LINX(7) manual page

Name

linx - LINX inter-process communication protocol

Synopsis

#include <linx.h>
#include <linx_socket.h>
#include <linx_ioctl.h>
#include <linx_types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

Description

LINX is a location transparent inter-process communication protocol. It is based on the message passing technology used in the Enea OSE family of real time operating systems.

LINX consists of a set of kernel modules and provides a standard socket based interface using its own protocol family, **PF_LINX.** There is also a LINX library that provides a more advanced messaging API. Applications normally access LINX through this library, but sometimes direct access via the socket interface may be necessary.

Linx Concepts

An application that wants to communicate using LINX must first create a **LINX endpoint.** One thread may own multiple LINX endpoints simultaneously. A LINX endpoint is created with a non-unique name (a string). A handle which is used to refer to the endpoint in subsequent calls is returned to the application.

LINX endpoints communicate by sending and receiving **LINX signals.** The properties of LINX signals are described below.

Each LINX endpoint has a binary identifier called a **spid** which is unique within the node. A sending endpoint must know the spid of the destination endpoint to be able to communicate. The spid of a peer LINX endpoint is normally obtained by **hunting** for its name. When an endpoint with a matching name is found or created, LINX sends a hunt signal back to the hunting endpoint that appears to have been sent from the found endpoint. The spid can thus be obtained by looking at the sender of this signal.

A LINX endpoint can supervise, or **attach** to, the spid of a peer endpoint in order to be notified by a LINX signal when it is terminated or becomes unreachable.

The communication path between two LINX nodes is called a **LINX link.** A LINX link is created with a name string, and the same link may have different names on the opposite nodes, i.e. the link between nodes A and B may be called "LinkToB" on A, and "LinkToA" on B.

When hunting for an endpoint on a remote node, the name of the endpoint is prepended with the path of link names needed to reach the node, e.g. "LinkToB/LinkToC/EndpointName". LINX will create a virtual endpoint that acts as a local representation of the remote endpoint. LINX signals sent to the spid of a virtual endpoint are automatically routed to the proper destination on the remote node.

When a LINX endpoint is closed, its owned LINX signals are freed.

It is not allowed to use a LINX endpoint from two contexts at the same time. When a Linux process has multiple threads, it is not allowed to access a LINX endpoint from other contexts than the one that opened it. When **fork**(2) is called, the child process inherits copies of the parents socket related resources, including LINX endpoints. In this case, either the parent or the child shall close its LINX endpoints. A LINX endpoint is not removed until it has been closed by all of its owners.

LINX Application Interfaces

LINX provides both a standard socket interface and a more advanced **LINX API** which is available through the LINX library, to be linked with the application. The LINX API is the recommended way for applications to access LINX, since it simplifies for the programmer by abstracting the direct socket interactions. It implements a set of functions specified in **linx.h**(3) .

The LINX socket interface is the underlying socket implementation provided by the LINX kernel module and is used by the LINX library. It is described in detail below.

It is possible for applications to use a combination of the LINX API and the LINX socket interface. In this case, the LINX API function **linx_get_descriptor**(3) can be used to obtain the socket descriptor of a LINX endpoint. This descriptor can be used together with other file descriptors in generic **poll**(2) or **select**(2) calls. Note that it is NOT allowed to call **close**(2) on a LINX socket descriptor obtained by a **linx_get_descriptor**(3) call.

LINX Signals

A LINX signal consists of a mandatory leading 4-byte signal number, optionally followed by data. Thus, the size of a LINX signal buffer must be at least 4 bytes. LINX signal numbers are of type **LINX_SIGSELECT.** Signal numbers are mainly defined by the applications, but a few values are not allowed. Zero (0) is illegal and must not be used and 250-255 are reserved by LINX.

LINX provides endian conversion of the signal number if needed when a signal is sent to a remote node. The signal data is not converted.

LINX Socket Interface

The LINX socket interface allows application programmers to access LINX using standard socket calls. It should be noted that only a subset of the socket calls are implemented and that additional features have been made available through ioctl calls. These deviations and features are described below.

struct sockaddr_linx
> When using the socket interface directly, a LINX endpoint is represented by a **sockaddr_linx** structure:

```
struct sockaddr_linx
{
        sa_family_t family;
        LINX_SPID   spid;
};
```

family shall be **AF_LINX** and **spid** is the spid of the LINX endpoint. The **sockaddr_linx** structure is type-casted into a generic **sockaddr** structure when passed to the socket interface function calls.

The following calls are provided by the LINX socket interface:

socket()

A LINX socket is created by calling the **socket**(2) function as:

linx_sd = socket(*PF_LINX*, *SOCK_DGRAM*, **0**)*;*

When a LINX socket is created, its name is unspecified. To assign a name to the socket, use the **LINX_IOCTL_HUNTNAME** ioctl request.

On success, the return value is a descriptor referencing the socket. On error, -1 is returned and *errno* is set to one of the following values:

EPROTONOTSUPPORTED
The protocol type is not supported. Only **PF_LINX** is accepted.

ESOCKTNOSUPPORT
The socket type is not supported. Only **SOCK_DGRAM** is accepted.

ENOMEM
Insufficient memory is available. Alternatively, the maximum number of spids has been reached. This value is configurable as a parameter to the LINX kernel module, see LINX Users Guide for more information.

sendto()

The **sendto**(2) function is called as:

len = sendto(*linx_sd*, *payload*, *size*, **0**, (**struct** *sockaddr******) **&***sockaddr_linx*, **sizeof**(**struct** *sockaddr_linx*))**;**

The *payload* shall be a LINX signal buffer and *size* shall be its length in bytes. Note that it is mandatory for a LINX signal to have a leading 4 byte signal number. The *spid* field of the *sockaddr_linx* structure shall be the spid of the destination endpoint.

On success, the number of bytes sent is returned. On error, -1 is returned and *errno* is set to one of the following values:

EBADF
An invalid descriptor was specified.

ECONNRESET
The destination endpoint has been killed.

EINVAL
Invalid argument passed.

ENOMEM
Insufficient memory is available.

EOPNOTSUPP
The sending LINX socket has not been assigned a name.

EPIPE
This error is reported at an attempt to send to the spid of a LINX endpoint that is being closed as the call occurs.

sendmsg()
The **sendmsg**(2) function is called as:

len = sendmsg(*linx_sd***, ****msg***, 0);**

msg is a **msghdr** structure as defined in **sendmsg**(2) . The *msg_iov* field of the msghdr structure shall point to an **iovec** structure containing the LINX signal buffer to transmit and the *msg_iovlen* field shall be set to 1. Note that it is mandatory for a LINX signal to have a leading 4 byte signal number. The *msg_name* field shall be a pointer to a **sockaddr_linx** structure containing the spid of the destination endpoint and the *msg_namelen* field shall be set to the size of the sockaddr_linx structure. The ancillary fields and the flags field of the msghdr structure shall not be used and be set to zero.

On success, the number of bytes sent is returned. On error, -1 is returned and *errno* is set to one of the following values:

EBADF
An invalid descriptor was specified.

ECONNRESET
The destination endpoint has been killed. Note that this case is accepted and the signal is silently discarded by LINX.

EINVAL
Invalid argument passed.

ENOMEM
Insufficient memory is available.

EOPNOTSUPP
The sending socket has not been assigned a name.

EPIPE
This error is reported at an attempt to send to the spid of a LINX endpoint that is being closed as the call occurs.

recvfrom()
The **recvfrom**(2) function is called as:

len = recvfrom(*linx_sd***, ***payload***, ***size***, 0, (struct ***sockaddr****) &***sockaddr_linx***, sizeof(struct ***sockaddr_linx***));**

It is used to receive any LINX signal from any LINX endpoint. It can not be used when signal number filtering and/or sender filtering is needed, see **recvmsg**(2) . The first signal in the sockets receive queue is returned in the supplied *payload* buffer. If no signal is currently available at the socket, the call blocks until a signal is received. The sender of the signal is returned in the **sockaddr_linx** structure. Note that the *size* of the payload buffer must be at least 4 bytes, since it is mandatory for a LINX signal to have a 4 byte leading signal number.

On success, the number of bytes received is returned. If the received signal is larger than the supplied payload buffer, zero is returned and the signal buffer size is written as a 32-bit value in the first 4 bytes of the payload buffer. On error, -1 is returned and *errno* is set to one of the following values:

EBADF
An invalid descriptor was specified.

EFAULT
Invalid payload buffer pointer provided.

EINVAL
Invalid argument passed.

ENOMEM
Insufficient memory is available.

EOPNOTSUPP
The receiving socket has not been assigned a name.

recvmsg()

The **recvmsg**(2) function is called as:

len = recvmsg(*sd***, ***msg***, 0);**

msg is a pointer to a **msghdr** structure as defined in **recvmsg**(2) . A LINX signal buffer shall be supplied in an **iovec** structure pointed to from the *msg_iov* field and the **msg_iovlen** field shall be set to 1. Note that the size of the supplied buffer must be at least 4 bytes, since it is mandatory for a LINX signal to have a 4 byte leading signal number.

The **recvmsg**(2) call supports signal number filtering and sender filtering. This allows the user to specify which signal numbers shall be received and/or from which sender. The signal filter is described by a **linx_receive_filter_param** structure:

```
struct linx_receive_filter_param
{
        LINX_SPID             from;
        LINX_OSBUFSIZE        sigselect_size;
        const LINX_SIGSELECT *sigselect;
};
```

The **from** field specifies that only signals from a specific spid should be received and **sigselect** is an array of **LINX_SIGSELECT** numbers to be received. The first position in the array contains the number of entries in the list that follows. If the first position is set to a negative count, all LINX signals except those listed will be received. The size of the array is **sigselect_size.**

The filtering uses ancillary fields in the **msghdr** structure and is described by the following example code:

```
struct msghdr msg;
char cmsg[CMSG_SPACE(sizeof(struct linx_receive_filter_param))];
struct linx_receive_filter_param * rfp;
struct iovec iov;
const LINX_SIGSELECT sig_sel[] = { 1, EXAMPLE_SIG };
```

```
rfp = ((struct linx_receive_filter_param *)
        (CMSG_DATA(((struct cmsghdr *)cmsg))));
rfp->sigselect_size = sizeof(sig_sel);
rfp->from = from_spid;
rfp->sigselect = sig_sel;
msg.msg_name = (void*)&linx_addr;
msg.msg_namelen = sizeof(struct sockaddr_linx);
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
msg.msg_flags = 0;
msg.msg_control = cmsg;
msg.msg_controllen =
        CMSG_SPACE(sizeof(struct linx_receive_filter_param));
((struct cmsghdr *)cmsg)->cmsg_len = msg.msg_controllen;
((struct cmsghdr *)cmsg)->cmsg_level = 0;
((struct cmsghdr *)cmsg)->cmsg_type = 0;
iov.iov_base = *signal;
iov.iov_len = sigsize;
read_size = recvmsg(linx->socket, &msg, 0);
```

On success, the number of bytes received is returned. If the received signal is larger than the supplied payload buffer, zero is returned and the signal buffer size is written as a 32-bit value in the first 4 bytes of the payload buffer. On error, -1 is returned and *errno* is set to one of the following values:

EBADF
An invalid descriptor was specified.

EFAULT
An invalid msghdr structure was provided.

EINVAL
Invalid argument passed.

ENOMEM
Insufficient memory is available.

EOPNOTSUPP
The sending socket has not been assigned a name.

poll()

LINX socket descriptors can be used in the **poll**(2) call.

The call returns a bitmask that indicates the state of the LINX socket receive queues. The following possible bits can be set at return from this function: **POLLERR** if the LINX socket is in an error state, **POLLHUP** if the LINX socket has been shutdown/released, **POLLIN** and **POLLRDNORM** if the LINX socket has data to read in receive queue.

select()

LINX socket descriptors can be used together with other file descriptors in the **select**(2) call.

ioctl()

IOCTL requests are sent to a LINX socket using the **ioctl**(2) call. See below for all IOCTL request codes supported by LINX sockets.

close()

A LINX socket created by a *socket*(2) call can be closed with *close*(2) . Note that this function shall NOT be used on any LINX socket descriptor created with *linx_open*(3) or obtained by the LINX API function *linx_get_descriptor*(3) .

On success, 0 is returned, otherwise -1 is returned and *errno* can be one of the following errors:

EBADF
An invalid descriptor was specified.

Only the calls described above are supported by a LINX socket. The following are NOT supported on LINX sockets and shall not be used: *bind*(2) , *connect*(2) , *socketpair*(2) , *accept*(2) , *getname*(2) , *listen*(2) , *shutdown*(2) , *setsockopt*(2) , *getsockopt*(2) , *mmap*(2) and *sendpage*(2) .

IOCTL Request Codes

The following IOCTL request codes can be accessed using *ioctl*(2) on LINX sockets:

LINX_IOCTL_SEND

Sends a signal from a LINX socket **sd**. The correct syntax is:

struct linx_sndrcv_param **sndrcv**;

error **= ioctl(***sd*, *LINX_IOCTL_SEND*, *sndrcv***);**

sndrcv is a **linx_sndrcv_param** structure:

```
struct linx_sndrcv_param
{
    __u32 from;
    __u32 to;
    __u32 size;
    __u32 sig_attr;
    __u32 sigselect_size;
    __u64 sigselect;
    __u32 tmo;
    __u64 buffer;
};
```

from is the spid of the sender, the sender does not need to be the current socket, the signal is sent from the current socket but the receiver will see the **from** as the sending spid, the default case is that **from** is the spid of the current LINX socket. **to** is the spid of the receiver, **size** is the size of the signal to be sent and **sig_attr** are the attributes of the signal. The **sigselect_size** and **sigselect** are not used by **LINX_IOCTL_SEND** and **notused** is for padding the struct since it needs to be 64-bit aligned but should be zeroed for future use. **buffer** is the pointer to the buffer to be sent, it is passed as a 64 bit unsigned value to be both 32-bit and 64-bit compatible.

On success, the number of bytes sent is returned.

LINX_IOCTL_RECEIVE

Receives a signal on a LINX socket **sd**. The call will block until a signal is received. The correct syntax is:

struct linx_sndrcv_param **sndrcv**;

```
error = ioctl(sd, LINX_IOCTL_RECEIVE, sndrcv);
```

sndrcv is a **linx_sndrcv_param** structure:

```
struct linx_sndrcv_param
{
    __u32 from;
    __u32 to;
    __u32 size;
    __u32 sig_attr;
    __u32 sigselect_size;
    __u64 sigselect;
    __u32 tmo;
    __u64 buffer;
};
```

The **from** field should be set to **LINX_ILLEGAL_SPID** if signals from anyone should be received or if only signals from a specific spid should be received then **from** should be set to that spid. If **LINX_ILLEGAL_SPID** was set the **from** will contain the spid of the sender after the return of the call. The **to** field is not used when receiving a signal. The **size** field is the size of the provided buffer. When the call returns the **sig_attr** field is set to the attribute the signal carriers. The **sigselect** field is an array of **LINX_SIGSELECT** numbers to be received. The first position in the array contains the number of entries in the list that follows. If the first position is set to a negative count, all LINX signals except those listed will be received. The size of the array is **sigselect_size.** When the **tmo** field is used the call waits maximum **tmo** milliseconds before returning even if no signal has been received, if a blocking receive is requested the **tmo** field should be ~0 (0xFFFFFFFF), this will block forever. If no signal is received the. **buffer** pointer will be set to NULL (the provided buffer is always consumed). The **buffer** field is a pointer to the buffer provided by the user.

On success, the number of bytes received is returned. If the received signal is larger than the supplied payload buffer, zero is returned and the signal buffer size is written as a 32-bit value in the first 4 bytes of the payload buffer.

LINX_IOCTL_REQUEST_TMO

Request a timeout, a signal is sent to the requesting LINX endpoint when a timeout has expired. The correct syntax is:

struct linx_tmo_param **tmo_param***;**

error **= ioctl(***sd, LINX_IOCTL_REQUEST_TMO, tmo_param***);**

tmo_param is a **linx_tmo_param** structure:

```
struct linx_tmo_param
{
    LINX_OSTIME tmo;
    LINX_OSBUFSIZE sigsize;
    union LINX_SIGNAL *sig;
    LINX_OSTMOREF tmoref;
};
```

tmo is the timeout in milliseconds and the actual timeout time is rounded upward to the nest larger tick, the call guarantees at least the number of milliseconds requested, **sig** is a pointer to the signal that will be returned when the timeout expires, if **sig** is LINX_NIL then the default timeout signal with signal number LINX_OS_TMO_SIG is received instead, **sigsize** is the size of the provided signal, if no signal is provided the value must be set to zero.

On success, an timemout reference is returned in **tmoref**. This reference can be used in **LINX_IOCTL_CANCEL_TMO** and **LINX_IOCTL_MODIFY_TMO**.

LINX_IOCTL_CANCEL_TMO

Cancel a pending timeout, the correct syntax is:

struct linx_tmo_param *_tmo_param_;

error **= ioctl(**_sd_, _LINX_IOCTL_CANCEL_TMO_, _tmo_param_**);**

_tmo_param_ is a **linx_tmo_param** structure:

```
struct linx_tmo_param
{
    LINX_OSTIME tmo;
    LINX_OSBUFSIZE sigsize;
    union LINX_SIGNAL *sig;
    LINX_OSTMOREF tmoref;
};
```

tmo, **sigsize** and **sig** are ignored, **tmoref** is used to identify which timeout is to be canceled, it is guaranteed that the timeout signal cannot be received after cancellation.

LINX_IOCTL_MODIFY_TMO

Modifies a pending timeout, the correct syntax is:

struct linx_tmo_param *_tmo_param_;

error **= ioctl(**_sd_, _LINX_IOCTL_MODIFY_TMO_, _tmo_param_**);**

_tmo_param_ is a **linx_tmo_param** structure:

```
struct linx_tmo_param
{
    LINX_OSTIME tmo;
    LINX_OSBUFSIZE sigsize;
    union LINX_SIGNAL *sig;
    LINX_OSTMOREF tmoref;
};
```

sigsize and **sig** are ignored, **tmoref** is used to identify which timeout is to be modified, **tmo** is the new timeout value.

LINX_IOCTL_REQUEST_NEW_LINK

Request a signal when a new link is available, the correct syntax is:

```
struct linx_new_link_param *new_link_param;

error = ioctl(sd, LINX_IOCTL_REQUEST_NEW_LINK, new_link_param);
```

new_link_param is a **linx_new_link_param** structure:

```
struct linx_new_link_param
{
    uint32_t token;
    uint32_t new_link_ref;
};
```

token is passed to and from the LINX kernel module keeping track of which links the caller already have been notified about, the **token** value is ignored the first time a LINX endpint requests a new link signal. The **token** received in the new link signal should then be used in the next new link signal request. **new_link_ref** is used to cancel a pending new link signal request.

The syntax of the new link signal received when a new link is available is:

```
struct linx_new_link {
    LINX_SIGSELECT signo;
    LINX_NLTOKEN token;
    int name;
    int attr;
    char buf[1];
};
```

signo is LINX_OS_NEW_LINK_SIG, **token** is the token value to be used in the next request for a new link signal, **name** is the offset into **buf** where the name of the new link is stored, the name is null terminated, **attr** is the offset into **buf** where the attributes, if any, of the link are stored, the attribute string is null terminted, **buf** is a character buffer containing the name and the attributes, if any, of the link.

LINX_IOCTL_CANCEL_NEW_LINK

Cancels a pending new link signal request, the correct syntax is:

```
struct linx_new_link_param *new_link_param;

error = ioctl(sd, LINX_IOCTL_CANCEL_NEW_LINK, new_link_param);
```

new_link_param is a **linx_new_link_param** structure:

```
struct linx_new_link_param
{
    uint32_t token;
    uint32_t new_link_ref;
};
```

token is ignored, **new_link_ref** is used to identify which pending new link signal request is to be cancelled, after the cancellation it is guarnateed that no more new link signals can be received.

LINX_IOCTL_HUNTNAME

Sets the name of a LINX socket **sd** and returns its binary LINX endpoint identifier, the spid. The correct syntax is:

struct linx_huntname *huntname;

error **= ioctl(**sd, LINX_IOCTL_HUNTNAME, huntname**);**

huntname is a **linx_huntname** structure:

```
struct linx_huntname
{
        LINX_SPID spid;
        size_t    namelen;
        char     *name;
};
```

namelen is the size in bytes of the string **name** that contains the name to assign to the socket. On success, **spid** is set to the spid assigned to the LINX socket.

LINX_IOCTL_HUNT

Hunts for a LINX endpoint (that has been assigned a name) to obtain its spid. The correct syntax is:

struct linx_hunt_param *hunt_param;

error **= ioctl(**sd, LINX_IOCTL_HUNT, hunt_param**);**

hunt_param is a **linx_hunt_param** structure:

```
struct linx_hunt_param
{
        LINX_OSBUFSIZE     sigsize;
        union LINX_SIGNAL *sig;
        LINX_SPID          from;
        size_t             namelen;
        char              *name;
```

```
    };
```

The **sig** parameter optionally holds a signal of size **sigsize** to be received when the other LINX socket is available. If no signal (NULL) is provided, the LINX default hunt signal of type **LINX_OS_HUNT_SIG** will be used. The **from** parameter shall be set to the owner of the hunt. In the normal case, this is the spid of the LINX socket performing the hunt call. If the spid associated with a different LINX socket is provided, the hunt can be cancelled by closing that socket. The hunt signal is always sent to the LINX socket performing the hunt call. The **namelen** is the size of the string **name** that contains the name of the LINX endpoint to be hunted for.

LINX_IOCTL_ATTACH

Attaches to a LINX endpoint in order to supervise it, i.e. to get an attach signal if it becomes unavailable. The correct syntax is:

```
struct linx_attach_param *attach_param;

error = ioctl(sd, LINX_IOCTL_ATTACH, attach_param);
```

attach_param is a **linx_attach_param** structure:

```
    struct linx_attach_param
    {
            LINX_SPID          spid;
            LINX_OSBUFSIZE     sigsize;
            union LINX_SIGNAL *sig;
            LINX_OSATTREF      attref;
    };
```

The **spid** field is the spid of the LINX endpoint to supervise. The **sig** parameter optionally holds a LINX signal of size **sigsize** to be received when the supervised LINX endpoint becomes unavailable. If no signal (NULL) is provided, the LINX default attach signal of type **LINX_OS_ATTACH_SIG** will be used.

On success, an attach reference is returned in **attref.** This reference can be used in **LINX_IOCTL_DETACH** later.

LINX_IOCTL_DETACH

Detaches from a supervised LINX endpoint, i.e. stops supervising it. The correct syntax is:

```
struct linx_detach_param *detach_param;

error = ioctl(sd, LINX_IOCTL_DETACH, detach_param);
```

The *detach_param* parameter is a **struct linx_detach_param** with the following fields:

```
struct linx_detach_param
{
        LINX_OSATTREF  attref;
}
```

The **attref** field is an attach reference returned from a **LINX_IOCTL_ATTACH** call.

LINX_IOCTL_SET_RECEIVE_FILTER

Sets up a receive filter prior to a **select**(2) call. The correct syntax is:

struct linx_receive_filter_param **rfp**;

error **= ioctl(***sd*, *LINX_IOCTL_SET_RECEIVE_FILTER*, *rfp***);**

rfp is a **linx_receive_filter_param** structure:

```
struct linx_receive_filter_param
{
        LINX_SPID            from;
        LINX_OSBUFSIZE       sigselect_size;
        const LINX_SIGSELECT *sigselect;
};
```

The **from** parameter specifies that only signals from a specific spid should be received and **sigselect** is an array of **LINX_SIGSELECT** numbers to be received. The first position in the array contains the number of entries in the list that follows. If the first position is set to a negative count, all LINX signals except those listed will be received. The size of the array is **sigselect_size.**

LINX_IOCTL_REGISTER_LINK_SUPERVISOR

Registers this socket as a supervisor of LINX links. The correct syntax is:

error **= ioctl(***sd*, *LINX_IOCTL_REGISTER_LINK_SUPERVISOR*, **0);**

When a link is established to another node, a notification signal will be sent to all LINX sockets that have registered as link supervisor. This signal has the format:

```
struct linx_new_link
{
        LINX_SIGSELECT  signo;
        int             name;
        int             attr;
        char            buf[1];
};
```

The **signo** is **LINX_OS_LINK_SIG.** The name of the link begins **name** bytes into **buf** and its attributes starts **attr** bytes into **buf.** Both the name and attributes are null terminated strings assigned to a link with the *linxcfg*(1) command.

To be notified when a link becomes unavailable, first do a **LINX_IOCTL_HUNT** on the link **name.** This returns a spid representing the link. Then the link can be supervised using **LINX_IOCTL_ATTACH** to that spid.

LINX_IOCTL_UNREGISTER_LINK_SUPERVISOR

Unregister this socket as a supervisor of LINX links. The correct syntax is:

```
error = ioctl(sd, LINX_IOCTL_UNREGISTER_LINK_SUPERVISOR, 0);
```

LINX_IOCTL_VERSION

Returns the version of the LINX kernel module. The correct syntax is:

```
unsigned int version;

error = ioctl(sd, LINX_IOCTL_VERSION, &version);
```

On success, the **version** parameter contains the version of the LINX kernel module. The LINX version number is a 32-bit number composed of an 8-bit major version, an 16-bit minor version, and a 8-bit seq (patch) number.

LINX_IOCTL_INFO

Retrieves information from the LINX kernel module. The correct syntax is:

```
struct linx_info info;

error = ioctl(sd, LINX_IOCTL_INFO, &info);
```

The **info** parameter is a **struct linx_info** with the following fields:

```
struct linx_info
{
        int    type;
        void  *type_spec;
};
```

The **type** field indicates the requested type of information and **type_spec** is a pointer to a struct that will contain input and return parameters.

Note that information retrieved with LINX_IOCTL_INFO may have become inaccurate when used in a subsequent call. The application must be prepared to handle errors related to this.

The different kinds of information that can be retrieved from the LINX kernel module are:

LINX_INFO_SUMMARY

Provides a summary of the most important information from the LINX kernel module.

struct linx_info *info*;

struct linx_info_summary *info_summary*;

info.type = *LINX_INFO_SUMMARY*;

info.type_spec = *&info_summary*;

The **linx_info_summary** structure is defined as:

```
struct linx_info_summary
{
        int no_of_local_sockets;
        int no_of_remote_sockets;
        int no_of_link_sockets;
        int no_of_pend_attach;
        int no_of_pend_hunt;
        int no_of_queued_signals;
};
```

The **no_of_local_sockets** field is the number of LINX sockets open locally, **no_of_remote_sockets** is the number of internal sockets open that have been created by the LINX kernel module to represent remote LINX endpoints. The **no_of_link_sockets** field is the number of open sockets representing LINX links to other nodes. The **no_of_pend_attach** field is the number of pending attaches, **no_of_pend_hunt** is the number of pending hunts and

no_of_queued_signals is the number of queued signals.

LINX_INFO_SOCKETS

Returns the number of open LINX sockets and their LINX endpoint identifiers (spids).

struct linx_info *info*;

struct linx_info_sockets *info_sockets*;

info.type = *LINX_INFO_SOCKETS*;

info.type_spec = *&info_sockets*;

The **linx_info_sockets** structure is defined as:

```
struct linx_info_sockets
{
        LINX_OSBOOLEAN local;
        LINX_OSBOOLEAN remote;
        LINX_OSBOOLEAN link;
        int            buffer_size;
        int            no_of_sockets;
        LINX_SPID     *buffer;
};
```

If **local** is true, local sockets are included in the output, if **remote** is true, remote sockets are included and if **link** is true, sockets representing LINX links are included. The number of LINX sockets matching the search is returned in **no_of_sockets** and the array of spids is returned in the provided **buffer**. of size **buffer_size** bytes. If the provided buffer is too small, not all sockets will be included.

LINX_INFO_TYPE

Returns the type of a LINX endpoint.

struct linx_info *info*;

struct linx_info_type *info_type*;

```
info.type = LINX_INFO_TYPE;

info.type_spec = &info_type;
```

The **linx_info_type** structure is defined as:

```
struct linx_info_type
{
        LINX_SPID  spid;
        int        type;
};
```

The **spid** field is the identifier of the LINX endpoint for which the type is requested. The type is returned in **type**. A LINX endpoint can be of types: **LINX_TYPE_UNKNOWN**, **LINX_TYPE_LOCAL**, **LINX_TYPE_REMOTE**, **LINX_TYPE_LINK**, **LINX_TYPE_ILLEGAL** or **LINX_TYPE_ZOMBIE**.

LINX_INFO_STATE

Returns the state of a LINX endpoint.

```
struct linx_info info;

struct linx_info_state info_state;

info.type = LINX_INFO_STATE;

info.type_spec = &info_state;
```

The **linx_info_state** structure is defined as:

```
struct linx_info_state
{
        LINX_SPID  spid;
        int        state;
};
```

The **spid** field is the identifier of the LINX endpoint for which the state is requested. The state is returned in **state**. A LINX socket can be in states: **LINX_STATE_UNKNOWN**, **LINX_STATE_RUNNING**,

LINX_STATE_RECV or **LINX_STATE_POLL**.

LINX_INFO_FILTERS

Returns information about the receive filters set up by a LINX endpoint.

struct linx_info *info*;

struct linx_info_filters *info_filters*;

info.type = *LINX_INFO_FILTERS*;

info.type_spec = *&info_filters*;

The **linx_info_filters** structure is defined as:

```
struct linx_info_filters
{
        LINX_SPID       spid;
        LINX_SPID       from_filter;
        int             buffer_size;
        int             no_of_sigselect;
        LINX_SIGSELECT *buffer;
};
```

The **spid** field is the identifier of the LINX endpoint for which the receive filter is requested. If the endpoint has setup a receive filter only accepting signals from a specific LINX endpoint, the spid of that endpoint is returned in **from_filter**. The number of LINX_SIGSELECT signal numbers in the receive filter is returned in **no_of_sigselect** and an array of these signal numbers are returned in the provided **buffer**. The **buffer_size** is the size in bytes of the buffer. If the buffer is too small, not all signal numbers in the filter are included.

LINX_INFO_RECV_QUEUE

Returns the receive queue of a LINX endpoint.

struct linx_info *info*;

struct linx_info_recv_queue *info_recv_queue*;

```
info.type = LINX_INFO_RECV_QUEUE;

info.type_spec = &info_recv_queue;
```

The **linx_info_recv_queue** structure is defined as:

```
struct linx_info_recv_queue
{
        LINX_SPID               spid;
        int                     buffer_size;
        int                     no_of_signals;
        struct linx_info_signal *buffer;
};
```

The **spid** field is the identifier of the LINX endpoint for which the receive queue is requested. The number of signals in the queue is returned in **no_of_signals**. An array with queue information is returned in the provided **buffer**. The **buffer_size** is the size in bytes of the buffer. If the buffer is too small, not all signals are included. The **linx_info_signal** structure is defined as:

```
struct linx_info_signal
{
        LINX_SIGSELECT signo;
        int            size;
        LINX_SPID      from;
};
```

The **signo** field holds the signal number, **size** is the size in bytes of the signal and **from** is the spid of the sending LINX endpoint.

LINX_INFO_PEND_ATTACH

Returns information about pending attaches **from** or **to** a LINX endpoint.

```
struct linx_info info;

struct linx_info_pend_attach info_pend_attach;

info.type = LINX_INFO_PEND_ATTACH;

info.type_spec = &info_pend_attach;
```

The **linx_info_pend_attach** structure is defined as:

```
struct linx_info_pend_attach
{
        LINX_SPID               spid;
        int                     from_or_to;
        int                     buffer_size;
        int                     no_of_attaches;
        struct linx_info_attach *buffer;
};
```

The **spid** field is the identifier of the LINX endpoint for which attach informations is requested. If **from_or_to** is set to **LINX_ATTACH_FROM,** information about attaches from the spid is returned. If it is set to **LINX_ATTACH_TO,** information about attaches to the spid is returned. The number of attaches to/from the **spid** is returned in **no_of_attaches**, Information about the attaches are returned in the provided **buffer**. The **buffer_size** is the size in bytes of the buffer. If the buffer is too small, not all attaches are included. The **linx_info_attach** structure is defined as:

```
struct linx_info_attach
{
        LINX_SPID               spid;
        LINX_OSATTREF           attref;
        struct linx_info_signal attach_signal;
};
```

The **spid** is the identifier of the LINX endpoint that has attached or has been attached to (depending on what **from_or_to** is set to). The **attref** field is the attach reference and **attach_signal** is the attach signal.

LINX_INFO_PEND_HUNT

Returns information about pending hunts issued from any LINX endpoint.

struct linx_info *info*;

struct linx_info_pend_hunt *info_pend_hunt*;

info.type = *LINX_INFO_PEND_HUNT*;

info.type_spec = *&info_pend_hunt*;

The **linx_info_pend_hunt** structure is defined as:

```
struct linx_info_pend_hunt
{
        LINX_SPID               spid;
        int                     buffer_size;
        int                     strings_offset;
        int                     no_of_hunts;
        struct linx_info_hunt *buffer;
};
```

The **spid** field is the identifier of the LINX endpoint for which hunt information is requested. The number of pending hunts is returned in **no_of_hunts** and information about each pending hunt is returned in the provided **buffer**. The **buffer_size** is the size in bytes of the buffer. If the buffer is too small, not all hunts are included. The **strings_offset** is the offset into the **buffer** where the name strings are stored. Each **linx_info_hunt** structure is defined as:

```
struct linx_info_hunt
{
        struct linx_info_signal hunt_signal;
        LINX_SPID               owner;
        char                    *hunt_name;
};
```

The **owner** field is the owner of the pending hunt and **hunt_name** is a string containing the name hunted for. The **hunt_signal** is the hunt signal. The **linx_info_signal** structure is described under **LINX_INFO_RECV_QUEUE**.

LINX_INFO_PEND_TMO

Returns information about pending timeouts issued from any LINX endpoint.

struct linx_info *info*;

struct linx_info_pend_tmo *info_pend_tmo*;

info.type = *LINX_INFO_PEND_TMO*;

info.type_spec = *&info_pend_tmo*;

The **linx_info_pend_tmo** structure is defined as:

```
struct linx_info_pend_tmo
{
        LINX_SPID              spid;
        int                    buffer_size;
        int                    no_of_timeouts;
        struct linx_info_tmo  *buffer;
};
```

The **spid** field is the identifier of the LINX endpoint for which hunt
information is requested. The number of pending timeouts is returned in
no_of_timeouts and information about each pending timeout is returned in
the provided **buffer**. The **buffer_size** is the size in bytes of the buffer. If the
buffer is too small, not all timeouts are included. Each **linx_info_tmo**
structure is defined as:

```
struct linx_info_tmo
{
        LINX_OSTIME            tmo;
        LINX_OSTMOREF          tmoref;
        struct linx_info_signal tmo_signal;
};
```

The **tmo** field is the remaining time and **tmoref** is the timeout reference. The
tmo_signal holds the timeout signal information.

LINX_INFO_SIGNAL_PAYLOAD

Returns the payload of a signal owned by a LINX endpoint.

struct linx_info *info*;

struct linx_info_signal_payload *info_signal_payload*;

info.type = *LINX_INFO_SIGNAL_PAYLOAD*;

info.type_spec = *&info_signal_payload*;

The **linx_info_signal_payload** structure is defined as:

```
struct linx_info_signal_payload
{
        LINX_SPID               spid;
        int                     buffer_size;
        struct linx_info_signal signal;
        int                     payload_size;
        char                    *buffer;
};
```

The **spid** field is the identifier of the LINX endpoint owning the signal and **signal** is a **linx_info_signal** structure returned from a previous LINX_INFO call. The signal buffer will be returned in the provided **buffer**. The **buffer_size** is the size in bytes of the buffer. If the provided buffer is too small, only the beginning of the signal buffer is returned. The **payload_size** shows the size in bytes of the returned signal payload. If the provided buffer is larger than the signal payload, **payload_size** will be less then **buffer_size**. If no signal payload matching the **signal** the **payload_size** will be set to zero.

LINX_INFO_NAME

Returns the name of a LINX endpoint.

struct linx_info *info*;

struct linx_info_name *info_name*;

info.type = *LINX_INFO_NAME*;

info.type_spec = *&info_name*;

The **linx_info_name** structure is defined as:

```
struct linx_info_name
{
        LINX_SPID spid;
        int       namelen;
        char      *name;
};
```

The **spid** field is the identifier of the LINX endpoint for which the name is requested. The **namelen** field is the length of the provided **name** buffer in which the name is be returned. If the LINX socket endpoint has not been assigned a name yet zero is returned and **name** is set to the empty string.

LINX_INFO_OWNER

Returns the process (PID) that owns a LINX endpoint.

struct linx_info *info;*

struct linx_info_owner *info_owner;*

info.type = *LINX_INFO_OWNER;*

info.type_spec = *&info_owner;*

The **linx_info_owner** structure is defined as:

```
struct linx_info_owner
{
        LINX_SPID spid;
        pid_t     owner;
};
```

The **spid** field is the identifier of the LINX endpoint for which the information is requested. On success, the PID of the owning process is returned in **owner**.

LINX_INFO_STAT

Returns statistics for a LINX endpoint. This requires that the LINX kernel module has been compiled with the "SOCK_STAT=yes" setting.

struct linx_info *info;*

struct linx_info_stat *info_stat;*

info.type = *LINX_INFO_STAT;*

info.type_spec = *&info_stat;*

The **linx_info_stat** structure is defined as:

```
struct linx_info_stat
```

```
{
        LINX_SPID spid;
        uint64_t no_sent_local_signals;
        uint64_t no_recv_local_signals;
        uint64_t no_sent_local_bytes;
        uint64_t no_recv_local_bytes;
        uint64_t no_sent_remote_signals;
        uint64_t no_recv_remote_signals;
        uint64_t no_sent_remote_bytes;
        uint64_t no_recv_remote_bytes;
        uint64_t no_sent_signals;
        uint64_t no_recv_signals;
        uint64_t no_sent_bytes;
        uint64_t no_recv_bytes;
        uint64_t no_queued_bytes;
        uint64_t no_queued_signals;
};
```

The **spid** is the identifier of the LINX endpoint for which statistics is required. If the LINX kernel module has not been compiled with "SOCK_STAT=yes", ioctl returns -1 and errno is set to ENOSYS.

Known Bugs

None.

See Also

Generic LINX for Linux man-page:
linx(7) (this document)

LINX API man-pages:
linx.h(3) , **linx_types.h**(3) ,
linx_alloc(3) , **linx_attach**(3) , **linx_cancel_tmo**(3) , **linx_close**(3) ,
linx_create_eth_link(3) , **linx_create_tcp_link**(3) , **linx_destroy_eth_link**(3) , **linx_destroy_tcp_link**(3) ,
linx_detach(3) ,
linx_free_buf(3) , **linx_free_name**(3) , **linx_free_stat**(3) ,
linx_get_descriptor(3) , **linx_get_name**(3) , **linx_get_spid**(3) ,
linx_get_stat(3) , **linx_hunt**(3) , **linx_hunt_from**(3) , **linx_modify_tmo**(3) ,
linx_open(3) , **linx_receive**(3) , **linx_receive_from**(3) ,
linx_receive_w_tmo(3) , **linx_request_tmo**(3) , **linx_send**(3) , **linx_send_w_opt**(3) ,
linx_send_w_s(3) , **linx_sender**(3) , **linx_set_sigsize**(3) , **linx_sigattr**(3) , **linx_sigsize**(3)

Related LINX applications
linxcfg(1) , **linxdisc**(8) , **linxdisc.conf**(5) , **linxls**(1) , **linxstat**(1) , **mkethcon**(1) , **mklink**(1) , **mktcpcon**(1) ,
rmethcon(1) , **rmlink**(1) , **rmtcpcon**(1)

Related generic Linux man-pages:
socket(2) , **close**(2) , **sendto**(2) , **sendmsg**(2) , **recvfrom**(2) , **recvmsg**(2) , **poll**(2) , **select**(2) , **ioctl**(2)

Author

Enea LINX team

LINX_ALLOC(3) manual page

Name

linx_alloc() - Allocate a signal buffer on a LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

union LINX_SIGNAL *linx_alloc(LINX *_linx_, LINX_OSBUFSIZE _size_, LINX_SIGSELECT _sig_no_);

Description

linx_alloc() is used to allocate a signal buffer of the indicated size The first 4 bytes of the signal buffer are initialized with the provided signal number _sig_no_. The signal number can be replaced later by writing a new value in the first 4 bytes of the buffer.

The new buffer is owned by the LINX endpoint with which it was created. The ownership is transferred to the receiving LINX endpoint when the signal is successfully sent using **linx_send**(3) . If the signal is not sent, the owner must free the buffer by calling **linx_free_buf**(3) using the same **linx** handle. The buffer will be freed if the owning LINX endpoint is closed or the process owning the endpoint exits.

linx is the handle to the LINX endpoint.

size is the size in bytes of the new signal buffer. The minimum size is 4 bytes, needed to store the signal number.

_sig_no_ is the signal number stored at the beginning of the signal buffer.

Return Value

On success, a pointer to the allocated signal buffer is returned. On error, LINX_NIL is returned and _errno_ will be set.

Errors

EMSGSIZE The _size_ is invalid.

ENOMEM Insufficient memory is available.

Bugs/Limitations

None.

Notes

A LINX signal can be only be sent by the LINX endpoint that owns it, i.e. the endpoint it was created from or received on. It can not be reused and sent by another LINX endpoint.

If a process, which has allocated signal buffers, calls *fork(2)* , both instances will own copies of the signal buffers. As only one process may use a LINX endpoint and its resources, either the parent or the child MUST close the LINX endpoint using **linx_close**(3) .

See Also

linx(7) , **linx_close**(3) , **linx_free_buf**(3) , **linx_hunt**(3) , **linx_send**(3) , **fork**(2)

Author

Enea LINX team

LINX_ATTACH(3) manual page

Name

linx_attach() - Attach to and supervise a LINX endpoint
linx_detach() - Detach from an attached LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSATTREF linx_attach(LINX *linx**, union LINX_SIGNAL ****sig**, LINX_SPID** spid**);**

int linx_detach(LINX *linx**, LINX_OSATTREF ***attref**);**

Description

linx_attach() is used to supervise another LINX endpoint. When the other endpoint is closed or becomes unavailable, an attach signal is sent to the supervising LINX endpoint *linx* as a notification. If a signal *sig* is provided, this signal will be received when the attach is triggered. If *sig* is NULL, the default LINX attach signal with signal number LINX_OS_ATTACH_SIG is received instead. The linx_attach() call consumes the signal, taking over its ownership, and sets the *sig* pointer to LINX_NIL. The signal is consumed if an error occurs too. If *sig* is corrupt, **abort**(3) is called.

linx_detach() is used to detach from an attached LINX endpoint. It's an error to detach from a LINX endpoint more than once or to detach from a LINX endpoint after the attach signal has been received. It is not an error to detach if the attach signal is waiting in the receive queue of the LINX endpoint. To prevent multiple detaches, linx_detach() sets the attref pointer to LINX_ILLEGAL_ATTREF.

linx is the handle of the LINX endpoint.

spid is the identifier of the LINX endpoint to supervise.

sig is either NULL or a user defined LINX signal.

attref is the LINX_OSATTREF attach reference obtained from **linx_attach**().

Return Value

linx_attach() returns an attach reference (LINX_OSATTREF) when successful. This attach reference can be used to cancel the attach (detach) after the attach is done. **linx_detach**() returns 0 on success.

On failure, **linx_attach**() returns LINX_ILLEGAL_ATTREF and **linx_detach**() returns -1, and *errno* is set appropriately.

Errors

EBADF, **ENOTSOCK** The *linx* handle refers to an invalid socket descriptor.

Bugs/Limitations

None.

Example

In this example the server attaches to the client and tells the client to exit and waits for the attach signal.

```
Server:
#include <linx.h>
#define CLIENT_DONE_SIG 0x1234
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  const LINX_SIGSELECT sel_att_sig [] = { 1, LINX_OS_ATTACH_SIG };
  /* Create a LINX endpoint with huntname "attacher" */
  linx = linx_open("attacher", NULL, 0);
  /* Hunt for the client */
  linx_hunt(linx, "client", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, sel_hunt_sig);
  /* Retrive the clients spid */
  client = linx_sender(linx, &sig)
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
  /* Attach to the client */
  linx_attach(linx, client, NULL);
  /* Create "done" signal */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), CLIENT_DONE_SIG);
  /* Send "done" signal to client */
  linx_send(linx, &sig, client);
  /* Wait for the attach signal */
  linx_receive(linx, &sig, sel_att_sig);
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
Client:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_done_sig[] = { 1, CLIENT_DONE_SIG };

  /* Open a LINX endpoint with huntname "client" */
  linx = linx_open("client", NULL, 0);
  /* Wait for server to send "done" signal */
  linx_receive(linx, &sig, sel_done_sig);
  /* Close the LINX endpoint – this will trigger the attach */
  linx_close(linx);
  return 0;
}
```

See Also

linx(7) , linx_attach(3) , linx_close(3) , linx_detach(3) , linx_hunt(3) , linx_send(3)

Author

Enea LINX team

LINX_CANCEL_TMO(3) manual page

Name

linx_request_tmo() - request timeout with specified signal
linx_cancel_tmo() - cancel the specified timeout.
linx_modify_tmo() - modify the specified timeout

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSTMOREF linx_request_tmo(LINX *linx**, LINX_OSTIME** tmo**, union LINX_SIGNAL ****sig**);**

int linx_cancel_tmo(LINX *linx**, LINX_OSTMOREF ***tmoref**);**

int linx_modify_tmo(LINX *linx**, LINX_OSTMOREF ***tmoref**, LINX_OSTIME** tmo**);**

Description

linx_request_tmo() is used to request a signal, *sig*, to be sent to the requesting LINX endpoint *linx* when a timeout has expired. Information necessary for cancelling the time-out is returned. The actual time-out time, *tmo*, is rounded upward to the next larger tick because a time-out can only trigger when a system clock tick is received. Hence, the routine guarantees at least the number of milliseconds requested, but may add a few more depending on the tick resolution. When several time-out signals are generated at the same clock tick, the order is unspecified. If *sig* is NULL, the default LINX timeout signal with signal number LINX_OS_TMO_SIG is received instead. The linx_request_tmo () call consumes the signal, and sets the *sig* pointer to LINX_NIL. The signal is also consumed if an error occurs.

linx_cancel_tmo() is used to cancel a timeout. The time-out is identified by the *tmoref* parameter, which was set by **linx_request_tmo**()**.** It's an error to cancel a timeout more than once or to cancel a timeout after the timeout signal has been received. It is not an error to cancel the timeout if the timeout signal is waiting in the receive queue of the LINX endpoint. To prevent multiple cancellations, **linx_cancel_tmo**() sets the tmoref pointer to LINX_ILLEGAL_TMOREF.

linx_modify_tmo() is used to modify a timeout. The time-out is identified by the tmoref parameter, which was set by **linx_request_tmo**()**.** It's an error to modify a timeout after the timeout signal has been received. It is not an error to modify the timeout if the timeout signal is waiting in the receive queue of the LINX endpoint.

linx is the handle of the LINX endpoint.

spid is the timeout time in milliseconds.

sig is either NULL or a user defined LINX signal.

tmoref is the reference to the timeout obtained from **linx_request_tmo**().

Return Value

linx_request_tmo() returns an timeout reference (LINX_OSTMOREF) when successful. This timeout reference can be used to cancel the timeout. **linx_cancel_tmo**() returns 0 on success.

On failure, **linx_request_tmo**() returns LINX_ILLEGAL_TMOREF while **linx_cancel_tmo**() and **linx_modify_tmo**() both returns -1. In all cases *errno* will be set appropriately.

Errors

EBADF, **ENOTSOCK** The *linx* handle refers to an invalid socket descriptor.

ENOMEM Not enough memory.

EINVAL Invalid argument.

Bugs/Limitations

None.

Example

This example shows how to request and wait for a timeout.

```
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_tmo_sig [] = { 1, LINX_OS_TMO_SIG };
  /* Create a LINX endpoint */
  linx = linx_open("tmo-test", NULL, 0);
  /* Request a one second timeout. When the timeout expires the
     default signal will be returned. */
  linx_request_tmo(linx, 1000, NULL);
  /* Wait for the timeout signal. */
  linx_receive(linx, &sig, sel_tmo_sig);
  /* Free the timeout signal */
  linx_free_buf(linx, &sig);
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
```

See Also

linx(7) , **linx_close**(3) , **linx_open**(3) **linx_receive**(3) ,

Author

Enea LINX team

LINXCFG(1) manual page

Name

linxcfg - a configuration utility for LINX links to other nodes

Synopsis

linxcfg [-t <cm>] [-v] <command> [COMMAND PARAMETERS and OPTIONS]

linxcfg [-t <cm>] [-v] destroy <linkname> [<link 2>...]

linxcfg -h

linxcfg help <cm>

linxcfg -t <cm> [help]

Description

linxcfg is the LINX configuration utility. Depending on the *command* given, **linxcfg** can destroy or create links using the specified Connection Manager (CM).

Options

linxcfg has common options to all commands

which should be placed **before** the actual *command*

-h, --help
>   Display help and exit.

-t, --type <cm>
>   Specifies *cm* as the Connection Manager to use. If this option is not provided, *eth* will be used as the default CM.

>   You can use **-h** to list all available CMs.

-v, --verbose
>   Verbose mode.

-c, --cf, --conn-feature <feature[=value]>
>   Adds a connection feature. This option can be specified several times.

-l, --lf, --link-feature <feature[=value]>
>   Adds a link feature. This option can be specified several times.

Commands

create PARAMETERS
>   Creates a connection link to another node. The *PARAMETERS* depend on the CM specified, see below for more details.

>   A remote link cannot be used until also the remote node creates the other end of the link (a link back to the originating node). The state is "connecting" until both sides are ready and communication via

the LINX protocol can begin.

destroy <linkname> [<link2>...]

Destroy the connection link with the name *linkname*.

help [<cm>]

Provides help about the given *cm* , either specified in **-t** or after the **help** command.

If no *cm* was specified, it will show a list of available CMs.

Options to Different Connection Managers

Ethernet CM

The **Ethernet CM** is specified with the *eth* keyword, which is actually the default if no other CM was specified.

The **create** command takes the following options:

create <macaddr> <interface> <linkname> [OPTIONAL PARAMETERS]

macaddr

Specify the remote mac address of the node to connect to, e.g. 0a:1b:2c:3d:4d:5e.

interface

Specify the Ethernet interface to use. Ex.: eth0.

linkname

Name of the connection link. This is the link part of the path in a remote hunt name to LINX endpoints on the other node. It will also be the name seen in */proc/net/linx/cm/eth/<linkname>*

Optional parameters

--window_size=nnn

This is the send/receive window_size, in number of packets on the link, and may need to be modified to adapt LINX to really slow or really fast Ethernet performance.
Both the local and remote side negotiate the window size, being the smallest of the two sizes the selected one. The default 128 messages should be sufficient in most configurations. The window size shall always be of a power of 2. Size 0 means to use the default window size.

--defer_queue_size=nnn

The defer queue size is in packages, with the size depending on the LINX link. The defer queue is used on the sender side when the send queue is full. Every message sent, when the send queue is full, is stored in the defer queue, until the send queue has room for more messages. The defer queue size is the maximum number of packages, with messages, that can be stored in the defer queue before the link is disconnected, due to lack or resources. The default value 2048 packages should be sufficient in most systems. Size 0 means to use the default defer queue size.

--send_tmo=nnn

The send acknowledge timeout specifies the time (in msec) to wait until unacknowledged messages are resent. The default 10 should be sufficient in most systems. Tmo 0 means to use the default timeout.

--nack_tmo=nnn

The retransmission timeout specifies the time (in msec) to wait until nack messages are resent. The default 20 should be sufficient in most systems. Tmo 0 means to use the default timeout.

--conn_tmo=nnn

The connect timeout specifies the time (in msec) to wait until an attempt to establish a connection is considered failed. The default 200 should be sufficient in most systems. Tmo 0 means to use the default timeout.

--live_tmo=nnn

The connection supervision timeout specifies the time (in msec) to wait until a connection is considered broken. Default is 100 (when the system is not idle) and should be sufficient in most systems. When the system is idle, the live timeout is ten times larger than the configured live timeout

value. Tmo 0 means to use the default timeout.

--mtu=nnn

> The MTU (Maximum Transmission Unit) specifies the size in bytes of the largets packet that the Ethernet protocol can pass onwards (this excludes the size of the Ethernet header). If not set the MTU is fetched from the interface. Typcially a MTU of 1500 bytes is used for Ethernet.

--attributes=s

> The attribute option gives the possibility to assign an arbitrary string to the link when it is created. This string is included in the new_link signal, which is sent to all link supervisors. More information on link supervisors can be found in **linx**(7) man page.

Tcp Cm

The TCP CM is specified using the *tcp* keyword.

The **create** command takes the following parameters:

create <ip> <linkname> [OPTIONAL PARAMETERS]

ip

> The IP address to connect to.

linkname

> Name of the connection link. This is the LINX link part of the path in a remote hunt name to LINX endpoints on the other node. It will also be the name seen in */proc/net/linx/cm/tcp/<linkname>*

Optional parameters

--live_tmo=<size>

> The live_tmo parameter is the time in milliseconds between every heartbeat that is used to detect if the connection has gone down. The default value is 1000 ms.

--use_nagle=<bool>

> Set to 1 if nagle algorithm shall be used on the socket for the connection. Default is off.

--attributes=<s>

> The attribute option gives the possibility to assign an arbitrary string to the link when it is created. This string is included in the new_link signal, which is sent to all link supervisors.

Files

None.

Diagnostics

linxcfg will display more verbose information to standard out, if the -v option is specified.

Known Bugs

None.

Examples

```
linxcfg create 01:23:a4:4f:b3:ac eth0 link_A
linxcfg destroy link_A
linxcfg -t tcp create 192.168.1.1 link_A
linxcfg -t tcp destroy link_A
```

Ethernet CM   36

See Also

linx(7) , linxstat(1) , linxdisc(8)

Author

Enea LINX team

LINX_CLOSE(3) manual page

Name

linx_close() - Close a LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_close(LINX * *linx* **);**

Description

Close a LINX endpoint that was created by a **linx_open**(3) call. All resources owned by this LINX endpoint are freed. When a LINX endpoint is closed, attach signals are sent to other LINX endpoints that have attached to it using **linx_attach**(3) . A LINX endpoint will be automatically closed if its owner process exits without calling **linx_close**(3) first. Note that the LINX socket is not closed until the last reference has been closed. See **linx_open**(3) for more information about forking processes that own LINX endpoints.

linx is the handle of the LINX endpoint to close.

Return Value

On success, zero is returned. On error, -1 is returned and *errno* is set.

Errors

EBADF The *linx* handle refers to an invalid socket descriptor.

Bugs/Limitations

None.

See Also

linx(7) , **linx_open**(3) , **linx_alloc**(3) , **linx_attach**(3)

Author

Enea LINX team

LINX_CREATE_ETH_LINK(3) manual page

Name

linx_create_eth_link() - creates an Ethernet link
linx_destroy_eth_link() - destroys an Ethernet link

Synopsis

#include <linxcfg.h>

int linx_create_eth_link(char **name***, char ****mac***, char ****ifc***, unsigned long** *window_size***, unsigned long** *defer_queue_size***, unsigned long** *send_tmo***, unsigned long** *nack_tmo***, unsigned long** *conn_tmo***, unsigned long** *live_tmo***);**

int linx_destroy_eth_link(char **name***);**

Description

The calls **linx_create_eth_link**() and **linx_destroy_eth_link**() creates and destroys links between two unique Ethernet interfaces that are present in the same subnet. A link is considered up when both sides have done **linx_create_eth_link**() to each others Ethernet interfaces. A link is identified by its name which is assigned by the parameter *name*. The name is local to the node so different nodes can have the same name to the same remote interface. The MAC address to the remote interface is given in *mac* in the form of a character string, i.e. "00:10:4A:0F:AB:00". The *ifc* parameter is used to specify which interface the link shall use.

The following options are not mandatory. If 0 is given, default values will be used.

The send and receive sliding window size is set by *window_size*, the window size shall always be of a power of 2. Default window size is 128 and it should be sufficient for most configurations. Both sides must use the same *window_size*.

A defer queue is used on the sender side when the send window is full, every message sent when the send window is full is stored in a defer queue until the send window has room for more messages. The *defer_queue* parameter specifies the maximum number of messages that can be stored in the defer queue before the link is disconnected due to lack of resources. The default size is 2048 and it should be sufficient for most systems.

The *send_tmo* timeout specifies the time to wait until unacked messages are resent (in msec). Default timeout is 10msec and it should be sufficient for most systems.

The *nack_tmo* timeout specifies the time to wait until nack messages are resent (in msec). The default timeout of 20msec should be sufficient for most systems.

The *conn_tmo* timeout specifies the time to wait (in msec) until the establishment of a connection is considered failed. The default value of 200msec should be sufficient in most systems.

The *live_tmo* parameter specifies the time to wait until a connection is considered broken. The default timeout of 100msec (when the system is not idle) should be sufficient for most systems. When the system is idle, the live timeout is ten times larger than the configured live timeout value.

Return Value

Returns zero if successful, otherwise -1 is returned and *errno* is set.

Errors

EINVAL if *name*, *mac* or *ifc* are NULL.

ENOBUFS or **ENOMEM** if there is not enough memory.

EPROTONOSUPPORT or **EAFNOSUPPORT** if *linx(7)* is not supported by the system.

ENFILE Not enough kernel memory to allocate a new LINX socket.

EMFILE Process file table overflow.

EACCES Permission to create a LINX socket is denied.

Bugs/Limitations

Several links can be created to and from an Ethernet interface but only one link is allowed between two unique interfaces, for instance, interface A can have two links to interface B and C but A cannot have two links to interface B.

Notes

In the future, **linx_create_eth_link**() and **linx_destroy_eth_link**() will be replaced by two generic function calls, **linx_create_link**() and **linx_destroy_link**().

Example

In this example a link is created to nodeB.

```
#include <linx.h>
#include <linxcfg.h>
int
main(int argc, char *argv[]) {
  int rv;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  /* Create a link to the ethernet interface with the MAC
   * address of 00:01:02:03:04:05, use the local interface
   * eth0 for the link and default values for all parameters. */
  rv = linx_create_eth_link("link_to_nodeB", "00:01:02:03:04:05", "eth0",
0, 0, 0, 0, 0, 0);
  if(rv != 0) {
    /* Link was not created */
    return -1;
  }
  /* Link was created, now we can use the link. */
  /* Open a LINX endpoint */
  linx = linx_open("myproc", 0, NULL);
  /* Hunt for process hisproc over the link */
```

```
  linx_hunt(linx, "link_to_nodeB/hisproc", NULL);
  /* This hunt will resolve first AFTER the other side has
   * done a linx_create_eth_link to us! */
  linx_receive(linx, &sig, sel_hunt_sig);
  linx_close(linx);
  /* Destroy the link */
  rv = linx_destroy_eth_link("link_to_nodeB");
}
```

See Also

linx(7) , **linx_create_tcp_link**(3) , **linx_destroy_tcp_link**(3)

Author

Enea LINX team

Example 41

LINX_CREATE_TCP_LINK(3) manual page

Name

linx_create_tcp_link() - creates a TCP link
linx_destroy_tcp_link() - destroys a TCP link

Synopsis

#include <linxcfg.h>

int linx_create_tcp_link(char * *name* **, char *** *ip_addr* **, unsigned long** *live_tmo* **, unsigned long** *use_nagle* **, char *** *attributes* **);**

int linx_destroy_tcp_link(char * *name* **);**

Description

The calls **linx_create_tcp_link**() and **linx_destroy_tcp_link**() respectively create and destroy TCP links. A link is considered up when both sides have done **linx_create_tcp_link**() to each others TCP port. A link is identified by its name which is assigned by the parameter *name*. The name is local to the node so different nodes can have the same name to the same remote interface. The IP address to the remote interface is given in *ip_addr* in the form of a character string, i.e. "192.168.1.1".

The following options are not mandatory. If 0 is given, default values will be used.

The *send_tmo* timeout specifies the time to wait until unacked messages are resent (in msec). Default timeout is 10msec and it should be sufficient for most systems.

In the *use_nagle* parameter a non zero value specifies that the nagle algorithm should be used on the socket for the connection. The default is off.

The *attributes* option gives the possibility to assign an arbitrary string to the link when it is created. This string is included in the *new_link* signal,

which is sent to all link supervisors

Return Value

Returns zero if successful, otherwise -1 is returned and *errno* is set.

Errors

EINVAL if *name* is NULL, or if *ip_addr* is NULL or has an invalid IP address.

ENOBUFS or **ENOMEM** if there is not enough memory.

EPROTONOSUPPORT or **EAFNOSUPPORT** if *linx(7)* is not supported by the system.

ENFILE Not enough kernel memory to allocate a new LINX socket.

EMFILE Process file table overflow.

EACCES Permission to create a LINX socket is denied.

Bugs/Limitations

Several links can be created to and from a TCP peer but only one link is allowed between two unique peers, for instance, interface A can have two links: one to peer B and the other to C, but A cannot have two links to peer B.

Notes

In the future, **linx_create_tcp_link**() and **linx_destroy_tcp_link**() will be replaced by two generic function calls, **linx_create_link**() and **linx_destroy_link**().

Example

In this example a link is created to nodeB.

```
#include <linx.h>
#include <linxcfg.h>
int
main(int argc, char *argv[]) {
  int rv;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  /* Create a link to the ethernet interface with the IP address
   * address 192.168.1.1, and use the default values for all
   * parameters */
  rv = linx_create_tcp_link("link_to_nodeB", "192.168.1.1", 0, 0, NULL);
  if(rv != 0) {
    /* Link was not created */
    return -1;
  }
  /* Link was created, now we can use the link. */
  /* Open a LINX endpoint */
  linx = linx_open("myproc", 0, NULL);
  /* Hunt for process hisproc over the link */
  linx_hunt(linx, "link_to_nodeB/hisproc", NULL);
  /* This hunt will resolve first AFTER the other side has
   * done a linx_create_tcp_link to us! */
  linx_receive(linx, &sig, sel_hunt_sig);
  linx_close(linx);
  /* Destroy the link */
  rv = linx_destroy_tcp_link("link_to_nodeB");
}
```

See Also

linx(7) , **linx_create_eth_link**(3) , **linx_destroy_eth_link**(3)

Author

Enea LINX team

Author

LINX_DESTROY_ETH_LINK(3) manual page

Name

linx_create_eth_link() - creates an Ethernet link
linx_destroy_eth_link() - destroys an Ethernet link

Synopsis

#include <linxcfg.h>

int linx_create_eth_link(char **name***, char ****mac***, char ****ifc***, unsigned long** *window_size***, unsigned long** *defer_queue_size***, unsigned long** *send_tmo***, unsigned long** *nack_tmo***, unsigned long** *conn_tmo***, unsigned long** *live_tmo***);**

int linx_destroy_eth_link(char **name***);**

Description

The calls **linx_create_eth_link**() and **linx_destroy_eth_link**() creates and destroys links between two unique Ethernet interfaces that are present in the same subnet. A link is considered up when both sides have done **linx_create_eth_link**() to each others Ethernet interfaces. A link is identified by its name which is assigned by the parameter *name*. The name is local to the node so different nodes can have the same name to the same remote interface. The MAC address to the remote interface is given in *mac* in the form of a character string, i.e. "00:10:4A:0F:AB:00". The *ifc* parameter is used to specify which interface the link shall use.

The following options are not mandatory. If 0 is given, default values will be used.

The send and receive sliding window size is set by *window_size*, the window size shall always be of a power of 2. Default window size is 128 and it should be sufficient for most configurations. Both sides must use the same *window_size*.

A defer queue is used on the sender side when the send window is full, every message sent when the send window is full is stored in a defer queue until the send window has room for more messages. The *defer_queue* parameter specifies the maximum number of messages that can be stored in the defer queue before the link is disconnected due to lack of resources. The default size is 2048 and it should be sufficient for most systems.

The *send_tmo* timeout specifies the time to wait until unacked messages are resent (in msec). Default timeout is 10msec and it should be sufficient for most systems.

The *nack_tmo* timeout specifies the time to wait until nack messages are resent (in msec). The default timeout of 20msec should be sufficient for most systems.

The *conn_tmo* timeout specifies the time to wait (in msec) until the establishment of a connection is considered failed. The default value of 200msec should be sufficient in most systems.

The *live_tmo* parameter specifies the time to wait until a connection is considered broken. The default timeout of 100msec (when the system is not idle) should be sufficient for most systems. When the system is idle, the live timeout is ten times larger than the configured live timeout value.

Return Value

Returns zero if successful, otherwise -1 is returned and *errno* is set.

Errors

EINVAL if *name*, *mac* or *ifc* are NULL.

ENOBUFS or **ENOMEM** if there is not enough memory.

EPROTONOSUPPORT or **EAFNOSUPPORT** if *linx(7)* is not supported by the system.

ENFILE Not enough kernel memory to allocate a new LINX socket.

EMFILE Process file table overflow.

EACCES Permission to create a LINX socket is denied.

Bugs/Limitations

Several links can be created to and from an Ethernet interface but only one link is allowed between two unique interfaces, for instance, interface A can have two links to interface B and C but A cannot have two links to interface B.

Notes

In the future, **linx_create_eth_link**() and **linx_destroy_eth_link**() will be replaced by two generic function calls, **linx_create_link**() and **linx_destroy_link**().

Example

In this example a link is created to nodeB.

```
#include <linx.h>
#include <linxcfg.h>
int
main(int argc, char *argv[]) {
  int rv;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  /* Create a link to the ethernet interface with the MAC
   * address of 00:01:02:03:04:05, use the local interface
   * eth0 for the link and default values for all parameters. */
  rv = linx_create_eth_link("link_to_nodeB", "00:01:02:03:04:05", "eth0",
0, 0, 0, 0, 0, 0);
  if(rv != 0) {
    /* Link was not created */
    return -1;
  }
  /* Link was created, now we can use the link. */
  /* Open a LINX endpoint */
  linx = linx_open("myproc", 0, NULL);
  /* Hunt for process hisproc over the link */
```

```
  linx_hunt(linx, "link_to_nodeB/hisproc", NULL);
  /* This hunt will resolve first AFTER the other side has
   * done a linx_create_eth_link to us! */
  linx_receive(linx, &sig, sel_hunt_sig);
  linx_close(linx);
  /* Destroy the link */
  rv = linx_destroy_eth_link("link_to_nodeB");
}
```

See Also

linx(7) , **linx_create_tcp_link**(3) , **linx_destroy_tcp_link**(3)

Author

Enea LINX team

Example   47

LINX_DESTROY_TCP_LINK(3) manual page

Name

linx_create_tcp_link() - creates a TCP link
linx_destroy_tcp_link() - destroys a TCP link

Synopsis

#include <linxcfg.h>

int linx_create_tcp_link(char *name*, **char** *ip_addr*, **unsigned long** *live_tmo*, **unsigned long** *use_nagle*, **char** *attributes*);

int linx_destroy_tcp_link(char *name*);

Description

The calls **linx_create_tcp_link**() and **linx_destroy_tcp_link**() respectively create and destroy TCP links. A link is considered up when both sides have done **linx_create_tcp_link**() to each others TCP port. A link is identified by its name which is assigned by the parameter *name*. The name is local to the node so different nodes can have the same name to the same remote interface. The IP address to the remote interface is given in *ip_addr* in the form of a character string, i.e. "192.168.1.1".

The following options are not mandatory. If 0 is given, default values will be used.

The *send_tmo* timeout specifies the time to wait until unacked messages are resent (in msec). Default timeout is 10msec and it should be sufficient for most systems.

In the *use_nagle* parameter a non zero value specifies that the nagle algorithm should be used on the socket for the connection. The default is off.

The *attributes* option gives the possibility to assign an arbitrary string to the link when it is created. This string is included in the *new_link* signal,

which is sent to all link supervisors

Return Value

Returns zero if successful, otherwise -1 is returned and *errno* is set.

Errors

EINVAL if *name* is NULL, or if *ip_addr* is NULL or has an invalid IP address.

ENOBUFS or **ENOMEM** if there is not enough memory.

EPROTONOSUPPORT or **EAFNOSUPPORT** if *linx(7)* is not supported by the system.

ENFILE Not enough kernel memory to allocate a new LINX socket.

EMFILE Process file table overflow.

EACCES Permission to create a LINX socket is denied.

Bugs/Limitations

Several links can be created to and from a TCP peer but only one link is allowed between two unique peers, for instance, interface A can have two links: one to peer B and the other to C, but A cannot have two links to peer B.

Notes

In the future, **linx_create_tcp_link**() and **linx_destroy_tcp_link**() will be replaced by two generic function calls, **linx_create_link**() and **linx_destroy_link**().

Example

In this example a link is created to nodeB.

```
#include <linx.h>
#include <linxcfg.h>
int
main(int argc, char *argv[]) {
  int rv;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  /* Create a link to the ethernet interface with the IP address
   * address 192.168.1.1, and use the default values for all
   * parameters */
  rv = linx_create_tcp_link("link_to_nodeB", "192.168.1.1", 0, 0, NULL);
  if(rv != 0) {
    /* Link was not created */
    return -1;
  }
  /* Link was created, now we can use the link. */
  /* Open a LINX endpoint */
  linx = linx_open("myproc", 0, NULL);
  /* Hunt for process hisproc over the link */
  linx_hunt(linx, "link_to_nodeB/hisproc", NULL);
  /* This hunt will resolve first AFTER the other side has
   * done a linx_create_tcp_link to us! */
  linx_receive(linx, &sig, sel_hunt_sig);
  linx_close(linx);
  /* Destroy the link */
  rv = linx_destroy_tcp_link("link_to_nodeB");
}
```

See Also

linx(7) , **linx_create_eth_link**(3) , **linx_destroy_eth_link**(3)

Author

Enea LINX team

LINX_DETACH(3) manual page

Name

linx_attach() - Attach to and supervise a LINX endpoint
linx_detach() - Detach from an attached LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSATTREF linx_attach(LINX *linx, union LINX_SIGNAL **sig, LINX_SPID spid);

int linx_detach(LINX *linx, LINX_OSATTREF *attref);

Description

linx_attach() is used to supervise another LINX endpoint. When the other endpoint is closed or becomes unavailable, an attach signal is sent to the supervising LINX endpoint *linx* as a notification. If a signal *sig* is provided, this signal will be received when the attach is triggered. If *sig* is NULL, the default LINX attach signal with signal number LINX_OS_ATTACH_SIG is received instead. The linx_attach() call consumes the signal, taking over its ownership, and sets the *sig* pointer to LINX_NIL. The signal is consumed if an error occurs too. If *sig* is corrupt, **abort**(3) is called.

linx_detach() is used to detach from an attached LINX endpoint. It's an error to detach from a LINX endpoint more than once or to detach from a LINX endpoint after the attach signal has been received. It is not an error to detach if the attach signal is waiting in the receive queue of the LINX endpoint. To prevent multiple detaches, linx_detach() sets the attref pointer to LINX_ILLEGAL_ATTREF.

linx is the handle of the LINX endpoint.

spid is the identifier of the LINX endpoint to supervise.

sig is either NULL or a user defined LINX signal.

attref is the LINX_OSATTREF attach reference obtained from **linx_attach**().

Return Value

linx_attach() returns an attach reference (LINX_OSATTREF) when successful. This attach reference can be used to cancel the attach (detach) after the attach is done. **linx_detach**() returns 0 on success.

On failure, **linx_attach**() returns LINX_ILLEGAL_ATTREF and **linx_detach**() returns -1, and *errno* is set appropriately.

Errors

EBADF, **ENOTSOCK** The *linx* handle refers to an invalid socket descriptor.

Bugs/Limitations

None.

Example

In this example the server attaches to the client and tells the client to exit and waits for the attach signal.

```
Server:
#include <linx.h>
#define CLIENT_DONE_SIG 0x1234
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  const LINX_SIGSELECT sel_att_sig [] = { 1, LINX_OS_ATTACH_SIG };
  /* Create a LINX endpoint with huntname "attacher" */
  linx = linx_open("attacher", NULL, 0);
  /* Hunt for the client */
  linx_hunt(linx, "client", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, sel_hunt_sig);
  /* Retrive the clients spid */
  client = linx_sender(linx, &sig)
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
  /* Attach to the client */
  linx_attach(linx, client, NULL);
  /* Create "done" signal */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), CLIENT_DONE_SIG);
  /* Send "done" signal to client */
  linx_send(linx, &sig, client);
  /* Wait for the attach signal */
  linx_receive(linx, &sig, sel_att_sig);
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
Client:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_done_sig[] = { 1, CLIENT_DONE_SIG };

  /* Open a LINX endpoint with huntname "client" */
  linx = linx_open("client", NULL, 0);
  /* Wait for server to send "done" signal */
  linx_receive(linx, &sig, sel_done_sig);
  /* Close the LINX endpoint - this will trigger the attach */
  linx_close(linx);
  return 0;
}
```

See Also

linx(7) , linx_attach(3) , linx_close(3) , linx_detach(3) , linx_hunt(3) , linx_send(3)

Author

Enea LINX team

LINXDISC(8) manual page

Name

linxdisc - The LINX discovery daemon

Synopsis

linxdisc [-d] [-c *config-file* **] [-r** *retries* **]**

Description

linxdisc is the LINX discovery daemon, which automatically detects and creates connections (links) to other remote LINX nodes, which also is running **linxdisc**. The daemon periodically sends advertisements and waits for advertisements from remote nodes. The period is about 3 seconds. Connections are created to allowed remote nodes, when advertisements are received.

See the configuration file *linxdisc.conf*(5) for filter rules, limiting which interfaces to use and which remote nodes to allow. Communication links are only allowed to remote nodes which advertise the same LINX network cluster name as configured and with a node name fulfilling the configured allow/deny rules. The configured LINX cluster name and node name are used in advertisements sent from this node. The cluster name and node name must be unique.

The advertised node name is the suggested linkname to use for links to the advertised node. The linkname is used in huntname paths by applications in other nodes, when they hunt for LINX endpoints in this node.

linxdisc must be run as root and there can only be one instance of linxdisc running on each node. When linxdisc catches a SIGHUP, the configuration will be reread. Use SIGTERM to terminate linxdisc. When linxdisc terminates, all connections are automatically destroyed.

The linxdisc protocol is described in detail in the Linx Protocol document (see http://linx.sourceforge.net).

Options

-d
> Run in debug mode. The program is interactive, not run as a daemon, and all progress and activity is logged to standard error.

-c config-file
> Use *config-file* instead of the default configuration file */etc/linxdisc.conf*.

Files

/etc/linxdisc.conf

> The default configuration file. See *linxdisc.conf*(5) for further details.

Diagnostics

linxdisc writes all activity and errors to the syslog, unless the -d option was given.

Known Bugs and Limitations

linxdisc cannot apply parameters such as timeouts etc when creating links. Interfaces that are unavailable when starting linxdisc will not be used when they become available, linxdisc has to be restarted in that case.

See Also

linx(7) , **linx_hunt**(3) , **linxdisc.conf**(5)

Author

Enea LINX team

LINXDISC.CONF(5) manual page

Name

linxdisc.conf - custom settings for linxdisc

Synopsis

/etc/linxdisc.conf

Description

This file contains configuration variables for the linxdisc daemon. For non-configured variables, default values are used.

The linxdisc.conf must be located in the /etc directory, unless specified when linxdisc is started.

The variables are set using this semantics: VALUE="value1 value2 ..."

When the linxdisc daemon catches a SIGHUP, the configuration will be reread. IFACE, LINX_NETWORK_NAME and NODE_NAME will not be changed during runtime, while ALLOW and DENY will be applied immediately to close disallowed connections and when creating new connections.

Variables

IFACE

> specifies what network interface(s) to use. Only real devices are supported. Several interfaces can be specified. e.g. IFACE="eth0 eth1". Default is to use all available devices.

LINX_NETWORK_NAME

> specifies the name of a LINX cluster. linxdisc only creates LINX communication links to nodes, advertising the same cluster name. All nodes must have the linxdisc daemon running. Default is the cluster name "undefined".

NODE_NAME

> specifies a node name to advertise. It is used by other nodes to filter against the allow/deny filter chains. It is also used as the recommended local link name, when linxdisc in a remote node creates LINX communication links to this node, as a response to the advertisement. Default node name is the hostname. Example: NODE_NAME="server1".

ALLOW

> chain. If specified, linxdisc will only accept connections to nodes with the specified advertised node names. Names are given in a string, separated by spaces. This variable can be set several times in the configuration file and the names will be concatenated to one ALLOW chain. Example: ALLOW="node1 node2".

DENY

> chain. NOTE: This setting will not have any effect if ALLOW is set. If DENY is specified instead of ALLOW, linxdisc will allow LINX connections to any node, except those with advertised node names listed here. Names are given in a string, separated by spaces. This variable can be set several times in the configuration file and the names will be concatenated to one DENY chain.

PARAM

> List of parameters to apply to all links created by linxdisc. For a description of parameters and allowed values, see linxcfg(1) . This variable can be set several times in this file and the specified parameters are concatenated to one PARAM chain.

See Also

linxdisc(8)

Author

Enea LINX team

LINX_FREE_BUF(3) manual page

Name

linx_free_buf() - Free a signal buffer owned by a LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_free_buf(LINX *linx**, union LINX_SIGNAL ****sig**);**

Description

Free a signal buffer, owned by the LINX endpoint, associated with the *linx* handle. Once the buffer is freed it may not be accessed again. To be owned by this LINX endpoint, the buffer has either been received by or allocated at this endpoint.

linx is the handle to the LINX endpoint, owning the *sig* signal buffer.

sig is the signal buffer. When the call returns, the pointer (*sig) is set to LINX_NIL to prevent further access.

If *sig* is corrupt, **abort**(3) is called.

Return Value

On success, 0 is returned. The function does not return on error.

Bugs/Limitations

None.

See Also

linx(7) , **linx_alloc**(3) , **linx_open**(3) , **linx_receive**(3) ,

Author

Enea LINX team

LINX_FREE_NAME(3) manual page

Name

linx_get_name() - Get the name of another LINX endpoint
linx_free_name() - Free the name returned by **linx_get_name**(3)

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_get_name(**LINX ***linx**, **LINX_SPID** *spid**, **char ****name**);**

int linx_free_name(**LINX ***linx**, **char ****name**);**

Description

Get the full hunt name of the LINX endpoint associated with the spid binary identifier (spid). The hunt name is a string which may include a path of link names if the LINX endpoint is located on a remote node, e.g. "LinkToA/LinkToB/LinkToC/EndpointName".

linx is the handle of the own LINX endpoint.

spid is the spid of the LINX endpoint for which the name is requested.

name is a pointer to a hunt name buffer on successful return, otherwise NULL. Unless name is NULL on return, the name buffer must be freed with **linx_free_name**(3) , when it is no longer needed.

Return Value

On success, 0 is returned and *name* contains a pointer to a buffer containing the huntname. The user is responsible for freeing the buffer with **linx_free_name**(3) .
In case of failure, -1 will be returned by both calls *errno* will be set.

Errors

EBADF The *linx* handle refers to an invalid socket descriptor.

EINVAL The name parameter is NULL.

ENOMEM Insufficient memory is available.

Bugs/Limitations

None.

ENEA

See Also

linx(7) , linx_open(3) , malloc(3) , free(3)

Author

Enea LINX team

LINX_FREE_STAT(3) manual page

Name

linx_get_stat() - Get the statistics of a LINX endpoint
linx_free_stat() - Free the statistics struct returned by **linx_get_stat**(3)

Synopsis

#include <linx_types.h>
#include <linx_ioctl.h>
#include <linx.h>

int linx_get_stat(LINX **linx***, LINX_SPID** *spid***, struct linx_info_stat *****stat***);**

int linx_free_stat(LINX **linx***, struct linx_info_stat *****stat***);**

Description

Get the statistics of the LINX endpoint, associated with the local *spid* binary identifier if the LINX kernel module is compiled with SOCK_STAT=yes.

linx is the handle to the LINX endpoint, which is used to retrive the statistics.

spid is the identifier of the other LINX endpoint, for which the statistics is requested.

stat will on return point to an struct *linx_info_stat* buffer contining the statistics. Unless returned as NULL, the buffer must be freed by the user with **linx_free_stat**(3) , when no longer needed. For details on the *linx_info_stat* struct see **linx**(7) .

Return Value

Returns 0 if successful, for **linx_get_stat**(3) the *stat* will contain a pointer to a buffer, allocated by LINX, containing the statistics. The user is responsible for freeing the buffer with **linx_free_stat**(3) .
In case of failure, -1 will be returned and *errno* will be set.

Errors

ENOSYS if the LINX kernel module has not been compiled with SOCK_STAT=yes.

ENOMEM Insufficient memory is available.

EBADF The *linx* handle is associated with an invalid socket descriptor.

Bugs/Limitations

None.

See Also

linx(7) , linx_open(3)

Author

Enea LINX team

LINX_GET_DESCRIPTOR(3) manual page

Name

linx_get_descriptor() - Get the socket descriptor associated with a LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_get_descriptor(LINX **linx***);**

Description

The **linx_get_descriptor**() call returns the socket descriptor associated with a LINX endpoint. This socket descriptor is usually not needed when using the LINX API.

The LINX socket descriptor must be used if the application wants to use the generic **select**(2) or **poll**(2) calls. The application can then use LINX API function calls with the corresponding LINX endpoint handle to take care of events reported for a LINX socket descriptor. Alternatively, the LINX socket interface can be used.

linx is the handle of the LINX endpoint for which the socket descriptor is retrieved.

Return Value

Returns a socket descriptor.

Bugs/Limitations

None.

Notes

A LINX socket descriptor retrieved by **linx_get_descriptor**(3) must NOT be closed by calling **close**(2) . Close the LINX endpoint instead using **linx_close**(3) .

See Also

linx(7) , **linx_close**(3) ,**linx_open**(3) , **select**(2) , **poll**(2)

Author

Enea LINX team

LINX_GET_NAME(3) manual page

Name

linx_get_name() - Get the name of another LINX endpoint
linx_free_name() - Free the name returned by **linx_get_name**(3)

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_get_name(LINX **linx***, LINX_SPID** *spid***, char *****name***);**

int linx_free_name(LINX **linx***, char *****name***);**

Description

Get the full hunt name of the LINX endpoint associated with the spid binary identifier (spid). The hunt name is a string which may include a path of link names if the LINX endpoint is located on a remote node, e.g. "LinkToA/LinkToB/LinkToC/EndpointName".

linx is the handle of the own LINX endpoint.

spid is the spid of the LINX endpoint for which the name is requested.

name is a pointer to a hunt name buffer on successful return, otherwise NULL. Unless name is NULL on return, the name buffer must be freed with **linx_free_name**(3) , when it is no longer needed.

Return Value

On success, 0 is returned and *name* contains a pointer to a buffer containing the huntname. The user is responsible for freeing the buffer with **linx_free_name**(3) .
In case of failure, -1 will be returned by both calls *errno* will be set.

Errors

EBADF The *linx* handle refers to an invalid socket descriptor.

EINVAL The name parameter is NULL.

ENOMEM Insufficient memory is available.

Bugs/Limitations

None.

See Also

linx(7) , linx_open(3) , malloc(3) , free(3)

Author

Enea LINX team

LINX_GET_SPID(3) manual page

Name

linx_get_spid() - Get the binary identifier of LINX endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_SPID linx_get_spid(LINX *linx);

Description

Get the binary identifier (spid) of the LINX endpoint referred to by the linx LINX handle.

Return Value

Returns the spid.

Bugs/Limitations

None.

See Also

linx(7) , **linx_open**(3)

Author

Enea LINX team

LINX_GET_STAT(3) manual page

Name

linx_get_stat() - Get the statistics of a LINX endpoint
linx_free_stat() - Free the statistics struct returned by **linx_get_stat**(3)

Synopsis

#include <linx_types.h>
#include <linx_ioctl.h>
#include <linx.h>

int linx_get_stat(LINX *linx**, LINX_SPID** spid**, struct linx_info_stat ****stat**);**

int linx_free_stat(LINX *linx**, struct linx_info_stat ****stat**);**

Description

Get the statistics of the LINX endpoint, associated with the local *spid* binary identifier if the LINX kernel module is compiled with SOCK_STAT=yes.

linx is the handle to the LINX endpoint, which is used to retrive the statistics.

spid is the identifier of the other LINX endpoint, for which the statistics is requested.

stat will on return point to an struct *linx_info_stat* buffer contining the statistics. Unless returned as NULL, the buffer must be freed by the user with **linx_free_stat**(3) , when no longer needed. For details on the *linx_info_stat* struct see **linx**(7) .

Return Value

Returns 0 if successful, for **linx_get_stat**(3) the *stat* will contain a pointer to a buffer, allocated by LINX, containing the statistics. The user is responsible for freeing the buffer with **linx_free_stat**(3) .
In case of failure, -1 will be returned and *errno* will be set.

Errors

ENOSYS if the LINX kernel module has not been compiled with SOCK_STAT=yes.

ENOMEM Insufficient memory is available.

EBADF The *linx* handle is associated with an invalid socket descriptor.

Bugs/Limitations

None.

See Also

linx(7) , **linx_open**(3)

Author

Enea LINX team

LINX.H(3) manual page

Name

linx.h - The LINX API

Synopsis

#include <linx.h>

Description

This is the main header file for applications using the LINX library. This library provides functions to handle communication between LINX endpoints using a messaging API.
Inclusion of the <linx.h> header may also make all symbols from *<stdint.h>* , *<sys/types.h>* and *<linx_types.h>* visible.

The **linx_open()** function creates a LINX endpoint and returns a handle that should be used in all other LINX API calls.

LINX *linx_open(const char **name***, uint32_t** *options***, void ****arg***);**

int linx_close(LINX **linx***);**

int linx_get_descriptor(LINX **linx***);**

union LINX_SIGNAL *linx_alloc(LINX **linx***, LINX_OSBUFSIZE** *size***, LINX_SIGSELECT** *sig_no***);**

int linx_free_buf(LINX **linx***, union LINX_SIGNAL *****sig***);**

int linx_send(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *to***);**

int linx_send_w_opt(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *from***, LINX_SPID** *to***, int32_t ****taglist***);**

int linx_send_w_s(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *from***, LINX_SPID** *to***);**

int linx_receive(LINX **linx***, union LINX_SIGNAL *****sig***, const LINX_SIGSELECT ****sig_sel***);**

int linx_receive_w_tmo(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_OSTIME** *tmo***, const LINX_SIGSELECT ****sig_sel***);**

int linx_receive_from(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_OSTIME** *tmo***, const LINX_SIGSELECT ****sig_sel***, LINX_SPID** *from***);**

LINX_SPID linx_sender(LINX **linx***, union LINX_SIGNAL *****sig***);**

int linx_sigattr(const LINX **linx***, const union LINX_SIGNAL *****sig***, uint32_t** *attr***, void *****value***);**

LINX_OSBUFSIZE linx_sigsize(LINX **linx***, union LINX_SIGNAL *****sig***);**

int linx_set_sigsize(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_OSBUFSIZE** *sigsize***);**

int linx_hunt(LINX **linx***, const char ****name***, union LINX_SIGNAL *****hunt_sig***);**

int linx_hunt_from(LINX **linx***, const char ****name***, union LINX_SIGNAL *****hunt_sig***, LINX_SPID** *from***);**

LINX_OSATTREF linx_attach(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *spid***);**

int linx_detach(LINX **linx***, LINX_OSATTREF ****attref***);**

LINX_SPID linx_get_spid(LINX **linx***);**

int linx_get_name(LINX **linx***, LINX_SPID** *spid***, char *****name***);**

int linx_free_name(LINX **linx***, char *****name***);**

int linx_get_stat(LINX **linx***, LINX_SPID** *spid***, struct linx_info_stat *****stat***);**

int linx_free_stat(LINX **linx***, struct linx_info_stat *****stat***);**

LINX_OSTMOREF linx_request_tmo(LINX **linx***, LINX_OSTIME** *tmo***, union LINX_SIGNAL *****sig***);**

int linx_cancel_tmo(LINX **linx***, LINX_OSTMOREF ****tmoref***);**

int linx_modify_tmo(LINX **linx***, LINX_OSTMOREF ****tmoref***, LINX_OSTIME** *tmo***);**

LINX_NLREF linx_request_new_link(LINX **linx***, LINX_NLTOKEN** *token***);**

int linx_cancel_new_link(LINX **linx***, LINX_NLREF *** *nlref***);**

Future Directions

The LINX API is still under development and may be subject to change in future releases.

Known Bugs

None.

Author

Enea LINX team

See Also

linx(7) , **linx_open**(3) , **linx_types.h**(3) , **linx_close**(3) , **linx_get_descriptor**(3) , **linx_alloc**(3) , **linx_free_buf**(3) , **linx_send**(3) , **linx_send_w_opt**(3) , **linx_send_w_s**(3) , **linx_receive**(3) , **linx_receive_w_tmo**(3) , **linx_receive_from**(3) , **linx_sender**(3) , **linx_sigattr**(3) , **linx_sigsize**(3) , **linx_set_sigsize**(3) , **linx_hunt**(3) , **linx_hunt_from**(3) , **linx_attach**(3) , **linx_detach**(3) , **linx_get_spid**(3) , **linx_get_name**(3) , **linx_free_name**(3) , **linx_get_stat**(3) , **linx_free_stat**(3) , **linx_request_tmo**(3) , **linx_modify_tmo**(3) , **linx_cancel_tmo**(3) , **linx_request_new_link**(3) , **linx_cancel_new_link**(3)

LINX_HUNT(3) manual page

Name

linx_hunt() - Search for a LINX endpoint matching a name

linx_hunt_from() - Search for a LINX endpoint matching a name and set another LINX endpoint to be the owner of the hunt.

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_hunt(LINX **linx***, const char ****name***, union LINX_SIGNAL *****hunt_sig***);**

int linx_hunt_from(LINX **linx***, const char ****name***, union LINX_SIGNAL *****hunt_sig***, LINX_SPID** *from***);**

Description

linx_hunt() and **linx_hunt_from**() are used to search for a LINX endpoint matching the provided hunt *name,* usually in order to be able to obtain its spid and communicate with it. The hunt name may optionally be prepended by a path of link names to reach a remote node on which the endpoint shall be found. When the endpoint has been found, the provided signal *hunt_sig* is sent from the found endpoint to the calling endpoint. If no hunt signal (NULL) is provided, the default LINX hunt signal of type LINX_OS_HUNT_SIG is used. The call consumes the signal, taking ownership of the signal, and sets the *hunt_sig* pointer to LINX_NIL to prevent further access. The signal is consumed also if an error occurs.

If there are multiple endpoints with the same name, it is not specified which of these endpoints is used to resolve the hunt call. When the hunt signal is received, the *linx_sender*(3) call can be used to get the spid of the found endpoint. Note that the returned spid is always local even though it may correspond to a LINX endpoint on a remote node. LINX will transparently route signals to their intended destination endpoints on remote nodes when needed.

A pending hunt will be automatically cancelled if the *linx* endpoint that owns it is closed, e.g. if the caller exits. In the case with **linx_hunt_from**(), the *from* parameter indicates a different owner of the pending hunt. If the *from* endpoint is closed while the hunt is pending, the hunt will be freed. Note that *from* does NOT receive the hunt signal.

name is a null-terminated string that shall exactly match a LINX endpoints name. If the endpoint is located on a remote node, the name of a link to that node (or names in a sequence of links) shall be included as the leading part of the *name* according to the following syntax:
"[<link_1>/][<link_2>/]...[<link_N>/]name"
Example: "LinkToA/LinkToB/EndpointName" where a path of two links are passed to reach the LINX endpoint.

linx is the handle to the LINX endpoint that will receive the hunt signal.

hunt_sig is the signal which will be sent to the caller when the hunt has been resolved. The sender of the signal will be the matching LINX endpoint found. If *hunt_sig* is NULL, a default LINX hunt signal with signal number LINX_OS_HUNT_SIG will be sent instead. If *hunt_sig* is corrupt, **abort**(3) is called.

from specifies a different owner of a possible pending hunt. If the endpoint, with spid *from*, is closed, the pending hunt will be cancelled.

Return Value

Returns 0 if successful. -1 if an error occurred in which case errno is set.

Errors

ENOMEM Insufficient memory is available.

EBADF The *linx* handle refers to an invalid socket descriptor.

EFAULT The .I name parameter is invalid.

ECONNRESET The *from* spid in the **linx_hunt_from**(3) refers to a LINX endpoint that has been closed (i.e. in state LINX_ZOMBIE).

EPIPE This error is reported at an attempt to do a **linx_hunt_from**(3) and the *from* spid is a LINX endpoint that is being closed as the call occurs.

Bugs/Limitations

None.

Example

In this example client1 hunts for client2 and client3 */

```
Client 1:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client2, client3;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  /* Open a LINX endpoint with huntname "client1" */
  linx = linx_open("client1", NULL, 0);
  /* Hunt for client2 */
  linx_hunt(linx, "client2", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, sel_hunt_sig);
  /* Retrieve client2's spid */
  client2 = linx_sender(linx, &sig);
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
   /* Hunt for client3 */
  linx_hunt(linx, "client3", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, sel_hunt_sig);
  /* Retrieve client3's spid */
  client3 = linx_sender(linx, &sig);
  /* Free the hunt signal */
```

```
  linx_free_buf(linx, &sig);
  /* Do work, e.g. exchanges signals with client2 or client3 ... */
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
Client 2:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  /* Open a LINX endpoint with huntname "client2" */
  linx = linx_open("client2", NULL, 0);
  /* Do work.... */
  /* When done close the LINX endpoint */
  linx_close(linx);
  /* "client2" can no longer be hunted on */
  return 0;
}
Client 3:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  /* Open a LINX endpoint with huntname "client3" */
  linx = linx_open("client3", NULL, 0);
  /* Do work.... */
  /* When done close the LINX endpoint */
  linx_close(linx);
  /* "client3" can no longer be hunted on */
  return 0;
}
```

See Also

linx(7) , **linx_attach**(3) , **linx_sender**(3)

Author

Enea LINX team

Example   73

LINX_HUNT_FROM(3) manual page

Name

linx_hunt() - Search for a LINX endpoint matching a name

linx_hunt_from() - Search for a LINX endpoint matching a name and set another LINX endpoint to be the owner of the hunt.

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_hunt(LINX *linx**, const char ***name**, union LINX_SIGNAL ****hunt_sig**);**

int linx_hunt_from(LINX *linx**, const char ***name**, union LINX_SIGNAL ****hunt_sig**, LINX_SPID** from**);**

Description

linx_hunt() and **linx_hunt_from**() are used to search for a LINX endpoint matching the provided hunt *name,* usually in order to be able to obtain its spid and communicate with it. The hunt name may optionally be prepended by a path of link names to reach a remote node on which the endpoint shall be found. When the endpoint has been found, the provided signal *hunt_sig* is sent from the found endpoint to the calling endpoint. If no hunt signal (NULL) is provided, the default LINX hunt signal of type LINX_OS_HUNT_SIG is used. The call consumes the signal, taking ownership of the signal, and sets the *hunt_sig* pointer to LINX_NIL to prevent further access. The signal is consumed also if an error occurs.

If there are multiple endpoints with the same name, it is not specified which of these endpoints is used to resolve the hunt call. When the hunt signal is received, the *linx_sender*(3) call can be used to get the spid of the found endpoint. Note that the returned spid is always local even though it may correspond to a LINX endpoint on a remote node. LINX will transparently route signals to their intended destination endpoints on remote nodes when needed.

A pending hunt will be automatically cancelled if the *linx* endpoint that owns it is closed, e.g. if the caller exits. In the case with **linx_hunt_from**(), the *from* parameter indicates a different owner of the pending hunt. If the *from* endpoint is closed while the hunt is pending, the hunt will be freed. Note that *from* does NOT receive the hunt signal.

name is a null-terminated string that shall exactly match a LINX endpoints name. If the endpoint is located on a remote node, the name of a link to that node (or names in a sequence of links) shall be included as the leading part of the *name* according to the following syntax:
"[<link_1>/][<link_2>/]...[<link_N>/]name"
Example: "LinkToA/LinkToB/EndpointName" where a path of two links are passed to reach the LINX endpoint.

linx is the handle to the LINX endpoint that will receive the hunt signal.

hunt_sig is the signal which will be sent to the caller when the hunt has been resolved. The sender of the signal will be the matching LINX endpoint found. If *hunt_sig* is NULL, a default LINX hunt signal with signal number LINX_OS_HUNT_SIG will be sent instead. If *hunt_sig* is corrupt, **abort**(3) is called.

from specifies a different owner of a possible pending hunt. If the endpoint, with spid *from*, is closed, the pending hunt will be cancelled.

Return Value

Returns 0 if successful. -1 if an error occurred in which case errno is set.

Errors

ENOMEM Insufficient memory is available.

EBADF The *linx* handle refers to an invalid socket descriptor.

EFAULT The .I name parameter is invalid.

ECONNRESET The *from* spid in the **linx_hunt_from**(3) refers to a LINX endpoint that has been closed (i.e. in state LINX_ZOMBIE).

EPIPE This error is reported at an attempt to do a **linx_hunt_from**(3) and the *from* spid is a LINX endpoint that is being closed as the call occurs.

Bugs/Limitations

None.

Example

In this example client1 hunts for client2 and client3 */

```
Client 1:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client2, client3;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_hunt_sig[] = { 1, LINX_OS_HUNT_SIG };
  /* Open a LINX endpoint with huntname "client1" */
  linx = linx_open("client1", NULL, 0);
  /* Hunt for client2 */
  linx_hunt(linx, "client2", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, sel_hunt_sig);
  /* Retrieve client2's spid */
  client2 = linx_sender(linx, &sig);
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
   /* Hunt for client3 */
  linx_hunt(linx, "client3", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, sel_hunt_sig);
  /* Retrieve client3's spid */
  client3 = linx_sender(linx, &sig);
  /* Free the hunt signal */
```

Description   75

```
  linx_free_buf(linx, &sig);
  /* Do work, e.g. exchanges signals with client2 or client3 ... */
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
Client 2:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  /* Open a LINX endpoint with huntname "client2" */
  linx = linx_open("client2", NULL, 0);
  /* Do work.... */
  /* When done close the LINX endpoint */
  linx_close(linx);
  /* "client2" can no longer be hunted on */
  return 0;
}
Client 3:
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  /* Open a LINX endpoint with huntname "client3" */
  linx = linx_open("client3", NULL, 0);
  /* Do work.... */
  /* When done close the LINX endpoint */
  linx_close(linx);
  /* "client3" can no longer be hunted on */
  return 0;
}
```

See Also

linx(7) , **linx_attach**(3) , **linx_sender**(3)

Author

Enea LINX team

Example   76

LINXLS(1) manual page

Name

linxls - list LINX links and connections.

Synopsis

linxls

Description

List any LINX links and connections and the state of these.

Options

None.

Known Bugs

None.

See Also

linx(7) , **linxstat**(1) , **linxcfg**(1)

Author

Enea LINX team

LINX_MODIFY_TMO(3) manual page

Name

linx_request_tmo() - request timeout with specified signal
linx_cancel_tmo() - cancel the specified timeout.
linx_modify_tmo() - modify the specified timeout

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSTMOREF linx_request_tmo(LINX *linx**, LINX_OSTIME** tmo**, union LINX_SIGNAL ****sig**);**

int linx_cancel_tmo(LINX *linx**, LINX_OSTMOREF ***tmoref**);**

int linx_modify_tmo(LINX *linx**, LINX_OSTMOREF ***tmoref**, LINX_OSTIME** tmo**);**

Description

linx_request_tmo() is used to request a signal, *sig*, to be sent to the requesting LINX endpoint *linx* when a timeout has expired. Information necessary for cancelling the time-out is returned. The actual time-out time, *tmo*, is rounded upward to the next larger tick because a time-out can only trigger when a system clock tick is received. Hence, the routine guarantees at least the number of milliseconds requested, but may add a few more depending on the tick resolution. When several time-out signals are generated at the same clock tick, the order is unspecified. If *sig* is NULL, the default LINX timeout signal with signal number LINX_OS_TMO_SIG is received instead. The linx_request_tmo () call consumes the signal, and sets the *sig* pointer to LINX_NIL. The signal is also consumed if an error occurs.

linx_cancel_tmo() is used to cancel a timeout. The time-out is identified by the *tmoref* parameter, which was set by **linx_request_tmo**()**.** It's an error to cancel a timeout more than once or to cancel a timeout after the timeout signal has been received. It is not an error to cancel the timeout if the timeout signal is waiting in the receive queue of the LINX endpoint. To prevent multiple cancellations, **linx_cancel_tmo**() sets the tmoref pointer to LINX_ILLEGAL_TMOREF.

linx_modify_tmo() is used to modify a timeout. The time-out is identified by the tmoref parameter, which was set by **linx_request_tmo**()**.** It's an error to modify a timeout after the timeout signal has been received. It is not an error to modify the timeout if the timeout signal is waiting in the receive queue of the LINX endpoint.

linx is the handle of the LINX endpoint.

spid is the timeout time in milliseconds.

sig is either NULL or a user defined LINX signal.

tmoref is the reference to the timeout obtained from **linx_request_tmo**().

Return Value

linx_request_tmo() returns an timeout reference (LINX_OSTMOREF) when successful. This timeout reference can be used to cancel the timeout. **linx_cancel_tmo**() returns 0 on success.

On failure, **linx_request_tmo**() returns LINX_ILLEGAL_TMOREF while **linx_cancel_tmo**() and **linx_modify_tmo**() both returns -1. In all cases *errno* will be set appropriately.

Errors

EBADF, **ENOTSOCK** The *linx* handle refers to an invalid socket descriptor.

ENOMEM Not enough memory.

EINVAL Invalid argument.

Bugs/Limitations

None.

Example

This example shows how to request and wait for a timeout.

```
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_tmo_sig [] = { 1, LINX_OS_TMO_SIG };
  /* Create a LINX endpoint */
  linx = linx_open("tmo-test", NULL, 0);
  /* Request a one second timeout. When the timeout expires the
     default signal will be returned. */
  linx_request_tmo(linx, 1000, NULL);
  /* Wait for the timeout signal. */
  linx_receive(linx, &sig, sel_tmo_sig);
  /* Free the timeout signal */
  linx_free_buf(linx, &sig);
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
```

See Also

linx(7) , **linx_close**(3) , **linx_open**(3) **linx_receive**(3) ,

Author

Enea LINX team

LINX_OPEN(3) manual page

Name

linx_open - Create a LINX endpoint and give it a huntname.

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX *linx_open(const char *name**, uint32_t** options**, void ***arg**);**

Description

A process uses **linx_open**() to create a LINX endpoint and give it a huntname. The call returns a handle to the LINX endpoint to be used in subsequent LINX API calls. As part of the **linx_open**(3) call, any pending hunts for the *name* are resolved, whether it is a local hunt or a hunt from a remote node connected via LINX.

name is the local part of the huntname to find this LINX endpoint. It is a null-terminated string, which in Linux may be of any length, but there may be restrictions due to other implementations, when communicating with other systems. *name* does not need to be unique in the system, but if multiple LINX endpoints have the same name it is unspecified which one is used when a **linx_hunt**(3) call for the name is resolved. From another LINX endpoint, use **linx_hunt**(3) with the huntname, followed by **linx_receive**(3) and **linx_sender**(3) , to get the binary identifier (spid) to be used to communicate with this LINX endpoint. The local LINX binary identifier (spid) for this endpoint can be fetched with **linx_get_spid**(3) but is usually not needed by the owner process.

The *options* and *arg* parameters are reserved for future extensions and shall be set to 0 and NULL respectively.

A LINX endpoint is closed by calling **linx_close**(3) or when the calling process exits.
If the process owning the LINX endpoint calls **fork**(2) , either the parent or the child must close the LINX handle with **linx_close**(3) . This is because only one process may use a LINX endpoint and after a fork both instances will have copies of the LINX handle. In any case, the LINX endpoint will not be terminated unless closed by both parent and child.

It is possible to retrieve the LINX socket descriptor associated with a LINX endpoint using **linx_get_descriptor**(3) if needed.

Return Value

On success, a LINX handle is returned. On error, NULL is returned and *errno* is set.

Errors

EINVAL Invalid argument passed.

ENOBUFS or **ENOMEM** Insufficient memory is available.

EPROTONOSUPPORT or **EAFNOSUPPORT** if *linx(7)* is not supported by the system.

ENFILE Not enough kernel memory to allocate a new LINX endpoint structure.

EMFILE Process file table overflow.

EACCES Permission to create a LINX endpoint is denied.

Bugs/Limitations

None.

See Also

linx(7) , **linx_close**(3) , **linx_hunt**(3) , **linx_get_descriptor**(3) , **linx_receive**(3) , **linx_send**(3) , **linx_sender**(3) , **fork**(2)

Author

Enea LINX team

LINX_RECEIVE(3) manual page

Name

linx_receive() - Receive a LINX signal
linx_receive_w_tmo() - Receive a LINX signal with timeout
linx_receive_from() - Receive a LINX signal, but only from a given endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_receive(LINX **linx***, union LINX_SIGNAL *****sig***, const LINX_SIGSELECT ****sig_sel***);**

int linx_receive_w_tmo(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_OSTIME** *tmo***, const LINX_SIGSELECT ****sig_sel***);**

int linx_receive_from(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_OSTIME** *tmo***, const LINX_SIGSELECT ****sig_sel***, LINX_SPID** *from***);**

Description

Receives a LINX signal. The calls will block until a signal is received or, if applicable, a timeout has elapsed. The first signal in the receive queue that matches the *sig_sel* filter is returned to the user. This way the user may process signals in a different order than when they arrived at the LINX endpoint. Signals that have arrived but have not yet been received using these calls will stay in the receive queue.

linx_receive() will wait indefinitely for a signal that matches the *sig_sel* filter. *sig_sel* filters are described in **linx**(7) .

linx_receive_w_tmo() waits until the provided timeout, *tmo*, has elapsed or for a signal that matches the *sig_sel* filter. When the timeout has elapsed, zero is returned and *sig* is set to LINX_NIL instead of a pointer to a signal buffer.

linx_receive_from() works in the same way as **linx_receive_w_tmo**() except that it will only accept signals sent from the LINX endpoint, indicated by the spid *from*.

linx is the handle to the LINX endpoint, via which the signals are received.

sig is the received signal. The signal buffer is allocated by **linx_receive**(3) . Never use a preallocated signal buffer as it will be lost.

sig_sel is a select filter, defining which types of buffers to receive. It is a list of signal numbers with a leading count indicating the number of signal numbers in the list. If the first position is set to a negative count, all signal numbers except those listed will be received. Read more about select filters in **linx**(7) .

tmo is the maximum number of milliseconds to wait. The value 0 will result in a check for a signal matching *sig_sel* followed by an immediate return.

from is the spid of the other LINX endpoint to receive from. Before **linx_receive_from**() is used, it is important to attach to the other LINX endpoint with **linx_attach**(3) and also to include the attach signal

number in the *sig_sel* filter. If this is not done and the other LINX endpoint (with spid *from*) is closed, this call will block the entire specified timeout.

Return Value

Returns the size of the received signal in bytes if successful. If a signal was received, it can be found in *sig*, otherwise *sig* will be LINX_NIL. Returns 0 if **linx_receive_w_tmo**(3) was called and no message was received before the timeout.

If unsuccessful, -1 is returned and *errno* is set.

Errors

ENOMEM if there is not enough memory.

EBADF, **EFAULT**, **ENOTCONN**, **ENOTSOCK** if the underlying *linx* structure contains a invalid socket descriptor.

EINTR the call was interrupted by a signal.

Bugs/Limitations

None.

Notes

In case the application needs to wait on multiple sockets of different kinds, the internal socket descriptor in a LINX endpoint can be fetched with **linx_get_descriptor***(3)* and used in a **select***(2)* or **poll***(2)* call to wake up if anything arrives at the LINX endpoint.

Example

In this example the server sends four signals to the client and the client chooses to receive them in the order it wants to.

```
Signal file (example.sig):
#include <linx.h>
#define SIG_X 0x1
#define SIG_Y 0x2
#define SIG_Z 0x3
#define SIG_W 0x4
/* select filter is { number-of-signals, signal, signal, ... } */
static const LINX_SIGSELECT sigsel_any[]     = { 0 };
static const LINX_SIGSELECT sigsel_sig_x_z[] = { 2, SIG_X, SIG_Z };
static const LINX_SIGSELECT sigsel_sig_w[]   = { 1, SIG_W };
Server:
#include <linx.h>
#include "example.sig"
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
```

```
  union LINX_SIGNAL *sig;
  /* Open a linx endpoint with huntname "server" */
  linx = linx_open("server", NULL, 0);
  /* Hunt for client */
  linx_hunt(linx, "client", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, LINX_OS_HUNT_SIG);
  /* Retrieve client's spid */
  client = linx_sender(linx, &sig);
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
  /*  Send four signals, they will be stored in the receive
   *  queue on the client in same order as sent but the
   *  client chooses in which order to retrieve them from
   *  the queue.
   */
  /* Send signal with signal number SIG_X */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_X);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_Y */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_Y);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_Z */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_Z);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_W */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_W);
  linx_send(linx, &sig, client);
  linx_free_buf(linx, &sig);
  /* Close the linx endpoint */
  linx_close (linx);
}
Client:
#include <linx.h>
#include "example.sig"
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
  /* Open a linx endpoint with huntname "client" */
  linx = linx_open("client", NULL, 0);
  /* Check for signal SIG_W first */
  linx_receive(linx, &sig, sigsel_sig_w);
  /* Do work, sig->sig_no is SIG_W */
  /* Free signal when done */
  linx_free_buf(linx, &sig);
  /* Receive the the first signal waiting in the receive queue */
  linx_receive(linx, &sig, sigsel_any);
  /* Do work, sig->sig_no is SIG_X */
  linx_free_buf(linx, &sig);
  /* Receive either SIG_X or SIG_Z from the receive queue. */
  linx_receive(linx, &sig, sigsel_sig_x_z);
  /* Do work, sig->sig_no is SIG_Z (SIG_X has been consumed) */
  linx_free_buf(linx, &sig);
  /* Receive the the first signal waiting in the receive queue */
  linx_receive(linx, &sig, sigsel_any);
  /* Do work, sig->sig_no is SIG_Y */
  linx_free_buf(linx, &sig);
  linx_close (linx);
}
```

See Also

Example   84

linx(7) , **linx_hunt**(3) , **linx_send**(3) , **linx_sender**(3) , **linx_free_buf**(3) , **linx_alloc**(3) , **linx_get_descriptor**(3) , **poll**(2) , **select**(2)

Author

Enea LINX team

linx(7) , **linx_hunt**(3) , **linx_send**(3) , **linx_sender**(3) , **linx_free_buf**(3) , **linx_alloc**(3) , **linx_get_descriptor**(3) , **poll**(2) , **select**(2)

See Also

LINX_RECEIVE_FROM(3) manual page

Name

linx_receive() - Receive a LINX signal
linx_receive_w_tmo() - Receive a LINX signal with timeout
linx_receive_from() - Receive a LINX signal, but only from a given endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_receive(LINX * *linx* **, union LINX_SIGNAL ***** *sig* **, const LINX_SIGSELECT *** *sig_sel* **);**

int linx_receive_w_tmo(LINX * *linx* **, union LINX_SIGNAL ***** *sig* **, LINX_OSTIME** *tmo* **, const LINX_SIGSELECT *** *sig_sel* **);**

int linx_receive_from(LINX * *linx* **, union LINX_SIGNAL ***** *sig* **, LINX_OSTIME** *tmo* **, const LINX_SIGSELECT *** *sig_sel* **, LINX_SPID** *from* **);**

Description

Receives a LINX signal. The calls will block until a signal is received or, if applicable, a timeout has elapsed. The first signal in the receive queue that matches the *sig_sel* filter is returned to the user. This way the user may process signals in a different order than when they arrived at the LINX endpoint. Signals that have arrived but have not yet been received using these calls will stay in the receive queue.

linx_receive() will wait indefinitely for a signal that matches the *sig_sel* filter. *sig_sel* filters are described in **linx**(7) .

linx_receive_w_tmo() waits until the provided timeout, *tmo*, has elapsed or for a signal that matches the *sig_sel* filter. When the timeout has elapsed, zero is returned and *sig* is set to LINX_NIL instead of a pointer to a signal buffer.

linx_receive_from() works in the same way as **linx_receive_w_tmo**() except that it will only accept signals sent from the LINX endpoint, indicated by the spid *from*.

linx is the handle to the LINX endpoint, via which the signals are received.

sig is the received signal. The signal buffer is allocated by **linx_receive**(3) . Never use a preallocated signal buffer as it will be lost.

sig_sel is a select filter, defining which types of buffers to receive. It is a list of signal numbers with a leading count indicating the number of signal numbers in the list. If the first position is set to a negative count, all signal numbers except those listed will be received. Read more about select filters in **linx**(7) .

tmo is the maximum number of milliseconds to wait. The value 0 will result in a check for a signal matching *sig_sel* followed by an immediate return.

from is the spid of the other LINX endpoint to receive from. Before **linx_receive_from**() is used, it is important to attach to the other LINX endpoint with **linx_attach**(3) and also to include the attach signal

number in the *sig_sel* filter. If this is not done and the other LINX endpoint (with spid *from*) is closed, this call will block the entire specified timeout.

Return Value

Returns the size of the received signal in bytes if successful. If a signal was received, it can be found in *sig*, otherwise *sig* will be LINX_NIL. Returns 0 if **linx_receive_w_tmo**(3) was called and no message was received before the timeout.

If unsuccessful, -1 is returned and *errno* is set.

Errors

ENOMEM if there is not enough memory.

EBADF, **EFAULT**, **ENOTCONN**, **ENOTSOCK** if the underlying *linx* structure contains a invalid socket descriptor.

EINTR the call was interrupted by a signal.

Bugs/Limitations

None.

Notes

In case the application needs to wait on multiple sockets of different kinds, the internal socket descriptor in a LINX endpoint can be fetched with **linx_get_descriptor***(3)* and used in a **select***(2)* or **poll***(2)* call to wake up if anything arrives at the LINX endpoint.

Example

In this example the server sends four signals to the client and the client chooses to receive them in the order it wants to.

```
Signal file (example.sig):
#include <linx.h>
#define SIG_X 0x1
#define SIG_Y 0x2
#define SIG_Z 0x3
#define SIG_W 0x4
/* select filter is { number-of-signals, signal, signal, ... } */
static const LINX_SIGSELECT sigsel_any[]     = { 0 };
static const LINX_SIGSELECT sigsel_sig_x_z[] = { 2, SIG_X, SIG_Z };
static const LINX_SIGSELECT sigsel_sig_w[]   = { 1, SIG_W };
Server:
#include <linx.h>
#include "example.sig"
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
```

```
  union LINX_SIGNAL *sig;
  /* Open a linx endpoint with huntname "server" */
  linx = linx_open("server", NULL, 0);
  /* Hunt for client */
  linx_hunt(linx, "client", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, LINX_OS_HUNT_SIG);
  /* Retrieve client's spid */
  client = linx_sender(linx, &sig);
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
  /*  Send four signals, they will be stored in the receive
   *  queue on the client in same order as sent but the
   *  client chooses in which order to retrieve them from
   *  the queue.
   */
  /* Send signal with signal number SIG_X */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_X);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_Y */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_Y);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_Z */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_Z);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_W */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_W);
  linx_send(linx, &sig, client);
  linx_free_buf(linx, &sig);
  /* Close the linx endpoint */
  linx_close (linx);
}
Client:
#include <linx.h>
#include "example.sig"
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
  /* Open a linx endpoint with huntname "client" */
  linx = linx_open("client", NULL, 0);
  /* Check for signal SIG_W first */
  linx_receive(linx, &sig, sigsel_sig_w);
  /* Do work, sig->sig_no is SIG_W */
  /* Free signal when done */
  linx_free_buf(linx, &sig);
  /* Receive the the first signal waiting in the receive queue */
  linx_receive(linx, &sig, sigsel_any);
  /* Do work, sig->sig_no is SIG_X */
  linx_free_buf(linx, &sig);
  /* Receive either SIG_X or SIG_Z from the receive queue. */
  linx_receive(linx, &sig, sigsel_sig_x_z);
  /* Do work, sig->sig_no is SIG_Z (SIG_X has been consumed) */
  linx_free_buf(linx, &sig);
  /* Receive the the first signal waiting in the receive queue */
  linx_receive(linx, &sig, sigsel_any);
  /* Do work, sig->sig_no is SIG_Y */
  linx_free_buf(linx, &sig);
  linx_close (linx);
}
```

See Also

Example   88

linx(7) , linx_hunt(3) , linx_send(3) , linx_sender(3) , linx_free_buf(3) , linx_alloc(3) ,
linx_get_descriptor(3) , poll(2) , select(2)

Author

Enea LINX team

linx(7) , linx_hunt(3) , linx_send(3) , linx_sender(3) , linx_free_buf(3) , linx_alloc(3) ,
linx_get_descriptor(3) , poll(2) , select(2)

See Also

Enea LINX team

LINX_RECEIVE_W_TMO(3) manual page

Name

linx_receive() - Receive a LINX signal
linx_receive_w_tmo() - Receive a LINX signal with timeout
linx_receive_from() - Receive a LINX signal, but only from a given endpoint

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_receive(LINX * *linx* **, union LINX_SIGNAL ** *** *sig* **, const LINX_SIGSELECT *** *sig_sel* **);**

int linx_receive_w_tmo(LINX * *linx* **, union LINX_SIGNAL ** *** *sig* **, LINX_OSTIME** *tmo* **, const LINX_SIGSELECT *** *sig_sel* **);**

int linx_receive_from(LINX * *linx* **, union LINX_SIGNAL ** *** *sig* **, LINX_OSTIME** *tmo* **, const LINX_SIGSELECT *** *sig_sel* **, LINX_SPID** *from* **);**

Description

Receives a LINX signal. The calls will block until a signal is received or, if applicable, a timeout has elapsed. The first signal in the receive queue that matches the *sig_sel* filter is returned to the user. This way the user may process signals in a different order than when they arrived at the LINX endpoint. Signals that have arrived but have not yet been received using these calls will stay in the receive queue.

linx_receive() will wait indefinitely for a signal that matches the *sig_sel* filter. *sig_sel* filters are described in **linx**(7) .

linx_receive_w_tmo() waits until the provided timeout, *tmo*, has elapsed or for a signal that matches the *sig_sel* filter. When the timeout has elapsed, zero is returned and *sig* is set to LINX_NIL instead of a pointer to a signal buffer.

linx_receive_from() works in the same way as **linx_receive_w_tmo**() except that it will only accept signals sent from the LINX endpoint, indicated by the spid *from*.

linx is the handle to the LINX endpoint, via which the signals are received.

sig is the received signal. The signal buffer is allocated by **linx_receive**(3) . Never use a preallocated signal buffer as it will be lost.

sig_sel is a select filter, defining which types of buffers to receive. It is a list of signal numbers with a leading count indicating the number of signal numbers in the list. If the first position is set to a negative count, all signal numbers except those listed will be received. Read more about select filters in **linx**(7) .

tmo is the maximum number of milliseconds to wait. The value 0 will result in a check for a signal matching *sig_sel* followed by an immediate return.

from is the spid of the other LINX endpoint to receive from. Before **linx_receive_from**() is used, it is important to attach to the other LINX endpoint with **linx_attach**(3) and also to include the attach signal

number in the *sig_sel* filter. If this is not done and the other LINX endpoint (with spid *from*) is closed, this call will block the entire specified timeout.

Return Value

Returns the size of the received signal in bytes if successful. If a signal was received, it can be found in *sig*, otherwise *sig* will be LINX_NIL. Returns 0 if **linx_receive_w_tmo**(3) was called and no message was received before the timeout.

If unsuccessful, -1 is returned and *errno* is set.

Errors

ENOMEM if there is not enough memory.

EBADF, **EFAULT**, **ENOTCONN**, **ENOTSOCK** if the underlying *linx* structure contains a invalid socket descriptor.

EINTR the call was interrupted by a signal.

Bugs/Limitations

None.

Notes

In case the application needs to wait on multiple sockets of different kinds, the internal socket descriptor in a LINX endpoint can be fetched with **linx_get_descriptor**(3) and used in a **select**(2) or **poll**(2) call to wake up if anything arrives at the LINX endpoint.

Example

In this example the server sends four signals to the client and the client chooses to receive them in the order it wants to.

```
Signal file (example.sig):
#include <linx.h>
#define SIG_X 0x1
#define SIG_Y 0x2
#define SIG_Z 0x3
#define SIG_W 0x4
/* select filter is { number-of-signals, signal, signal, ... } */
static const LINX_SIGSELECT sigsel_any[]     = { 0 };
static const LINX_SIGSELECT sigsel_sig_x_z[] = { 2, SIG_X, SIG_Z };
static const LINX_SIGSELECT sigsel_sig_w[]   = { 1, SIG_W };
Server:
#include <linx.h>
#include "example.sig"
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
```

Description   91

```
  union LINX_SIGNAL *sig;
  /* Open a linx endpoint with huntname "server" */
  linx = linx_open("server", NULL, 0);
  /* Hunt for client */
  linx_hunt(linx, "client", NULL);
  /* Receive hunt signal */
  linx_receive(linx, &sig, LINX_OS_HUNT_SIG);
  /* Retrieve client's spid */
  client = linx_sender(linx, &sig);
  /* Free the hunt signal */
  linx_free_buf(linx, &sig);
  /*  Send four signals, they will be stored in the receive
   *  queue on the client in same order as sent but the
   *  client chooses in which order to retrieve them from
   *  the queue.
   */
  /* Send signal with signal number SIG_X */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_X);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_Y */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_Y);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_Z */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_Z);
  linx_send(linx, &sig, client);
  /* Send signal with signal number SIG_W */
  sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), SIG_W);
  linx_send(linx, &sig, client);
  linx_free_buf(linx, &sig);
  /* Close the linx endpoint */
  linx_close (linx);
}
Client:
#include <linx.h>
#include "example.sig"
int
main (int argc, char *argv[])
{
  LINX *linx;
  LINX_SPID client;
  /* Open a linx endpoint with huntname "client" */
  linx = linx_open("client", NULL, 0);
  /* Check for signal SIG_W first */
  linx_receive(linx, &sig, sigsel_sig_w);
  /* Do work, sig->sig_no is SIG_W */
  /* Free signal when done */
  linx_free_buf(linx, &sig);
  /* Receive the the first signal waiting in the receive queue */
  linx_receive(linx, &sig, sigsel_any);
  /* Do work, sig->sig_no is SIG_X */
  linx_free_buf(linx, &sig);
  /* Receive either SIG_X or SIG_Z from the receive queue. */
  linx_receive(linx, &sig, sigsel_sig_x_z);
  /* Do work, sig->sig_no is SIG_Z (SIG_X has been consumed) */
  linx_free_buf(linx, &sig);
  /* Receive the the first signal waiting in the receive queue */
  linx_receive(linx, &sig, sigsel_any);
  /* Do work, sig->sig_no is SIG_Y */
  linx_free_buf(linx, &sig);
  linx_close (linx);
}
```

See Also

Example   92

linx(7) , linx_hunt(3) , linx_send(3) , linx_sender(3) , linx_free_buf(3) , linx_alloc(3) , linx_get_descriptor(3) , poll(2) , select(2)

Author

Enea LINX team

linx(7) , linx_hunt(3) , linx_send(3) , linx_sender(3) , linx_free_buf(3) , linx_alloc(3) , linx_get_descriptor(3) , poll(2) , select(2)

See Also   93

Enea LINX team

LINX_REQUEST_TMO(3) manual page

Name

linx_request_tmo() - request timeout with specified signal
linx_cancel_tmo() - cancel the specified timeout.
linx_modify_tmo() - modify the specified timeout

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSTMOREF linx_request_tmo(LINX *linx**, LINX_OSTIME** tmo**, union LINX_SIGNAL ****sig**);**

int linx_cancel_tmo(LINX *linx**, LINX_OSTMOREF ***tmoref**);**

int linx_modify_tmo(LINX *linx**, LINX_OSTMOREF ***tmoref**, LINX_OSTIME** tmo**);**

Description

linx_request_tmo() is used to request a signal, *sig*, to be sent to the requesting LINX endpoint *linx* when a timeout has expired. Information necessary for cancelling the time-out is returned. The actual time-out time, *tmo*, is rounded upward to the next larger tick because a time-out can only trigger when a system clock tick is received. Hence, the routine guarantees at least the number of milliseconds requested, but may add a few more depending on the tick resolution. When several time-out signals are generated at the same clock tick, the order is unspecified. If *sig* is NULL, the default LINX timeout signal with signal number LINX_OS_TMO_SIG is received instead. The linx_request_tmo () call consumes the signal, and sets the *sig* pointer to LINX_NIL. The signal is also consumed if an error occurs.

linx_cancel_tmo() is used to cancel a timeout. The time-out is identified by the *tmoref* parameter, which was set by **linx_request_tmo**()**.** It's an error to cancel a timeout more than once or to cancel a timeout after the timeout signal has been received. It is not an error to cancel the timeout if the timeout signal is waiting in the receive queue of the LINX endpoint. To prevent multiple cancellations, **linx_cancel_tmo**() sets the tmoref pointer to LINX_ILLEGAL_TMOREF.

linx_modify_tmo() is used to modify a timeout. The time-out is identified by the tmoref parameter, which was set by **linx_request_tmo**()**.** It's an error to modify a timeout after the timeout signal has been received. It is not an error to modify the timeout if the timeout signal is waiting in the receive queue of the LINX endpoint.

linx is the handle of the LINX endpoint.

spid is the timeout time in milliseconds.

sig is either NULL or a user defined LINX signal.

tmoref is the reference to the timeout obtained from **linx_request_tmo**().

Return Value

linx_request_tmo() returns an timeout reference (LINX_OSTMOREF) when successful. This timeout reference can be used to cancel the timeout. **linx_cancel_tmo**() returns 0 on success.

On failure, **linx_request_tmo**() returns LINX_ILLEGAL_TMOREF while **linx_cancel_tmo**() and **linx_modify_tmo**() both returns -1. In all cases *errno* will be set appropriately.

Errors

EBADF, **ENOTSOCK** The *linx* handle refers to an invalid socket descriptor.

ENOMEM Not enough memory.

EINVAL Invalid argument.

Bugs/Limitations

None.

Example

This example shows how to request and wait for a timeout.

```
#include <linx.h>
int
main (int argc, char *argv[])
{
  LINX *linx;
  union LINX_SIGNAL *sig;
  const LINX_SIGSELECT sel_tmo_sig [] = { 1, LINX_OS_TMO_SIG };
  /* Create a LINX endpoint */
  linx = linx_open("tmo-test", NULL, 0);
  /* Request a one second timeout. When the timeout expires the
     default signal will be returned. */
  linx_request_tmo(linx, 1000, NULL);
  /* Wait for the timeout signal. */
  linx_receive(linx, &sig, sel_tmo_sig);
  /* Free the timeout signal */
  linx_free_buf(linx, &sig);
  /* Close the LINX endpoint */
  linx_close(linx);
  return 0;
}
```

See Also

linx(7) , **linx_close**(3) , **linx_open**(3) **linx_receive**(3) ,

Author

Enea LINX team

LINX_SEND(3) manual page

Name

linx_send() - Send a LINX signal to a LINX endpoint (spid)

linx_send_w_s() - Send a LINX signal and specify a different LINX endpoint as sender

linx_send_w_opt() - Send a signal to a LINX endpoint (spid) with alternative

methods

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_send(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *to***);**

int linx_send_w_s(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *from***, LINX_SPID** *to***);**

int linx_send_w_opt(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *from***, LINX_SPID** *to***,**
int32_t **taglist***);**

Description

Send a signal to the LINX endpoint identified by the *to* binary identifier (spid). **linx_send**(3) always consumes the signal buffer, even if the call fails, by freeing the memory and setting *sig to LINX_NIL to prevent further access.

linx is the handle to the LINX endpoint, via which the signal is transmitted.

sig is the signal to send. The signal buffer must be owned by the transmitting LINX endpoint. It could have been allocated with **linx_alloc**(3) or received via the same LINX endpoint. If *sig* is corrupt, **abort**(3) is called.

to is the identifier (spid) of the recipient LINX endpoint, usually found as a result of a **linx_hunt**(3) . If the *to* spid refers to an endpoint that has been closed (i.e. in state LINX_ZOMBIE), -1 is returned and *errno* is set to ECONNRESET. If *to* does not apply to the specific method used in **linx_send_w_opt**(3) , the LINX_ILLEGAL_SPID define shall be used as its value.

from is the identifier (spid) of a different sender endpoint (instead of the transmitting endpoint). The recipient will see this spid as the sender of the signal. If the *from* spid refers to an endpoint that has been closed (i.e. in state LINX_ZOMBIE), -1 is returned and *errno* is set to ECONNRESET. If *from* does not apply to the specific method used in **linx_send_w_opt**(3) , the LINX_ILLEGAL_SPID define shall be used as its value.

taglist is an array of tags and values used with the call **linx_send_w_opt**(3) . Each tag has an associated value that directly follows it. See the example for details. If the *taglist* contains an invalid parameter, -1 is returned and *errno* is set to EINVAL. The *taglist* parameter list can contain the folllowing values.

LINX_SIG_OPT_OOB

The **LINX_SIG_OPT_OOB** tag is used to enable the OOB send method. The value of the **LINX_SIG_OPT_OOB** tag shall be 1.

An OOB signal is a signal that is transmitted, transported, forwarded and received ahead of in band signals. When the OOB signal is placed in the signal queue of the receiving process, it is placed ahead of in band signals but after OOB signals present in the signal queue.

If the OOB signal is sent inter node and the operating system on the receiving node is not supporting OOB, the signal is delivered as an in band signal.

When sending an OOB signal the *sig* parameter is the signal to be sent OOB. The signal buffer must be owned by the transmitting LINX endpoint. It could have been allocated with **linx_alloc**(3) or received.

When an OOB signal is sent using **linx_send_w_opt**(3) , a signal attribute is set on the signal, **LINX_SIG_ATTR_OOB**. The **linx_sigattr**(3) is used to test if the attribute is set or not.

LINX_SIG_OPT_END
Marks the end of the taglist and has no value. Must be at the end of the *taglist*.

Return Value

Returns 0 if successful. On failure, -1 is returned and *errno* will be set. In any case, *sig* is set to LINX_NIL.

Errors

EBADF, **ENOTSOCK** The LINX endpoint is associated with an invalid socket descriptor.

ENOBUFS, **ENOMEM** Insufficient memory is available.

ECONNRESET The destination spid refers to a LINX endpoint that has been closed (i.e. in state LINX_ZOMBIE). This situation is best avoided by using *linx_attach*(3) to supervise the receiving LINX endpoint. Closed spid are not reused until forced by spid instance counter overflow.

EPIPE This error is reported at an attempt to send to the spid of a LINX endpoint that is being closed as the call occurs.

EINVAL This error is reported if an unrecognized taglist parameter was passed in the *taglist*.

Bugs/Limitations

None.

Example

```
union LINX_SIGNAL *sig;
 LINX *linx;
 LINX_SPID to_spid;
 LINX_SPID from_spid;
 int32_t taglist[3];
 ...
 taglist[0]=LINX_SIG_OPT_OOB;
 taglist[1]=1;
```

```
taglist[2]=LINX_SIG_OPT_END;
sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), 100);
(void)linx_send_w_opt(linx, &sig, to_spid, from_spid, taglist);
...
```

Notes

It is illegal to allocate/receive a signal buffer on one LINX endpoint and then send it using **linx_send**(3) from another LINX endpoint. In that case a new signal buffer needs to be allocated and the buffer contents moved to the new signal buffers prior to sending it.

```
union LINX_SIGNAL *sig1, *sig2;
 LINX_OSBUFSIZE size;
 LINX_SIGSELECT sel[] = {0};
 LINX *linx1, *linx2;
 LINX_SPID to_spid;
 ...
 sig1 = linx_receive(linx1, &sig1, sel);
 size = linx_sigsize(linx1, sig1);
 sig2 = linx_alloc(linx2, size, sel);
 memcpy((void)sig2, (void)sig1, size);
 (void)linx_free_buf(linx1, &sig1);
 (void)linx_send(linx2, &sig2, to_spid);
 ...
```

See Also

linx(7) , **linx_alloc**(3) , **linx_hunt**(3) , **linx_receive**(3) , **linx_receive_w_tmo**(3) , **linx_receive_from**(3) , **linx_sigattr**(3)

Author

Enea LINX team

Example   98

LINX_SENDER(3) manual page

Name

linx_sender() - Retrieve the sender of a LINX signal

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_SPID linx_sender(LINX **linx***, union LINX_SIGNAL *****sig***);**

Description

Get the binary identifier (spid) of the LINX endpoint which sent the signal *sig*.

linx is the handle of the LINX endpoint on which *sig* was received.

If *sig* is corrupt, **abort**(3) is called.

Return Value

The identifier (spid) of the LINX endpoint, which sent the signal *sig*. If the signal was never sent, **linx_sender**() will return the spid of the own LINX endpoint.

Bugs/Limitations

None.

See Also

linx(7) , **linx_alloc**(3) , **linx_receive**(3) , **linx_receive_from**(3) , **linx_receive_w_tmo**(3) ,

Author

Enea LINX team

LINX_SEND_W_OPT(3) manual page

Name

linx_send() - Send a LINX signal to a LINX endpoint (spid)

linx_send_w_s() - Send a LINX signal and specify a different LINX endpoint as sender

linx_send_w_opt() - Send a signal to a LINX endpoint (spid) with alternative

methods

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_send(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *to***);**

int linx_send_w_s(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *from***, LINX_SPID** *to***);**

int linx_send_w_opt(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_SPID** *from***, LINX_SPID** *to***,**
int32_t **taglist***);**

Description

Send a signal to the LINX endpoint identified by the *to* binary identifier (spid). **linx_send**(3) always consumes the signal buffer, even if the call fails, by freeing the memory and setting *sig to LINX_NIL to prevent further access.

linx is the handle to the LINX endpoint, via which the signal is transmitted.

sig is the signal to send. The signal buffer must be owned by the transmitting LINX endpoint. It could have been allocated with **linx_alloc**(3) or received via the same LINX endpoint. If *sig* is corrupt, **abort**(3) is called.

to is the identifier (spid) of the recipient LINX endpoint, usually found as a result of a **linx_hunt**(3) . If the *to* spid refers to an endpoint that has been closed (i.e. in state LINX_ZOMBIE), -1 is returned and *errno* is set to ECONNRESET. If *to* does not apply to the specific method used in **linx_send_w_opt**(3) , the LINX_ILLEGAL_SPID define shall be used as its value.

from is the identifier (spid) of a different sender endpoint (instead of the transmitting endpoint). The recipient will see this spid as the sender of the signal. If the *from* spid refers to an endpoint that has been closed (i.e. in state LINX_ZOMBIE), -1 is returned and *errno* is set to ECONNRESET. If *from* does not apply to the specific method used in **linx_send_w_opt**(3) , the LINX_ILLEGAL_SPID define shall be used as its value.

taglist is an array of tags and values used with the call **linx_send_w_opt**(3) . Each tag has an associated value that directly follows it. See the example for details. If the *taglist* contains an invalid parameter, -1 is returned and *errno* is set to EINVAL. The *taglist* parameter list can contain the folllowing values.

LINX_SIG_OPT_OOB

The **LINX_SIG_OPT_OOB** tag is used to enable the OOB send method. The value of the **LINX_SIG_OPT_OOB** tag shall be 1.

An OOB signal is a signal that is transmitted, transported, forwarded and received ahead of in band signals. When the OOB signal is placed in the signal queue of the receiving process, it is placed ahead of in band signals but after OOB signals present in the signal queue.

If the OOB signal is sent inter node and the operating system on the receiving node is not supporting OOB, the signal is delivered as an in band signal.

When sending an OOB signal the *sig* parameter is the signal to be sent OOB. The signal buffer must be owned by the transmitting LINX endpoint. It could have been allocated with **linx_alloc**(3) or received.

When an OOB signal is sent using **linx_send_w_opt**(3) , a signal attribute is set on the signal, **LINX_SIG_ATTR_OOB**. The **linx_sigattr**(3) is used to test if the attribute is set or not.

LINX_SIG_OPT_END
Marks the end of the taglist and has no value. Must be at the end of the *taglist*.

Return Value

Returns 0 if successful. On failure, -1 is returned and *errno* will be set. In any case, *sig* is set to LINX_NIL.

Errors

EBADF, **ENOTSOCK** The LINX endpoint is associated with an invalid socket descriptor.

ENOBUFS, **ENOMEM** Insufficient memory is available.

ECONNRESET The destination spid refers to a LINX endpoint that has been closed (i.e. in state LINX_ZOMBIE). This situation is best avoided by using *linx_attach*(3) to supervise the receiving LINX endpoint. Closed spid are not reused until forced by spid instance counter overflow.

EPIPE This error is reported at an attempt to send to the spid of a LINX endpoint that is being closed as the call occurs.

EINVAL This error is reported if an unrecognized taglist parameter was passed in the *taglist*.

Bugs/Limitations

None.

Example

```
union LINX_SIGNAL *sig;
 LINX *linx;
 LINX_SPID to_spid;
 LINX_SPID from_spid;
 int32_t taglist[3];
 ...
 taglist[0]=LINX_SIG_OPT_OOB;
 taglist[1]=1;
```

```
taglist[2]=LINX_SIG_OPT_END;
sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), 100);
(void)linx_send_w_opt(linx, &sig, to_spid, from_spid, taglist);
...
```

Notes

It is illegal to allocate/receive a signal buffer on one LINX endpoint and then send it using **linx_send**(3) from another LINX endpoint. In that case a new signal buffer needs to be allocated and the buffer contents moved to the new signal buffers prior to sending it.

```
union LINX_SIGNAL *sig1, *sig2;
LINX_OSBUFSIZE size;
LINX_SIGSELECT sel[] = {0};
LINX *linx1, *linx2;
LINX_SPID to_spid;
...
sig1 = linx_receive(linx1, &sig1, sel);
size = linx_sigsize(linx1, sig1);
sig2 = linx_alloc(linx2, size, sel);
memcpy((void)sig2, (void)sig1, size);
(void)linx_free_buf(linx1, &sig1);
(void)linx_send(linx2, &sig2, to_spid);
...
```

See Also

linx(7) , **linx_alloc**(3) , **linx_hunt**(3) , **linx_receive**(3) , **linx_receive_w_tmo**(3) , **linx_receive_from**(3) , **linx_sigattr**(3)

Author

Enea LINX team

Example   102

LINX_SEND_W_S(3) manual page

Name

linx_send() - Send a LINX signal to a LINX endpoint (spid)

linx_send_w_s() - Send a LINX signal and specify a different LINX endpoint as sender

linx_send_w_opt() - Send a signal to a LINX endpoint (spid) with alternative

methods

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_send(LINX *linx**, union LINX_SIGNAL ****sig**, LINX_SPID** to**);**

int linx_send_w_s(LINX *linx**, union LINX_SIGNAL ****sig**, LINX_SPID** from**, LINX_SPID** to**);**

int linx_send_w_opt(LINX *linx**, union LINX_SIGNAL ****sig**, LINX_SPID** from**, LINX_SPID** to**,**
int32_t *taglist**);**

Description

Send a signal to the LINX endpoint identified by the *to* binary identifier (spid). **linx_send**(3) always consumes the signal buffer, even if the call fails, by freeing the memory and setting *sig to LINX_NIL to prevent further access.

linx is the handle to the LINX endpoint, via which the signal is transmitted.

sig is the signal to send. The signal buffer must be owned by the transmitting LINX endpoint. It could have been allocated with **linx_alloc**(3) or received via the same LINX endpoint. If *sig* is corrupt, **abort**(3) is called.

to is the identifier (spid) of the recipient LINX endpoint, usually found as a result of a **linx_hunt**(3) . If the *to* spid refers to an endpoint that has been closed (i.e. in state LINX_ZOMBIE), -1 is returned and *errno* is set to ECONNRESET. If *to* does not apply to the specific method used in **linx_send_w_opt**(3) , the LINX_ILLEGAL_SPID define shall be used as its value.

from is the identifier (spid) of a different sender endpoint (instead of the transmitting endpoint). The recipient will see this spid as the sender of the signal. If the *from* spid refers to an endpoint that has been closed (i.e. in state LINX_ZOMBIE), -1 is returned and *errno* is set to ECONNRESET. If *from* does not apply to the specific method used in **linx_send_w_opt**(3) , the LINX_ILLEGAL_SPID define shall be used as its value.

taglist is an array of tags and values used with the call **linx_send_w_opt**(3) . Each tag has an associated value that directly follows it. See the example for details. If the *taglist* contains an invalid parameter, -1 is returned and *errno* is set to EINVAL. The *taglist* parameter list can contain the folllowing values.

LINX_SIG_OPT_OOB

ENEA

The **LINX_SIG_OPT_OOB** tag is used to enable the OOB send method. The value of the **LINX_SIG_OPT_OOB** tag shall be 1.

An OOB signal is a signal that is transmitted, transported, forwarded and received ahead of in band signals. When the OOB signal is placed in the signal queue of the receiving process, it is placed ahead of in band signals but after OOB signals present in the signal queue.

If the OOB signal is sent inter node and the operating system on the receiving node is not supporting OOB, the signal is delivered as an in band signal.

When sending an OOB signal the *sig* parameter is the signal to be sent OOB. The signal buffer must be owned by the transmitting LINX endpoint. It could have been allocated with **linx_alloc**(3) or received.

When an OOB signal is sent using **linx_send_w_opt**(3) , a signal attribute is set on the signal, **LINX_SIG_ATTR_OOB**. The **linx_sigattr**(3) is used to test if the attribute is set or not.

LINX_SIG_OPT_END
Marks the end of the taglist and has no value. Must be at the end of the *taglist*.

Return Value

Returns 0 if successful. On failure, -1 is returned and *errno* will be set. In any case, *sig* is set to LINX_NIL.

Errors

EBADF, **ENOTSOCK** The LINX endpoint is associated with an invalid socket descriptor.

ENOBUFS, **ENOMEM** Insufficient memory is available.

ECONNRESET The destination spid refers to a LINX endpoint that has been closed (i.e. in state LINX_ZOMBIE). This situation is best avoided by using *linx_attach*(3) to supervise the receiving LINX endpoint. Closed spid are not reused until forced by spid instance counter overflow.

EPIPE This error is reported at an attempt to send to the spid of a LINX endpoint that is being closed as the call occurs.

EINVAL This error is reported if an unrecognized taglist parameter was passed in the *taglist*.

Bugs/Limitations

None.

Example

```
    union LINX_SIGNAL *sig;
     LINX *linx;
     LINX_SPID to_spid;
     LINX_SPID from_spid;
     int32_t taglist[3];
     ...
     taglist[0]=LINX_SIG_OPT_OOB;
     taglist[1]=1;
```

```
taglist[2]=LINX_SIG_OPT_END;
sig = linx_alloc(linx, sizeof(LINX_SIGSELECT), 100);
(void)linx_send_w_opt(linx, &sig, to_spid, from_spid, taglist);
...
```

Notes

It is illegal to allocate/receive a signal buffer on one LINX endpoint and then send it using **linx_send**(3) from another LINX endpoint. In that case a new signal buffer needs to be allocated and the buffer contents moved to the new signal buffers prior to sending it.

```
union LINX_SIGNAL *sig1, *sig2;
LINX_OSBUFSIZE size;
LINX_SIGSELECT sel[] = {0};
LINX *linx1, *linx2;
LINX_SPID to_spid;
...
sig1 = linx_receive(linx1, &sig1, sel);
size = linx_sigsize(linx1, sig1);
sig2 = linx_alloc(linx2, size, sel);
memcpy((void)sig2, (void)sig1, size);
(void)linx_free_buf(linx1, &sig1);
(void)linx_send(linx2, &sig2, to_spid);
...
```

See Also

linx(7) , **linx_alloc**(3) , **linx_hunt**(3) , **linx_receive**(3) , **linx_receive_w_tmo**(3) , **linx_receive_from**(3) , **linx_sigattr**(3)

Author

Enea LINX team

Example   105

LINX_SET_SIGSIZE(3) manual page

Name

linx_sigsize() - Get the size of a LINX signal buffer in bytes
linx_set_sigsize() - Set the size of a LINX signal buffer in bytes

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSBUFSIZE linx_sigsize(LINX **linx***, union LINX_SIGNAL *****sig***);**

int linx_set_sigsize(LINX **linx***, union LINX_SIGNAL *****sig***, LINX_OSBUFSIZE *sigsize*);**

Description

linx_sigsize() gets the size of the signal buffer *sig*. The signal buffer must be owned by the LINX endpoint, referred to by *linx* but can either be a received buffer or a buffer, allocated by the **linx_alloc**(3) call on this LINX endpoint. If *sig* is corrupt, **abort**(3) is called.

linx_set_sigsize() resizes the signal buffer *sig* to *sigsize* bytes and returns 0 on success. An error is returned if the size is outside permitted limits. If the new *size* of the signal buffer *sig* is smaller than the original size, the data stored in signal buffer *sig* is truncated.

linx is the handle to the LINX endpoint, which owns the *sig* signal buffer.

sigsize is the new size in bytes of the signal buffer.

Return Value

linx_sigsize returns the size in bytes of the signal buffer. *linx_set_sigsize* returns zero on success. If an error occurred, -1 is returned and *errno* is set.

Errors

EMSGSIZE *sigsize* is less than 4.

ENOMEM Insufficient memory is available.

Bugs/Limitations

None.

See Also

linx(7) , **linx_alloc**(3) , **linx_receive**(3)

Author

Enea LINX team

LINX_SIGATTR(3) manual page

Name

linx_sigattr() - Get an attribute of a LINX signal

Synopsis

#include <linx_types.h>
#include <linx.h>

int linx_sigattr(const LINX * *linx* **, const union LINX_SIGNAL ** *** *sig* **, uint32_t** *attr* **, void ** *** *value* **);**

Description

The **linx_sigattr**(3) API function is used to determine what attributes a received signal sig has, due to a **linx_send_w_opt**(3) LINX API call. The *value* pointer will be set to an attribute specific value. The *linx* parameter is the LINX handle used when receiving the signal.

linx is the handle to the LINX endpoint, which owns the *sig* signal buffer.

attr is the attribute of the *sig* signal buffer.

value is the returned value that corresponds to *attr* which in turn is the name of the attribute set on the *sig* signal buffer. *attr* can have the following values.

LINX_SIG_ATTR_OOB

A signal has the **LINX_SIG_OPT_OOB** attribute with value set to *1* if it was sent using the alternative **linx_send_w_opt**(3) with **LINX_SIG_OPT_OOB**. If the attribute is not set, the returned value is *~0*.

The OOB attribute has no affect on further communication using the signal, it is kept to make it possible for the receiving LINX endpoint to know if a signal is OOB or not. If the signal is sent to another LINX endpoint and the OOB attribute needs to be preserved, the **linx_send_w_opt**(3) call needs to be used for that signaling. If the received OOB signal is sent with **linx_send**(3) or **linx_send_w_s**(3) the OOB attribute is lost and the signal is transmitted as a in band signal.

Return Value

Returns 0 if successful. On failure, -1 is returned and *errno* will be set.

Errors

EBADF, **ENOTSOCK** The LINX endpoint is associated with an invalid socket descriptor.

Bugs/Limitations

None.

See Also

linx(7) , linx_send_w_opt(3)

Author

Enea LINX team

LINX_SIGSIZE(3) manual page

Name

linx_sigsize() - Get the size of a LINX signal buffer in bytes
linx_set_sigsize() - Set the size of a LINX signal buffer in bytes

Synopsis

#include <linx_types.h>
#include <linx.h>

LINX_OSBUFSIZE linx_sigsize(LINX *linx**, union LINX_SIGNAL ****sig**);**

int linx_set_sigsize(LINX *linx**, union LINX_SIGNAL ****sig**, LINX_OSBUFSIZE** sigsize**);**

Description

linx_sigsize() gets the size of the signal buffer *sig*. The signal buffer must be owned by the LINX endpoint, referred to by *linx* but can either be a received buffer or a buffer, allocated by the **linx_alloc**(3) call on this LINX endpoint. If *sig* is corrupt, **abort**(3) is called.

linx_set_sigsize() resizes the signal buffer *sig* to *sigsize* bytes and returns 0 on success. An error is returned if the size is outside permitted limits. If the new *size* of the signal buffer *sig* is smaller than the original size, the data stored in signal buffer *sig* is truncated.

linx is the handle to the LINX endpoint, which owns the *sig* signal buffer.

sigsize is the new size in bytes of the signal buffer.

Return Value

linx_sigsize returns the size in bytes of the signal buffer. *linx_set_sigsize* returns zero on success. If an error occurred, -1 is returned and *errno* is set.

Errors

EMSGSIZE *sigsize* is less than 4.

ENOMEM Insufficient memory is available.

Bugs/Limitations

None.

See Also

linx(7) , **linx_alloc**(3) , **linx_receive**(3)

Author

Enea LINX team

Author

LINXSTAT(1) manual page

Name

linxstat - display specific information about LINX

Synopsis

linxstat [-a] [-b] [-f] [-h] [-H] [-q] [-s] [-S] [-t *type* **] [-T]**

Description

linxstat is a utility to display the status of the LINX communication system, including information about all local and known remote LINX endpoints and LINX links to remote nodes. The names, types, status and local identifiers are displayed as well as queues, attaches and hunts. The information is fetched from the LINX kernel module. The following columns can be output, in addition to a leading summary (compare the example):

PID is the id of the process owning the LINX endpoint or LINX link. As each remote endpoint are represented by a local LINX endpoint with a local identifier (spid), they are all owned by one common local process.

spid is the local LINX binary identifier for the LINX endpoint or LINX link.

name is the huntname of the LINX endpoint or LINX link. If a name is too long for the column, a "+" sign is displayed and the tailing part of the name.

type is the type of entry, i.e. *local* for a local LINX endpoint, *link* for a LINX link to another node, *remote* for a local representation of a remote LINX endpoint, *zombie* for a LINX endpoint which has been closed, *illegal* if an endpoint structure is illegal. *unknown* if linxstat cannot reach the LINX endpoint.

state is the state of the endpoint or link. The main states are *running* not waiting for anything, *recv* while waiting to receive signal buffers, *poll* while waiting for events with the *poll*(2) or *select*(2) system call, *unknown* if linxstat cannot get the state.

queue is the receive queue. This is the number of signal buffers delivered to the LINX endpoint, but not yet received.

attach(from/to) are the pending attach information. Both the number of attaches to each LINX endpoint from other endpoints and the number of attaches from each endpoint to other endpoints are displayed.

hunt are pending hunts owned by each LINX endpoint. It also includes pending hunts, sent by others but with this endpoint as sender.

tmo are pending timeouts for each LINX endpoint.

Options

-a
> Display more verbose pending attach information.

-b

Batch mode, only display the number of open LINX endpoints.

-f

Print receive filters.

-h

Display a short help message.

-H

Display more verbose pending hunt information.

-q

Display more verbose receive queue information.

-s

Display less information (simple). Only endpoint-specific information are displayed, no summary, no title and no counter columns.

-S

If the LINX kernel module has been compiled with per-socket statistics (SOCK_STAT) then linxstat can display statistics for each socket.

-t type

Display only sockets of the specified type.

-T

Print pending timeouts.

Files

None.

Diagnostics

linxstat writes all information to standard output.

Known Bugs

linxstat queries are not atomic, the state of the system may change while linxstat is running.

Example

This is a displayed list from linxstat.

```
 local:5 remote:1 link:1
 pending attach:2
 pending hunt:0
 pending timeout:0
 queued signals:0
  PID spid     name               type    state  queue attach(f/t) hunt tmo
24903 0x10003 linx16              link    -      0     1/0         0    0
24903 0x10004 +16/apitest_server  remote  -      0     0/0         0    0
24909 0x10001 apitest_client      local   run    0     0/1         0    0
24909 0x10024 +est_attach_parent  local   poll   0     1/1         0    0
24909 0x10005 not bound           local   run    0     0/0         0    0
24958 0x10023 +test_attach_child  local   recv   0     0/0         0    0
24963 0x10025 linxstat            local   run    0     0/0         0    0
```

With -a, more information is displayed after each line with pending attaches. Extracts from a table, created by linxstat -a:

```
local:4 remote:1 link:1
pending attach:1
pending hunt:0
pending timeout:0
queued signals:0
PID   spid    name             type   state  queue attach(f/t) hunt tmo
25053 0x10002 linx16           link   -      0     1/0          0    0
      Attach info (from):
         [attref:0x00000400   from:0x00010001 signo:0       size: 0]
25053 0x10003 +16/apitest_server remote -    0     0/0          0    0
25059 0x10001 apitest_client   local run    0     0/1          0    0
      Attach info (to):
         [attref:0x00000400   to:0x00010001 signo:0       size:  0]
25059 0x10004 not bound        local run    0     0/0          0    0
25059 0x10014 +inx_receive_w_tmo local poll 0     0/0          0    0
25083 0x10015 linxstat         local run    0     0/0          0    0
```

See Also

linx(7) , **linx_attach**(3) , **linx_hunt**(3) , **linx_open**(3) , **linxdisc**(8) , **linxdisc.conf**(5) , **linxcfg**(1)

Author

Enea LINX team

Example   114

LINX_TYPES.H(3) manual page

Name

linx_types.h - data types for LINX

Synopsis

#include <linx_types.h>

Description

In addition to the definitions below, the **linx_types.h** includes the system headers **<stdint.h>**, **<stdlib.h>** and **<linux/types.h>**.

This **linx_types.h** header may be included automatically by the **linx.h** header file.

The header file contains the following definitions of LINX types and constants:

LINX
> The LINX handle, returned from **linx_open**(3) . It is used to access the LINX endpoint with all other LINX API fuctions.

LINX_FALSE
> defined to 0.

LINX_TRUE
> defined to 1.

LINX_ILLEGAL_ATTREF
> defined to ((LINX_OSATTREF)0). It can be returned instead of a valid LINX_OSATTREF value, e.g. from **linx_attach**(3) .

LINX_ILLEGAL_SPID
> defined to ((LINX_SPID)0). It can be returned instead of a valid LINX_SPID value, e.g. from **linx_get_sped**(3) .

LINX_NIL
> defined to ((union LINX_SIGNAL *) 0). This null pointer value indicates that a signal pointer no longer points to any buffer, e.g. after freeing the signal buffer.

LINX_OS_ATTACH_SIG
> defined to ((LINX_SIGSELECT)252). This is the reserved signal number of the default attach signal used if the application does not provide a signal with the **linx_attach**(3) call.

LINX_OS_HUNT_SIG
> defined to ((LINX_SIGSELECT)251). This is the reserved signal number of the default hunt signal used if the application does not provide a signal with the **linx_hunt**(3) call.

LINX_OSATTREF
> A uint32_t used for attach references.

LINX_OSBOOLEAN
> An int used for boolean operations.

LINX_OSBUFSIZE
> An ssize_t used for signal buffer sizes.

LINX_OSTIME
> A uint32_t used for time in micro-seconds (us).

LINX_SIGSELECT
> A uint32_t used for signal numbers.

LINX_SPID

A uint32_t used for LINX endpoint binary identifiers (also called spid).

LINX_SIG_OPT_END

A define used as endmark in the *taglist* parameter list, passed in the **linx_send_w_opt**(3) call.

LINX_SIG_OPT_OOB

A define used for OOB signaling in the *taglist* parameter list, passed in the **linx_send_w_opt**(3) call.

LINX_SIG_ATTR_OOB

Is an attribute of a received signal that was sent OOB. Defined to 1.

The following LINX definition is not explicitly done in this header file, but is used in the LINX API.

union LINX_SIGNAL

is an opaque structure, containing any signal buffer of any size, with a 32-bit signal number of type **LINX_SIGSELECT** in the first four bytes. This signal number can be accessed as the *sig_no* struct member of the signal buffer. The application allocates various sized signal buffers with **linx_alloc**(3) , which returns a **union LINX_SIGNAL ***.

See Also

linx(7) , **linx_alloc**(3) , **linx_attach**(3) , **linx_get_spid**(3) , **linx_hunt**(3) , **linx_open**(3)

Author

Enea LINX team

MKETHCON(1) manual page

Name

mkethcon - create LINX Ethernet connections.

Synopsis

mkethcon [OPTIONS] <connection>

Description

Create a LINX Ethernet connection to a remote node.

Options

-h, --help
>Display help and exit.

Mandatory options

-m, --mac=MAC address
>Specify the remote MAC address of the node to connect to, e.g. 0a:1b:2c:3d:4d:5e.

-i, --if=interface
>Specify the Ethernet interface to use, e.g. eth0.

Optional options

--window_size=nnn
>This is the send/receive window_size, in number of packets on the link, and may need to be modified to adapt LINX to really slow or really fast Ethernet performance.
>Both the local and remote side negotiate the window size, being the smallest of the two sizes the selected one. The default 128 messages should be sufficient in most configurations. The window size shall always be of a power of 2. Size 0 means to use the default window size.

--defer_queue_size=nnn
>The defer queue size is in packages, with the size depending on the LINX link. The defer queue is used on the sender side when the send queue is full. Every message sent, when the send queue is full, is stored in the defer queue, until the send queue has room for more messages. The defer queue size is the maximum number of packages, with messages, that can be stored in the defer queue before the link is disconnected, due to lack or resources. The default value 2048 packages should be sufficient in most systems. Size 0 means to use the default defer queue size.

--send_tmo=nnn
>The send acknowledge timeout specifies the time (in msec) to wait until unacknowledged messages are resent. The default 10 should be sufficient in most systems. Tmo 0 means to use the default timeout.

--nack_tmo=nnn
>The retransmission timeout specifies the time (in msec) to wait until nack messages are resent. The default 20 should be sufficient in most systems. Tmo 0 means to use the default timeout.

--conn_tmo=nnn
>The connect timeout specifies the time (in msec) to wait until an attempt to establish a connection is considered failed. The default 200 should be sufficient in most systems. Tmo 0 means to use the default timeout.

--live_tmo=nnn
>The connection supervision timeout specifies the time (in msec) to wait until a connection is considered broken. Default is 100 (when the system is not idle) and should be sufficient in most

systems. When the system is idle, the live timeout is ten times larger than the configured live timeout value. Tmo 0 means to use the default timeout.

--mtu=nnn

The MTU (Maximum Transmission Unit) specifies the size in bytes of the largets packet that the Ethernet protocol can pass onwards (this excludes the size of the Ethernet header). If not set the MTU is fetched from the interface. Typcially a MTU of 1500 bytes is used for Ethernet.

Files

None.

Diagnostics

None.

Known Bugs

None.

Examples

```
mkethcon --mac=01:23:a4:4f:b3:ac --if=eth0 ethcon_A
mkethcon -m 01:23:a4:4f:b3:ac -i eth0 ethcon_A
```

See Also

linx(7) , **linxcfg**(1) , **linxstat**(1) , **linxdisc**(8) , **mklink**(1) , **rmethcon**(1)

Author

Enea LINX team

MKLINK(1) manual page

Name

mklink - create LINX links.

Synopsis

mklink [OPTIONS] <link>

Description

Create a LINX link to a remote node.

Options

-h, --help
>   Display help and exit.

-a, --attributes=<attributes>
>   Specify an optional cookie that is returned unmodified in the new_link signal. The new_link signal is
>   sent to link supervisor(s) when the link is available.

-c, --connection=<connection>
>   Specify the connection(s) that this link should use. This option can be repeated if a link should use
>   more than one connection. Note that the connection(s) must be created before they can be assigned to
>   a link. The connection name is made up of two parts, see example. The first part identifies the CM
>   and the second part is the name that was used in the CM specific create command.

Files

None.

Diagnostics

None.

Known Bugs

None.

Examples

```
mkethcon --mac=01:23:a4:4f:b3:ac --if=eth0 ethcon_A
mklink --connection=ethcm/ethcon_A link_A
mkethcon --mac=01:23:a4:4f:b3:ac --if=eth0 ethcon_A
mktcpcon tcpcon_A
mklink -c ethcm/ethcon_A -c tcpcm/tcpcon_A link_A
```

See Also

linx(7) , linxcfg(1) , mkethcon(1) , mktcpcon(1) , rmlink(1)

Author

Enea LINX team

MKTCPCON(1) manual page

Name

mktcpcon - create LINX TCP/IP connections.

Synopsis

mktcpcon [OPTIONS] <connection>

Description

Create a LINX TCP/IP connection to a remote node.

Options

-h, --help
> Display help and exit.

-i, --ipaddr=<IP address>
> The IP address to connect to, e.g. 192.168.0.12.

-t, --live_tmo=<size>
> The live_tmo parameter is the time in milliseconds between every heartbeat that is used to detect if the connection has gone down. The default value is 1000 ms.

-n, --use_nagle=<bool>
> Set to 1 if nagle algorithm shall be used on the socket for the connection. Default is off.

-s, --silent
> Silent mode.

Files

None.

Diagnostics

None.

Known Bugs

None.

Examples

```
mktcpcon --ipaddr=192.168.0.12 tcpcon_A
```

See Also

linx(7) , **linxcfg**(1) , **mklink**(1) , **rmtcpcon**(1)

Author

Enea LINX

team

RBLOG(3) manual page

Name

rblog - write to ram ringbuffer log

Synopsis

#include <rblog.h>

int rblog(const char *<i>fmt</i>, ...);

Description

Write a new message into the ring buffer. Older messages successively gets overwritten when the writing wraps around.

The function parameters *fmt* and ... follows the same format as the parameters to printk, i.e. a format string and optional data parameters.

Notes

A maximum of 1024 bytes including null terminator will be written.

This function is reentrant and can be called in atomic context.

Return Value

rblog() returns the number of characters written to the log after format conversions.

RBLOG_CREATE(3) manual page

Name

rblog_create - initialize ram ringbuffer log

Synopsis

#include <rblog.h>

int rblog_create(size_t *size***);**

Description

Allocate memory (at least *size* bytes; the value is rounded up to fit a whole number of memory pages), register file for reading the log in /proc, and register SysRq keys for printing the log to console. The name of the /proc file used and which keys are use for printing can be selected with #defines in rblog.h. *size* must be at least 1.

Return Value

rblog_create() returns 0 if successful, or a negative error value otherwise.

Errors

EPERM The buffer is already initialized, or *size* is less than 1.

ENOMEM Memory allocation for the buffer failed, or not enough memory to create /proc file entry.

rblog_create() can also return any error value returned by register_sysrq_key().

RBLOG_DESTROY(3) manual page

Name

rblog_destroy - remove ring buffer

Synopsis

#include <rblog.h>

void rblog_destroy(void);

Description

rblog_destroy() unregisters SysRq keys and the /proc file registered by rblog_create(), and frees the memory allocated for the log. After rblog_destroy() is called, rblog_create() can be called again to create a new log buffer.

RBLOG-EXTRACT(1) manual page

Name

rblog-extract - extract ram ringbuffer log from ELF core file

Synopsis

rblog-extract

core-file

Description

rblog-extract searches the memory segments of the given ELF core file for valid rblog structs, and (optionally) saves the log contents to file. The ELF core file should have been produced by Kdump after a kernel panic on a kernel where an rblog is being used (see the section BACKGROUND below).

Notes

rblog-extract can read both 32- and 64-bit ELF files; however, the width of pointers and 'int's must be the same on the crashed machine and the machine running rblog-extract. rblog-extract will detect if this is not the case and exit with an error.

rblog-extract will not run (or compile) on non-Linux platforms; it uses Linux's definition of ELF files and also the Linux 'stat' system call.

Background

If a kernel has crashed it can be very desirable to save the memory image of the crashed environment. This gives the possibility to debug it afterwards to (hopefully) find the cause of the problem. The kexec and kdump tools can be used to load and execute a new kernel from an existing kernel. This second kernel, called a "capture kernel" or "panic kernel", is executed after the first (running) kernel has crashed. In the environment of the second kernel, the memory of the first kernel can be accessed.

This is an example of manual usage. You only need to have the kexec-tools package installed:

1.
        Add "crashkernel=64M@16M" option to your bootloader configuration and reboot.
2.
        Load the panic kernel with, for example:

                kexec -p /boot/vmlinuz-2.6.22.17-0.1-default \

                        --initrd=/boot/initrd-2.6.22.17-0.1-default \

        --append="root=/dev/sda5 1 irqpoll maxcpus=1"

The command line should include the root file system (root=), the irqpoll and reset_devices options and should end with the runlevel you would like to have in the kdump environment (probably 1). Use "man kexec" to see all options.

Now, if we get a crash, the "capture" (or "panic") kernel will be executed, and an ELF core file named "vmcore" that contains the memory segments of the crashed kernel will be saved in /var/log/dump/$date$time for later analysis.

Author

Enea LINX team

RMETHCON(1) manual page

Name

rmethcon - remove LINX Ethernet connections.

Synopsis

rmethcon [OPTIONS] <connection>...

Description

Remove the LINX Ethernet connections. Make sure that the connection is not assigned to any link before it is removed (i.e. remove the link first).

Options

-h, --help
       Display help and exit.

Files

None.

Diagnostics

None.

Known Bugs

None.

Examples

```
rmethcon ethcon_A (or rmethcon ethcm/ethcon_A)
rmethcon ethcon_A ethcon_B
```

See Also

linx(7) , **linxcfg(1)** , **mkethcon(1)** , **rmlink(1)**

Author

Enea LINX

team

ENEA

RMLINK(1) manual page

Name

rmlink - remove LINX links.

Synopsis

rmlink [-ahs] link...

Description

Remove LINX links and optionally any connections that they uses.

Options

-a, --all
>   Remove all, i.e. both the link and its assigned connection(s) are removed. This is the preferred method. If this option isn't used, any connections assigned to the link must be removed with a separate command, e.g. rmethcon, rmtcpcon, etc.

-h, --help
>   Display help and exit.

-s, --silent
>   Silent mode.

Files

None.

Diagnostics

None.

Known Bugs

None.

Examples

```
rmlink -a link_A
rmlink link_A link_B
```

See Also

linx(7) , **linxcfg**(1) , **mklink**(1) , **rmlink**(1)

Author

Enea LINX

team

Author

RMTCPCON(1) manual page

Name

rmtcpcon - remove LINX TCP/IP connections.

Synopsis

rmtcpcon [OPTIONS] <connection>...

Description

Remove LINX TCP/IP connections. Make sure that the connection is not assigned to any link before it is removed (i.e. remove the link first).

Options

-h, --help
Display help and exit.

Files

None.

Diagnostics

None.

Known Bugs

None.

Examples

```
rmethcon tcpcon_A
rmethcon tcpcon_A tcpcon_B
```

See Also

linx(7) , **linxcfg**(1) , **mktcpcon**(1) , **rmlink**(1)

Author

Enea LINX

team

Copyright