier behavior properties

You can also configure the Notifier to use a variety of settings. All of these properties are optional.

- "connect_string property" on page 61
- "enable property" on page 61
- "gui property" on page 61
- "isolation property" on page 62
- "poll_every property" on page 62

See also

☞ For more information about the Notifier, see "Notifiers" on page 17.

☞ For more information about setting properties, see "Setting properties" on page 14.

Notifier events

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 14
- "isolation property" on page 62

confirmation_handler property

You can implement this property to programmatically handle delivery confirmation information uploaded by remote listeners. If the status parameter is 0, the push request identified by request_id was successfully received by the Listener identified by the remote_device parameters.

You can use the request_option out parameter to take an appropriate action in response to the delivery confirmation. If request_option is 0, the confirmation_handler takes the default Notifier action: the request_delete event is executed to delete the original push request. However, if the Listener device sending the delivery confirmation does not match the Listener device identified by the request_id , the default action is to send the original push request on a secondary gateway.

Note: to enable remote Listeners to upload delivery confirmation information use the dblsn -x option. If you want delivery confirmation but do not want IP tracking, use the dblsn -ni option.

☞ For more information about dblsn -x and -ni, see "Listener syntax" on page 36.

Following are the confirmation_handler parameters. You can use all of the parameters or a subset. This property requires the use of a stored procedure.

Script parameter	Description
request_option (out)	Integer. Controls what the Notifier does to the request after the handler returns. Can be one of: ♦ **0**: Perform default Notifier action based on the value of the status parameter. If status indicates that the responding device is the target one, then the Notifier will delete the request; otherwise the Notifier will attempt to deliver on a secondary gateway. ♦ **1**: Do nothing. ♦ **2**: Execute Notifier.request_delete. ♦ **3**: Attempt to deliver to a secondary gateway.
status (in)	Integer. Summary of the situation. The status can be used during development to catch problems such as incorrect filters and handler attributes. The status can be one of: ♦ **0**: Received and confirmed. ♦ **-2**: Right respondent but the message was rejected. ♦ **-3**: Right respondent and the message was accepted but the action failed. ♦ **-4**: Wrong respondent and the message was accepted. ♦ **-5**: Wrong respondent and the message was rejected. ♦ **-6**: Wrong respondent. The message was accepted and the action succeeded. ♦ **-7**: Wrong respondent. The message was accepted but the action failed.
request_id (in)	Integer. Identifies the request.
remote_code (in)	Integer. Summary reported by the remote Listener. Can be one of: ♦ **1**: Message accepted. ♦ **2**: Message rejected. ♦ **3**: Message accepted and action succeeded. ♦ **4**: Message accepted and action failed.
remote_device (in)	Varchar. Device name of the responding Listener.
remote_mluser (in)	Varchar. MobiLink user name of the responding Listener.
remote_action_return (in)	Varchar. Return code of the remote action.
remote_action (in)	Varchar. Reserved for the action command.
gateway (in)	Varchar. Gateway associated with the request.
address (in)	Varchar. Address associated with the request.
subject (in)	Varchar. Subject associated with the request.
content (in)	Varchar. Content associated with the request.

See also

- Device tracker gateways: “confirm_delivery property” on page 64
- SMTP gateways: “confirm_delivery property” on page 65

- UDP gateways: "confirm_delivery property" on page 68
- "ml_add_property" [*MobiLink - Server Administration*]

Example

In the following example, you create a table called CustomConfirmation and then you log confirmations to it using a stored procedure called CustomConfirmationHandler. In this example, the output parameter request_option is always set to 0, which means that default Notifier handling is used.

```
CREATE TABLE CustomConfirmation(
        error_code            integer,
        request_id            integer,
        remote_code            integer,
        remote_device        varchar(128),
        remote_mluser        varchar(128),
        remote_action_return    varchar(128),
        remote_action        varchar(128),
        gateway            varchar(255),
        address            varchar(255),
        subject            varchar(255),
        content            varchar(255),
        occurAt            timestamp not null default timestamp )

CREATE PROCEDURE CustomConfirmationHandler(
        out    @request_option    integer,
        in @error_code        integer,
        in @request_id        integer,
        in @remote_code        integer,
        in @remote_device        varchar(128),
        in @remote_mluser        varchar(128),
        in @remote_action_return    varchar(128),
        in @remote_action        varchar(128),
        in @gateway            varchar(255),
        in @address            varchar(255),
        in @subject            varchar(255),
        in @content            varchar(255) )

begin
        INSERT INTO CustomConfirmation(
        error_code,
        request_id,
        remote_code,
        remote_device,
        remote_mluser,
        remote_action_return,
        remote_action,
        gateway,
        address,
        subject,
        content )


        VALUES    (
        @error_code,
        @request_id,
        @remote_code,
        @remote_device,
        @remote_mluser,
        @remote_action_return,
        @remote_action,
        @gateway,
        @address,
```



```
            @subject,
            @content );
            SET @request_option = 0;
        end
```

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.

Notifier polling events

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 14.

Example

This example creates a push request for a Notifier called Notifier A. It uses a SQL statement that 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.

To use the stored procedure ml_add_property with a SQL Anywhere consolidated database, use the following command.

```
ml_add_property( 'SIS',
  'Notifier(Notifier A)',
  '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'
);
```

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.

error_handler property

You can implement this property to catch situations where a transmission failed or was not confirmed. For example, when a transmission fails you can cause a line to be inserted in an audit table or a notification sent to someone.

You can capture the following information. You can use all of the parameters or a subset. This property requires the use of a stored procedure.

Script parameter	Description
request_option (out)	Integer. Controls what the Notifier does to the request after the handler returns. Can be one of: ♦ 0: Perform default action based on the error code and log the error. ♦ 1: Do nothing. ♦ 2: Execute Notifier.request_delete. ♦ 3: Attempt to deliver to a secondary gateway.
error_code (in)	Integer. Can be one of: ♦ -1: The request timed out with confirmation of success. ♦ -8: Error during delivery attempt.
request_id (in)	Integer. Identifies the request.
gateway (in)	Varchar. Gateway associated with the request.
address (in)	Varchar. Address associated with the request.
subject (in)	Varchar. Subject associated with the request.
content (in)	Varchar. Content associated with the request.

See also

- "ml_add_property" [*MobiLink - Server Administration*]

Example

In the following example, you create a table called CustomError. You log errors to the table using a stored procedure called CustomErrorHandler. In this example, the output parameter notifier_opcode is always 0, which means that default Notifier handling is used.


```
CREATE TABLE CustomError(
    error_code        integer,
    request_id        integer,
    gateway        varchar(255),
    address        varchar(255),
    subject        varchar(255),
    content        varchar(255),
    occurAt        timestamp not null default timestamp );


CREATE PROCEDURE CustomErrorHandler(
    out    @notifier_opcode    integer,
    in @error_code        integer,
    in @request_id        integer,
    in @gateway            varchar(255),
    in @address            varchar(255),
    in @subject            varchar(255),
    in @content            varchar(255) )
begin
    INSERT INTO CustomError(
    error_code,
    request_id,
    gateway,
    address,
    subject,
    content )
    VALUES(
    @error_code,
    @request_id,
    @gateway,
    @address,
    @subject,
    @content );
    set @notifier_opcode = 0;
end
```

To use the stored procedure ml_add_property with a SQL Anywhere consolidated database, use the following command.

```
call ml_add_property(
  'SIS',
  'Notifier(myNotifier)',
  'error_handler',
     'call CustomConfirmationHandler(?, ?, ?, ?, ?, ?, ?)');
```

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.

request_delete property

This is a SQL statement that specifies cleanup operations. The statement takes the request id as its only parameter. A parameter can be referenced by a named parameter or using 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 *samples-dir\MobiLink\SIS_CarDealer*. For more information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

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.

Notifier behavior properties

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 mlsrv10 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 - Server Administration*].

See also

- "Setting properties" on page 14.

enable property

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

See also

- "Setting properties" on page 14.

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 to poll immediately. It can also be used to shut down the Notifier without shutting down the MobiLink server. (Once stopped, the Notifier can only be restarted by shutting down and restarting the MobiLink server.)

This property is optional. The default is ON.

See also

- "Setting properties" on page 14.

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 14

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.

shared_database_connection property

Set this option to on if you want Notifiers to share connections.

When Notifiers share connections they consume fewer system resources without incurring performance penalties. However, in some situations it is not possible to share connections, such as when applications use SQL variables that have non-unique names among Notifiers.

Notifiers can share connections only if they are at the same isolation level.

The default is off.

See also

- "isolation property" on page 62

Gateway properties

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 21.

☞ For more information about setting properties, see "Setting properties" on page 14.

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 information about handling delivery confirmation on the server, see "confirmation_handler property" on page 54.

For example,

```
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.

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.

See also

- "SMTP gateway properties" on page 65

sync_gateway property

This names a SYNC gateway that the device tracker can use. The gateway must be enabled. A device tracker gateway can only use one SYNC gateway.

The SYNC gateway is a TCP/IP-based gateway that supports notification through a persistent connection. It is the recommended gateway.

There is a predefined instance of the SYNC gateway called Default-SYNC. It is configured to work with the predefined device tracker gateway, called Default-DeviceTracker. When the Default-SYNC gateway is used through the Default-DeviceTracker gateway, delivery confirmation is turned on by default.

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.

See also

- "UDP gateway properties" on page 68

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 email messages. In particular, they can send SMS messages to SMS listeners via a wireless carrier's email-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 19.

☞ For more information about setting properties, see "Setting properties" on page 14.

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).

☞ For information about handling delivery confirmation on the server, see "confirmation_handler property" on page 54.

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.

description property

Use this property to optionally describe the gateway.

enable property

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

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 emails (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 the MobiLink 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.

SYNC gateway properties

SYNC gateway configuration is required only if you need to send notification over the same protocol as your synchronizations.

A default SYNC gateway is provided and enabled by default. This default gateway can be used only if Notifier configuration is stored in the consolidated database (and not in a Notifier properties file).

The SYNC gateway uses persistent connections. The same connection is used for notifications, confirmations, and device tracking. The SYNC gateway is the preferred gateway: when using device tracking, a notification attempt will start with the SYNC gateway, and fallback to the UDP gateway and then the SMTP gateway.

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

You can have multiple SYNC gateways. To set up additional SYNC gateways, you can 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 19.

☞ For more information about setting properties, see "Setting properties" on page 14.

confirm_action property

Specify yes if you want confirmation upon delivery. This property has an effect only when sending directly through this gateway, and not when sending indirectly through a device tracking gateway.

The default is no.

See also

- "confirmation_handler property" on page 54

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).

For information about handling delivery confirmation on the server, see "confirmation_handler property" on page 54.

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.

description property

Use this property to optionally describe the gateway.

enable property

Specify **enable=yes** to use a SYNC gateway.

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 19.

☞ For more information about setting properties, see "Setting properties" on page 14.

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).

For information about handling delivery confirmation on the server, see "confirmation_handler property" on page 54.

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.

description property

Use this property to optionally describe the gateway.

enable property

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

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 assigns a free port.

Carrier properties

Carriers are required only when you are using an SMTP gateway.

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 email addresses.

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

email-address **=**
sms_email_user_prefix*phone-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 70
- "sms_email_user_prefix property" on page 71
- "ml_device_address" [*MobiLink - Server Administration*]

☞ For more information about carriers, see "Gateways and carriers" on page 19.

☞ For more information about setting properties, see "Setting properties" on page 14.

enable property

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

network_provider_id property

Specifies the network provider ID.

To use SMS on CE Phone Edition, set the network provider ID to _generic_. For example,

```
network_provider_id=_generic_
```

sms_email_domain property

Specifies the domain name of the carrier.

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

email-address **=**
sms_email_user_prefix*phone-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 71
- "ml_device_address" [*MobiLink - Server Administration*]

sms_email_user_prefix property

Specifies the prefix used in email addresses.

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

email-address **=**
sms_email_user_prefix*phone-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 70
- "ml_device_address" [*MobiLink - Server Administration*]

CHAPTER 6

Server-Initiated Synchronization System Procedures

Contents

About this chapter

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

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

☞ For more information, see "Device tracking" on page 21 and "Using device tracking with Listeners that don't support it" on page 23.

☞ For more information about MobiLink system tables, see "MobiLink Server System Tables" [*MobiLink - Server Administration*].

☞ For information about other MobiLink system procedures, see "MobiLink Server System Procedures" [*MobiLink - Server Administration*].

ml_delete_device

Function

Use this system 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

This function is useful only if you are manually setting up device tracking.

☞ See "Using device tracking with Listeners that don't support it" on page 23.

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 system 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

This system procedure is useful only if you are manually setting up device tracking.

☞ See "Using device tracking with Listeners that don't support it" on page 23.

Example

Delete an address record:

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

ml_delete_listening

Function

Use this system 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

This system procedure is useful only if you are manually setting up device tracking.

☞ See "Using device tracking with Listeners that don't support it" on page 23.

Example

Delete a recipient record:

```
CALL ml_delete_listening( 'myULDB' );
```

ml_set_device

Function

Use this system 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 over-writing manually entered data.
6	source	VARCHAR(255). Optional remarks on the source of this record.

Description

The system 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.

This system procedure is useful only if you are manually setting up device tracking.

☞ See "Using device tracking with Listeners that don't support it" on page 23.

See also

- "ml_set_device_address" on page 79
- "ml_set_listening" on page 81
- "ml_device" [*MobiLink - Server Administration*]
- "ml_device_address" [*MobiLink - Server Administration*]
- "ml_listening" [*MobiLink - Server Administration*]

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 system 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 system 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.

This system procedure is useful only if you are manually setting up device tracking.

☞ See "Using device tracking with Listeners that don't support it" on page 23.

See also

- "ml_set_device" on page 77
- "ml_set_listening" on page 81
- "ml_device" [*MobiLink - Server Administration*]
- "ml_device_address" [*MobiLink - Server Administration*]
- "ml_listening" [*MobiLink - Server Administration*]

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 system 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 system 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.

This system procedure is useful only if you are manually setting up device tracking.

☞ See "Using device tracking with Listeners that don't support it" on page 23.

See also

- "ml_set_device" on page 77
- "ml_set_device_address" on page 79
- "ml_device" [*MobiLink - Server Administration*]
- "ml_device_address" [*MobiLink - Server Administration*]
- "ml_listening" [*MobiLink - Server Administration*]

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 SDK for Palm

Contents

About this chapter

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

Introduction

You can use the Listener SDK for Palm to create Listeners for new Palm devices. The Listener SDK is a simple API that is provided to help you extend the Listener utility. The programming interface includes a message processing interface and device dependent functions. You can use the Listener SDK to create Listeners for new Palm devices or new wireless network adapters.

Palm Listener SDK files

The MobiLink Listener SDK and sample implementations are located in the following files:

Palm Files	Description
MobiLink\ListenerSDK\Palm\68k\cw\lib\PalmLsn.lib	Runtime library for Palm Listeners. This provides a message handling routine, Listener controls, and a handler editor.
MobiLink\ListenerSDK\Palm\68k\cw\rsc	Contains UI resources for the Palm Listener.
MobiLink\ListenerSDK\Palm\src\PalmLsn.h	Runtime library header and Palm Listener API.
MobiLink\ListenerSDK\Palm\src\Treo600.c	Treo 600 implementation.
MobiLink\ListenerSDK\Palm\src\Treo650.c	Treo 650 implementation.

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 84.

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:

  "PalmLsnAllocate function" on page 85

  "PalmLsnFree function" on page 86

- **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 86

  "PalmLsnDupSender function" on page 88

  "PalmLsnDupTime function" on page 88

- **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 89.

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 86

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 85

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 88
- "PalmLsnDupTime function" on page 88
- "a_palm_msg structure" on page 85

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 86
- "PalmLsnDupTime function" on page 88
- "a_palm_msg structure" on page 85

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 86
- "PalmLsnDupSender function" on page 88
- "a_palm_msg structure" on page 85

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 95.
- **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 91.

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 48.

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 42.

See Also

- "Palm Listener Configuration utility" on page 48
- "Message handlers" on page 28
- "Action variables" on page 42
- "PalmLsnCheckConfigDB function" on page 90
- "a_palm_msg structure" on page 85

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 95.

- **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 91.

Remarks

You can use this function to detect errors opening a configuration database or reading its records.

See Also

- "PalmLsnProcess function" on page 89

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 89

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 89
- "Palm Listener SDK files" on page 84

Example

The following example, used in the Treo 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 94

  "PalmLsnTargetDeviceID" on page 95

  "PalmLsnGetConfigFileName" on page 95

- **Registration or initialization** You can use the following functions to register or unregister the Listener.

  "PalmLsnNormalStart" on page 96

  "PalmLsnNormalStop" on page 96

- **Event handling** You can use the following function to handle application events:

  "PalmLsnNormalHandleEvent" on page 97

  You can use the following function to respond to launch codes which may be device dependent.

  "PalmLsnSpecialLaunch" on page 97

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 95

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 94

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.

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 89

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 96
- "PalmLsnSpecialLaunch" on page 97

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 96

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 );
}
```


Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W

