LINX for Linux User's Guide

Document version: 2.6.9 update 1

1. LINX Overview

1.1 Introduction

LINX is an open inter-process communications (IPC) protocol, designed to be platform and interconnect independent. It enables applications to communicate transparently regardless of whether they are running on the same CPU or are located on different nodes in a cluster. Any type of cluster configuration is supported, from a single multi-core board to large systems with many nodes interconnected by any network topology. LINX is based on the traditional message passing technology used in the Enea OSE / OSEck family of real-time operating systems.

LINX consists of a set of Linux kernel modules, a LINX library to be linked with applications and a few command tools for configuration of inter-node links and statistics reports.

There is one main LINX kernel module that implements the IPC mechanisms and the Rapid Link Handler (RLNH) protocol, which allows LINX functionality to span multiple nodes transparently over logical links. To use LINX for inter-node communication, a Connection Manager (CM) kernel module that supports the underlying interconnect must be loaded as well. Currently, LINX contains two CMs, one for raw Ethernet and one for TCP/IP. The CM is located below the main LINX kernel module in the protocol stack and its main task is to provide reliable, in-order delivery of messages. LINX can be adapted to any underlying transport by adding new CMs.

The RLNH protocol and the CM protocols are specified in a separate document (see section 9.2) .

The LINX kernel module provides a standard socket based interface using its own protocol family, `PF_LINX`.

The LINX library provides a set of function calls to applications. Application programmers should normally use the LINX API provided by this library, but it is possible to use the underlying socket interface too if necessary. Information on how to use the socket interface directly is found in the LINX reference documentation.

1.2 LINX Concepts

Endpoint	An endpoint is an entity which can participate in LINX communication. Each endpoint is assigned a name by the application creating it.
SPID	A binary identifier assigned to each endpoint by LINX. The SPID is used to refer to an endpoint when communicating with it.
LINX Signal	Endpoints communicate by exchanging messages called LINX signals. When sending a LINX signal, the application specifies the SPID of the destination endpoint.
Connection	A LINX connection provides reliable, in-order delivery of LINX data between two nodes over an underlying media or protocol stack.
Connection Manager	A LINX component that implements support for setting up connections over a particular type of interconnect.
Link	A logical association between two LINX nodes. Each link uses an underlying connection as transport. LINX IPC services are transparent across links.
Hunting	A LINX mechanism that allows applications to look up the SPID of an endpoint by name. A LINX signal is sent back to the application when a matching endpoint is found or created. Applications can search for endpoints on remote nodes by specifying a path of links to traverse.

Attaching A LINX mechanism that allows application to supervise endpoints in order to find out when they are terminated. A LINX signal is sent back to the application when the supervised endpoint is terminated.

2. Installation

Download the LINX distribution `linx-n.n.n.tar.gz`, where n.n.n is the LINX version. See section 9.3 for information on where to download LINX. Extract the contents of the archive at a suitable place in your Linux system:

```
$ tar -zxvf linx-n.n.n.tar.gz
```

This creates a LINX directory called `linx-n.n.n/`. The file `linx-n.n.n/doc/index.html` contains pointers to all documentation available in the release. Make sure to read the `README`, `RELEASE_NOTES` and `Changelog` files for information about this version. Reference documentation is available as man pages and in HTML format. There is also a document describing the LINX protocols.

The following is found under the top level LINX directory:

`Makefile`, `config.mk`, `common.mk`	Make files for building LINX
`COPYING`, `MANIFEST`, `README`, `RELEASE_NOTES`	Licensing, readme, release notes.
`bmark/`	Example benchmark application
`doc/`	LINX documentation
`drivers/`	Dummy network driver for LINX message trace
`example/`	Example client/server application
`include/`	LINX include files
`liblinx/`	LINX library source code
`linxcfg/`, `linxdisc/`, `linxstat/`, `linxgw/`	LINX commands source code
`net/linx/`	LINX source code
`patch/`	Patches for libpcap and tcpdump
`scripts/`	Build scripts

3. Building LINX

To build LINX self hosted, e.g. for the running kernel, just go to the top level LINX directory and do ./configure followed by make:

```
$ cd /path/to/linx-n.n.n/
$ ./configure
$ make
```

If you encounter problems due to automake files, regenerate the files by using the autogen.sh script.

```
$ ./autogen.sh
```

This will build the entire LINX API libs and user space LINX tools.

Note that headers in the target Linux kernel source tree must be available to be able to compile LINX. This is needed also when compiling for the running Linux kernel.

Cross-compiling the LINX API libs and tools requires the PATH to include the cross compilier tool kit and the make files must be configured with a host parameter:

```
$ cd /path/to/linx-n.n.n/
$ PATH=/path/to/cross/compiler:$PATH
$ ./configure --host=powerpc-linux
$ make
```

For multithreaded LINX client applications that will use a large number of LINX endpoints within the same process, it is possible to decrease the virtual memory consumption by enabling LINX endpoints to share the same virtual address pool:

```
$ cd /path/to/linx-n.n.n/
$ PATH=/path/to/cross/compiler:$PATH
$ ./configure --enable-share-virt-pool
$ make
```

Note that having a large number of LINX endpoints sharing the same virtual address pool may lead to pool depletion.

To build the LINX kernel modules go to the LINX kernel modules directory and do make:

```
$ cd /path/to/linx-n.n.n/net/linx
$ make
```

Cross-compiling LINX for another target requires a few variables to be set, either as environment variables or by changing the config.mk file in the top level LINX directory. The following is needed:

- `ARCH` – Target architecture, e.g. `ppc`
- `CROSS_COMPILE` – Cross compiler tool prefix, e.g. `powerpc-linux-`
- `KERNEL` – Kernel source tree

In addition, the `PATH` environment variable must be set to reach the cross compiler tool kit. When this has been set up correctly, go to the top level LINX directory and do make.

When building the entire LINX package, the following is built:

- In `net/linx/`
  - The LINX kernel module, `linx.ko`
  - The LINX Ethernet Connection Manager kernel module, `linx_eth_cm.ko`
  - The LINX TCP Connection Manager kernel module, `linx_tcp_cm.ko`
  - The LINX Rapid IO Connection Manager kernel module, `linx_rio_cm.ko`
  - The LINX Shared Memory Connection Manager kernel module, `linx_shm_cm.ko`
  - The LINX Connection Manager Control Layer kernel module, `linx_cmcl.ko`
- In `lib/` (created)
  - The LINX library, `liblinx.a`
  - The LINX configuration library, `liblinxcfg.a`
  - The LINX Gateway library, `libgw.a`
  - The
- In `bin/` (created)
  - The `mklink` command for creating links
  - The `rmlink` command for destroying links
  - The `mkethcon` command for creating connections using the LINX Ethernet Connection Manager.
  - The `rmethcon` command for destroying connections created with the mkethcon command.
  - The `mkshmcon` command for creating connections using the LINX Shared Memory Connection Manager.
  - The `rmshmcon` command for destroying connections created with the mkshmcon command.
  - The `mktcpcon` command for creating connections using the LINX TCP Connection Manager.
  - The `rmtcpcon` command for destroying connections created with the mktcpcon command.
  - The `mkriocon` command for creating connections using the LINX Rapid IO Connction Manager.
  - The `rmriocon` command for destroying connections created with the `mkriocon` command.
  - The `mkcmclcon` command for creating connections using the LINX Connection Manager Control Layer.
  - The `rmcmclcon` command for destroying connections created with the `mkcmclcon` command.
  - The `linxdisc` daemon for dynamic LINX topology setup
  - The `linxgws` daemon for connecting to applications using the Gateway protocol.
  - The `linxgwcmd` command for listing Gateway servers.
  - The `linxstat` command for LINX statistics
  - The `linxcfg` command for creating connections and links.
- In `drivers/net`
  - The LINX message trace dummy network driver, `linxtrace.ko`
- In `example/bin` (created)
  - LINX example application binaries, `linx_basic_client` and `linx_basic_server`
- In `bmark/bin` (created)
  - A simple benchmark application, `linx_bmark`

4. Using LINX

This section describes the fundamental concepts of LINX communication. The examples show how to use the LINX API, defined in the file `linx.h`.

See section 5 for information on how to load and configure LINX for your system.

4.1 LINX Endpoints

An application that wants to communicate using LINX first creates a **LINX endpoint** by calling `linx_open()`. `linx_open()` assigns a name to the endpoint, the name is a null-terminated string. The name is not required to be unique. On Linux, the name may be of any length, but note that there may be restrictions on other platforms. One thread may own multiple LINX endpoints simultaneously.

```
LINX *client = linx_open("client", 0, NULL);
```

LINX assigns each endpoint a binary identifier called a **SPID**. The SPID is used to refer to the endpoint when communicating with it. SPIDs are unique within the node on which they are created.

Each endpoint is internally associated with a LINX socket. An application can obtain the socket descriptor of a LINX endpoint using the `linx_get_descriptor()` call if needed, e.g. for generic `poll()` or `select()` calls together with other descriptors. Note that a LINX socket descriptor retrieved this way must not be closed by calling `close()`.

A LINX endpoint is closed by calling `linx_close()`. This frees all resources owned by the endpoint. If a thread exits, all of its owned LINX endpoints will automatically be closed.

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()` 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.

4.2 LINX Signals

Applications communicate by exchanging messages called **LINX signals** between endpoints. A LINX signal buffer contains a mandatory leading 4 byte **signal number**, optionally followed by data that the sender wishes to convey to the destination. Thus, the minimum size of a LINX signal is 4 bytes. Signal numbers are used to identify different types of signals and are mainly defined by applications. The signal number range 0x10000000 to 0xFFFFFFFF is available for user applications.

New LINX signals are easily defined. It is simply a matter of declaring a struct (see the example below). Each application shall also declare the `union LINX_SIGNAL` type to contain all LINX signals used in that particular application. LINX signals shall be cast to pointers to this generic structure in LINX API function calls.

```
#define REQUEST_SIG 0x10000001 /* Signal number */
#define REPLY_SIG   0x10000002

struct request_sig
{
   LINX_SIGSELECT sig_no;
   int code;
}
```

```
struct reply_sig
{
   LINX_SIGSELECT sig_no;
   int status;
}

union LINX_SIGNAL
{
   LINX_SIGSELECT     sig_no;
   struct request_sig request;
   struct reply_sig   reply;
};
```

Before a LINX signal can be sent, it must be allocated and initialized. The `linx_alloc()` call returns a LINX signal buffer and initializes the signal number with a provided value.

```
union LINX_SIGNAL *sig;

sig = linx_alloc(endpoint, sizeof(struct request_sig), REQUEST_SIG);
sig->request.code = 1;
```

The returned LINX signal buffer is owned by the LINX endpoint that allocated it and may not be used by other endpoints. Sending a LINX signal transfers its ownership to the destination endpoint. A LINX signal is never shared between different threads or endpoints. When a LINX signal buffer is not needed anymore, it should be freed by calling `linx_free_buf()`.

Before sending a LINX signal, the SPID of the destination endpoint must be known. LINX provides a method to obtain the SPID of an endpoint by searching for its name, this is called *hunting* and is described in the next section. The receiver of a LINX signal can look up the SPID of the sender using the `linx_sender()` call. When the destination SPID is known, the LINX signal can be sent:

```
linx_send(endpoint, &sig, server_spid);
```

Transferred LINX signals are stored in a receive queue associated with the destination endpoint. The destination endpoint chooses when to receive a LINX signal and what signal numbers to accept at any given time. This means that an endpoint may choose to receive LINX signals in a different order than they were sent, based on signal number filtering. A received LINX signal may be reused, for example to send a reply, if the buffer is large enough. Just overwrite the signal number field with the new value.

```
union SIGNAL *sig;
LINX_SIGSELECT any_sig[]= { 0 }; /* Signal filter. Here any signal is allowed */

linx_receive(endpoint, &sig, any_sig);
```

LINX provides automatic endian conversion of the signal number if needed when a LINX signal is sent to an endpoint located on a remote node. The rest of the signal data is not converted, this must be taken care of in the applications.

4.3 Hunting for an Endpoint

Before sending a LINX signal, the sender must know the SPID of the destination endpoint. The SPID of a peer endpoint is obtained by asking LINX to **hunt** for its name using the `linx_hunt()` call. When the peer endpoint has been found, LINX makes sure that a LINX signal is sent to the hunting endpoint. This LINX signal appears to have been sent from the found peer endpoint, i.e the SPID can be obtained by looking at the sender of the LINX signal. The hunting endpoint may provide a LINX signal to be sent back when the sought endpoint has been found. If no LINX signal is provided, a default LINX signal of type `LINX_OS_HUNT_SIG` is sent instead.

```
union SIGNAL *sig;
LINX_SPID server_spid;
LINX_SIGSELECT sig_sel_hunt = { 1, LINX_OS_HUNT_SIG };

linx_hunt(endpoint, "server", NULL);
linx_receive(endpoint, &sig, sig_sel_hunt);
server_spid = linx_sender(endpoint, &sig);
```

If the peer endpoint does not exist when hunted for, LINX stores the hunt internally as pending. The LINX hunt signal is sent back to the hunting endpoint when an endpoint with matching name is created.

If there are several LINX endpoints with the same name, it is not defined which one is used to resolve a hunt call.

4.4 Attaching to an Endpoint

If a LINX endpoint sends a LINX signal to another endpoint, but the receiving endpoint has terminated for some reason, the LINX signal will be thrown away (freed) by LINX.

LINX provides a mechanism to supervise a peer endpoint, i.e. to request notification of when it is terminated. The `linx_attach()` call is used to **attach** to an endpoint. When a supervised endpoint terminates, LINX makes sure that a LINX signal is sent back to the supervising endpoint. This LINX signal appears to have been sent from the supervised (terminated) endpoint, i.e the SPID can be obtained by looking at the sender of the LINX signal. The endpoint that attaches may provide a LINX signal to be sent back when the supervised process terminates. If no LINX signal is provided, a default LINX signal of type `LINX_OS_ATTACH_SIG` is sent instead.

```
linx_attach(endpoint, server_spid, NULL);
```

If the supervising endpoint wants to resume communication, it should issue a new hunt for the peer endpoint name, in order to be notified when a new endpoint with the same name is found or created.

4.5 Inter-node Communication

LINX endpoints are able to communicate transparently regardless of whether they are located on the same node or on different nodes interconnected in some way in a LINX cluster. A cluster may use different operating systems that support LINX – a LINX endpoint on a Linux node may for example communicate with a process on a connected DSP running the Enea OSEck real-time operating system.

A LINX cluster should be seen as a logical network established between a set of nodes interconnected by some underlying transport that is supported by LINX, such as Ethernet. For two nodes to be able to communicate, a LINX **link** must first be established between them. Each node may set up any number of links to other nodes which are directly reachable on the underlying transport. Links can be manually set up by using the `mkethcon` or `mkshmcon` or `mktcpcon` and the `mklink` command, or dynamically established using the LINX discovery daemon, `linxdisc`.

Each link has a name that is unique within the node. The name of a link may be (and usually is) different on the two sides of the link, i.e. the link between nodes A and B may be called “LinkToB” on A and “LinkToA” on B. Often the link name is the same as the name of the remote node connected via the link.

Note that nodes do not have addresses in LINX. To reach a remote node, the complete path of link names to be used is specified.

To hunt for a LINX endpoint located on a remote node, the name of the endpoint is prepended with the path of link names that shall be used to reach that node, separated by “/”.

Example:

Hunting for “LinkToB/LinkToC/EndpointName” tells LINX to search for “EndpointName” on the node two hops away from us that is reachable by traversing first “LinkToB” and then “LinkToC”.

4.6 Virtual Endpoints

Since LINX SPIDs are unique within a single node only, it is not possible to address remote endpoints by using their remote SPIDs directly. LINX inter-node communication is based on automatic creation of local **virtual endpoints** that represent remote endpoints. Each LINX endpoint involved in inter-node communication has a virtual endpoint, internally created by LINX, representing it on the peer node. A virtual endpoint acts as a proxy for a particular remote endpoint and is communicated with in the same way as normal (user-created) endpoints. This way, applications do not need to know the true SPIDs of endpoints on other nodes - they always communicate with local virtual endpoints, which have local SPIDs. The life span of a virtual endpoint matches the life span of the remote endpoint it represents.

A LINX signal sent to a virtual endpoint is intercepted by LINX and automatically forwarded to the remote node where the endpoint it represents is located. On the destination node, LINX delivers the LINX signal to its intended destination and makes it appear as if it was sent from a virtual endpoint representing the true sender.

A LINX signal received from an endpoint on a remote node always appears to have been sent from the corresponding local virtual endpoint.

Virtual endpoints are created by LINX when needed, typically when a remote hunt call has been made and the peer endpoint has been found (or created) on the remote node. Virtual endpoints use the same naming syntax as the hunt path described above. This means that when LINX creates a virtual endpoint, the pending hunt request for its name are resolved and the hunt LINX signal is sent to the hunting endpoint from the SPID of the virtual endpoint.

LINX also creates virtual endpoints representing links to remote nodes. These endpoints carry the same name as the link, prepended with "/". An application can monitor the state of a link by hunting for and attaching to the virtual endpoint representing it.

When an endpoint terminates, LINX makes sure that all virtual endpoints representing it on remote nodes are terminated too (so that attach LINX signals are sent to supervising endpoints). If a remote node is shut down or becomes unreachable, LINX will detect this and terminate all virtual endpoints that represent endpoints located on that node.

Example:

An application on node A hunts for "LinkToB/server". This tells LINX to search for the endpoint "server" on node B, reachable by traversing link "LinkToB". When an endpoint named "server" has been found (or created) on B, LINX creates a virtual endpoint on node A named "LinkToB/server" and sends the hunt LINX signal from this virtual endpoint to the hunting endpoint. After receiving the hunt LINX signal, the application is able to communicate with the remote endpoint "server" on node B by sending LINX signals to the virtual endpoint "LinkToB/server".

Note that the scenario above will also create a virtual endpoint named "LinkToA/client" on node B (if "client" is the name of the hunting endpoint and "LinkToA" is the name of the link on node B).

4.7 Out-of-band signalling

LINX has from version 2.1 and forth support for out-of-band signalling, enabling the user to set an out-of-band attribute on signals when sending them. LINX will make a best-effort attempt to deliver any out-of-band signals ahead of normal (in-band) signals between two LINX endpoints. LINX does not guarantee that out-of-band signals really are delivered before in-band signals. Except being able to be delivered ahead of in-band signals, out-of-band signals follow the same rules as internal signals, delivery is guaranteed and the order in which out-of-band signals are sent is kept on the receiving side, i.e. two out-of-band signals sent one after the other are guaranteed to arrive in the same order on the receiver.

If the receiving side should have no support for out-of-band signalling, i.e. an older version of LINX, the signal will be treated in-band.

Out-of-band signals can be sent both intranode and internode. In the intranode case the signal is put in the receiver's in-queue ahead of in-band signals but not before any out-of-band signals already in the queue. When sending out-of-band signals internode the out-of-band attribute is passed down to the connection layer and it is up to the connection layer if it chooses to treat out-of-band signals differently than in-band signals, trying to deliver them "faster" than in-band signals.

Upon receiving a signal the user can find out if the received signal has the out-of-band attribute set.

LINX has from version 2.1 and forth support for tying two separate connections to one logical link. When doing so one of the connections is used for in-band signals while the other connection is used for out-of-band signalling. The two connections can be over different media and the link is considered in the state up only when both connections are in state connected. If one of the connections is disconnected the link is considered down, thus no fail-over is done.

4.8 LINX Signal Pool

This feature adds zero copy functionality to LINX, by implementing a signal pool that is allocated and managed by the LINX kernel module. When a LINX endpoint is opened, a new mapping of the signal pool is created in the virtual address space of the calling process By using this feature, the overhead of copying the signals from user space to kernel space on `linx_send()` and from kernel space to user space on `linx_receive()` is eliminated.

One can allocate signals from the LINX pool by calling `linx_s_alloc()` and free them by calling `linx_free_buf()` . If the Signal Pool feature is disabled, the `linx_s_alloc()` will return NULL and the errno will be set to EOPNOTSUPP.

LINX pool configuration:

If no base address is specified, the pool is allocated with vmalloc(). If one requires the pool to reside into physically contiguous memory, it should reserve the memory at boot time by using the boot command-line parameter:
memmap=<size>$<addr>. Then specify the size and the base address of the pool when the LINX module is inserted, using the following parameters:
`linx_pool_size`=<size> `linx_pool_base_addr`=<addr>

If no `linx_pool_size` is specified, a default pool of 16 MB is allocated. The pool can be configured to use different signal block sizes by specifying the following module parameter:
`linx_pool_blocks_size= <size_1>,<size2>,...,<sizeN<`.

It is possible for all LINX endpoints within a process to share the same virtual address space of the pool mapping if the `--enable-share-virt-pool` configuration option is used at compile time.

It should be noted that the LINX Signal Pool feature is enabled by default and it can be disabled by setting the linx_pool_size parameter to 0.

5. Getting Started

5.1 Loading the LINX Kernel Modules

To enable LINX, simply load the LINX kernel module into the Linux kernel (requires root permissions):

```
$ insmod net/linx/linx.ko
```

Applications are now able to use LINX, but only to communicate within the node.

To use LINX for communication between several interconnected nodes, also load the appropriate LINX Connection Manager kernel module depending on which underlying transport to use. LINX currently supports raw Ethernet and TCP/IP.

To use raw Ethernet as transport, load the Ethernet CM kernel module:

```
$ insmod net/linx/linx_eth_cm.ko
```

To use TCP/IP as transport, load the TCP CM kernel module:

```
$ insmod net/linx/linx_tcp_cm.ko
```

5.2 Creating Links

5.2.1 Normal case - one link over one connection

To setup a LINX cluster with two participating nodes, here called A and B, start by installing the appropriate kernel modules as described above on both nodes. Then use the connection manager specific setup tool to create a connection on each node to create a connection to the other node and then the mklink command on both nodes to create a logical link. To destroy a link the rmlink command is used.

It is up to each connection manager to provide a tool for setting up a connection. The LINX release contains Ethernet and TCP connection managers, their corresponding tools for creating and destroying connections are mkethcon, rmethcon, mkshmcon, rmshmcon, mktcpcon and rmtcpcon.

When using Ethernet, the MAC address of the remote node, the device name to use and a suitable link name shall be provided.

On node A (replace the MAC address with the actual value on node B):

```
$ ./bin/mkethcon --mac=00:18:4D:72:13:1B --if=eth0 ConnToB
$ ./bin/mklink --connection=ethcm/ConnToB LinkToB
```

On node B (replace the MAC address with the actual value on node A):

```
$ ./bin/mkethcon --mac=00:0C:6E:C3:FB:A2 --if=eth0 ConnToA
$ ./bin/mklink --connection=ethcm/ConnToA LinkToA
```

When using Shared Memory, the mailbox identifier shall be provided. It is also necessary to provide the -x option on one of the sides.

On CPU A:

```
$ ./bin/mkshmcon -b 1 ConnToB
```

```
$ ./bin/mklink --connection=shmcm/ConnToB LinkToB
```

On CPU B:

```
$ ./bin/mkshmcon -b 1 -x ConnToA
$ ./bin/mklink --connection=shmcm/ConnToA LinkToA
```

When using TCP/IP, the IP address of the remote node and a suitable link name shall be provided:

On node A (replace the IP address with the actual value on node B):

```
$ ./bin/mktcpcon --ipaddr=192.168.1.2 ConnToB
$ ./bin/mklink --connection=tcpcm/ConnToB LinkToB
```

On node B (replace the IP address with the actual value on node A):

```
$ ./bin/mktcpcon --ipaddr=192.168.1.1 ConnToA
$ ./bin/mklink --connection=tcpcm/ConnToA LinkToA
```

After these steps, the LINX cluster is available and applications can communicate with each other transparently, regardless of on which node they are located.

One can also use the linxcfg tool which creates both the connection and link at the same time. For documentation of the linxcfg tool see the man-page for linxcfg(1).

5.2.2 Out-of-band - one link over two connections

LINX supports one logical link over two connections, the connection managers do not need to aware of this and connections are setup the same way as before. Then the mklink tool is used to tie two connections to one logical link. The connections does not need to be on the same media.

In this example a logical link is setup between node A and node B using the Ethernet Connection Manager and VLAN. The non-VLAN connection will be used for in-band signalling and the VLAN connection for out-of-band siganlling. The order in which the connections are passed in on the command line to mklink determines which connection will be used for out-of-band, the first connection is used for in-band and the second for out-of-band.

On node A (replace the MAC address and IP address to that of node B):

```
$ /sbin/vconfig add eth0 5
$ /sbin/ifconfig eth0.5 up
$ ./bin/mkethcon --mac=00:18:4D:72:13:1B --if=eth0 EthConnToB
$ ./bin/mkethcon --mac=00:18:4D:72:13:1B --if=eth0.5 VlanEthConnToB
$ ./bin/mklink --connection=ethcm/EthConnToB --connection=ethcm/VlanEthConnToB LinkToB
```

On node B (replace the MAC address and IP address to that of node A):

```
$ /sbin/vconfig add eth0 5
$ /sbin/ifconfig eth0.5 up
$ ./bin/mkethcon --mac=00:0C:6E:C3:FB:A2 --if=eth0 EthConnToA
$ ./bin/mkethcon --mac=00:0C:6E:C3:FB:A2 --if=eth0.5 VlanEthConnToA
$ ./bin/mklink --connection=ethcm/EthConnToA --connection=ethcm/VlanConnToA LinkToA
```

Now all in-band signals sent over the link LinkToA will be sent on the non-VLAN Ethernet connection and all out-of-band signals will be sent on the VLAN Ethernet connection.

5.3 Running the LINX Example Application

The LINX example application is found in the `example/` directory. It is a simple client / server based application that serves both as an introduction to the LINX API programming model, and as a quick way of testing that LINX is up and running in a system with one or more nodes.

See above for information on how to build the LINX kernel modules and binaries (including the example).

Doing `make example` in the top level LINX directory builds only the example. This produces two executables in the `example/bin` directory: `linx_basic_client` and `linx_basic_server`.

The actual operation of the application is simple; the client sends LINX signals to the server and the server sends reply LINX signals back. There can be any number of clients distributed on different nodes. Each client will hunt for the server, either on a given link name (path of link names) or on the local machine if no linkname is provided. The server can be terminated and restarted, the clients use LINX attach to detect when the server disappears and will resume operation when the server is available again.

To run the example on a single node, the followings steps are needed:

1. Build LINX
2. Load the LINX kernel module into the kernel
3. Start the example server in the background: `example/bin/linx_basic_server &`
4. Start the example client: `example/bin/linx_basic_client`

To run the example on two nodes, the following steps are needed:

1. Build LINX
2. Load the LINX kernel module on both nodes
3. Load the appropriate LINX CM kernel module on each node
4. On each node, use the `mkethcon/mktcpcon` and mklink commands to establish the link
5. On one node, start the example server: `example/bin/linx_basic_server`
6. On the other node, start the example client: `example/bin/linx_basic_client linkname`
   Where `linxname` is either LinkToA or LinkToB according to the link names given above in 5.2).

6. LINX Utilities

Description of the tools provided with this release.

6.1 mklink

The `mklink` command creates LINX links to remote nodes. The connection(s) must already be created by the Connection Manager used. The type of connection(s) used is transparent for the `mklink`. A logical LINX link can use two connections, one for normal signalling and one for out-of-band signalling, the first connection on the command-line will used for in-band and the second for out-of-band signalling.

Example:

```
$ ./bin/mklink --connection=ethcm/eth_connA linkA
```

Example with two links, the first for in-band and second for out-of-band signalling:

```
$ ./bin/mklink --connection=ethcm/eth_connA --connection=ethcm/eth_connA_2 linkA
```

The `mklink` command must be used on both participating nodes for a link to be established.

See the mklink(1) reference documentation for details.

6.2 rmlink

The `rmlink` command is used to destroy links created by the `mklink` command, both sides must run this command.

Example:

```
$ ./bin/rmlink linkA
```

See the rmlink(1) reference documentation for details.

6.3 mkethcon

The `mkethcon` command is used to create Ethernet Connections to remote nodes. The connection name is then used as a handle when creating a LINX link.

Example:

```
$ ./bin/mkethcon --mac=01:23:45:67:89:A0 --if=eth0 eth_connA
```

The `mkethcon` command must be used on both participating nodes for a Ethernet connection to be established.

See the mkethcon(1) reference documentation for details.

6.4 rmethcon

The `rmethcon` command is used to destroy connections created by the `mkethcon` command, both sides must run this command.

Example:

```
$ ./bin/rmethconn eth_connA
```

See the rmethcon(1) reference documentation for details.

6.5 mkshmcon

The `mkshmcon` command is used to create Shared Memory Connections to remote nodes. The connection name is then used as a handle when creating a LINX link.

Example:

```
$ ./bin/mkshmcon -b 1 -n 16 -m 120 shm_connA
```

The `mkshmcon` command must be used on both participating nodes for a Shared Memory connection to be established.

See the mkshmcon(1) reference documentation for details.

6.6 rmshmcon

The `rmshmcon` command is used to destroy connections created by the `mkshmcon` command, both sides must run this command.

Example:

```
$ ./bin/rmshmconn shm_connA
```

See the rmshmcon(1) reference documentation for details.

6.7 mktcpcon

The `mktcpcon` command is used to create TCP Connections to remote nodes. The connection name is then used as a handle when creating a LINX link.

Example:

```
$ ./bin/mktcpcon --ipaddr=12.34.56.78 tcp_connA
```

The `mktcpcon` command must be used on both participating nodes for a TCP connection to be established.

See the mktcpcon(1) reference documentation for details.

6.8 rmtcpcon

The `rmtcpcon` command is used to destroy connections created by the `mktcpcon` command, both sides must run this command.

Example:

```
$ ./bin/rmtcpconn tcp_connA
```

See the rmtcpcon(1) reference documentation for details.

6.9 mkriocon

The mkriocon command is used to create RIO Connections to remote nodes. The connection name is then used as a handle when creating a LINX link.

Example:

$ ./bin/mkriocon -p 1 -l 1 -m 0 -I 2 -t 5 -i rio0 rio-conn

The mkriocon command must be used on both participating nodes for a TCP connection to be established.

See the mkriocon(1) reference documentation for details.

6.10 rmriocon

The rmriocon command is used to destroy connections created by the mkriocon command, both sides must run this command.

Example:

$ ./bin/rmrioconn rio-conn

See the rmriocon(1) reference documentation for details.

6.11 mkcmclcon

The mkcmcmlcon command is used to create CMCL Connections to remote nodes. The connection name is then used as a handle when creating a LINX link. CMCL also takes a connection name representing the layer underneath as input parameter, the CMCL connection is created using the underlying connection.

Example:

$ ./bin/mkcmclcon -s -c cmtl-conn cmcl-conn -t 3000

The mkcmclcon command must be used on both participating nodes for a CMCL connection to be established.

See the mkcmclcon(1) reference documentation for details.

6.12 rmcmclcon

The rmcmclcon command is used to destroy connections created by the mkcmclcon command, both sides must run this command.

Example:

$ ./bin/rmcmclconn cmcl-conn

See the rmcmclcon(1) reference documentation for details.

6.13 linxdisc

On Ethernet, a LINX cluster can be automatically established and supervised by running the `linxdisc` daemon on all participating nodes. The daemon periodically broadcasts advertisements and waits for advertisements from remote nodes. Each node advertises a cluster name and a node name. These values are defined in a configuration file provided to `linxdisc`. Each cluster and each node must have a unique name. The configuration file also defines filtering rules for which network interfaces to use and which remote node names to connect to. When an advertisement is received from a node that belongs to the same cluster and is allowed according to the node name filtering rules, a link is automatically set up between the nodes.

See the linxdisc(8) and linxdisc.conf(5) reference documentation for details.

6.14 linxstat

LINX status information can be fetched from the LINX kernel module and displayed using the `linxstat` command. Status is shown for local and virtual (remote) endpoints as well as links to other nodes. The information includes names, SPIDs, queued LINX signals and pending hunts and attaches. See the linxstat(1) reference documentation for details.

6.15 linxgws

In Linx for Linux 2.2 the LINX Gateway was added allowing connectivity between Linux applications using the Gateway client protocol to LINX systems. The Gateway Server runs as a daemon and is started on the command line, configuration is done using a configuration file. The default location is /etc/linxgws.conf but can also be specified at startup. The LINX Gateway Server, its configuration and the Gateway Client are described in detail in the Users Guide for the LINX Gateway (PDF-version).

Example:

```
$ ./bin/linxgws linxgws.conf
```

See the linxgws(8) and linxgws.conf(5) reference documentation for details.

6.16 linxgwcmd

The linxgwcmd tool is used to discover Gateway servers within the current UDP broadcast domain. Can also be used to test connectivity towards a Gateway server and also simple benchmarking.

Example: list all Gateway servers broadcasting on port 30000

```
$ ./bin/linxgwcmd -l -b udp://*:30000
```

Example: connect to a Gateway server named "default_gws" and echo 10 signals of 100 bytes each.

```
$ ./bin/linxgwcmd -s default_gws -e10,100
```

See the linxgwcmd(1) reference documentation for details.

6.17 LINX Message Trace

When using an external protocol analyzer such as `tcpdump`, only LINX messages sent between nodes are visible. Intra-node transmissions never reach the point where the Linux kernel duplicates messages for listening protocol analyzers (the Linux General Device Driver Interface). LINX provides a feature to make these messages visible as well. This is done by duplicating all LINX signals that are sent within LINX to a

special dummy network driver called linx0.

There are also patches for `tcpdump` and `libpcap` included in the LINX release (since LINX 1.2). These patches make tcpdump and libpcap understand the messages that are sent to linx0.

To start using LINX message trace, the following steps are required:

1. Compile the LINX kernel module with the `LINX_MESSAGE_TRACE=yes` option and load it into the kernel.
2. Compile the LINX dummy network driver and load it into the kernel. Enable it by doing: `ifconfig linx0 up`
3. Compile tcpdump and libpcap with the patches provided in the LINX release `patch/` directory.
4. Start tcpdump and configure it to listen on linx0.

7. LINX Kernel Module Configuration

Parameters can be passed to the LINX kernel module at load time. Example:

```
$ insmod linx.ko linx_max_links=64
```

To list available parameters and types, the modinfo command can be used:

```
$ modinfo -p linx.ko
```

The following parameters can be passed to the LINX kernel module:

linx_max_links

The maximum number of links that can be established to remote nodes. When the limit is reached, LINX will refuse to create new links. The default value is 32 and the maximum value is 1024.

linx_max_sockets_per_link

The maximum number of endpoints (sockets) that are allowed to communicate over one link. If the limit is exceeded, the link will be disconnected and reestablished. The value must be a power of 2. The default value is 1024 and the maximum value is 65536.

linx_max_spids

The maximum number of LINX endpoints (sockets) that can be created. When the limit is reached, LINX will refuse to create new endpoints. The value must be a power of 2. The default value is 512 and the maximum value is 65536.

linx_max_attrefs

The maximum number of pending attaches. When the limit is reached, attach() calls will fail. The value needs to be a power of 2. The default value is 1024 and the maximum value is 65536.

linx_max_tmorefs

The maximum number of timeout references. When the limit is reached, `linx_request_tmo()` will fail. The value must be a power of 2. The default value is 1024 and the maximum value is 65536.

linx_receive_buffer_size

The size of the signal that will be allocated by `linx_receive` in user space, used to copy the received signal from kernel space. The default value is 4096 bytes. This parameter may prove useful in use-cases with small signal sizes, when allocating 4096 bytes at each `linx_receive` may consume a lot of extra memory and add performance penalties (i.e. page faults). To decrease this allocation overhead, it is recommended to adjust the buffer size from the default, in accordance with the actual size needed per case.

linx_pool_alignment

Alignment of the `signal pool`. The pool will be aligned at 1 << linx_pool_alignment bytes. The default value for the linx_pool_alignment parameter is 8.

linx_pool_base_addr

The base address of the physical memory area where the signal pool is mapped. If none is specified, the pool is allocated with vmalloc(). The base address must be a multiple of the pool alignment.

linx_pool_size

Size of the signal pool. The default value is 16 MB. The size must be a multiple of the pool alignment. Setting the value to 0, it will disable the LINX Signal Pool feature.

linx_pool_blocks_size

Array containing the size of the blocks in the signal pool. The default number of blocks is 8 and the default block sizes are: 512, 1024, 2048, 4096, 8192, 16384, 65536, 131072. The block sizes will be rounded up if they are not multiples of the pool alignment.

linx_pool_s_alloc_count

The number of buffers to be reserved from the signal pool to user space. Doing a system call for each linx_s_alloc() call adds a significant overhead. To speed-up pool allocation, an intermediary list of pool signals for each pool block size has been added to user space, per LINX endpoint. When linx_s_alloc() is called, if the list for the requested pool block is not empty, a signal is returned immediately, without doing a system call to the kernel. Otherwise, linx_pool_s_alloc_count buffers are reserved from the pool and added to the list, so that they can used in the subsequent linx_s_alloc() calls. It should be noted that these lists are specific to each LINX endpoint: one endpoint cannot use the buffers from another endpoint. The default value of linx_pool_s_alloc_count is 1, maintaning the default functionality of one system call per each pool allocation.

linx_pool_s_free_count

The threshold value at which the signal pool buffers are returned to the kernel and can be re-used. When freeing pool signals, doing a system call for each linx_free_buf() is costly and can have an impact on performance. By using the linx_pool_s_free_count parameter, one can specify the number of consecutive linx_free_buf() calls after which the signals are freed (by returning them to the kernel pool). Until the threshold value is reached, the pool signals are saved in an internal list, per endpoint. When linx_pool_s_free_count is reached, all the saved signals are actually freed. It should be noted that this value must be chosen in correlation with the pool size, since it might lead to pool exhaustion if the actual freeing is done too rarely. The default value of linx_pool_s_free_count is 1, maintaning the default functionality of one system call per each pool signal freed.

8. LINX Statistics

8.1 Per-endpoint Statistics

Statistics per endpoint can be enabled at compile-time for the LINX kernel module with the `-DSOCK_STAT` build flag.

Statistics are collected for both ordinary and virtual LINX endpoints, as well as for links, independent of which Connection Manager is used. The results can be found in the procfs file system in the file `/proc/net/linx/sockets` and are also available through the `linx_get_stat()` function call or by using the `linxstat -S` command.

The following statistics are presented:

no_recv_bytes	Number of received bytes
no_sent_bytes	Number of sent bytes
no_recv_signals	Number of received LINX signals
no_sent_signals	Number of sent LINX signals
no_recv_remote_bytes	Number of bytes received from remote endpoints.
no_sent_remote_bytes	Number of bytes sent to remote endpoints.
no_recv_remote_signals	Number of LINX signals received from remote endpoints.
no_sent_remote_signals	Number of LINX signals sent to remote endpoints.
no_recv_local_bytes	Number of bytes received from local endpoints.
no_sent_local_bytes	Number of bytes sent to local endpoints.
no_recv_local_signals	Number of LINX signals received from local endpoints.
no_sent_local_signals	Number of LINX signals sent to local endpoints.
no_queued_bytes	Number of bytes waiting in the receive queue of the endpoint.
no_queued_signals	Number of LINX signals waiting in the receive queue of the endpoint.

A sent LINX signal is shown as queued until the destination application has received it with a `linx_receive()` call (or a `recvfrom()` / `recvmsg()` if using the socket interface directly). When a hunt for a remote endpoint is resolved, the hunt reply will be counted as a “received remote” LINX signal even though the hunt reply itself is not sent over the link. From the LINX endpoints perspective, the hunt reply is received from a virtual endpoint which represents a remote endpoint. The same applies to attach LINX signals from virtual endpoints.

When a LINX endpoint is destroyed, so are the statistics for that endpoint. If an application needs to save the statistics, it should do so before calling `linx_close()` (or `close()` if using the socket interface directly).

8.2 Ethernet CM Statistics

Statistics can be enabled at compile-time for the LINX Ethernet Connection Manager by using the following flags when building LINX:

-DSTAT_SEND_PKTS
: Shows the number of sent packets per link, stored in `/proc/net/linx/cm/eth/send_pkts`

-DSTAT_RECV_PKTS
: Shows the number of received packets per link, stored in `/proc/net/linx/cm/eth/recv_pkts`

-DSTAT_SEND_PKTS
Shows the number of sent packets per link, stored in `/proc/net/linx/cm/eth/send_pkts`

-DSTAT_RECV_PKTS
Shows the number of received packets per link, stored in `/proc/net/linx/cm/eth/recv_pkts`

-DSTAT_SEND_RETRANS
Shows the number of received packets per link, stored in `/proc/net/linx/cm/eth/send_retrans`

-DSTAT_BASIC
Turns on all of the above statistics fields.

9. Where to Find More Information

9.1 Reference Documentation

The reference manuals can be found under the `doc/` directory, as MAN pages in the `man1` - `man8` subdirectories. The linx(7) manual page is the top document. To be able to read the reference documentation with the `man` command in Linux, the path to the LINX `doc/` directory needs to be added to the `MANPATH` environment variable. Alternatively, index.html in the LINX `doc/` directory contains pointers to HTML and PDF versions of the reference manual pages, which have been generated from the man page format files.

Note: The PDF version of all MAN pages are collected in one pdf file, linxmanpages.pdf.

The LINX API is described in the reference manual pages, see linx.h(3) and linx_types.h(3).

How to use the LINX socket interface directly is described in the linx(7) manual page.

The mklink(1), rmlink(1), mkethcon(1), rmethcon(1), mktcpcon(1), rmtcpcon(1), linxstat(1), linxcfg(1) commands and the linxdisc(8) daemon are also described in reference manual pages.

9.2 LINX Protocols

Specifications of all LINX protocols for inter-node communication are found in the separate LINX protocols (HTML) document. (PDF version)

9.3 The LINX Project

The LINX project can be found on the following address:

- https://linux.enea.com/linx/

Email: linx@enea.com

9.4 Other Information

On www.enea.com/linx you can find an overview of LINX, and links to technical white papers and other information.