Enea® LINX Gateway User's Guide

- 1. LINX Gateway Overview
    - 1.1 Technical Components
- 2. Configuring the LINX Gateway Server
    - 2.1 Starting the LINX Gateway Server
    - 2.2 Restarting the LINX Gateway server
    - 2.3 Configuration File
- 3. Write Gateway Client Applications
    - 3.1 Create and Destroy Clients
    - 3.2 Signals in Gateway Clients
    - 3.3 Obtain Endpoint IDs via Hunt
    - 3.4 Supervision
    - 3.5 Non-Blocking Receive Mode
    - 3.6 Error Handling
        - 3.6.1 Using the Error Handler
        - 3.6.2 Error Handling Using Checkup Calls
- 4. Find Gateway
- 5. linxgwcmd - Gateway Command Tool

Document version: 1.0

Copyright © Enea Software AB 2010.

Enea®, Enea OSE®, and Polyhedra® are the registered trademarks of Enea AB and its subsidiaries. Enea OSE®ck, Enea OSE® Epsilon, Enea® Element, Enea® Optima, Enea® LINX, Enea® Accelerator, Polyhedra® FlashLite, Enea® dSPEED, Accelerating Network Convergence™, Device Software Optimized™, and Embedded for Leaders™ are unregistered trademarks of Enea AB or its subsidiaries. Linux is a registered trademark of Linus Torvalds. Any other company, product or service names mentioned in this document are the registered or unregistered trademarks of their respective owner.
The source code included in LINX for Linux is released partly under the GPL (see COPYING file) and partly under a BSD type license - see license text in each source file.

Disclaimer: The information in this document is subject to change without notice and should not be construed as a commitment by Enea Software AB.

1. LINX Gateway Overview

The following describes the gateway concepts and the way the gateway can be used as a part of a system. It describes how to use, configure and write clients for the Enea® LINX Gateway server.

The LINX Gateway server allows existing user applications using the OSE Gateway client API to communicate seamlessly with a LINX cluster without rewriting or even recompiling.

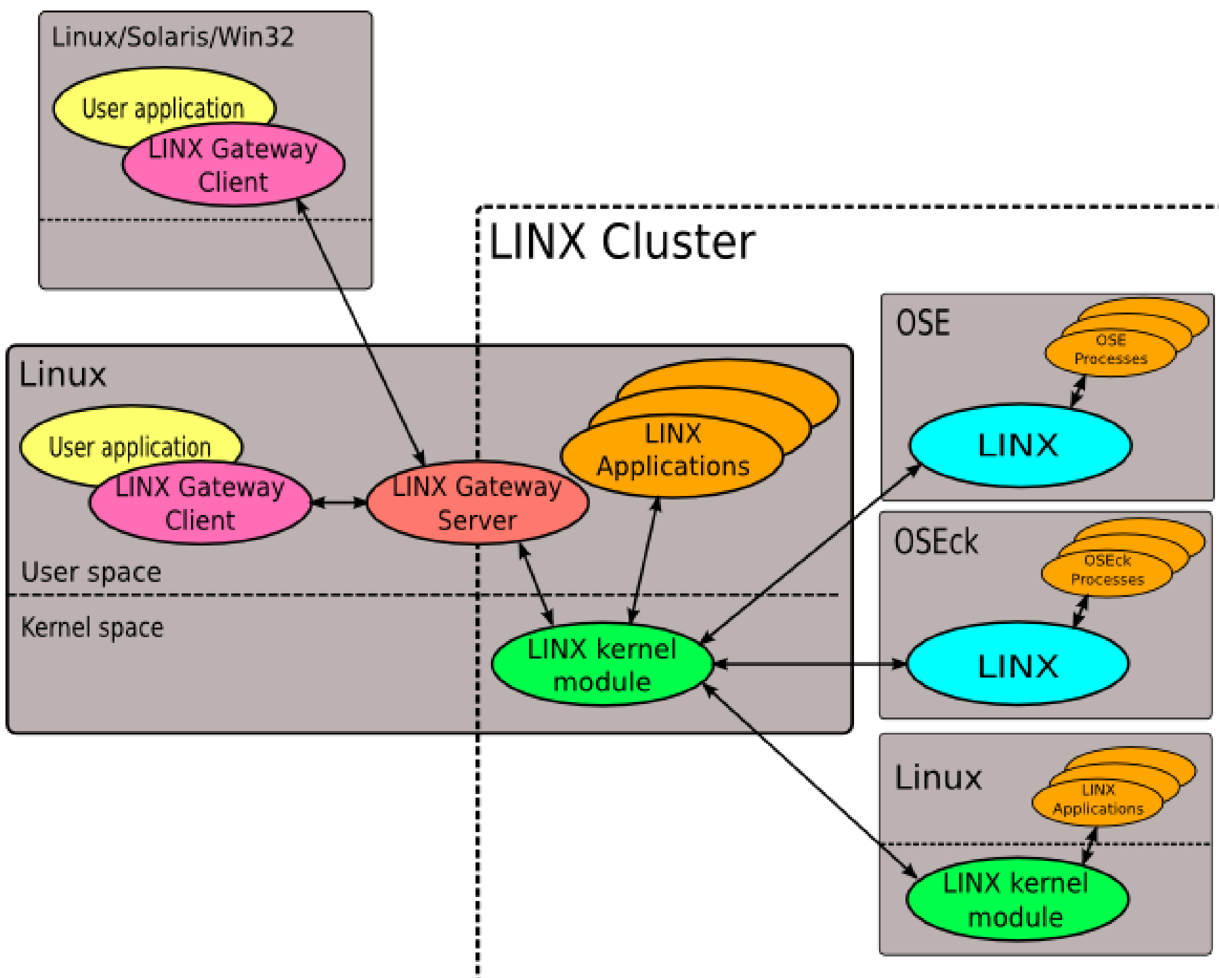
1.1 Technical Components

On a Linux system with the LINX kernel module installed the LINX Gateway Server is used to make the LINX network available to clients using the Gateway client. The clients could either run on the same node as the LINX Gateway Server or connect to a LINX Gateway Server running on a remote node.

The client library contains an LINX/OSE-like API for signal communication with the Gateway server and with the LINX endpoints connected to it.

The Gateway server handles new clients, and responds to broadcast messages from clients, to notify clients about its existence. The Gateway Server client processes handle the Gateway clients in the LINX environment after they have been created.

Example of a LINX cluster and how application using the Gateway client can connect to the LINX cluster via the LINX Gateway server.



2. Configuring the LINX Gateway Server

The LINX Gateway server is configured through a file that is read at startup and whenever the gateway receives a SIGHUP signal. When a SIGHUP signal is received all clients processes are killed and all client connections are closed.

2.1 Starting the LINX Gateway Server

At startup, LINX Gateway server must read a configuration file to set up the system. The configuration file can be specified on the command line, otherwise the LINX Gateway server looks for a configuration file named linxgws.conf in the /etc directory on the Linux node.

2.2 Restarting the LINX Gateway server

If the server receives a SIGHUP during execution, the configuration file is re-read and the server is restarted with the new configuration.

2.3 Configuration File

The configuration file is needed when starting an LINX Gateway Server. The configuration file is a text file, an example configuration file is included in linxgw/linxgws/example.conf.

Gateway name

Specifies the name of the LINX Gateway server, the name is advertised on the broadcast UDP port to all clients.

Example: `GATEWAY_NAME=my_gateway`

Interface name

Specifies what network interface to use when broadcasting advertisements, the interface has to be configured with an IP address.

Example: `INTERFACE_NAME=eth0`

Public Port

pecifies the TCP port number on which the LINX Gateway Server will accept connections, if not specified the default port number 16384 will be used.

Example: `PUBLIC_PORT=35700`

Broadcast Port

Specifies the UDP port number on which the LINX Gateway server will advertise its name and the TCP port on that it accepts connections. If the variable is left empty the default port number 21768 will be used.

Example: `BROADCAST_PORT=21444`

3. Write Gateway Client Applications

The following describes how to write clients for the LINX Gateway using the client API.

3.1 Create and Destroy Clients

Before a client can communicate through the LINX Gateway Server, it has to create an OSEGW connection object, and this is done by calling the osegw_create function. The function takes the name of the client as an argument, and this name is the one that other clients or LINX endpoints in the LINX cluster can hunt for. Other arguments to the function are a user number, the TCP address, port to the LINX Gateway server, an authentication string, a function pointer to an error handler, and a pointer to a user object. The latter pointer is passed to the error handler if called.

The error handler is optional. If not wanted, use NULL for both the error handler function and the user object handle.

```
gw = osegw_create("client", 0, "tcp://localhost:12357", NULL, NULL, NULL);
```

When the osegw_create function is called, an object containing necessary information about the connection is created on the client side. This object also hosts a list of signal buffers that belongs to the connection. On the gateway side, a process for the client is started. This process is used as a representation of the client in the LINX cluster. All communication to and from the client goes through that process.

The osegw_destroy call kills the process that represents the client, closes the connection to the gateway, frees all allocated signals belonging to the connection (if not already freed), and frees the object.

```
osegw_destroy(gw);
```

3.2 Signals in Gateway Clients

A client communicates with LINX endpoints and other clients through a signaling API similar to the OSE signaling API. The commands used for this are osegw_alloc, osegw_free, osegw_send, osegw_receive, and osegw_receive_w_tmo. In addition a few other calls can be used for non-blocking receive, 3.5 Non-Blocking Mode Receive.

Before a signal can be sent, it has to be allocated with osegw_alloc. To avoid memory losses, the call registers the signal in a list handled by the OSEGW object that is specified in the call. When the signal is sent or freed, it is removed from the list. If the OSEGW object is destroyed when there are signals in the list, they are silently freed.

Below is an example of a signal allocation:

```
struct OSEGW_SIGNAL *signal;
signal = osegw_alloc(osegw_object, sizeof(OSEGW_SIGSELECT), signal_number);
```

And the signal is freed with:

```
osegw_free(osegw_object, &signal);
```

The size of the signal includes the signal number, which is of type OSEGW_SIGSELECT. Targets and hosts might also be of different endians. This is handled by the Gateway, which means that a signal size used by an

Gateway client might not be the same on the target. It is therefore recommended to use a sizeof (OSEGW_SIGSELECT) call for the size parameter. The Gateway does not handle endian conflicts or other compatibility issues for the user data.

When a signal is sent by the osegw_send call, it is automatically freed and removed from the OSEGW object's signal list. Example of an osegw_send call:

```
osegw_send(osegw_object, &signal, destination_pid);
```

The destination_pid argument can be obtained by a hunt call, see the section below about process synchronization for more information about this.

When a signal is received, it is automatically added to the OSEGW objects signal list. The difference between osegw_receive and osegw_receive_w_tmo is that the latter has an argument in which a timeout can be specified. The call will only block for the amount of milliseconds specified, and then it returns, even if no signal was received. Time for communication between the client and the Gateway has to be added to the specified time. The osegw_receive call will block until a signal is received. Example of a receive_w_tmo statement:

```
OSEGW_SIGSELECT sigselect_array[] = {2, 1, 2};
struct OSEGW_SIGNAL *signal;
signal = osegw_receive_w_tmo(gw, 1000, sigelect_array);
```

The call above will block for 1000 ms or until a signal is received.

If the connection to the Gateway server is lost, while blocking in an osegw_receive, the error handler is called with the error code OSEGW_ECONNECTION_LOST and osegw_receive returns OSEGW_NIL. This also applies to osegw_receive_w_tmo.

The client library implements a "ping"-mechanism to detect if the server has died. If a signal has not been received within OSEGW_PING_TIMEOUT milliseconds, the client sends a ping to the server to check that it is alive. If the server has not responded to a ping within OSEGW_REPLY_TIMEOUT milliseconds, the error handler is called with the error code OSEGW_ECONNECTION_TIMEDOUT. As long as the error handler returns a non-zero value, this sequence is repeated until a server response is received. If the error handler returns zero, the error handler is called once more with the error code OSEGW_ECONNECTION_LOST. The OSEGW_PING_TIMEOUT is configurable at compile-time, see ose_gw.h.

3.3 Obtain Endpoint IDs via Hunt

Before communication can take place between Gateway clients and LINX endpoints, they have to know each others' endpoint IDs. Endpoint IDs are found using the osegw_hunt call, and the name of the destination endpoint or client must be known. The syntax looks like this:

```
OSEGW_OSBOOLEAN osegw_hunt(struct OSEGW ose_gw,
                           const char name,
                           OSEGW_OSUSER user,
                           OSEGW_PROCESS pid_,
                           union OSEGW_SIGNAL hunt_sig);
```

Below is a complete example that shows how a hunt can be used.

```
#include "ose_gw.h"
#define HUNT_SIG_NO 42
```

```
struct HUNT_SIGNAL
{
   OSEGW_SIGSELECT sig_no;
};

union OSEGW_SIGNAL
{
   OSEGW_SIGSELECT sig_no;
   struct HUNT_SIGNAL hunt_signal;
};

/*
 * This function hunts a process specified by the parameter
 * name and then blocks until the process is terminated.
 */
void supervise_proc(struct OSEGW ose_gw, const char name)
{
   static const OSEGW_SIGSELECT select_hunt_sig[] =
      {1, HUNT_SIGN_O};
   static const OSEGW_SIGSELECT select_attach_sig[] =
      {1, OSEGW_ATTACH_SIG};
   union OSEGW_SIGNAL sig;
   OSEGW_PROCESS hunted_proc;
   OSEGW_OSATTREF attref;

   /* Hunt for the process with hunt signal */
   sig = osegw_alloc(sizeof(struct
                    HUNT_SIGNAL), HUNT_SIG_NO);
   (void)osegw_hunt(ose_gw, name, 0, NULL, &sig);

   /* Wait for the hunt signal */
   sig = osegw_receive(ose_gw,
   select_hunt_sig);

   /* Save the hunted process' process id. */
   hunted_proc = osegw_sender(ose_gw, &sig);
   osegw_free_buf(ose_gw, &sig);

   /* Supervise the existence of the found process, use the
    * default return signal from the kernel.
    */
   attref = osegw_attach(NULL, hunted_proc);

   /* Wait for the hunted process to be terminated */
   sig = osegw_receive(ose_gw, select_attach_sig);
   osegw_free_buf(ose_gw, &sig);
}
```

3.4 Supervision

A Gateway client should supervise other clients or endpoint with the `osegw_attach` call. The syntax for the call is:

```
OSEGW_OSATTREF osegw_attach(struct OSEGW *ose_gw,
                            union OSEGW_SIGNAL **sig,
                            OSEGW_PROCESS pid);
```

When this call is applied to a client or an endpoint, the signal given as an argument will be returned to the caller if the client or endpoint is killed. This can be used for controlled communication, with which the sender can be sure that signals have been received by the other party.

To cancel an attach, `osegw_detach` can be used:

```
void osegw_detach(struct OSEGW *ose_gw, OSEGW_OSATTREF *attref);
```

An Example of Controlled Communication

```
#include "ose_gw.h"
#define HUNT_SIG_NO 42

struct HUNT_SIGNAL
{
   OSEGW_SIGSELECT sig_no;
};

union OSEGW_SIGNAL
{
   OSEGW_SIGSELECT sig_no;
   struct HUNT_SIGNAL hunt_signal;
};

/*
 * This function hunts a process specified by the parameter name and
 * does some communication. Afterwards it checks if the hunted
 * process is still alive to make sure that the communication
 * succeeded.
 */

void controlled_comm(struct OSEGW *ose_gw, const char *name)
{
   static const OSEGW_SIGSELECT
   select_hunt_sig[] = {1, HUNT_SIG_NO};
   static const OSEGW_SIGSELECT
   select_attach_sig[] = {1, OSEGW_ATTACH_SIG};
   union OSEGW_SIGNAL *sig;
   OSEGW_PROCESS hunted_proc;
   OSEGW_OSATTREF attref;

   /* Hunt for the process with hunt signal */
   sig = osegw_alloc(sizeof(struct HUNT_SIGNAL), HUNT_SIG_NO);
   (void) osegw_hunt(ose_gw, name, 0, NULL, &sig);

   /* Wait for the hunt signal */
   sig = osegw_receive(ose_gw, select_hunt_sig);

   /* Save the hunted process' process id. */
   hunted_proc = osegw_sender(ose_gw, &sig);
   osegw_free_buf(ose_gw, &sig);

   /*
    * Supervise the existence of the found process, use the
    * default return signal from the kernel.
    */
   attref = osegw_attach(NULL,hunted_proc);

   /* communicate with the hunted process here. */

   /*
    * Check if the hunted process is still alive, because
    * it can be assumed that the communication succeeded.
    */
   sig = osegw_receive_w_tmo(ose_gw, 0, select_attach_sig);
   if (if sig != OSEGW_NIL)
   {
      if (osegw_sender(ose_gw, &sig) != hunted_proc)
      {
         /*
          * The process is still alive and we don't need to
```

```
                 * supervise it any more, detach the signal.
                 */
                osegw_detach(ose_gw, &hunted_proc);
            }
            osegw_free_buf(ose_gw, &sig);
        }
    }
```

3.5 Non-Blocking Receive Mode

Sometimes, clients not only have to wait for events from the Gateway, but also from other applications in the host operating system. It is also possible that a solution requires two Gateway clients to execute in the same thread. The non-blocking receive mode makes this possible through a number of calls.

A native communication object is needed in order to be able to use native select or poll calls with the `OSEGW` object. This can be obtained by the following call:

```
void * osegw_get_blocking_object(struct OSEGW *ose_gw, OSEGW_OSADDRESS *type);
```

The return value is a pointer to a native communication object, and the type of the object, for example a native socket, can be obtained by the type argument. The type can be used to verify that the object returned is of the type expected. If no object is available for the channel, `NULL` is returned and `OSEGW_BO_UNAVAILABLE` is put into the type argument.

To initialize the asynchronous receive, use the `osegw_init_async_receive` call:

```
void osegw_init_async_receive(struct OSEGW *ose_gw, const OSEGW_SIGSELECT *sig_sel);
```

This call registers a sigselect array in the client process in the gateway. When `osegw_init_async_receive` has been called, the client is not allowed to use `osegw_send`, `osegw_hunt`, `osegw_attach`, `osegw_detach`, `osegw_receive`, or `osegw_receive_w_tmo` calls. Calls that do not communicate with the gateway, such as `osegw_alloc`, still can be used.

The asynchronous receive mode can be cancelled with `osegw_cancel_async_receive`. After this call, all Gateway API calls can be used again. If a signal was received before the asynchronous mode was cancelled, it is returned by this call. If no signal was received, the call returns `NULL`. The syntax for `osegw_cancel_async_receive` is:

```
union OSEGW_SIGNAL *
osegw_cancel_async_receive(struct OSEGW *ose_gw);
```

If a client native call, such as select or poll, has detected that the native communication object is readable, an osegw_async_receive call can be used to read the signal:

```
union OSEGW_SIGNAL *
osegw_async_receive(struct OSEGW *ose_gw);
```

If the `osegw_async_receive` call is used when the native communication object is not readable, the call will simply block until a signal is received.

The listing below shows a complete example of two clients that execute in the same thread and are sending signals to each other using the asynchronous receive mode.

```
#include <sys/socket.h>
#include <stdio.h>
#include "ose_gw.h"

#define OSEGW_ADDRESS "tcp://localhost:12357/"
#define LONG_TIME 1000000

static const OSEGW_SIGSELECT any_sig[] = { 0 };

static void handle_error(void)
{
   exit(1);
}

static OSEGW_PROCESS get_pid(struct OSEGW *gw)
{
   union OSEGW_SIGNAL *signal;
   OSEGW_PROCESS pid;

   signal = osegw_alloc(gw, 10, 0x01);
   pid = osegw_sender(gw, &signal);
   osegw_free_buf(gw, &signal);
   return pid;
}

static int get_max_value(int v1, int v2)
{
   if (v1 > v2)
      return v1;
   else
      return v2;
}

int main(int argc, char *argv[])
{
   struct OSEGW *gw_client1;
   struct OSEGW *gw_client2;
   union OSEGW_SIGNAL *signal;
   OSEGW_PROCESS pid_client1;
   OSEGW_PROCESS pid_client2;
   int sock_client1;
   int sock_client2;
   int max_val;
   int nr_signals_left;
   int rv;
   fd_set rfds;

   /* Create the Clients */
   gw_client1 = osegw_create("client1", 0,
                             OSEGW_ADDRESS,
                             NULL, NULL, NULL);
   if (gw_client1 == NULL)
   {
      printf("*** ERROR *** Could not connect to OSE Gateway\n");
      exit(1);
   }
   gw_client2 = osegw_create("client2", 0,
                             OSEGW_ADDRESS,
                             NULL, NULL, NULL);
   if (gw_client2 == NULL)
   {
      printf("*** ERROR *** Could not connect to OSE Gateway\n");
      exit(1);
   }

   pid_client1 = get_pid(gw_client1);
   pid_client2 = get_pid(gw_client2);
```

```
    sock_client1 = *(int *)osegw_get_blocking_object(gw_client1, NULL);
    sock_client2 = *(int *)osegw_get_blocking_object(gw_client2, NULL);

    /* Client1 sends signals to client 2 */
    signal = osegw_alloc(gw_client1, 10, 0x01);
    osegw_send(gw_client1, &signal, pid_client2);
    signal = osegw_alloc(gw_client1, 10, 0x01);

    osegw_send(gw_client1, &signal, pid_client2);

    /* Client2 sends signals to client 1 */
    signal = osegw_alloc(gw_client2, 10, 0x01);
    osegw_send(gw_client2, &signal, pid_client1);
    signal = osegw_alloc(gw_client2, 10, 0x01);
    osegw_send(gw_client2, &signal, pid_client1);

    /* Start the asynchronous receive mode for both clients */
    osegw_init_async_receive(gw_client1, any_sig);
    osegw_init_async_receive(gw_client2, any_sig);

    /* Receive all four signals */
    nr_signals_left = 4;
    while (nr_signals_left > 0)
    {
       FD_ZERO(&rfds);
       FD_SET(sock_client1, &rfds);
       FD_SET(sock_client2, &rfds);
       max_val = get_max_value(sock_client1, sock_client2);

       rv = select(max_val + 1, &rfds, NULL, NULL, NULL);
       if (rv <= 0)
          handle_error();
       else
       {
          if (FD_ISSET(sock_client1, &rfds))
          {
             signal = osegw_async_receive(gw_client1);
             osegw_free_buf(gw_client1, &signal);
             nr_signals_left--;
             osegw_init_async_receive(gw_client1, any_sig);
          }
          if (FD_ISSET(sock_client2, &rfds))
          {
             signal = osegw_async_receive(gw_client2);
             osegw_free_buf(gw_client2, &signal);
             nr_signals_left--;
             osegw_init_async_receive(gw_client2, any_sig);
          }
       }
    }

    /* Cancel async receive for both clients */
    osegw_cancel_async_receive(gw_client1);
    osegw_cancel_async_receive(gw_client2);

    /* Destroy both clients */
    osegw_destroy(gw_client1);
    osegw_destroy(gw_client2);
    return 0;
}
```

3.6 Error Handling

The Gateway API supports two ways of handling errors. One uses a centralized error handler and the other uses function calls to check if an error has occurred.

3.6.1 Using the Error Handler

The error handler is implemented by the user and must follow the type definition for `OSEGW_ERRORHANDLER` in the Gateway API.

```
typedef OSEGW_BOOLEAN
OSEGW_ERRORHANDLER(struct OSEGW *ose_gw,
                   void *usr_hd,
                   OSEGW_ERRCODE ecode,
                   OSEGW_ERRCODE extra);
```

A pointer to the function is passed to the gateway object when the `osegw_create` function is called.

The error handler is primarily used when all errors are considered fatal. In such cases, the error handler can simply print the error codes and terminate the program. This is usually a good strategy when a client is prototyped. One simple function handles the errors, and no checks have to be implemented after every call to the gateway API functions. If no error handler is implemented, the create call must use `NULL` as parameter value for the error handler. If a error handler is provided and something fails during the creation of the `OSEGW` object, the error handler is still called. The return value from `osegw_create` will be `NULL`.

Example: Simple Client Prototype with the Error Handler

```
#include <stdio.h>
#include "ose_gw.h"

/*
 * error_handler
 * =============
 *
 * An error handler. The types is defined in ose_gw.h.
 *
 */


OSEGW_BOOLEAN
error_handler(struct OSEGW *ose_gw,
              void *usr_hdl,
              OSEGW_OSERRCODE ecode,
              OSEGW_OSERRCODE extra)
{
   fprintf(stderr, "An error occurred in OSE Gateway Client:"
                   "ERROR:%d EXTRA:%d\n", ecode, extra);
   exit(1);
}

int
main(int argc, char **argv)
{
   struct OSEGW gw_obj;

   /*  Check input argument */
   if (argc != 1)
   {
      printf("Syntax: %s IP-address:port\n"
             "Example: %s localhost:4000\n", argv[0], argv[0]);
      exit(1);
```

```
    }

    /*
     * Establish a connection to the OSE Gateway and pass a
     * pointer to the error handler.
     */
    gw_obj = osegw_create("prototype",
                          argv[1],
                          error_handler,
                           NULL);

    for (;;)
    {
       ...
    }
}
```

3.6.2 Error Handling Using Checkup Calls

The second way to handle errors is more like traditional error handling in Unix and Windows. The gateway object stores an error code internally when an error occurs. This code can be reset and fetched by the `osegw_reset_error` and `osegw_get_error` calls.

```
OSEGW_OSERRCODE osegw_get_error(struct OSEGW *ose_gw);
void osegw_reset_error(struct OSEGW *ose_gw);
```

The error is reset when a connection is established, so there is no need to reset the error just after a call to the `osegw_create`. If something fails during create, `NULL` is returned, and these error handling functions can not be used.

The code is not automatically reset after a successful call. This mean the code will describe the latest error, if any, if several calls to the gateway API are made without resetting the error between them.

Example - How to Use `osegw_get_error` and `osegw_reset_error`

```
#include <stdio.h&rt;
#include "ose_gw.h"

int
main(int argc, char **argv)
{
   struct OSEGW *gw_obj;
   OSEGW_PROCESS pid;

   /* Check input argument */
   if (argc != 1)
   {
      printf("Syntax: %s IP-address:port\n"
             "Example: %s localhost:4000\n", argv[0], argv[0]);
      exit(1);
   }

   /*
    * Establish a connection to the OSE Gateway, no error handler
    * is used.
    */
   gw_obj = osegw_create("example_client",
                         0,
                         argv[1],
                         NULL, NULL, NULL);
   if (gw_obj == NULL)
```

```
    {
       /*
        * The error handling function can not be used here
        * because there is no object to use them on.
        */
       printf("Error establishing connection to OSE Gateway "
              "at location: %s\n", argv[1]);
       exit(1);
    }
    (void)osegw_hunt(gw_obj, "target_proc", &pid, NULL);

    if (osegw_get_error(gw_obj) != OSEGW_OK)
    {
       printf("Error hunting for example_client\n");
       osegw_reset_error(gw_target);
    }

    ...
}
```

4. Find Gateway

The find gateway feature makes it easy for clients to find gateways on a local network. The `osegw_find_gw` call broadcasts a message and a gateway that receives the message replies with its address and name. The name is specified in the gateway configuration file, see 2.3 Configuration File. The address can be used by the `osegw_create` call to establish a connection to the gateway.

If the client already knows the address to the gateway, using this functionality might be unnecessary.

The broadcast is handled by the `osegw_find_gw` call. The user must implement a function following the `OSEGW_FOUND_GW` type definition, and pass a pointer to it to the `osegw_find_gw` function.

```
typedef OSEGW_BOOLEAN
(*OSEGW_FOUND_GW)(void *usr_hd,
                  const char *host_address,
                  const char *name);
```

See 2.3 Configuration File that demonstrates an example of how to use the `osegw_find_gw` call to create a function that establishes a connection to an gateway by passing the name, as specified in the gateway configuration file, of the gateway instead of the address.

Another use for this functionality is appropriate when the client has some kind of GUI. All gateways can be stored by the `OSEGW_FOUND_GW` function and presented in a list, for example, from which users can select the gateways to which they want to connect.

Example: Broadcast

```
#include <string.h>
#include "ose_gw.h"

#define BROADCAST_ADDRESS "255.255.255.0"
#define MAX_HOST_ADDRESS_LEN 64
#define TIMEOUT 3000

struct GW_DATA_OBJ
{
   char const *gw_name;
   char const
   host_address[MAX_HOST_ADDRESS_LEN];
```

```
};

OSBOOLEAN
find_gw_by_name(void *user_hd,
                char const *host_address,
                char const *name)
{
   struct GW_DATA_OBJ *obj = (struct GW_DATA_OBJ *)user_hd;
   if (strcmp(name, obj->gw_name) ==0)
   {
      strncpy(obj->host_address,
              host_address,
              MAX_HOST_ADDRESS_LEN);
      return OSEGW_TRUE;
   }
   return OSEGW_FALSE;
}

struct OSEGW *
create_by_gateway_name(char *client_name,
                       char *gateway_name,
                       OSEGW_ERRORHANDLER err_hdl,
                       void *user_handle)
{
   struct OSEGW *gw_obj;
   struct GW_DATA_OBJ gw_data;
   OSEGW_BOOLEAN rv;

   /* Try to find a gateway named 'gateway_name' */
   gw_data.gw_name = gateway_name;
   rv = osegw_find_gw(BROADCAST_ADDRESS,
                      TIMEOUT, find_gw_by_name, (void *)&gw_data);

   if(!rv)
   {
      /* No Gateway with the specified name found */
      return NULL;
   }
   else
   {
      /* Gateway found, connect to it */
      gw_obj = osegw_create(client_name,
                            gw_data.host_address,
                            errHdl, user_handle);
   }
   return gw_obj;
}

int
main(int argc, char **argv)
{
   struct OSEGW gw_obj;
   OSEGW_PROCESS pid;

   /*
    * Check input argument
    */
   if (argc != 1)
   {
      printf("Syntax: %s 'Name of an OSE Gateway'\n"
             "Example: %s DSP.XX.X\n", argv[0], argv[0]);
      exit(1);
   }

   /*
    * Establish a connection to the OSE Gateway, no error handle
    * is used.
```

```
         */
        gw_obj = create_by_gateway_name("prototype",
                                        argv[1],
                                        NULL,
                                        NULL);
        if (gw_obj == NULL)
        {
           printf("Error establishing connection to OSE Gateway "
                  "with name: %s\n", argv[1]);
           exit(1);
        }

        ...
     }
```

5. linxgwcmd - A Gateway Command Tool

The `linxgwcmd` tool can be used to find and test Gateway servers on a network. The following command line options exist:

-a <auth>	Use auth as authentication string, which should be in "user:passwd" form
-b <brc_addr>	Use brc_addr as broadcast string, which should be in "udp://*:port" form, where "port" should be the port number. Default broadcast string is "udp://*:21768"
-c <name>	Use name as the client's name (default "gw_client")
-e[<n>][,<b>]	Echo test to an OSE Gateway use `n` as number of loops (default 10) and `b` for number of bytes/chunk (default 4).
-h	Print this usage message.
-l[<t>][,<n>]	List Gateway servers. Use `t` as timeout value (default is 5 secs) and `n` for max items (default lists all). `linxgwcmd -l25,1` => look for Gateways in 25 secs and list only the first found.
-p <proc>	Hunt for process "proc".
-s <url/name>	Connect to a OSE Gateway server using either the servers URL or its name.