

**Oracle® Communications Unified
Communications Suite**

Event Notification Service Guide

Release 8.0

July 2015

ORACLE®

Oracle Communications Unified Communications Suite Event Notification Service Guide, Release 8.0

Copyright © 2007, 2015, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

1. Introduction to Event Notification Service	4
2. Current Event Notification Service C API Reference	16
3. Legacy Event Notification Service C API Reference	19
4. Messaging Server Specific Information	44
5. Calendar Server Specific Information	50
6. Debugging ENS	65

Chapter 1. Introduction to Event Notification Service

Introduction to Event Notification Service

This information provides an overview of the Event Notification Service (ENS) components, architecture, and Application Programming Interface (API).

Topics:

- [Event Notification Service Overview](#)
- [Event Notification Service Architecture](#)
- [Event Notification Service API Overview](#)

Event Notification Service Overview

The Event Notification Service (ENS) is the underlying publish-and-subscribe service available in the following Oracle Communications Unified Communications Suite products:

- Calendar Server
- Messaging Server



Note

See [Administering Event Notification Service in Messaging Server](#) for instructions on enabling and administering ENS in Messaging Server.

ENS acts as a dispatcher used by Oracle applications as a central point of collection for certain types of *events* that are of interest to them. Events are changes to the value of one or more properties of a resource. In this structure, a URI (Uniform Resource Identifier) represents an event. Any application that wants to know when these types of events occur registers with ENS, which identifies events in order and matches notifications with subscriptions.

Event examples include:

- Arrival of new mail to a user's inbox
- User's mailbox has exceeded its quota
- Calendar reminders

Specifically, ENS accepts reports of events that can be categorized, and notifies other applications that have registered an interest in certain categories of events.

ENS provides a server and API for publishers and subscribers. A publisher makes an event available to the notification service. A subscriber tells the notification service that it wants to receive notifications of a specific event. See [Event Notification Service API Overview](#) for more information on the ENS API.

ENS in Calendar Server 6

By default, ENS is enabled in Calendar Server 6. For Calendar Server 6 you do not need to do anything else to use ENS.

A user who wants to subscribe to notifications other than the alarms generated by Calendar Server 6 needs to write a subscriber.

Sample ENS C publisher and subscriber code is bundled with Calendar Server 6. (See [ENS Sample Code for Calendar Server](#).) Once Calendar Server 6 is installed, the code can be found in the following directory:

```
/opt/SUNWics5/cal/csapi/samples/ens
```

ENS in Messaging Server

The following instructions are applicable to Messaging Server 6.x:

ENS and iBiff (the ENS publisher for Messaging Server, also referred to as the notification plug-in to Messaging Server) are bundled in Messaging Server and ENS is enabled. However, the iBiff plug-in file, `libibiff`, is not automatically loaded at installation.

To subscribe to notifications, you need to first perform the following two actions on the Messaging Server host:

- Load the iBiff notification plug-in
- Stop and restart the Messaging Server

See [Administering Event Notification Service in Messaging Server](#) for further instructions.

The following instructions apply to Messaging Server 7:

There are two notification services for event notifications and alarms: GlassFish Message Queue (JMQ) and Event Notification Service (ENS). Messaging Server, Calendar Server, and Instant Messaging still have internal dependencies to ENS. By default, ENS is not enabled in this release. To enable ENS, set the `configutil` parameter `local.ens.enable` to 1.



Note

JMQ was briefly supported for IMAP IDLE through Messaging Server 7 Update 3 but use of it for that purpose is not recommended and was removed in Messaging Server 7 Update 4.

The following instructions apply to Messaging Server 7 Update 4:

Messaging Server enables IMAP IDLE by default, using the Event Notification Service (ENS). No configuration is required. In addition, IMAP IDLE now only works using ENS, because the ability to use IMAP IDLE with JMQ has been removed. For existing installations, administrators should, at a minimum, ensure that the ENS server is enabled by setting the `configutil` parameter `local.ens.enable` to 1 and restarting the Messaging Server.

Using the ENS API to Write a Subscriber for Messaging Server Notifications

A user who wants to subscribe to Messaging Server notifications needs to write a subscriber to the ENS API. To do so, the subscriber needs to know what the various Messaging Server notifications are. See [Messaging Server Specific Information](#) for that information.

Messaging Server comes bundled with sample ENS C publisher and subscriber code. See [Sample Code](#) for more information.

Sample Messaging Server code is provided with the product in the following directory:

```
msg-server-base/examples
```

Event References

Event references identify an event handled by ENS. Event references use the following URI syntax (as specified by RFC 2396):

scheme : // *authority resource* / [? *param1=value1¶m2=value2¶m3=value3*]

where:

- *scheme* is the access method, such as `http`, `imap`, `ftp`, or `wcap`. For Calendar Server and Messaging Server, the ENS scheme is `enp`.
- *authority* is the DNS domain or host name that controls access to the resource.
- *resource* is the path leading to the resource in the context of the authority. It can be composed of several path components separated by a slash (`/`).
- *param* is the name of a parameter describing the state of a resource.
- *value* is its value. There can be zero or more parameter/value pairs.

In general, all Calendar Server events start with the following:

```
enp:///ics
```

The Messaging Server notification plugin iBiff uses the following scheme and resource by default:

```
enp://127.0.0.1/store
```



Note

Although the event reference has a URI syntax, the scheme, authority, and resource have no special significance. They are merely used as strings with no further interpretation in ENS.

Calendar Server Event Reference Example

The following is an example event reference URI to subscribe to all event alarms with a calendar ID of `jac`:

```
enp:///ics/alarm?calid=jac
```



Note

This URI is not meant to be used by end users.

Messaging Server Event Reference Example

The following is an example event reference that requests a subscription to all `NewMsg` events for a user whose user ID is `blim`:

```
enp://127.0.0.1/store?evtType=NewMsg&mailboxName=blim
```

When using ENS with Messaging Server, the user ID you specify is case sensitive.



Note

This URI is not meant to be used by end users.

ENS Connection Pooling

The ENS connection pooling feature applies to **Messaging Server 7 Update 3** and prior versions.

The connection pooling feature of ENS enables a pool of subscribers to receive notifications from a single event reference. For every event, ENS chooses one subscriber from the pool to send the notification to. Thus, only one subscriber in the pool receives the notification. The ENS server balances sending of notifications among the subscribers. This enables the client to have a pool of subscribers that work together to receive all notifications from a single event reference.

For example, if notifications are being published to the event reference `enp://127.0.0.1/store`, a subscriber will normally subscribe to this event reference to receive notifications. To have a pool of subscribers receive all the notifications to this event reference, each subscriber in the pool only needs to subscribe to the event reference `enp+pool://127.0.0.1/store` instead. The ENS server chooses one subscriber from the pool to send the notification to.



Note

The publisher still sends notifications to the simple event reference, in the example above `enp://127.0.0.1/store`, that is, the publisher has no knowledge of the subscriber pool.

Multiple Pool Extension

Connection pooling can support multiple pools of subscribers. That is, you can have two pools of subscribers, each pool receiving all the notifications from the event reference. The syntax of the event reference for the subscriber is:

```
enp+pool[._poolid_]://domain/event
```

where *poolid* is a string using only base64 alphabet. (See RFC1521, Table 1, for what the base64 alphabet contains.) So, for example, to have two pools of subscribers to the event reference `enp://127.0.0.1/store`, each pool could subscribe to the following event references:

```
enp+pool.1://127.0.0.1/store— for first pool of subscribers  
enp+pool.2://127.0.0.1/store— for second pool of subscribers
```

Event Notification Service Architecture

On the Solaris OS platform, ENS runs as a daemon, `enpd`, along with other daemons in various calendar or messaging server configurations, to collect and dispatch events that occur to properties of resources. On Windows platforms, ENS runs as a service, `enpd.exe`.

For ENS, an event is a change that happens to a resource, while a resource is an entity such as a calendar or inbox. For example, adding an entry to a calendar (the resource) generates an event, which is stored by ENS. This event can then be subscribed to, and a notification would then be sent to the subscriber.

The ENS architecture enables the following three things to occur:

- Notification - This is a message that describes an event occurrence. Sent by the event publisher, it contains a reference to the event, as well as any additional parameter/value pairs added to the

URI, and optional data (the payload) used by the event consumers, but opaque to the notification service. Whoever is interested in the event can subscribe to it.

- Subscription - This is a message sent to subscribe to an event. It contains an event reference, a client-side request identifier, and optional parameter/value pairs added to the URI. The subscription applies to upcoming events (that is, a subscriber asks to be notified of upcoming events).
- Unsubscription - This message cancels (unsubscribes) an existing subscription. An event subscriber tells ENS to stop relaying notifications for the specified event.



Note

ENS is a "best effort" network service. A best effort network service does not provide any guarantees that the data is delivered. More specifically, under high load situations, ENS silently discards notifications.

Notify

ENS notifies its subscribers of an event by sending a notification. Notify is also referred to as "publish." A notification can contain the following items:

- An event reference (which, optionally, can contain parameter/value pairs)
- Optional application-specific data ("opaque" for ENS, but the publisher and subscriber agree apriori to the format of the data)

The optional application-specific data is referred to as the "payload."

There are two kinds of notifications:

- *Unreliable notification* - Notification sent from an event publisher to a notification server. If the publisher does not know nor care about whether there are any consumers, or whether they get the notification, this request does not absolutely need to be acknowledged. However, a publisher and a subscriber, who are mutually aware of each other, can agree to set up a reliable event notification link (RENL) between themselves. In this case, once the subscriber has processed the publisher's notification, it sends an acknowledgment notification back to the publisher.
- *Reliable notification* - Notification sent from a server to a subscriber as a result of a subscription. This type of notification should be acknowledged. A reliable notification contains the same attributes as an unreliable notification.

See [Publisher API](#) for more information.

Subscribe

ENS receives a request to be notified of events. The request sent by the event subscriber is a subscription. The subscription is valid during the life of the session, or until it is cancelled (unsubscribed).

A subscription can contain the following items:

- An event reference (which, optionally, can contain parameter/value pairs)
- A request identifier

See [Subscriber API](#) for more information.

Unsubscribe

ENS receives a request to cancel an existing subscription. [Subscriber API](#) for more information.

How Calendar Server Interacts with ENS

This section contains the following topics:

- [Calendar Server Alarm Queue](#)
- [Calendar Server Daemons](#)
- [Alarm Transfer Reliability](#)
- [Calendar Server Example](#)

Calendar Server Alarm Queue

ENS is an alarm dispatcher. This decouples alarm delivery from alarm generation. It also enables the use of multiple delivery methods, such as email and wireless communication. The `csadmin` daemon detects events by sensing changes in the state of the alarm queue. The alarm queue's state changes every time an alarm is placed in the queue. An alarm is queued when a calendar event generates an alarm. The following URIs represent these kind of events:

for events:

```
enp:///ics/eventalarm?calid=calid&uid=uid&rid=rid&aid=aid
```

for todos (tasks):

```
enp:///ics/todoalarm?calid=calid&uid=uid&rid=rid&aid=aid
```

where:

- *calid* is the calendar ID.
- *uid* is the event/todo (task) ID within the calendar.
- *rid* is the recurrence id for a recurring event/todo (task).
- *aid* is the alarm ID within the event/todo (task). In case there are multiple alarms, the *aid* identifies the correct alarm.

The publisher `csadmin` dequeues the alarms and sends notifications to `enpd`. The `enpd` daemon then checks to see if anyone is subscribed to this kind of event and sends notifications to the subscriber, `csnotifyd`, for any subscriptions it finds. Other subscribers to alarm notifications (reminders) can be created and deployed within a Calendar Server installation. These three daemons interacting together implement event notification for Calendar Server.

Calendar Server Daemons

Calendar Server includes two daemons that communicate to the ENS daemon, `enpd`:

- `csadmin`
The `csadmin` daemon contains a publisher that submits notifications to the notification service by sending alarm events to ENS. It manages the Calendar Server alarm queue. It implements a scheduler, which lets it know when an alarm has to be generated. At such a point, `csadmin` publishes an event. ENS receives and dispatches the event notification. To ensure alarm transfer reliability, `csadmin` requires acknowledgment for certain events or event types. (See [Alarm Transfer Reliability](#).) The `csadmin` daemon uses Reliable Event Notification Links (RENs) to accomplish acknowledgment.
- `csnotifyd`
The `csnotifyd` daemon is the subscriber that expresses interest in particular events (subscribes), and receives notifications about these subscribed-to events from ENS, and sends notice of these events and todos (tasks) to its clients by email. Though the ability to unsubscribe is part of the ENS architecture, `csnotifyd` does not bother to unsubscribe to events for the following two reasons: there is no need to unsubscribe or resubscribe during normal runtime; and due to the temporary nature of the subscriptions store (it is

held in memory), all subscriptions are implicitly unsubscribed when the connection to ENS is shutdown.

The `csnotifyd` daemon subscribes to `enp:///ics/alarm/`. The todo (task) or event is specified in a parameter.

Alarm Transfer Reliability

To ensure that no alarm ever gets lost, `csadmin` and `csnotifyd` use the RENL feature of ENS for certain types of alarms. For these alarms, `csadmin` requests an end-to-end acknowledgment for each notification it sends, while `csnotifyd`, after successfully processing it, generates a notification acknowledgment for each RENL alarm notifications it receives.

For these RENL alarms, should the network, the ENS daemon, or `csnotifyd` fail to handle a notification, `csadmin` will not receive any acknowledgment, and will not remove the alarm from the alarm queue. The alarm will, therefore, be published again after a timeout.

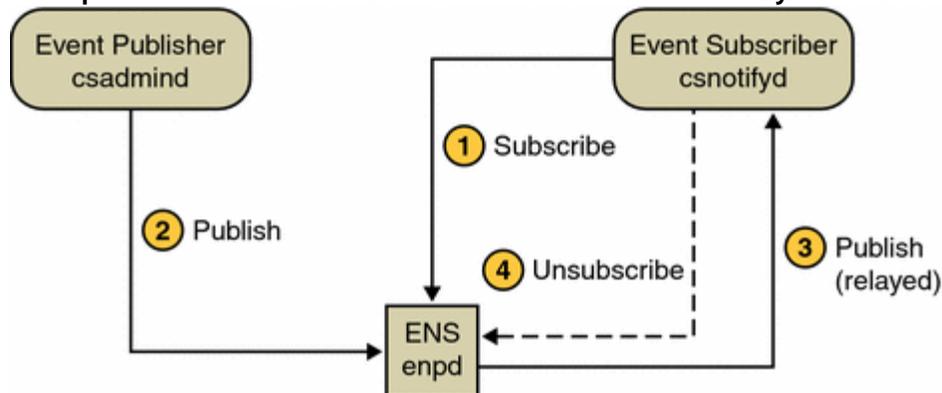
Calendar Server Example

A typical ENS publish and subscribe cycle for Calendar Server resembles the following:

1. The event subscriber, `csnotifyd`, expresses interest in an event (subscribes).
2. The event publisher, `csadmin`, detects events and sends notification (publishes).
3. ENS publishes the event to the subscriber.
4. The event subscriber cancels interest in the event (unsubscribes). This step happens implicitly when the connection to ENS is shut down.

The following figure illustrates this cycle.

Example Event Notification Service Publish and Subscribe Cycle for Calendar Server



The following table provides the narrative for the figure.

Sample ENS Publish and Subscribe Cycle

Action	ENS Response
1. The <code>csnotifyd</code> daemon sends a subscription request to ENS.	ENS stores the subscription in the subscriptions database.
2. The <code>csadmin</code> daemon sends a notification request to ENS.	ENS queries the subscriptions database for subscriptions matching the notification.
3. The <code>csnotifyd</code> daemon receives a notification from ENS.	When ENS receives a notification from a publisher, it looks up its internal subscription table to find subscriptions matching the event reference of the notification. Then for each subscription, it relays a copy of the notification to the subscriber who owns this subscription.
4. Currently, <code>csnotifyd</code> does not bother sending cancellation requests to ENS.	Because the subscriptions store is in memory only (not in a database), all subscriptions are implicitly unsubscribed when the connection to ENS is shutdown.

How Messaging Server Interacts with ENS

The ENS server delivers notifications from the Messaging Server to ENS clients (that is, iBiff subscribers). There is no guarantee of the order of notification prior to the ENS server because the events are coming from different processes (MTA, `stored`, and `imapd`).

Notifications flow from iBiff in the MTA, `stored`, and `imap` processes to ENS `enpd`. The ENS client subscribes to the ENS, and receives notifications. Messaging Server publishes the notifications with iBiff, but no Messaging Server services subscribe to these notifications. A customer-provided ENS subscriber or client should be written to consume the notifications and do whatever is necessary.

The Messaging Server architecture enforces that a given set of mailboxes is served by a given host computer. A given mailbox is not served by multiple host computers. There are several processes manipulating a given mailbox but only one computer host serving a given mailbox. Thus, to receive notifications, end-users only need to subscribe to the ENS daemon that serves the mailbox they are interested in.

Messaging Server enables you to have either one ENS server for all mailboxes—that is, one ENS server for all the computer hosts servicing the message store—or multiple ENS servers, perhaps one ENS server per computer host. The second scenario is more scalable. Also, in this scenario, end users must subscribe to multiple ENS servers to get the events for mailboxes they are interested in.

Thus, the architecture requires an ENS server per computer host. The ENS servers and the client processes do not have to be co-located with each other or with messaging servers.

Event Notification Service API Overview

This section provides an overview of the C API for ENS. For detailed information on the ENS C API, see [Legacy Event Notification Service C API Reference](#).

ENS C API Overview (Messaging Server 7 Update 3 and Prior Releases)

ENS implements the following three APIs:

- Publisher API

A publisher sends notification of a subscribed-to event to ENS, which then distributes it to the subscribers. Optionally, in Calendar Server, the application can request acknowledgment of receipt of the notification. To do this, a Reliable Event Notification Link (RENL) is necessary. An RENL has a publisher, a subscriber, and a unique ID, which identify notifications that are subject to acknowledgment. The publisher informs the application of the receipt of an acknowledgment by invoking the `end2end_ack` callback passed to `publish_a`. Currently, only Calendar Server supports RENL.

- **Unreliable Publisher API**

A publisher sends notification of an event to ENS, which distributes it to the subscribers. The term "unreliable" means that the API does not indicate whether a published event was successfully sent or received by a subscriber.

This communication model is suitable for some applications, but not for others. If you are concerned about transfer reliability, use the full-blown publisher API (`publisher.h`) instead. However, the Unreliable Publisher API has the merit of being easier to use if transfer reliability is not a concern.

- **Subscriber API**

A subscriber is a client to the notification service which expresses interest in particular events. When the notification service receives a notification about one of these events from a publisher, it relays the notification to the subscriber.

A subscriber may also unsubscribe, which cancels an active subscription.

In Calendar Server, to enable an RENL, the subscriber declares its existence to ENS, which then transparently generates notification acknowledgment on behalf of the subscriber application. The subscriber can revoke the RENL at any time.

- **Publish and Subscribe Dispatcher API**

When an asynchronous publisher is used, ENS needs to borrow threads from a thread pool in order to invoke callbacks. The application can either choose to create its own thread pool and pass it to ENS, or it can let ENS create and manage its own thread pool. In either case, ENS creates and uses a dispatcher object to instantiate the dispatcher used (`pas_dispatcher_t`). `GDisp` (`libasync`) is the dispatcher supported.

Building and Running Custom Applications

To assist you in building your own custom publisher and subscriber applications, Messaging Server and Calendar Server include sample code. This section tells you where to find the sample code, where the APIs' include (header) files are located, and where the libraries are that you need to build and run your custom programs.



Note

This section applies to the C API only.

Location of Sample Code

Calendar Server

Calendar Server includes four simple sample programs to help you get started. The code for these samples resides in the following directory:

`cal-server-base/cal/csapi/samples/ens`

Messaging Server

Starting with version 5.1, Messaging Server contains sample programs to help you learn how to receive notifications. These sample programs are located in the following directory:

`msg-server-base/examples`

Location of Include Files

Calendar Server

The include (header) files for the publisher and subscriber APIs are: `publisher.h`, `subscriber.h`, and `pasdisp.h` (publish and subscribe dispatcher). They are located in the CSAPI include directory. The default include path is:

```
cal-server-base/cal/csapi/include
```

Messaging Server

The default include path for Messaging Server is:

```
msg-server-base/include
```

Dynamically Linked/Shared Libraries

Calendar Server

Your custom code must be linked with the dynamically linked library `libens`, which implements the publisher and subscriber APIs. On some platforms all the dependencies of `libens` must be provided as part of the link directive. These dependencies, in order, are:

1. `libgap`
2. `libcyrus`
3. `libyasr`
4. `libasync`
5. `libnspr3`
6. `libplsd4`
7. `libplc3`

Calendar Server uses these libraries; therefore, they are located in the server's `bin` directory. The default `libens` path is:

```
/opt/cal-server-base/cal/bin
```



Note

For Windows, in order to build publisher and subscriber applications, you also need the archive files (`.lib` files) corresponding to all the earlier mentioned libraries. These are located in the CSAPI library directory, `lib`. The default `lib` path is:

```
drive:\ProgramFiles\iPlanet\CalendarServer5\cal\csapi\lib
```

Messaging Server

The libraries for Messaging Server are located in the following directory:

```
msg-server-base/lib
```

Messaging Server 7 Update 3 and prior releases: Refer to `msg-server-base/examples/enssdk/Makefile.sample` to help determine what libraries are needed. This makefile contains instructions on how to compile and run the `apub` and `asub` programs. This file also describes what libraries are needed, and what the `LD_LIBRARY_PATH` should be. The following listing shows a sample `makefile.sample` file.

Makefile.sample File (Messaging Server 7 Update 3 and Prior Releases)

```
#
# Sample makefile
#
# your C compiler
CC = gcc

# LIBS
# Your library path should include <msg-server-base>/lib
LIBS = -lens -lgap -lxenp -lcyrus -lchartable -lyasr -lasync

all: apub asub

apub: apub.c
    $(CC) -o apub -I ../include apub.c $(LIBS)

asub: asub.c
    $(CC) -o asub -I ../include asub.c $(LIBS)

run:
    @echo 'run <msg-server-base>/start-ens'
    @echo run asub localhost 7997
    @echo run apub localhost 7997
```



Note

The Windows distribution includes the following additional files:

```
msg-server-base\bin\msg\enssdk\examples
bin\msg\enssdk\examples\libens.lib
bin\msg\enssdk\examples\libgap.lib
bin\msg\enssdk\examples\libxenp.lib
bin\msg\enssdk\examples\libcyrus.lib
bin\msg\enssdk\examples\libchartable.lib
bin\msg\enssdk\examples\libyasr.lib
bin\msg\enssdk\examples\libasync.lib
bin\msg\enssdk\examples\asub.dsw
bin\msg\enssdk\examples\apub.dsp
bin\msg\enssdk\examples\asub.dsp
```

To build on Windows platforms:

1. A sample VC++ workspace is provided in `asub.dsw`. It has two projects in it: `asub.dsp` and `apub.dsp`.
The required `.lib` files to link is in the same directory as `asub.c` and `apub.c`.
2. To run, it requires that the following DLLs are in your path.

```
libens.dll
libgap.dll
libxenp.dll
libcyrus.dll
libchartable.dll
libyasr.dll
libasync.dll
```

The simplest way to accomplish this is to include `msg-server-base` in `{\msglib}` in your `PATH`.

Chapter 2. Current Event Notification Service C API Reference

Current Event Notification Service C API Reference

This information details the new ENS C API in Messaging Server 7 Update 4.

Topics:

- [API Overview](#)
- [API Basic Usage](#)
- [API Usage Notes](#)

API Overview

The ENS C API, `ens.h`, is located in the `msg-server-base/examples/enssdk/` directory. The `ens_pub.c` sample publisher and `ens_sub.c` sample subscriber demonstrate use of the ENS C API.

Here is the API header (`ens.h`):

```
====
// ens.h -- ENS C client API
//
// Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights
reserved.

# ifndef ENS_HEADER_INCLUDED
# define ENS_HEADER_INCLUDED 1

# ifdef __cplusplus
extern "C" {
# endif

//
// Connecting
//

// an ENS client
struct ensclient_t;
typedef struct ensclient_t ensclient_t;

// callback invoked if the ENS connection dies
// subscriptions are no longer valid but must not be unsubscribed
typedef void (*lost_cnx_cb_t)(void*);

ensclient_t* ens_open(const char* host, int port, lost_cnx_cb_t
lost_cnx_cb, void* lost_cnx_arg);

// automatically cleans up all existing subscription handles
```

```

void ens_close(ensclient_t*);

//
// Publishing events
//

typedef void (*destructor_t)(void*);          // cleanup function

void ens_publish(ensclient_t*, const char* evt, char* body, size_t
bodysz, destructor_t body_delete);

//
// Receiving events
//

// a subscription handle
struct sub_t;
typedef struct sub_t subscription_t;

// handler called when a subscribed event is received
typedef void (*notify_cb_t)(void *rock, char *event, char *body, size_t
body_len);

subscription_t* ens_subscribe(ensclient_t*, const char* evt, notify_cb_t
cb, void* rock);
void ens_unsubscribe(ensclient_t*, subscription_t*);

#ifdef __cplusplus
}
#endif

```

```
# endif // ENS_HEADER_INCLUDED
====
```

API Basic Usage

The client calls `ens_open()` to start a connection. If reliability across ENS connection outages is important, the client should provide a lost connection handler callback. The lost connection handler normally marks any client-specific subscription information as invalid, calls `ens_close`, and triggers a task to attempt a reconnect by using `ens_open` (possibly after a delay).

The client calls `ens_close()` on shut down.

The client calls `ens_subscribe()` to subscribe to events and gets a callback when a matching event is received. The client calls `ens_unsubscribe()` to unsubscribe from an event (usually not necessary as a client can just call `ens_close`).

To publish an event, use `ens_publish`. (In general, you do not need to do so and the sample code should be sufficient.)

To build the sample programs, link against the `libens` library, which is normally installed in `/opt/sun/comms/messaging64/lib/libens.so`.

The `ens_sub.c` sample program is helpful to see what events are generated and how the event strings and message payloads are formatted.

Both Messaging Server publishers (that is, `imapd`) and the ENS server (`enpd`) are designed to drop events if an overload situation occurs.

API Usage Notes

Clients written against the old ENS C API (the headers and `libens.so` file included with Messaging Server 7 Update 3 and prior releases) should work against the new ENS daemon as long as they link against the `libens.so` library from Messaging Server 7 Update 3 and prior releases. However, the new API is simpler so a rewrite to the new API is recommended.

The ENS C API is presently the recommended API for C-based software that needs to subscribe to Messaging Server events. Use of the Glassfish Message Queue, OpenMQ, or Java Enterprise System Message Queue C API is not recommended.

Chapter 3. Legacy Event Notification Service C API Reference

Legacy Event Notification Service C API Reference



Note

New ENS software should use the new simpler API starting with Messaging Server 7 Update 4 instead of this API. The sample code and libraries for this API are no longer included in packaging. If you have existing code that used this API, you can continue to do so, provided that you perform a manual copy of the necessary shared libraries from a legacy version of Messaging Server that used this API. For information on the new API, see [Current Event Notification Service C API Reference](#).

This information details the ENS C API.

Topics:

- [Sample Code](#)
- [Publisher API Functions List](#)
- [Unreliable Publisher API](#)
- [Subscriber API Functions List](#)
- [Publish and Subscribe Dispatcher Functions List](#)
- [Publisher API](#)
- [Unreliable Publisher API](#)
- [Subscriber API](#)
- [Publish and Subscribe Dispatcher API](#)

Sample Code

The following two code samples illustrate how to use the ENS API. The sample code is provided with the product in the following directory:

```
<msg-svr-base>/examples
```

To Use the Sample Code

1. Before running the makefile, set your library search path to include the directory:

```
<msg-svr-base>/lib
```

2. Compile the code using the `Makefile.sample`.
3. Run `apub` and `asub` as follows in separate windows:

```
apub localhost 7997
```

```
asub localhost 7997
```

Whatever is typed into the `apub` window should appear on the `asub` window. If you use the default settings, all `iBiff` notifications should appear in the `asub` window.
4. Remove the `msg-svr-base/lib` path from your library search path.

**Note**

If you do not remove this from the library search path, you will not be able to stop and start the directory server.

Sample Publisher

This sample code provides a simple interactive asynchronous publisher.

```
/*
 *
 * Syntax:
 *  apub host port
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "upub.h"

static upub_t *_publisher = NULL;

/* function prototypes */
static void _read_stdin(void);
static void _publish_ack(void *arg);

/**
 *
 **/
static void _read_stdin()
{
    static char input[1024];

    while (1) {
        printf("apub> ");
        fflush(stdout);
        if ( !fgets(input, sizeof(input), stdin) ) {
            continue;
        } else {
            char *message;
            unsigned int message_len;

            input[strlen(input) - 1] = 0; /* Strip off the \n */

            if (*input == '.' && input[1] == 0) {
                break;
            }

            message = strdup(input);
            message_len = strlen(message);
            upub_publish(_publisher, "enp://yoyo.com/xyz",
                message, message_len, _publish_ack);
        }
    }
}
```

```

    }
}

return;
}

/**
 * call back after publish is done
 **/
static void
_publish_ack(void *arg)
{
    free(arg);
    return;
}

int
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];

    if (argc < 2) {
        printf("\nUsage:\n\tapub host port\n");
        exit(2);
    }
    if (*(argv[1]) == '0') {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }
    if (argc > 2) {
        port = (unsigned short)atoi(argv[2]);
    }

    _publisher = upub_init(NULL, host, port, 1);
    if (_publisher == NULL) {
        printf("could not create publisher\n");
        exit(1);
    }

    _read_stdin();
}

```

```

    upub_shutdown(_publisher);
}

```

Sample Subscriber

This sample code provides a simple subscriber.

```

/*
 *
 * asub : example asynchronous subscriber
 *
 * Syntax:
 *     asub host port
 */
*
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "pasdisp.h"
#include "subscriber.h"

#define DEFAULT_EVENT_REF "enp://yoyo.com/xyz"

static pas_dispatcher_t *disp = NULL;
static subscriber_t *_subscriber = NULL;
static subscription_t *_subscription = NULL;
static renl_t *_renl = NULL;
static char *_event_ref = DEFAULT_EVENT_REF;

static void _exit_usage()
{
    printf("\nUsage:\nasub host port\n");
    exit(5);
}

static void _exit_error(const char *msg)
{
    printf("%s\n", msg);
    exit(1);
}

static void _subscribe_ack(void *arg, int rc, void *subscription)
{
    (void)arg;
    if (!rc) {
        _subscription = subscription;
        printf("Subscription successful\n");
        subscriber_keepalive(_subscriber, 30000);
    }else {
        printf("Subscription failed - status %d\n", rc);
        pas_shutdown(disp);
    }
}

```

```

    }
}

static void _unsubscribe_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;
    (void *)arg;

    if (rc != 0) {
        printf("Unsubscribe failed - status %d\n", rc);
    }

    subscriber_delete(_subscriber);
    pas_shutdown(dispatch);
}

static int _handle_notify(void *arg, char *url, char *str, int len)
{
    (void *)arg;
    printf("[%s] %.*s\n", url, len, (str) ? str : "(null)");
    return 0;
}

static void _open_ack(void *arg, int rc, void *enc)
{
    _subscriber = (subscriber_t *)enc;

    (void *)arg;
    if (rc) {
        printf("Failed to create subscriber with status %d\n", rc);
        pas_shutdown(dispatch);
        return;
    }

    subscribe(_subscriber, "enp://127.0.0.1/store",
              _handle_notify, NULL,
              _unsubscribe_ack, NULL);
    return;
}

static void _unsubscribe(int sig)
{
    (int)sig;
    unsubscribe(_subscriber, _subscription, _unsubscribe_ack, NULL);
}

int
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];

    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0') {
        strcpy(host, "127.0.0.1");
    } else {

```

```
    strcpy(host, argv[1]);
}
if (argc > 2) {
    port = (unsigned short)atoi(argv[2]);
}
if (argc > 3) {
    _event_ref = argv[3];
}

disp = pas_dispatcher_new(NULL);
if (disp == NULL) _exit_error("Can't create publisher");

subscriber_new_a(disp, NULL, host, port, _open_ack, NULL);

pas_dispatch(disp);

pas_dispatcher_delete(disp);
```

```
exit(0);  
}
```

Publisher API Functions List

This information includes a description of the following Publisher functions, listed in the following table:

ENS Publisher API Functions List

Function	Description
<code>publisher_t</code>	Definition for a publisher.
<code>publisher_cb_t</code>	Generic callback function acknowledging an asynchronous call.
<code>publisher_new_a</code>	Creates a new asynchronous publisher.
<code>publisher_new_s</code>	Creates a new synchronous publisher.
<code>publish_a</code>	Sends an asynchronous notification to the notification service.
<code>publish_s</code>	Sends a synchronous notification to the notification service.
<code>publisher_delete</code>	Terminates a publish session.
<code>publisher_get_subscriber</code>	Creates a subscriber using the publisher's credentials.
<code>renl_create_publisher</code>	Creates an RENL, which enables the invocation of <code>end2end_ack</code> .
<code>renl_cancel_publisher</code>	Cancels an RENL.

Unreliable Publisher API

This information includes a description of the following Unreliable Publisher functions, listed in the following table:

ENS Unreliable Publisher API Functions List

Function	Description
<code>upub_t</code>	Definition for an unreliable publisher.
<code>upub_publish</code>	Sends a notification.
<code>upub_init</code>	Creates the publisher.
<code>upub_shutdown</code>	Shuts down and frees the publisher.

Subscriber API Functions List

This information includes a description of following Subscriber functions, listed in the following table:

ENS Subscriber API Functions List

Function	Description
subscriber_t	Definition of a subscriber.
subscription_t	Definition of a subscription.
subscriber_cb_t	Generic callback function acknowledging an asynchronous call.
subscriber_notify_cb_t	Synchronous callback; called upon receipt of a notification.
subscriber_new_a	Creates a new asynchronous subscriber.
subscriber_new_s	Creates a new synchronous subscriber.
subscribe_a	Establishes an asynchronous subscription.
unsubscribe_a	Cancel an asynchronous subscription.
subscriber_delete	Terminates a subscriber.
subscriber_get_publisher	Creates a publisher using the subscriber's credentials.
renl_create_subscriber	Creates the subscription part of the RENL.
renl_cancel_subscriber	Cancel an RENL.

Publish and Subscribe Dispatcher Functions List

This information includes a description of the following Publish and Subscribe Dispatcher functions, listed in the following table:

ENS Publish and Subscribe Dispatcher Functions List

Function	Description
pas_dispatcher_t	Definition of a publish and subscribe dispatcher.
pas_dispatcher_new	Creates a dispatcher.
pas_dispatcher_delete	Destroys a dispatcher created with <code>pas_dispatcher_new</code> .
pas_dispatch	Starts the dispatch loop of an event notification environment.
pas_shutdown	Stops the dispatch loop on an event notification environment started with <code>pas_dispatch</code> .

Publisher API

The Publisher API consists of one definition and nine functions:

- `publisher_t`
- `publisher_cb_t`
- `publisher_new_a`
- `publisher_new_s`
- `publish_a`
- `publish_s`
- `publisher_delete`
- `publisher_get_subscriber`
- `renl_create_publisher`
- `renl_cancel_publisher`

publisher_t

Purpose.

A publisher.

Syntax

```
typedef struct enc_struct publisher_t;
```

Parameters

None.

Returns

Nothing.

publisher_cb_t

Purpose.

Generic callback function invoked by ENS to acknowledge an asynchronous call.

Syntax

```
typedef void (*publisher_cb_t) (void *arg, int rc, void *data);
```

Parameters

arg	Context variable passed by the caller.
rc	The return code.
data	For an open, contains a newly created context.

Returns

Nothing.

publisher_new_a

Purpose

Creates a new asynchronous publisher.

Syntax

```
void publisher_new_a (pas_dispatcher_t *disp,
                    void *worker,
                    const char *host,
                    unsigned short port,
                    publisher_cb_t cbdone,
                    void *cbarg);
```

Parameters

disp	P&S thread pool context returned by pas_dispatcher_new.
worker	Application worker. If not NULL, grouped with existing workers created by ENS to service this publisher session. Used to prevent multiple threads from accessing the publisher data at the same time.
host	Notification server host name.
port	Notification server port.
cbdone	The callback invoked when the publisher has been successfully created, or could not be created. There are three Parameters to cbdone: * cbarg The first argument. * A status code. If non-zero, the publisher could not be created; value specifies cause of the failure. * The new active publisher.
cbarg	First argument of cbdone.

Returns

Nothing. It passes the new active publisher as third argument of cbdone callback.

publisher_new_s

Purpose

Creates a new synchronous publisher.

Syntax

```
publisher_t *publisher_new_s (pas_dispatcher_t *disp,
                             void *worker,
                             const char *host,
                             unsigned short port);
```

Parameters

disp	P&S thread pool context returned by <code>pas_dispatcher_new</code> .
worker	Application worker. If not <code>NULL</code> , grouped with existing workers created by <code>ENS</code> to service this publisher session. Used to prevent multiple threads from accessing the publisher data at the same time.
host	Notification server host name.
port	Notification server port.

Returns

A new active publisher (`publisher_t`).

publish_a

Purpose

Sends an asynchronous notification to the notification service.

Syntax

```
void publish_a (publisher_t *publisher,
               const char *event_ref,
               const char *data,
               unsigned int datalen,
               publisher_cb_t cbdone,
               publisher_cb_t end2end_ack,
               void *cbarg,
               unsigned long timeout);
```

Parameters

<code>publisher_t</code>	The active publisher.
<code>event_ref</code>	The event reference. This is a URI identifying the modified resource.
<code>data</code>	The event data. The body of the notification message. It is opaque to the notification service, which merely relays it to the events' subscriber.
<code>datalen</code>	The length in bytes of the data.
<code>cbdone</code>	The callback invoked when the data has been accepted or deemed unacceptable by the notification service. What makes a notification acceptable depends on the protocol used. The protocol may choose to use the transport acknowledgment (TCP) or use its own acknowledgment response mechanism.
<code>end2end_ack</code>	The callback function invoked after acknowledgment from the consumer peer (in an RENL) has been received. Used only in the context of an RENL.
<code>cbarg</code>	The first argument of <code>cbdone</code> or <code>end2end_ack</code> when invoked.
<code>timeout</code>	The length of time to wait for an RENL to complete.

Returns

Nothing.

publish_s

Purpose

Sends a synchronous notification to the notification service.

Syntax

```
int publish_s (publisher_t *publisher,  
              const char *event_ref,  
              const char *data,  
              unsigned int datalen);
```

Parameters

publisher	The active publisher.
event_ref	The event reference. This is a URI identifying the modified resource.
data	The event data. The body of the notification message. It is opaque to the notification service, which relays it to the events' subscriber.
datalen	The length in bytes of the data.

Returns

Zero if successful; a failure code if unsuccessful. If an RENL, the call does not return until the consumer has completely processed the notification and has successfully acknowledged it.

publisher_delete

Purpose

Terminates a publish session.

Syntax

```
void publisher_delete (publisher_t *publisher);
```

Parameters

publisher	The publisher to delete.
-----------	--------------------------

Returns

Nothing.

publisher_get_subscriber

Purpose

Creates a subscriber using the credentials of the publisher.

Syntax

```
struct subscriber_struct * publisher_get_subscriber(publisher_t
*publisher);
```

Parameters

publisher	The publisher whose credentials are used to create the subscriber.
-----------	--

Returns

The subscriber, or NULL if the creation failed. If the creation failed, use the subscriber_new to create the subscriber.

renl_create_publisher

Purpose

Declares an RENL, which enables the end2end_ack invocation. After this call returns, the end2end_ack argument is invoked when an acknowledgment notification matching the specified publisher and subscriber is received.

Syntax

```
void renl_create_publisher (publisher_t *publisher,
                           const char *renl_id,
                           const char *subscriber,
                           publisher_cb_t cbdone,
                           void *cbarg);
```

Parameters

publisher	The active publisher.
renl_id	The unique RENL identifier. This allows two peers to be able to set up multiple RENLs between them.
subscriber	The authenticated identity of the peer.
cbdone	The callback invoked when the RENL is established.
cbarg	The first argument of cbdone, when invoked.

Returns

Nothing.

renl_cancel_publisher

Purpose

This cancels an RENL. This does not prevent more notifications being sent, but should a client acknowledgment be received, the end2end_ack argument of publish will no longer be invoked. All RENLs are automatically destroyed when the publisher is deleted. Therefore, this function does not need

to be called to free RENL-related memory before deleting a publisher.

Syntax

```
void renl_cancel_publisher (renl_t *renl);
```

Parameters

renl	The RENL to cancel.
------	---------------------

Returns

Nothing.

Unreliable Publisher API

The Unreliable Publisher (`upub`) API consists of the following functions:

- [upub_t](#)
- [upub_publish](#)
- [upub_init](#)
- [upub_shutdown](#)

upub_t

Purpose

Defines an unreliable publisher type.

Syntax

```
typedef struct upub_struct upub_t;
```

Parameters

None.

Returns

Nothing.

upub_publish

Purpose

Sends a notification.

Syntax

```
void upub_publish (upub_t *upub,
                  const char *event_ref,
                  void *payload,
                  unsigned int payload_len,
                  void (*delete_payload) (void *));
```

Parameters

upub	Publisher.
event_ref	Event reference. This is a URI identifying the resource that is modified.
payload	Notification payload.
payload_len	Length in bytes of the payload.
delete_payload	Callback to free the payload when it is no longer needed.

Returns

Nothing.

upub_init

Purpose

Creates the publisher.

This function allocates the `upub` object and sets up the thread dispatcher. The thread dispatcher can be provided by the application if the application already has one. This will have the advantage of not creating a thread dedicated to this API.

Note: If a connection-based transport is used, the connection to the notification service is not set up here. This is deferred to the first time a notification is actually published.

Syntax

```
upub_t *upub_init(void *gdc,
                  const char *host, unsigned short port,
                  int num_workers);
```

Parameters

gdc	GDisp context (thread dispatcher).
host	Notification server host name.
port	Notification server port.
num_workers	Number of parallel sessions.

Returns

The `upub` ready to be used.

upub_shutdown

Purpose

Shuts down and frees the publisher.

Syntax

```
void upub_shutdown(upub_t *upub);
```

Parameters

upub

Publisher.

Returns

Nothing.

Subscriber API

The Subscriber API includes two definitions and ten functions:

- [subscriber_t](#)
- [subscription_t](#)
- [subscriber_cb_t](#)
- [subscriber_notify_cb_t](#)
- [subscriber_new_a](#)
- [subscriber_new_s](#)
- [subscribe_a](#)
- [unsubscribe_a](#)
- [subscriber_delete](#)
- [subscriber_get_publisher](#)
- [renl_create_subscriber](#)
- [renl_cancel_subscriber](#)

subscriber_t

Purpose

A subscriber.

Syntax

```
typedef struct enc_struct subscriber_t;
```

Parameters

None.

Returns

Nothing.

subscription_t

Purpose

A subscription.

Syntax

```
typedef struct subscription_struct subscription_t;
```

Parameters

None.

Returns

Nothing.

subscriber_cb_t

Purpose

Generic callback function invoked by ENS to acknowledge an asynchronous call.

Syntax

```
typedef void (*subscriber_cb_t) (void *arg,  
                                int rc,  
                                void *data);
```

Parameters

arg	Context variable passed by the caller.
rc	The return code.
data	For an open, contains a newly created context.

Returns

Nothing.

subscriber_notify_cb_t

Purpose

Subscriber callback; called upon receipt of a notification.

Syntax

```
typedef void (*subscriber_notify_cb_t) (void *arg,
                                       char *event,
                                       char *data,
                                       int datalen);
```

Parameters

arg	Context pointer passed to subscribe (notify_arg).
event	The event reference (URI). The notification event reference matches the subscription, but may contain additional information called event attributes, such as a uid.
data	The body of the notification. A MIME object.
datalen	Length of the data.

Returns

Zero if successful, non-zero otherwise.

subscriber_new_a

Purpose

Creates a new asynchronous subscriber.

Syntax

```
void subscriber_new_a (pas_dispatcher_t *disp,
                     void *worker,
                     const char *host,
                     unsigned short port,
                     subscriber_cb_t cbdone,
                     void *cbarg);
```

Parameters

disp	Thread dispatcher context returned by <code>pas_dispatcher_new</code> .
worker	Application worker. If not NULL, grouped with existing workers created by ENS to service this subscriber session. Used to prevent multiple threads from accessing the subscriber data at the same time. Only usable if the caller creates and dispatches the GDisp context.
host	Notification server host name or IP address.
port	Subscription service port number.
cbdone	The callback invoked when the subscriber session becomes active and subscriptions can be issued. There are three parameters to <code>cbdone</code> :* <code>cbarg</code> The first argument. * A status code. If non-zero, the subscriber could not be created; value specifies cause of the failure. * The new active subscriber (<code>subscriber_t</code>).
cbarg	First argument of <code>cbdone</code> .

Returns

Nothing. It passes the new active subscriber as third argument of `cbdone` callback.

subscriber_new_s

Purpose

Creates a new synchronous subscriber.

Syntax

```
subscriber_t *subscriber_new_s (pas_dispatcher_t *disp,
                               const char *host,
                               unsigned short port);
```

Parameters

disp	Publish and subscribe dispatcher returned by <code>pas_dispatcher_new</code> .
worker	Application worker. If not NULL, grouped with existing workers created by ENS to service this publisher session. Used to prevent multiple threads from accessing the publisher data at the same time. Only usable if the caller creates and dispatches the GDisp context.
host	Notification server host name or IP address.
port	Subscription service port number.

Returns

A new active subscriber (`subscriber_t`).

subscribe_a

Purpose

Establishes an asynchronous subscription.

Syntax

```
void subscribe_a (subscriber_t *subscriber,  
                 const char *event_ref,  
                 subscriber_notify_cb_t notify_cb,  
                 void *notify_arg,  
                 subscriber_cb_t cbdone,  
                 void *cbarg):
```

Parameters

subscriber	The subscriber.
event_ref	The event reference. This is a URI identifying the event's source.
notify_cb	The callback invoked upon receipt of a notification matching this subscription.
notify_arg	The first argument of <code>notify_arg</code> . May be called at any time, by any thread, while the subscription is still active.
cbdone	Called when an unsubscribe completes. It has three Parameters: * <code>cbarg</code> (see below). * Status code. * A pointer to an opaque subscription object.
cbarg	The first argument of <code>cbdone</code> .

Returns

Nothing.

unsubscribe_a

Purpose

Cancels an asynchronous subscription.

Syntax

```
void unsubscribe_a (subscriber_t *subscriber,  
                  subscription_t *subscription,  
                  subscriber_cb_t cbdone,  
                  void *cbarg);
```

Parameters

subscriber	The disappearing subscriber.
subscription	The subscription to cancel.
cbdone	Called when an unsubscribe completes. It has three parameters: * cbarg (see below). * Status code. * A pointer to an opaque subscription object.
cbarg	The first argument of cbdone.

Returns

Nothing.

subscriber_delete

Purpose

Terminates a subscriber.

Syntax

```
void subscriber_delete (subscriber_t *subscriber);
```

Parameters

subscriber	The subscriber to delete.
------------	---------------------------

Returns

Nothing.

subscriber_get_publisher

Purpose

Creates a publisher, using the credentials of the subscriber.

Syntax

```
struct publisher_struct *subscriber_get_publisher (subscriber_t *subscriber);
```

Parameters

subscriber	The subscriber whose credentials are used to create the publisher.
------------	--

Returns

The publisher, or NULL if creation failed. In case the creation fails, use the `publisher_new`.

renl_create_subscriber

Purpose

Creates the subscription part of an RENL.

Syntax

```
renl_t *renl_create_subscriber (subscription_t *subscription,  
                               const char *renl_id,  
                               const char *publisher);
```

Parameters

subscription	The subscription.
renl_id	The unique RENL identifier. This allows two peers to be able to set up multiple RENLs between them.
publisher	The authenticated identity of the peer.

Returns

The opaque RENL object.

renl_cancel_subscriber

Purpose

This cancels an RENL. It does not cancel a subscription. It tells ENS not to acknowledge any more notifications received for this subscription. It destroys the RENL object, the application may no longer use this RENL. All RENLs are automatically destroyed when the subscription is canceled. Therefore, this function does not need to be called to free RENL-related memory before deleting a subscriber.

Syntax

```
void renl_cancel_subscriber (renl_t *renl);
```

Parameters

renl	The RENL to cancel.
------	---------------------

Returns

Nothing.

Publish and Subscribe Dispatcher API

The Publish and Subscribe Dispatcher API includes one definition and four functions:

- [pas_dispatcher_t](#)
- [pas_dispatcher_new](#)

- pas_dispatcher_delete
- pas_dispatch
- pas_shutdown



Note

The only thread dispatcher supported is GDisp (libasync).

pas_dispatcher_t

Purpose

A publish and subscribe dispatcher.

Syntax

```
typedef struct pas_dispatcher_struct pas_dispatcher_t;
```

Parameters

None.

Returns

Nothing.

pas_dispatcher_new

Purpose

Creates or advertises a dispatcher.

Syntax

```
pas_dispatcher_t *pas_dispatcher_new (void *disp);
```

Parameters

dispcx	The dispatcher context. If NULL, } }to start dispatching notifications, the application must call { {pas_dispatch. If not NULL, the dispatcher is a libasync dispatcher.
--------	--

Returns

The dispatcher to use when creating publishers or subscribers (pas_dispatcher_t).

pas_dispatcher_delete

Purpose

Destroys a dispatcher created with pas_dispatcher_new.

Syntax

```
void pas_dispatcher_delete (pas_dispatcher_t *disp);
```

Parameters

disp	The event notification client environment.
------	--

Returns

Nothing.

pas_dispatch

Purpose

Starts the dispatch loop of an event notification environment. It has no effect if the application uses its own thread pool.

Syntax

```
void pas_dispatch (pas_dispatcher_t *disp);
```

Parameters

disp	The new dispatcher.
------	---------------------

Returns

Nothing.

pas_shutdown

Purpose

Stops the dispatch loop of an event notification environment started with `pas_dispatch`. It has no effect if an application-provided dispatcher was passed to `pas_dispatcher_new`.

Syntax

```
void pas_shutdown (pas_dispatcher_t *disp);
```

Parameters

disp	The dispatcher context to shutdown.
------	-------------------------------------

Returns

Nothing.

Chapter 4. Messaging Server Specific Information

Oracle Communications Messaging Server Specific Information

This information describes the Messaging Server specific items you need to use the ENS APIs.

Topics:

- [Event Notification Types and Parameters](#)
- [Implementation Notes](#)

Event Notification Types and Parameters

For Messaging Server, there is only one event reference, which can be composed of several parameters. There are various types of event notifications. The following table lists the event types supported by Messaging Server and gives a description of each:

Event Types

Event Types	Description
ChangeFlag This event type is available starting in Messaging Server 7 Update 4 Patch 24 .	Shows change status as "1" add, "2" remove, or "3" replace.
DeleteMsg	Messages marked as "Deleted" are removed from the mailbox. This is the equivalent to IMAP expunge.
Login	User logged in from IMAP, HTTP, or POP.
Logout	User logged out from IMAP, HTTP, or POP.
NewMsg	New message was received by the system into the user's mailbox. Can have a payload of message headers and body.
OverQuota	Operation failed because the user's mailbox exceeded one of the quotas (diskquota, msgquota). The MTA channel holds the message until the quota changes or the user's mail box count goes below the quota. If the message expires while it is being held by the MTA, it will be expunged.
PurgeMsg	Message expunged (as a result of an expired date) from the mailbox by the server process imexpire. This is a server side expunge, whereas DeleteMsg is a client side expunge. This is not a purge in the true sense of the word.
ReadMsg	Message in the mailbox was read (in the IMAP protocol, the message was marked Seen).
TrashMsg	Message was marked for deletion by IMAP or HTTP. The user may still see the message in the folder, depending on the mail client's configuration. The messages are to be removed from the folder when an expunge is performed.
UnderQuota	Quota went back to normal from OverQuota state.
UpdateMsg	Message was appended to the mailbox (other than by NewMsg). for example, the user copied an email message to the mailbox. Can have a payload of message headers and body.

The following applies to the above supported event types:

- For `NewMsg` and `UpdateMsg`, message pay load is turned off by default to prevent overloading ENS. For information on how to enable the payload, see [Payload](#). No other event types support a payload.
- Event notifications can be generated for changes to the `INBOX` alone, or to the `INBOX` and all other folders. The following configuration variable allows for `INBOX` only (value = 0), or for both the `INBOX` and all other folders (value = 1):

```
local.store.notifyplugin.noneInbox.enable
```

The default setting is for `INBOX` only (value = 0).

Note

There is no mechanism to select folders; all folders are included when the variable is enabled (value = 1).

- The `NewMsg` notification is issued only after the message is deposited in the user mailbox (as opposed to "after it was accepted by the server and queued in the message queue").

- Every notification carries several pieces of information (called parameters) depending on the event type, for example, `NewMsg` indicates the IMAP `uid` of the new message. For details on the parameters each event type takes, see [Available Parameters for Each Event Type](#).
- Events are not generated for POP3 client access.
- All event types can be suppressed by issuing `XNOTNOTIFY`. For example, an IMAP script used for housekeeping only (the users are not meant to be notified) might issue it to suppress all events.

Parameters

iBiff uses the following format for the ENS event reference:

```
enp://127.0.0.1/store_?param=_value&param1=_value1&param2=_value2
```

The event key `enp://127.0.0.1/store` has no significance other than its uniqueness as a string. For example, the hostname portion of the event key has no significance as a hostname. It is simply a string that is part of the URI. However, the event key is user configurable. The list of iBiff event reference parameters is listed in tables [Mandatory Event Reference Parameters](#) and [Optional Event Reference Parameters](#) that follow.

The second part of the event reference consists of parameter-value pairs. This part of the event reference is separated from the event key by a question mark (?). The parameter and value are separated by an equals sign (=). The parameter-value pairs are separated by an ampersand (&). Note that there can be empty values, for which the value simply does not exist.

The following table describes the mandatory event reference parameters that need to be included in every notification.

Mandatory Event Reference Parameters

Parameter	Data Type	Description
<code>evtType</code>	string	Specifies the event type.
<code>hostname</code>	string	The hostname of the machine that generated the event.
<code>mailboxName</code>	string	Specifies the mailbox name in the message store. The <code>mailboxName</code> has the format <code>uid@domain</code> , where <code>uid</code> is the user's unique identifier, and <code>domain</code> is the domain the user belongs to. The <code>@domain</code> portion is added only when the user does not belong to the default domain (i.e. the user is in a hosted domain).
<code>pid</code>	integer	ID of the process that generated the event.
<code>process</code>	string	Specifies the name of the process that generated the event.
<code>timestamp</code>	64-bit integer	Specifies the number of milliseconds since the epoch (midnight GMT, January 1, 1970).

The table [Optional Event Reference Parameters](#) describes optional event reference parameters, which might be seen in the event depending on the event type (see [Available Parameters for Each Event Type](#)).

Optional Event Reference Parameters

Parameter	Data Type	Description
client	IP address	The IP address of the client logging in or out.
diskQuota	signed 32-bit integer	Specifies the disk space quota in kilobytes. The value is set to -1 to indicate no quotas.
diskUsed	signed 32-bit integer	Specifies the amount of disk space used in kilobytes.
hdrLen	unsigned 32-bit integer	Specifies the size of the message header. Note that this might not be the size of the header in the payload, because it might have been truncated.
imapUid	unsigned 32-bit integer	Specifies the IMAP uid parameter.
lastUid	unsigned 32-bit integer	Specifies the last IMAP uid value that was used.
numDel	unsigned 32-bit integer	Specifies the number of messages marked as deleted in the mailbox.
numMsgs	unsigned 32-bit integer	Specifies the number of total messages in the mailbox.
numMsgsMax	signed 32-bit integer	Specifies the quota for the maximum number of messages. The value is set to -1 to indicate no quotas.
numSeen	unsigned 32-bit integer	Specifies the number of messages in the mailbox marked as seen (read).
size	unsigned 32-bit integer	Specifies the size of the message. Note that this may not be the size of payload, since the payload is typically a truncated version of the message.
uidValidity	unsigned 32-bit integer	Specifies the IMAP uid validity parameter.



Note

Subscribers should allow for undocumented parameters when parsing the event reference. This allows for future compatibility when new parameters are added.

The following table shows the parameters that are available for each event type. For example, to see which parameters apply to a `TrashMsg` event, look in the column header for "ReadMsg, TrashMsg" and then note that these events can use `numDel`, `numMsgs`, `numSeen`, and `userValidity`.

Available Parameters for Each Event Type

**Note**

Oracle reserves the right to change "no" to "yes" at any time needed.

Parameter	NewMsg, UpdateMsg	ReadMsg, TrashMsg	DeleteMsg, PurgeMsg	Login, Logout	OverQuota, UnderQuota
client	No	No	No	Yes	No
diskQuota	No	No	No	No	Yes
diskUsed	No	No	No	No	Yes
hdrLen	Yes	No	No	No	No
imapUid	Yes	No	Yes	No	No
lastUid	No	No	Yes	No	No
numDel	No	Yes	No	No	No
numMsgs	Yes	Yes	Yes	No	Yes
numMsgsMax	No	No	No	No	Yes
numSeen	No	Yes	No	No	No
size	Yes	No	No	No	No
uidValidity	Yes	Yes	Yes	No	No
userid	No	No	No	Yes	No

Payload

ENS allows a payload for two event types: `NewMsg`, and `UpdateMsg`; the other event types do not carry a payload. The payload portion of these two notifications can contain any of the following data:

- No header or body data (default setting)
- Message header data only
- Message body data only
- Both message header and body data

The amount and type of data sent as the payload of the ENS event is determined by the configuration parameters found in [Payload Configuration Parameters](#).

Payload Configuration Parameters

The following table describes the payload configuration parameters; these parameters were updated in Messaging Server 7 Update 4.

Configuration Parameter	Description
<code>local.store.notifyplugin.*.maxbodysize</code>	Specifies the maximum size (in bytes) of the body that will be transmitted with the notification. Syntax: <code>uint32</code> Default: 0
<code>local.store.notifyplugin.*.maxheadersize</code>	Specifies the maximum size (in bytes) of the header that will be transmitted with the notification. Syntax: <code>uint32</code> Default: 0

Note that both parameters are set to zero as the default so that no header or body data is sent with ENS notifications.

Examples

The following example shows a `NewMsg` event reference (it is actually a single line that is broken up to several lines for readability):

```
enp://127.0.0.1/store?evtType=NewMsg&timestamp=1047488403000&
hostname=eman&process=imta&pid=476&mailboxName=testuser&numMsgs=16
&uidValidity=1046993605&imapUid=62&size=877&hdrLen=814
```

In this example, for the `DeleteMsg` event. Messages marked as deleted by IMAP or HTTP were expunged. The user would not see the message in the folder any more.

```
enp://127.0.0.1/store?evtType=DeleteMsg&timestamp=1047488588000&
hostname=eman&process=imapd&pid=419&mailboxName=testuser&
numMsgs=6&uidValidity=1046993605&imapUid=61&lastUid=62
```

And a third example shows a `ReadMsg` event. Message was marked as Seen by IMAP or HTTP.

```
enp://127.0.0.1/store?evtType=ReadMsg&timestamp=1047488477000&
hostname=eman&process=imapd&pid=419&mailboxName=testuser&
uidValidity=1046993605&numSeen=11&numDel=9&numMsgs=16
```

Implementation Notes

The current implementation does not provide security on events that can be subscribed to. Thus, a user could register for all events, and portions of all other users' mail. Because of this it is strongly recommended that the ENS subscriber be on the "safe" side of the firewall at the very least.

Beginning with Messaging Server 7 Update 4:

The ENS server supports two options to control TCP access to the ENS server (`service.ens.domainallowed` and `service.ens.domainnotallowed`). These options work the same way as the equivalent options for [POP](#), [IMAP](#), and [HTTP](#). These options replace the functionality of the `ENS_ACCESS` environment variable that was included in the legacy ENS server.

Chapter 5. Calendar Server Specific Information

Calendar Server Specific Information

This information describes the Calendar Server specific items you need to use the ENS APIs.

Topics:

- [Calendar Server Notifications](#)
- [ENS Sample Code for Calendar Server](#)

Calendar Server Notifications

There are two parts to the format of an Calendar Server notification:

- The event reference: A URL identifying the event.
- The payload: The data describing the event. Three different payload formats are supported: binary, text/calendar, and text/XML.

There are two types of calendar notifications:

- [Alarm Notifications](#)--relay reminders
- [Calendar Update Notifications](#)--distribute changes to the calendar database

Alarm Notifications

Alarm notifications relay reminders. They are published by the `csadmin` daemon whenever it wants to send a reminder. The default subscriber for these alarms in Communications Suite is the `csnotifyd` daemon. Notifications consumed by `csnotifyd` have a binary payload and are acknowledged (reliable).

Additionally, the server can be configured to generate one additional notification for each reminder, which can be consumed by a third party notification infrastructure.

The [Alarm Notifications](#) table shows the configuration variables that enable these notifications.

Alarm Notifications

ics.conf	Default Value	Description
<code>caldb.serveralarms.binary.url</code>	<code>enp:///ics/alarm</code>	Used by <code>csadmin</code> and <code>csnotifyd</code> to send SMTP reminders.
<code>caldb.serveralarms.binary.enable</code>	<code>yes</code>	Enable or disable the default alarm (binary) transport provided by the Calendar Server product.
<code>caldb.serveralarms.url</code>	NULL	ENS topic URL for custom implementation. If this is NULL, then no formatted messages will be published. The <code>ics.conf</code> value will be set to <code>enp:///ics/alarm</code> .
<code>caldb.serveralarms.contenttype</code>	<code>text/xml</code>	Content MIME type of formatted message.
<code>caldb.berkeleydb.alarmretrytime</code>	<code>300</code>	Retry interval in seconds for failed deliveries. Specify zero (0) to disable retry.

Event URL parameters are the same for either one:

- `calid` - Calendar ID
- `uid` - Component, either event or todo (task) ID
- `rid` - Recurrence ID
- `aid` - Alarm ID
- `comptype` - An event or a todo (task)
- URI

Calendar Update Notifications

Calendar update notifications distribute changes to the calendar database. They are published by the `cshttpd` or `csdwpd` daemons whenever a change is made to the database (if the notification is enabled for this type of change).

There are eleven types of notifications. The following table lists each type of calendar update notification, its `ics.conf` parameters, and their default values.

Calendar Update Notifications

Types	ics.conf Parameters	Default Value
Attendee refresh actions	caldb.berkeleydb.ensmsg.refreshevent caldb.berkeleydb.ensmsg.refreshevent.url caldb.berkeleydb.ensmsg.refreshevent.contenttype	no enp:///ics/caleventrefresh text/xml
Attendee reply action	caldb.berkeleydb.ensmsg.replyevent caldb.berkeleydb.ensmsg.replyevent.url caldb.berkeleydb.ensmsg.replyevent.contenttype	no enp:///ics/caleventreply text/xml
Calendar creation	caldb.berkeleydb.ensmsg.createcal caldb.berkeleydb.ensmsg.createcal.url	yes enp:///ics/calendarcreate
Calendar deletion	caldb.berkeleydb.ensmsg.deletecal caldb.berkeleydb.ensmsg.deletecal.url	yes enp:///ics/calendardelete
Calendar modification	caldb.berkeleydb.ensmsg.modifycal caldb.berkeleydb.ensmsg.modifycal.url	yes enp:///ics/calendarmodify
Event creation	caldb.berkeleydb.ensmsg.createevent caldb.berkeleydb.ensmsg.createevent.url	yes enp:///ics/caleventcreate
Event modification	caldb.berkeleydb.ensmsg.modifyevent caldb.berkeleydb.ensmsg.modifyevent.url	yes enp:///ics/caleventmodify
Event deletion	caldb.berkeleydb.ensmsg.deleteevent caldb.berkeleydb.ensmsg.deleteevent.url	yes enp:///ics/caleventdelete
Todo (task) creation	caldb.berkeleydb.ensmsg.createtodo caldb.berkeleydb.ensmsg.createtodo.url	yes enp:///ics/caltodocreate
Todo (task) modification	caldb.berkeleydb.ensmsg.modifytodo caldb.berkeleydb.ensmsg.modifytodo.url	yes enp:///ics/caltodomodify
Todo (task) deletion	caldb.berkeleydb.ensmsg.deletetodo caldb.berkeleydb.ensmsg.deletetodo.url	yes enp:///ics/caltododelete

Event URL parameters include:

- calid - Calendar ID
- uid - Component, either event or todo (task) ID
- rid - Recurrence ID

Advanced Topics

Normally, ENS notifications for attendee replies and organizer refreshes are published to the `caldb.berkeleydb.ensmsg.modifyevent` topic along with other modifications. By setting the `ics.conf` parameter `caldb.berkeleydb.ensmsg.advancedtopics` to "yes" (the default is "no"), the ENS notifications can be published to separate modify, reply and refresh topics. This allows the consumer of the notification to understand more precisely what type of transaction triggered the notification.

The following table shows the topics ENS publishes notifications to depending on the setting of the `ics.conf` parameter `caldb.berkeleydb.ensmsg.advancedtopics`.

Advanced Topics Parameter

Value of Advanced Topics Parameter	Topics to Which ENS Publishes Attendee Notifications
yes	caldb.berkeleydb.ensmsg.modifyevent caldb.berkeleydb.ensmsg.refreshevent caldb.berkeleydb.ensmsg.replyevent
no	caldb.berkeleydb.ensmsg.modifyevent

WCAP appid parameter and X-Tokens

When ENS sends out notifications of modifications made to existing events, it returns two X-Tokens with the notification, X-NSCP-COMPONENT-SOURCE and X-NSCP-TRIGGERED-BY.

The contents of the X-NSCP-COMPONENT-SOURCE X-Token vary depending on who originated the event and the absence or presence of the appid parameter in the original WCAP command that requested the event.

If the appid parameter is present in the original WCAP command, ENS returns its value in the X-NSCP-COMPONENT-SOURCE X-Token. (Only certain commands take the appid parameter. See the *Calendar Server Programmer's Manual* for further information on the appid parameter.) Using this mechanism, applications can "tag" ENS notifications in order to detect which ones it originated. The value of the appid command is a character string of the application's choosing. If the appid parameter is missing, standard values are assigned to the X-Token depending on the origin, see the [Presence of appid and Value of X-Token X-NSCP-COMPONENT-SOURCE](#) for the standard values).

The X-Token, X-NSCP-TRIGGERED-BY holds the name (uid) of the organizer or attendee that triggered the notification regardless of the absence or presence of the appid parameter.

The [Presence of appid and Value of X-Token X-NSCP-COMPONENT-SOURCE](#) table shows the effect of the presence of the appid parameter in WCAP commands on the value of the X-Token X-NSCP-COMPONENT-SOURCE.

Presence of appid and Value of X-Token X-NSCP-COMPONENT-SOURCE

appid Present?	Value of X-Token X-NSCP-COMPONENT-SOURCE (with Request Origin)
no	WCAP (default)CALENDAR EXPRESS (from UI)ADMIN (from Admin tools)
yes	Value of appid

ENS Sample Code for Calendar Server

Calendar Server ships with a complete ENS implementation. If you wish to customize it, you may use the ENS APIs to do so. The following four code samples, a simple publisher and subscriber pair, and a reliable publisher and subscriber pair, illustrate how to use the ENS API. The sample code is provided with the product in the following directory:

```
/opt/SUNWics5/cal/csapi/samples/ens
```

Sample Publisher and Subscriber

This sample code pair establishes a simple interactive asynchronous publisher and subscriber.

Publisher Code Sample

```

/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * apub : simple interactive asynchronous publisher using
 *
 * Syntax:
 *   apub host port
 */
#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "publisher.h"

static pas_dispatcher_t *disp = NULL;
static publisher_t *_publisher = NULL;
static int _shutdown = 0;

static void _read_stdin();

static void _exit_usage()
{
    printf("\nUsage:\napub host port\n");
    exit(5);
}

static void _exit_error(const char *msg)
{
    printf("%s\n", msg);
    exit(1);
}

static void _call_shutdown()
{
    _shutdown = 1;
    pas_shutdown(disp);
}

static void _open_ack(void *arg, int rc, void *enc)
{
    _publisher = (publisher_t *)enc;
    (void *)arg;
    if (!_publisher)
    {
        printf("Failed to create publisher with status %d\n", rc);
        _call_shutdown();
        return;
    }
    _read_stdin();
    return;
}

static void _publish_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;
    free(arg);
    if (rc != 0)
    {

```

```

        printf("Publish failed with status %d\n", rc);
        _call_shutdown();
        return;
    }
    _read_stdin();
    return;
}
static void _read_stdin()
{
    static char input[1024];
    printf("apub> ");
    fflush(stdout);
    while (!_shutdown)
    {
        if ( !fgets(input, sizeof(input), stdin) )
        {
            continue;
        } else {
            char *message;
            unsigned int message_len;
            input[strlen(input) - 1] = 0; /* Strip off the \n */
            if (*input == '.' && input[1] == 0)
            {
                publisher_delete(_publisher);
                _call_shutdown();
                break;
            }
            message = strdup(input);
            message_len = strlen(message);
            publish(_publisher, "enp://siroe.com/xyz",message,
                message_len,
                _publish_ack, NULL, (void *)message, 0);
            return;
        }
    }
    return;
}
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
        port = (unsigned short)atoi(argv[2]);
    }
    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");
    publisher_new_a(disp, NULL, host, port, _open_ack, disp);
    pas_dispatch(disp);
}

```

```
_shutdown = 1;
```

```

    pas_dispatcher_delete(dispatcher);
    exit(0);
}

```

Subscriber Code Sample

```

/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * asub : example asynchronous subscriber
 *
 * Syntax:
 *   asub host port
 */
#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "subscriber.h"

static pas_dispatcher_t *disp = NULL;
static subscriber_t *_subscriber = NULL;
static subscription_t *_subscription = NULL;
static renl_t *_renl = NULL;

static void _exit_usage()
{
    printf("\nUsage:\nasub host port\n");
    exit(5);
}

static void _exit_error(const char *msg)
{
    printf("%s\n", msg);
    exit(1);
}

static void _subscribe_ack(void *arg, int rc, void *subscription)
{
    (void)arg;
    if (!rc)
    {
        _subscription = subscription;
        printf("Subscription successful\n");
    } else {
        printf("Subscription failed - status %d\n", rc);
        pas_shutdown(disp);
    }
}

static void _unsubscribe_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;
    (void *)arg;
}

```

```

    if (rc != 0)
    {
        printf("Unsubscribe failed - status %d\n", rc);
    }
    subscriber_delete(_subscriber);
    pas_shutdown(dispatcher);
}
static int _handle_notify(void *arg, char *url, char *str, int len)
{
    (void *)arg;
    printf("[%s] %.*s\n", url, len, (str) ? str : "(null)");
    return 0;
}
static void _open_ack(void *arg, int rc, void *enc)
{
    _subscriber = (subscriber_t *)enc;
    (void *)arg;
    if (rc)
    {
        printf("Failed to create subscriber with status %d\n", rc);
        pas_shutdown(dispatcher);
        return;
    }
    subscribe(_subscriber, "enp://siroe.com/xyz",
              _handle_notify, NULL,
              _subscribe_ack, NULL);
    return;
}
static void _unsubscribe(int sig)
{
    (int)sig;
    unsubscribe(_subscriber, _subscription, _unsubscribe_ack, NULL);
}
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
        port = (unsigned short)atoi(argv[2]);
    }
    dispatcher = pas_dispatcher_new(NULL);
    if (dispatcher == NULL) _exit_error("Can't create publisher");
    subscriber_new_a(dispatcher, NULL, host, port, _open_ack, NULL);
    pas_dispatch(dispatcher);
}

```

```

    pas_dispatcher_delete(dispatcher);
    exit(0);
}

```

Reliable Publisher and Subscriber

This sample code pair establishes a reliable asynchronous publisher and subscriber.

Reliable Publisher Sample

```

/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * rpub : simple *reliable* interactive asynchronous publisher.
 *       It is designed to be used in combination with rsub,
 *       the reliable subscriber.
 *
 * Syntax:
 *       rpub host port
 */
#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "publisher.h"

static pas_dispatcher_t *disp = NULL;
static publisher_t *_publisher = NULL;
static int _shutdown = 0;
static renl_t *_renl;
static void _read_stdin();

static void _exit_usage()
{
    printf("\nUsage:\nrpub host port\n");
    exit(5);
}

static void _exit_error(const char *msg)
{
    printf("%s\n", msg);
    exit(1);
}

static void _call_shutdown()
{
    _shutdown = 1;
    pas_shutdown(disp);
}

static void _renl_create_cb(void *arg, int rc, void *ignored)
{
    (void *)arg;
    (void *)ignored;
}

```

```

    if (!_publisher)
    {
        printf("Failed to create RENL - status %d\n", rc);
        _call_shutdown();
        return;
    }
    _read_stdin();
    return;
}
static void _publisher_new_cb(void *arg, int rc, void *enc)
{
    _publisher = (publisher_t *)enc;
    (void *)arg;
    if (!_publisher)
    {
        printf("Failed to create publisher - status %d\n", rc);
        _call_shutdown();
        return;
    }
    renl_create_publisher(_publisher, "renl_id", NULL,
                        _renl_create_cb, NULL);

    return;
}
static void _recv_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;
    if (rc < 0)
    {
        printf("Acknowledgment Timeout\n");
    } else if ( rc == 0) {
        printf("Acknowledgment Received\n");
    }
    fflush (stdout);
    _read_stdin();
    free(arg);
    return;
}
static void _read_stdin()
{
    static char input[1024];
    printf("rpub> ");
    fflush(stdout);
    while (!_shutdown)
    {
        if ( !fgets(input, sizeof(input), stdin) )
        {
            continue;
        } else {
            char *message;
            unsigned int message_len;
            input[strlen(input) - 1] = 0; /* Strip off the \n */
            if (*input == '.' && input[1] == 0)
            {
                publisher_delete(_publisher);
                _call_shutdown();
                break;
            }
        }
    }
}

```

```

    }
    message = strdup(input);
    message_len = strlen(message);

    /* five seconds timeout */
    publish(_publisher, "enp://siroe.com/xyz",
           message, message_len,
           NULL, _recv_ack, message, 5000);
    return;
}
}
return;
}
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
        port = (unsigned short)atoi(argv[2]);
    }
    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");
    publisher_new_a(disp, NULL, host, port, _publisher_new_cb,
                   NULL);
    pas_dispatch(disp);
    _shutdown = 1;
}

```

```

    pas_dispatcher_delete(dispatcher);
    exit(0);
}

```

Reliable Subscriber Sample

```

/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * asub : example asynchronous subscriber
 *
 * Syntax:
 *   asub host port
 */
#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "subscriber.h"

static pas_dispatcher_t *disp = NULL;
static subscriber_t *_subscriber = NULL;
static subscription_t *_subscription = NULL;
static renl_t *_renl = NULL;

static void _exit_usage()
{
    printf("\nUsage:\nasub host port\n");
    exit(5);
}

static void _exit_error(const char *msg)
{
    printf("%s\n", msg);
    exit(1);
}

static void _subscribe_ack(void *arg, int rc, void *subscription)
{
    (void)arg;
    if (!rc)
    {
        _subscription = subscription;
        printf("Subscription successful\n");
        _renl = renl_create_subscriber(_subscription, "renl_id", NULL);
    } else {
        printf("Subscription failed - status %d\n", rc);
        pas_shutdown(dispatcher);
    }
}

static void _unsubscribe_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;
}

```

```

    (void *)arg;
    if (rc != 0)
    {
        printf("Unsubscribe failed - status %d\n", rc);
    }
    subscriber_delete(_subscriber);
    pas_shutdown(dispatcher);
}
static int _handle_notify(void *arg, char *url, char *str, int len)
{
    (void *)arg;
    printf("[%s] %.*s\n", url, len, (str) ? str : "(null)");
    return 0;
}
static void _open_ack(void *arg, int rc, void *enc)
{
    _subscriber = (subscriber_t *)enc;
    (void *)arg;
    if (rc)
    {
        printf("Failed to create subscriber with status %d\n", rc);
        pas_shutdown(dispatcher);
        return;
    }
    subscribe(_subscriber, "enp://siroe.com/xyz", _handle_notify,
              NULL, _subscribe_ack, NULL);
    return;
}
static void _unsubscribe(int sig)
{
    (int)sig;
    unsubscribe(_subscriber, _subscription, _unsubscribe_ack, NULL);
}
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];

    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
        port = (unsigned short)atoi(argv[2]);
    }
    dispatcher = pas_dispatcher_new(NULL);
    if (dispatcher == NULL) _exit_error("Can't create publisher");
    subscriber_new_a(dispatcher, NULL, host, port, _open_ack, NULL);
    pas_dispatch(dispatcher);
    pas_dispatcher_delete(dispatcher);
}

```

```
    exit(0);  
}
```

Chapter 6. Debugging ENS

Debugging ENS

This information contains instructions for obtaining trace information that can be valuable for debugging problems with any program that uses the ENS API. This includes all servers that send notifications through `enpd`, `csadmin`, `csnotifyd`, the `iBiff` plug-in, `stored`, `imapd`. Trace information can be obtained by setting several environment variables.

This information applies to **Messaging Server 7 Update 3** and prior versions.

Topics:

- [Environment Variables](#)
- [How to Enable Debug Tracing](#)
- [Sample Debugging Sessions](#)

Environment Variables

Tracing can be done at both the GAP (generic request and reply protocol layer) and ENP (publish and subscribe protocol layer) levels. Also, service bus traces can be set. The default is for no logging or tracing.

The following environment variables can be set for GAP tracing:

- `GAP_DEBUG`
- `GAP_LOG_MODULES`
- `GAP_LOGFILE` (Calendar Server only)

The following environment variables can be set for ENP tracing:

- `XENP_TRACE`
- `ENS_DEBUG`
- `ENS_LOG_MODULES`
- `ENS_LOGFILE` (Calendar Server only)
- `ENS_STATS`

The following environment variable can be set for service bus tracing: `SERVICEBUS_DEBUG`.

GAP_DEBUG

The value is a positive integer which indicates the trace level. Each higher trace level includes the output from the levels below it. For example, if you set the trace level to 7, level 1-6 traces are also included. The default value for this variable is 4, but since `GAP_LOG_MODULES` defaults to zero (0), no logging is done.

While it is possible to set the variable to any integer value greater than 7 and less than 100, the effect will be the same as setting it to 7.

The following table lists the trace levels for the variable `GAP_DEBUG`:

Trace Level Values

Trace Level	Trace Level Name	Description
0	N/A	No output except emergency messages
1	NSLOG_ALERT	Alert messages
2	NSLOG_CRIT	Critical messages
3	NSLOG_ERR	Software error conditions
4	NSLOG_WARNING	Default; warning messages (user error conditions)
5	NSLOG_NOTICE	Normal but significant conditions
6	NSLOG_INFO	Informational messages
7	NSLOG_DEBUG	Debug messages
100	NSLOG_TRACE	Full trace

GAP_LOG_MODULES

Use this variable to obtain trace information specific to one or more functional modules in the GAP code. This variable is a bit map. That is, each bit set in the variable turns on tracing for a particular module.

More than one module can be specified at once. To specify multiple modules, add the individual values of the modules you want. For example, if you want to trace both the connection layer and the transaction modules, you set the value of this variable to 10; to get all modules, set the value to 15.

The following table lists the values for the variable `GAP_LOG_MODULES`:

GAP_LOG_MODULES Values

Value	Value Name	Description
0	N/A	Default; no modules logged.
1	GAPLOG_CONNECTION	Connection layer– socket input output calls
2	GAPLOG_SESSION	Session layer– session setup and closing
4	GAPLOG_TRANSACTION	Transaction creation– continuation and termination
8	GAPLOG_DISPATCHER	Thread dispatcher code– GDisp tracing

GAP_LOGFILE

This variable is used for Calendar Server only. This variable tells the system where to output GAP tracing. To send the output to a log file, set the variable to a text file name. The default (variable set to zero) sends GAP tracing to standard out.

XENP_TRACE

Use this variable to generate encoded data traces. Any non-zero value activates the trace.

ENS_DEBUG

Use this variable to trace functional (unencoded) client or server request responses.

The value is a positive integer which indicates the trace level. Each higher trace level includes the output from the levels below it. For example, if you set the trace level to 4, level 1-3 traces are also included.

While it is also possible to set the variable to any integer between 7 and 100, the effect will be the same as setting it to 7. That is, anything less than 100 but greater than 6 is treated the same.

The following table lists the trace level values for the `ENS_DEBUG` variable:

ENS_DEBUG Trace Level Values

Trace Level	Trace Level Name	Description
0	N/A	No output except emergency messages
1	NSLOG_ALERT	Alert messages
2	NSLOG_CRIT	Critical messages
3	NSLOG_ERR	Software error conditions
4	NSLOG_WARNING	Warning messages (user error conditions)
5	NSLOG_NOTICE	Normal but significant conditions
6	NSLOG_INFO	Informational messages
7	NSLOG_DEBUG	Debug messages
100	NSLOG_TRACE	Full trace

ENS_LOG_MODULES

Use this variable to obtain trace information specific to one or more functional modules in the ENS code. This variable is a bit map. That is, each bit set in the variable turns on tracing for a particular module.

More than one module can be specified at once. To specify multiple modules, add the individual values of the modules you want. For example, if you want to trace both the server and the RENL modules, you set the value of this variable to 10; to get all modules, set the value to 31.

The following table lists the values for the variable `ENS_LOG_MODULES`:

ENS_LOG_MODULES Values

Values	Value Names	Description
0	N/A	Default; no modules logged.
1	ENSLOG_CLIENT_API	Client API generated transactions
2	ENSLOG_SERVER	Server generated transactions
4	ENSLOG_UPUB	Publisher transactions
8	ENSLOG_RENL	Reliable event notifications
16	ENSLOG_STORE	ENS message store transactions

ENS_LOGFILE

This variable is used for Calendar Server only. This variable tells the system where to output ENS tracing. To send the output to a log file, set the variable to a text file name. The default (variable set to

zero) sends ENS tracing to standard out.

ENS_STATS

To have statistics printed periodically, set this variable to a non-zero value.

SERVICEBUS_DEBUG

Service Bus is a process monitoring system based on ENS, and is used in ENS. Any non-zero value causes service bus traces to be sent to standard out. There is no log file variable for service bus. To send the traces to a log file, temporarily redefine standard out to a text file name. During this time, all standard out messages will appear in the text file you create.

How to Enable Debug Tracing

The following procedure describes how to enable debug tracing.

To Start Tracing

1. If ENS is running, `stop enpd`.

To start and stop `enpd`, you must be in the `bin` directory.

For example:

- For Calendar Server on Unix, `/opt/SUNWics/cal/bin`.
- For Calendar Server on Windows, `C:\Program Files\Sun ONE Calendar Server\..\cal\bin`.



Note

You can enable debugging for specific services by stopping only that service, for example `stop csnotifyd`, instead of the entire ENS server.

2. Set all variables to the desired value.

For Unix:

- Bourne shell
`variable_name=value; export variable_name`

For example:

```
GAP_DEBUG=2; export GAP_DEBUG
```

- C shell
`setenv variable_name value`

For example:

```
setenv GAP_DEBUG 2
```

For Windows:

```
set variable_name=value
```

For example,

```
set GAP_DEBUG=2
```

3. If you want the traces to print to a log file, set the appropriate logfile variables (for `END_LOGFILE`, or `GAP_LOGFILE`) or temporarily redefine standard out to a text file.
4. Restart ENS— `start enpd`

If you only disabled one service rather than the whole ENS server, you start that service only, for example `start csnotifyd`.

Sample Debugging Sessions

The following are sample debugging sessions on the Messaging Server and Calendar Server.

Each example has three parts:

- Set Environment Variables
- Sample Trace Output
- Short Commentary

Example 1: For Messaging Server

Set Environment Variables

```
setenv LD_LIBRARY_PATH msg-svr-base/lib/  
stop-ens  
setenv SERVICEBUS_DEBUG 1  
setenv ENS_DEBUG 1  
setenv ENS_LOG_MODULES 1  
setenv GAP_DEBUG 1  
setenv GAP_LOG_MODULES 1  
setenv XENP_TRACE 1  
setenv ENS_STATS 1  
msg-svr-base/bin/enpd
```

Sample Trace Output

```

1 | servbus 3451633705 [26321]: Starting Service Bus
2 | servbus 3451636227 [26321]: Service Bus subscriber created
  successfully
3 | servbus 3451636286 [26321]: Service Bus Ready
4 |      XENP -> len=36 servbus:///monitor/ens|subs|00010000
5 |      XENP -> len=60 servbus:///service/ens&pid=26321
  &state=running|ntfy|00000000
6 |      XENP <- len=36 servbus:///monitor/ens|subs|00010000
7 |      XENP <- len=4  PACK
8 |      XENP <- len=60 servbus:///service/ens&pid=26321
  &state=running|ntfy|00000000
9 |secs: pub: pub/s: pub/s(i): ntfy: ntfy/s :ntfy/s(i):
10| 5 : 1: 0 : 0 : 0 : 0 : 0 :
11|10 : 1: 0 : 0 : 0 : 0 : 0 :
12|      XENP <-
  len=232enp://127.0.0.1/store?evtType=NewMs&mailboxName=ServiceAdmin&
  timestamp=1027623669000&process=2637&hostname=ketu&numMsgs=14&size=621
  &uidValidity=1025118712&imapUid=14&hdrLen=547&qUsed=16&qMax=-1&
  qMsgUsed=15&qMsgMax=-1|ntfy|00000000
13| 15 : 2: 0 : 0 : 0 : 0 : 0 :
14| 20 : 2: 0 : 0 : 0 : 0 : 0 :
15| 25 : 2: 0 : 0 : 0 : 0 : 0 :
16| 30 : 2: 0 : 0 : 0 : 0 : 0 :
17| 35 : 2: 0 : 0 : 0 : 0 : 0 :
18| 40 : 2: 0 : 0 : 0 : 0 : 0 :
19| 45 : 2: 0 : 0 : 0 : 0 : 0 :
20| 51 : 2: 0 : 0 : 0 : 0 : 0 :
21| 56 : 2: 0 : 0 : 0 : 0 : 0 :
22| 61 : 2: 0 : 0 : 0 : 0 : 0 :
23| 66 : 2: 0 : 0 : 0 : 0 : 0 :
24| 71 : 2: 0 : 0 : 0 : 0 : 0 :
25| 76 : 2: 0 : 0 : 0 : 0 : 0 :
26|secs: pub: pub/s: pub/s(i): ntfy: ntfy/s :ntfy/s(i):
27| 81 : 2: 0 : 0 : 0 : 0 : 0 :
28| 86 : 2: 0 : 0 : 0 : 0 : 0 :
29| 91 : 2: 0 : 0 : 0 : 0 : 0 :
30| 96 : 2: 0 : 0 : 0 : 0 : 0 :
31|101: 2: 0 : 0 : 0 : 0 : 0 :
32|106: 2: 0 : 0 : 0 : 0 : 0 :
33|111: 2: 0 : 0 : 0 : 0 : 0 :
34|116: 2: 0 : 0 : 0 : 0 : 0 :
35|121: 2: 0 : 0 : 0 : 0 : 0 :
36|126: 2: 0 : 0 : 0 : 0 : 0 :
37|131: 2: 0 : 0 : 0 : 0 : 0 :
38|136: 2: 0 : 0 : 0 : 0 : 0 :
39|141: 2: 0 : 0 : 0 : 0 : 0 :
40|146: 2: 0 : 0 : 0 : 0 : 0 :
41|151: 2: 0 : 0 : 0 : 0 : 0 :
42|^C
43| XENP -> len=60 servbus:///service/ens&pid=26321
  &state=stopped|ntfy|00000000
44|servbus 3466881202 [26321]: Service Bus going away
45|servbus 3466881542 [26321]: Failed to create subscriber-
  error-1

```

Short Commentary

The following table contains comments that apply to the lines of the preceding trace output:

Line Number	Comment
1 - 8	Printed upon startup
9 - 11 and 13 - 41	Periodic statistics print out
12	A message is sent
42	Control-c stopped operation. This was done to end the sample only. Not recommended for stopping processes normally.

Example 2: For Messaging Server

Set Environment Variables

```
1 | (293 root) setenv ENS_DEBUG 99
2 | (294 root) setenv ENS_LOG_MODULES 63
3 | (295 root) msg-svr-base/bin/enpd
Sample Trace Output
4 | ENS 3588422667 [26400]: LOGIN 2
5 | ENS 3588423361 [26400]: _enp_session_open_cb : new session id=2
created
6 | ENS 3588423380 [26400]: recorded new subscription : 0001;
servbus:///monitor/ens
7 | ENS 3588423395 [26400]: subscribe
(event=servbus:///monitor/ens, sid=2) = 0
8 | ENS 3588423403 [26400]: publish
(event=servbus:///service/ens&pid=26400&state=running, sid=2)
9 | ENS 3588423414 [26400]: publish
(event=servbus:///service/ens&pid=26400&state=running, sid=2) = 0
10 | ENS 3588423825 [26400]: _ens_rcv_request_cb: sid=2
op=1 id=00010000
11 | ENS 3588423842 [26400]: simple|store_req
(servbus:///monitor/ens#2) =2,servbus:///monitor/ens
12 | ENS 3588423848 [26400]: simple|store_evt
(servbus:///monitor/ens#2) = 2,servbus:///monitor/ens
13 | ENS 3588423853 [26400]: SUBS 2 servbus:///monitor/ens
00010000
14 | ENS 3588424389 [26400]: _ens_rcv_request_cb: sid=2
op=2 id=00000000
15 | ENS 3588424395 [26400]: NTFY 2 servbus:///service/ens
&pid=26400&state=running
16 | ENS 3588424409 [26400]:ens_notify
(event=servbus:///service/ens&pid=26400&state=running,
id=00000000,sid=2):no match
17 | ENS 3588503451 [26400]: LOGIN 3
18 | ENS 3588504099 [26400]: LOGIN 4
19 | ENS 3588504938 [26400]: LOGIN 5
```

```

20 | ENS 3588505284 [26400]: LOGIN 6
21
22 | ENS 3591631839 [26400]: LOGIN 7
23 | ENS 3591637445 [26400]: _ens_rcv_request_cb: sid=7
   op=2 id=00000000
24 | ENS 3591637452 [26400]: NTFY 7
   enp://127.0.0.1/store?evtType=NewMsg
   &mailboxName=ServiceAdmin&timestamp=1027625056000&process=2646
   &hostname=ketu&numMsgs=19&size=621&uidValidity=1025118712
   &imapUid=19&hdrLen=547&qUsed=19&qMax=-1&qMsgUsed=20&qMsgMax=-1
25 | ENS 3591637467 [26400]:ens_notify
   (event=enp://127.0.0.1/store?evtType=NewMsg
   &mailboxName=ServiceAdmin&timestamp=1027625056000&process=2646
   &hostname=ketu&numMsgs=19&size=621&uidValidity=1025118712
   &imapUid=19&hdrLen=547&qUsed=19&qMax=-1&qMsgUsed=20
   &qMsgMax=-1, id=00000000, sid=7): no match
26 |
27 | ENS 3595049771 [26400]: session closing 7
28 | ^CENS 3596193757 [26400]:publish
   (event=servbus:///service/ens&pid=26400&state=stopped, sid=2)
29 | ENS 3596193782 [26400]:publish
   (event=servbus:///service/ens&pid=26400&state=stopped, sid=2) = 0
30 | ENS 3596193987 [26400]: pas_dispatcher_delete : clean up
   starting
31 | ENS 3596194018 [26400]: _enp_session_closing_cb : closing
   session id=2
32 | ENS 3596194024 [26400]: destroying subscription :0001;
   servbus:///monitor/ens
33 | ENS 3596194041 [26400]: pas_dispatcher_delete : 0 client(s) have
   been bumped
34 | ENS 3596194065 [26400]: session closing 2
35 | ENS 3596194075 [26400]: simple|remov_evt
   (2, servbus:///monitor/ens)
36 | ENS 3596194107 [26400]: session closing 3

```

```
37 | ENS 3596194216 [26400]: session closing 4
38 | ENS 3596194281 [26400]: session closing 5
39 | ENS 3596195039 [26400]: session closing 6
```

Short Commentary

The following table contains comments that apply to the lines of the preceding trace output:

Line Number	Comment
1 - 20	Initialization
22-26	Sent email message
27	Printed asynchronously
28	Control-c stopped operation. This was done to end the sample only. Not recommended for stopping processes normally.
29-39	enpd exiting