Linux Automation GmbH lxa-iobus-server

Chris Fiege

Aug 01, 2022

CONTENTS:

This packages provides a daemon which interfaces IOBus-devices from Linux Automation GmbH with test-automation tools like labgrid. IOBus is a CANopen-inspired communications protocol on top of CAN.

This packages provides the following features:

- lxa-iobus-server: This is the central daemon that manages the nodes on the bus. It provides a (human-readable) web interface and a REST API for remote control of the nodes. It also updates the firmware running on the devices on the bus.

- The most recent firmware for all available IOBus devices.

If you want to get in touch with us feel free to do so:

- IRC channel `#lxa` on libera.chat (bridged to the Matrix channel #lxa:matrix.org)

- If our *Troubleshooting* guide doesn't solve your problem or if you found a bug feel free to open an issue on github.

- You can send us an email to info@linux-automation.com.

GETTING STARTED

This chapter describes the steps to set up the `lxa-iobus-server` on your system. Since the server interfaces with real hardware we will first set up your CAN bus and afterwards set up the server itself.
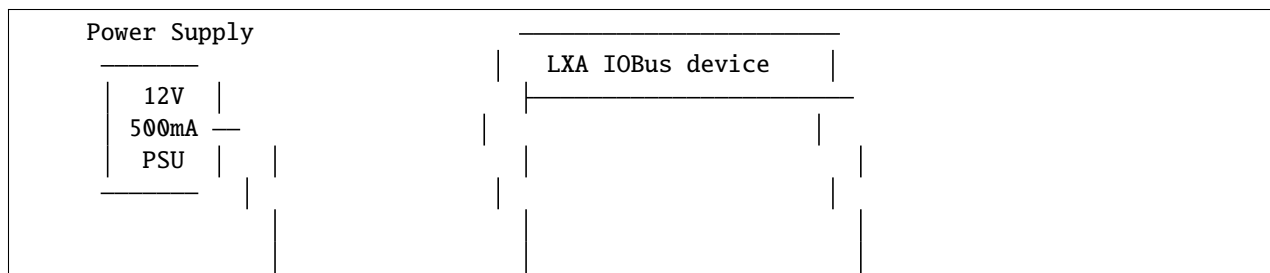
1.1 System requirements

The lxa-iobus-server has been developed to work on a modern Linux-based distribution. Additional to this the following requirements need to be meet to run the lxa-iobus-server:

- Python 3.7 or later
- on Debian: python3-virtualenv
- SocketCAN (The built-in CAN layer in recent Linux Kernels)
- SocketCAN compatible CAN interface
- At least one IOBus device
- git
- `make` for easy setup of the lxa-iobus-server
- optional: `systemd` to setup a service for lxa-iobus-server
- optional: `systemd` >= 239 to bring up your CAN-device on boot
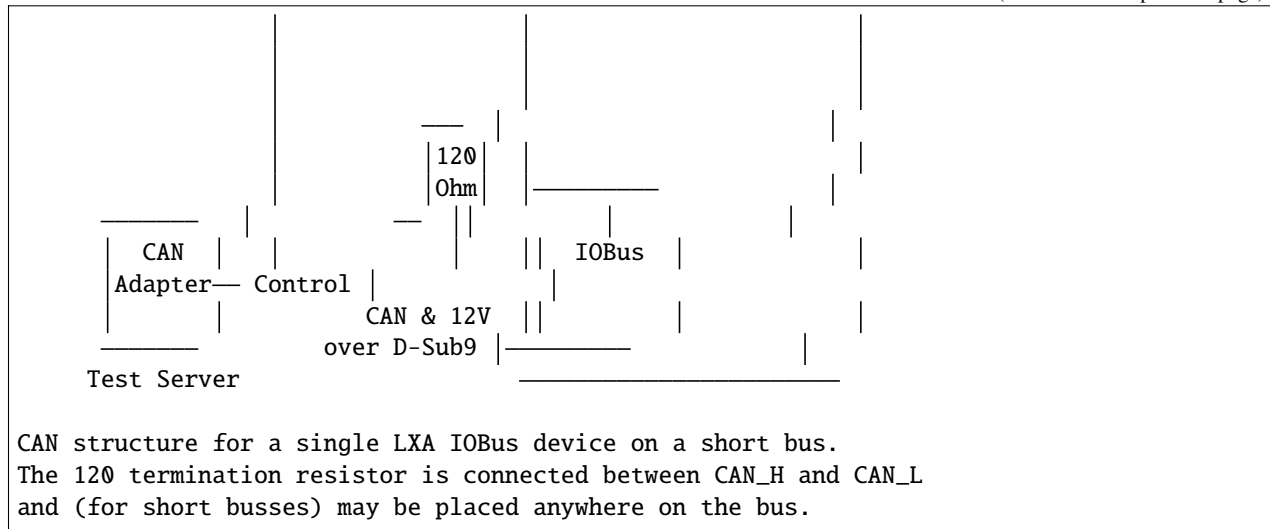
1.2 Hardware Preparations

For the `lxa-iobus-server` to work you need to set up your CAN bus correctly. This chapter shows you how to set up your CAN bus. If you are not familiar with CAN please refer to the chapter *CAN-Bus Introduction* for some basics about CAN.

The following figure shows a minimum CAN-Bus Setup for a single LXA IOBus device:

```
   Power Supply                    _____
                              |   | LXA IOBus device   |
     _____                   |   |_____|
    | 12V  |                  |    |                   |
    | 500mA —                 |    |                   |
    | PSU  |  |               |    |                   |
    |_____|  |               |    |                   |
             |                |    |                   |
             |                |    |                   |
             |                |    |                   |
```

```
                      |              |                      |
                      |              |                      |
                      |              |                      |
                     ___    |        |         ___          |
                    |120|   |        |_____ |           |
                    |Ohm|   |        |         |           |
          _____    |    __  ||         ||  IOBus  |      |
         |  CAN  |   |  |   |        || |         |      |
         |Adapter|— Control |        ||         |      |
         |       |   |       CAN & 12V  ||         |      |
          ‾‾‾‾‾‾‾    |      over D-Sub9 |_____|       |
          Test Server                  ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

CAN structure for a single LXA IOBus device on a short bus.
The 120 termination resistor is connected between CAN_H and CAN_L
and (for short busses) may be placed anywhere on the bus.
```

In this example the LXA IOBus device and the CAN adapter are the only devices on the CAN bus. The Test-Server is the host running the control application and is connected to the CAN bus.

Power for the LXA IOBus device is provided by a 12V DC power supply. The power supply is connected to the power pins on the CAN bus.

A single 120 termination resistor, connecting the two CAN signal lines, is sufficient when the bus length is kept short.

The following chapters give more information on how to build this minimum setup.

1.2.1 Pinout

The following figure shows the common pinout of the D-Sub 9 connector on the LXA IOBus:

Fig. 1: Pinout of the D-Sub 9 Pin connector looking from the outside onto the connector. (Public Domain, from: Wikimedia)

The connector uses the standard pinout for CAN on D-Sub 9 connectors, that is defined in the CANopen standard CiA-303-1 and is used throughout the automotive industry. The following table shows the common pins used on the LXA IOBus:

Table 1: D-Sub 9 CAN Pinout

Pin Number	Name	Internal Function
1	-	*Not connected*
2	**CAN_L**	CAN bus (negative)
3	**CAN_GND**	Connected to system GND
4	-	*Not connected*
5	**CAN_SHIELD**	Can be connected to system GND
6	**POWER_GND**	Connected to system GND
7	**CAN_H**	CAN bus (positive)
8	-	*Not connected*
9	**+12V**	Power Supply

Pins marked as *not connected* are not part of the common LXA IOBus specification.

Note: Check the manual of your LXA IOBus products for their safe working voltage ranges and absolute maximum values on these pins.

Note: The LXA IOBus uses a fixed bitrate of 100 kBits/s for communication. Other bus nodes should allow for at least $\pm 2\%$ bitrate error. See *Bitrate-Intolerant CAN Bus* for an example of how this may cause issues with some CAN-interfaces and how to fix these issues.

1.2.2 Termination resistor and bus topology

Important: Especially in installations with multiple meters of cabling, a clear topology and termination are required for highly reliability.

A CAN bus should be designed as a single line with short stubs connecting the devices to the bus.

The CAN bus needs to be terminated properly. This is usually done using 120 resistors between **CAN_H** and **CAN_L** at both ends of the line, close to the last devices on the bus.

Experience has shown that very short buses (eg. shorter than 0.5m) can be realized with a single termination resistor on the bus and without a strict line topology.

1.2.3 Cabling

For longer distances an unshielded twisted-pair (UTP) cable with 120 differential impedance should be used for the CAN bus. For GND and power supply use wires with a sufficient cross section to keep the power supply and CAN bus common mode voltage in the allowed ranges.

For short busses flat ribbon cables present a cheap and easy-to-install alternative to UTP cabling. Plugs and sockets are available from many manufacturers, for example *L17DEFRA09P* and *L17DEFRA09S* from Amphenol.

1.3 IOBus Server Quickstart

We assume that the linux network interface connected to your CAN bus is `can0`. If your CAN bus has a different name please skip to the next chapter. Make sure you have at least one other CAN device on your bus (e.g. an IOBus device) and that your bus has sufficient termination resistors. If you connect an IOBus device to a currently unmanaged bus (a CAN bus without a running lxa-iobus-server) the network LED on the IOBus device will blink until the node has been initialized.

First: Setup your SocketCAN interface `can0`:

```
$ sudo ip l set can0 down # Deactivate the interface so that the bitrate can be changed
$ sudo ip link set can0 type can bitrate 100000
$ sudo ip l set can0 up # Activate the interface with new bitrate
```

The next step is to download the server software by cloning this repository:

```
$ git clone https://github.com/linux-automation/lxa-iobus.git
Cloning into 'lxa-iobus'...
remote: Enumerating objects: 476, done.
remote: Counting objects: 100% (476/476), done.
remote: Compressing objects: 100% (227/227), done.
remote: Total 476 (delta 257), reused 448 (delta 229), pack-reused 0
Receiving objects: 100% (476/476), 1.04 MiB | 2.48 MiB/s, done.
Resolving deltas: 100% (257/257), done.
```

Now you are able to call `make server` which will create a `python venv` inside the directory and start a server that binds to `http://localhost:8080/`.

```
$ cd lxa-iobus/
$ make server
rm -rf env && \
python3.7 -m venv env && \
. env/bin/activate && \
pip install -e .[full] && \
date > env/.created
Obtaining file:///home/chris/tmp/lxa-iobus
[...]
Successfully installed aenum-2.2.4 aiohttp-3.5.4 aiohttp-json-rpc-0.12.1 async-timeout-3.
→0.1
attrs-20.2.0 backcall-0.2.0 canopen-1.1.0 chardet-3.0.4 decorator-4.4.2 idna-2.10
ipython-6.5.0 ipython-genutils-0.2.0 jedi-0.17.2 lxa-iobus multidict-4.7.6 parso-0.7.1
pexpect-4.8.0 pickleshare-0.7.5 prompt-toolkit-1.0.18 ptyprocess-0.6.0 pygments-2.7.2
python-can-3.3.4 simplegeneric-0.8.1 six-1.15.0 traitlets-5.0.5 typing-extensions-3.7.4.3
wcwidth-0.2.5 wrapt-1.12.1 yarl-1.6.2
. env/bin/activate && \
lxa-iobus-server can0
starting server on http://localhost:8080/
```

After this step the lxa-iobus-server will start to scan the bus for connected IOBus-compatible nodes. Depending on the number of nodes this can take up to 30 seconds. Observe the status of the network LED on your iobus compatible node. Once the node has been initialized by the server the LED stops blinking.

Now navigate your web browser to `http://localhost:8080/`. Your node should be listed under `nodes`. Your lxa-iobus-server is now ready for use.

If you want the server to be started at system startup take a look into the installation section.

1.4 Installation

The permanent installation of the lxa-iobus-server consists of three parts:

1) Clone the repository and create a `python venv` with the installation.

2) Bring up the SocketCAN-device at system start.

3) Setup the lxa-iobus-server and make it start at system start.

1.4.1 Create a python venv

Clone this repository:

```
$ git clone https://github.com/linux-automation/lxa-iobus.git
Cloning into 'lxa-iobus'...
remote: Enumerating objects: 476, done.
remote: Counting objects: 100% (476/476), done.
remote: Compressing objects: 100% (227/227), done.
remote: Total 476 (delta 257), reused 448 (delta 229), pack-reused 0
Receiving objects: 100% (476/476), 1.04 MiB | 2.48 MiB/s, done.
Resolving deltas: 100% (257/257), done.
$ cd lxa-iobus/
```

Create a venv and install `lxa-iobus-server`:

```
$ make env
rm -rf env && \
python3 -m venv env && \
. env/bin/activate && \
pip install -e .[full] && \
date > env/.created
Obtaining file:///home/chris/work/Projects/github/lxa-iobus
[...]
Successfully installed [...]
```

You can now run the `lxa-iobus-server` located in `./env/bin/lxa-ibus-server`.

1.4.2 Setup SocketCAN device with systemd-networkd

In this step `systemd-networkd` is used to set up the SocketCAN device at system startup. If you are not using `systemd-networkd` skip to the next chapter.

This installation method requires you to have systemd with a version of at least 239 on your system and a SocketCAN device must be available.

You can check the status using:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group␣
→default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
[...]
185: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP mode DEFAULT group␣
→default qlen 10
    link/can
```

In this example the SocketCAN device is `can0`.

To setup the interface using `systemd-networkd` copy the rules `80_can0-iobus.link` and `80_can0-iobus.network` from `./contrib/systemd/` to `/etc/systemd/network/`. Make sure to update the `[Match]` sections in both files and the `[Link]` section in the `.link` file to match the name of your SocketCAN device.

These files will do the following:

- Use the SocketCAN device `can0`

- Rename it to `can0-iobus`. Especially on systems with multiple interfaces this makes it a lot easier to identify the interface used for the lxa-iobus-server.

- Set the bitrate to 100 kbit/s.

- Bring the interface up.

To apply this changes restart `systemd-networkd` using `systemctl restart systemd-networkd`. Afterwards make sure your device has been renamed and is up using `ip link`.

1.4.3 Setup SocketCAN device manually

If you are using another way of setting up your network you may skip this step and make sure you meet the following requirements instead:

- Set the bitrate to 100 kbit/s

- Bring the interface up

- Optionally: Rename the interface with the suffix `-iobus`. Especially on systems with multiple interfaces this makes it a lot easier to identify the interface used for the lxa-iobus-server.

1.4.4 Setup lxa-iobus-server

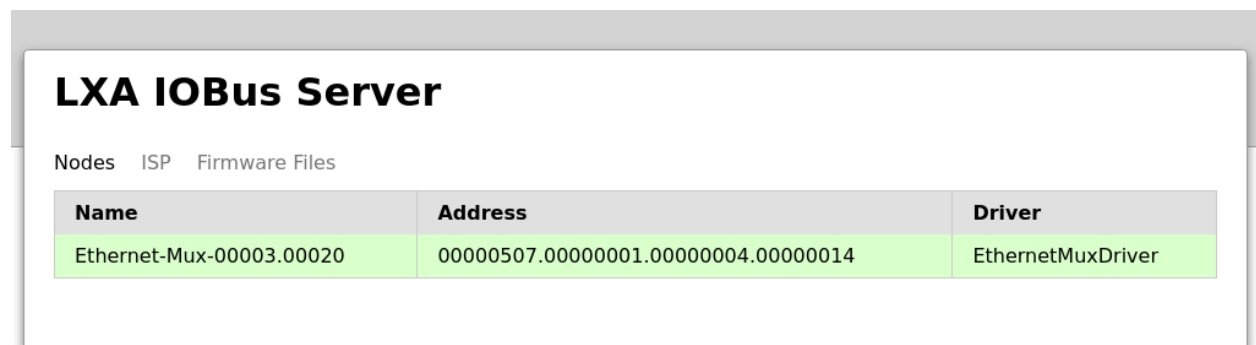In this chapter `systemd` will be used to start the lxa-iobus-server.

To setup a systemd-service use the example `.service`-unit provided in `./contrib/systemd/lxa-iobus.service`. To install the service copy this file to `/etc/systemd/system/`.

Make sure to set the correct SocketCAN interface and path to the `lxa-iobus-server`-executeable in the service file. Make sure you have at least one other CAN device on your bus an that your bus is terminated.

Afterwards the service can be started using `systemctl start lxa-iobus.service`. If no errors are shown in `systemctl status lxa-iobus.service` the web interface should be available on `http://localhost:8080`.

1.5 Usage

Once started the server should start enumerating devices connected to the bus. Visit the IOBus Server web interface at http://localhost:8080/ for a list of detected IOBus devices:

Fig. 2: List of nodes in the IOBus Server web interface

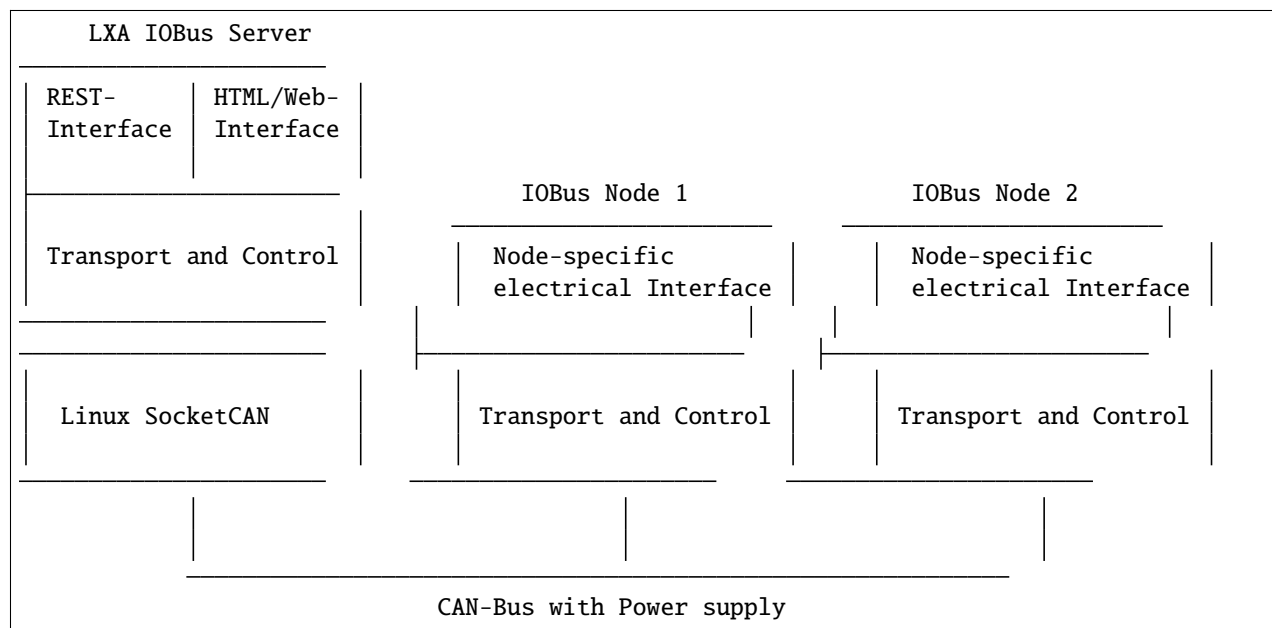Click on a node for detailed information about this node and the options to toggle the outputs.

SYSTEM ARCHITECTURE

The `lxa-iobus-server` is a gateway between a lab automation software (e.g. labrid) and the actual LXA IOBus devices connected to the bus.

This chapter gives introduces the architecture used to in the LXA IOBus system.

2.1 Network Layers

The following figure shows the structure of the LXA IOBus system:

```
  LXA IOBus Server
 _____

| REST-    | HTML/Web-|
| Interface| Interface|
|_____|          |              IOBus Node 1              IOBus Node 2
|          |          |           _____           _____
|                     |          |                         |
| Transport and Control          |  Node-specific        | |  Node-specific        |
|                     |          |  electrical Interface | |  electrical Interface |
|_____|          |_____|       | |_____|       |
|                     |          |                        |
|                     |          |                        |
| Linux SocketCAN     |          | Transport and Control  | | Transport and Control  |
|                     |          |                        |
|_____|          |_____| |_____|
         |                                |                          |
         |                                |                          |
         |_____|_____|
                      CAN-Bus with Power supply
```

- **REST-Interface**: Using this communication interface external software is able to interact with the nodes connected to the IOBus.

- **Web-Interface**: This interface provides the information available on the REST-Interface in a human-readable form.

- **Transport and Control**: This part implements the CANopen-inspired protocol and keeps track of the current state of the bus and connected devices.

- **Linux SocketCAN**: The LXA IOBus Server uses SocketCAN to interfact with the CAN-bus.

- **Node-specific electrical interface**: Every LXA IOBus node has an application-specific specialiced electiral interface that is designed to perform different automation tasks.

- **CAN-Bus**: This is the actual electrical interface that connects server and nodes. This is the same CAN bus inteface you may know from many automotive applications.

CAN BASICS

CAN and CANopen are, when compared to modern Ethernet and IP, quite simple protcols. Most software developers are however more familiar with Ethernet and IP and less with CAN and CANopen.

This chapter tries to give a short introduction into CAN and CANopen and tries to focus on topics that are relevant for the operation of the LXA IOBus system.

3.1 CAN-Bus Introduction

The CAN bus was developed to connect multiple small computers in a reliable way. A very common scenario is to connect multiple control units inside a road vehicle: There is a lot of noise and possible bad connectors on the bus but CAN must still be able to carry information from one control unit to the others.

CAN is a low-speed and low-bandwith bus: A single message can only carry up to 8 bytes of payload. The maximum symbol-rate on the bus is 1 M/s.

3.1.1 Messages not Addresses

In common computer networks (e.g. Ethernet) every node has a static address (e.g. MAC-address). Information in such network is usually sent from one address to another address on the same network segment, e.g. each message contains one source address and one destination address. (There are exception but let's leave those aside here.)

On a CAN bus messages are published to a message-id, where a single message-id can be consumed by one or many nodes on the bus. It is even possible that multiple nodes on the bus publish information to the same message-id! In modern terms: A CAN bus follows the publish / subscribe paradigm where the message-ids are the topics. (Beware that there is no central *broker* here. Subscription is done using ingress filters on the receiving nodes.) This means CAN messages only contain a destination address and no source address and the destination address may be shared between multiple nodes.

CAN itself only defines the length of a message-id (11 or 29 bit depending on the addressing scheme used) but not how these message-id should be used. In a vehicle all message-ids are pre-allocated by the manufacturer: For every message-id a structure defining the contents of the payload is defined and shared between all control units.

In the case of the LXA IOBus the meaning of the message-ids and the payload is defined by a CANopen-inspired protocol.

3.1.2 Reliable transmission

CAN makes a good amount of effort to make sure all nodes on the bus share a common understanding of the information transmitted: Every message contains a checksum to make sure no bit-errors occour on the bus.

Additionally all nodes on the bus do a handshake for every message that ensures that either all or no node received the message.

To archive this every receiving node on the bus sends an acknowledge-flag to the bus once the complete frame has been received and the checksum is correct.

If the received checksum is not correct or another receive-error occurs an error-message is send to the bus. If an error message is received the current message is discarded in the MAC - before forwarding it to the higher level.

Let's take a look at the following scenarios:

- **No other node on the bus:** The node sends a message on the bus. Since there is no other node on the bus the sender will not receive an ACK. The sending node will assume that the message has not been received by any node. This can happen if there are only two nodes on a bus and one is not powered or disconnected due to a faulty conenction.

- **Two other nodes on the bus and the checksum is OK**: Both receiving nodes send an acknowlege after the end of the message. All three nodes assume that every other node has received the message correctly. This message is delivered to the next higher level.

- **Two other nodes on the bus and one receives an invalid checksum:** In this case the node receiving the invalid checksum will generate en error-frame instead of the acknowledge. Both other nodes will discard the message. The sending node will probably re-transmit the message.

SOFTWARE AND FIRMWARE UPGRADES

4.1 Upgrading the lxa-iobus-server

Upgrading the LXA iobus-server is done by installing a new version of the Python package.

Before installing a new version of the server stop the currently running lxa-iobus-server. If you are using the provided systemd-service run:

```
$ sudo systemctl stop lxa-iobus.service
```
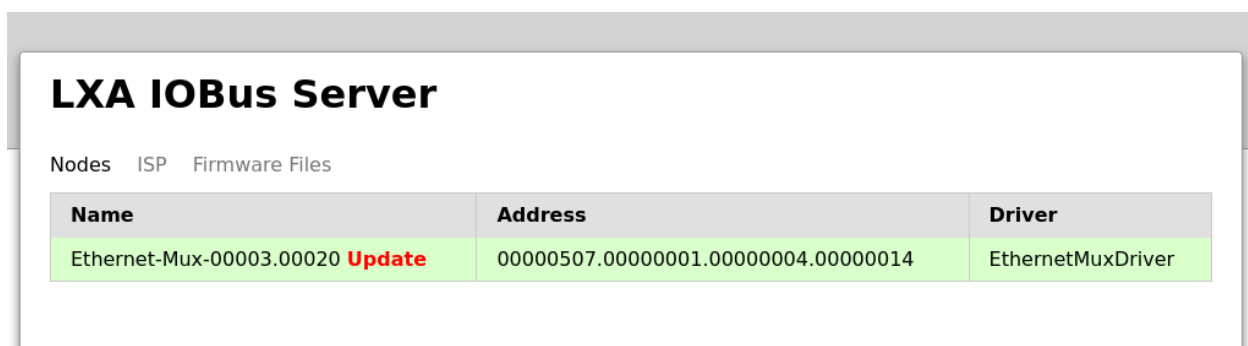
Afterwards you can build a new `env`:

```
$ cd /path/to/the/lxa-iobus/repository
$ git pull
$ make clean
$ make env
```

Now you can start your service again:

```
$ sudo systemctl start lxa-iobus.service
```

4.2 Bundled Firmware Upgrades

The `lxa-iobus-server` software comes bundled with the latest firmware binaries for the IOBus devices. The availability of new firmware upgrades for devices is indicated in the Web-Interface by a red **Update** text in the node list:

Fig. 1: List of nodes. Devices "00003.00020" has a pending firmware upgrade.

A firmware upgrade is performed by selecting the corresponding entry in the node list and clicking the *Update to …* button at the top:
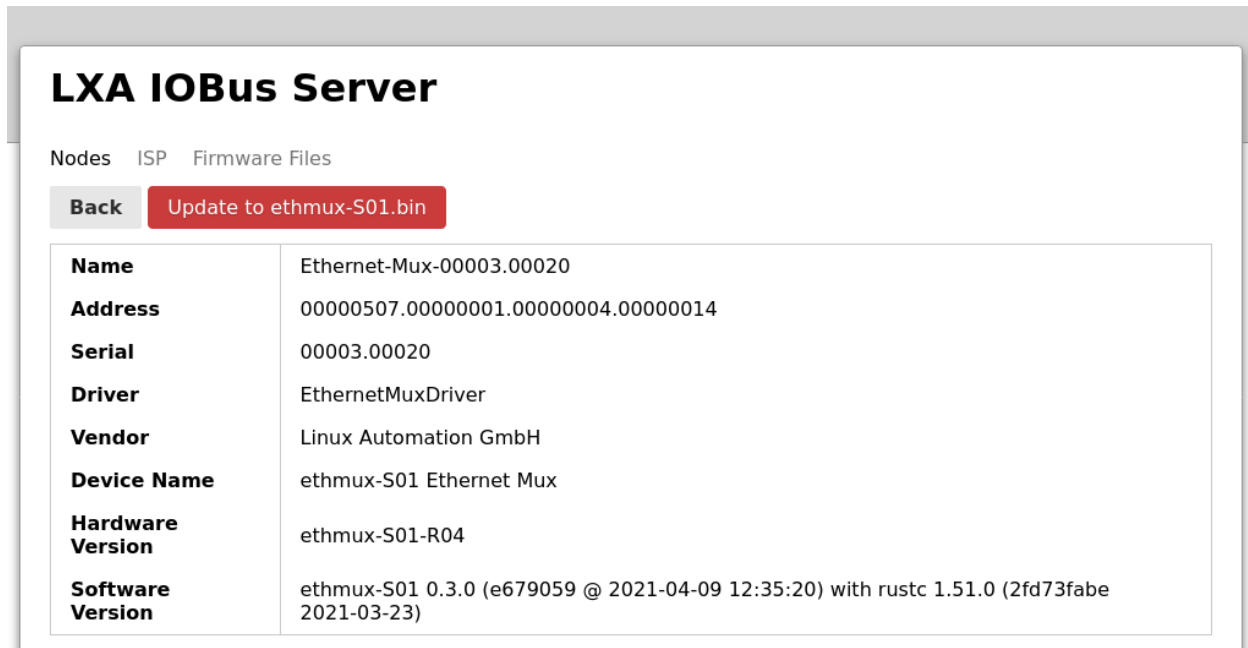

Fig. 2: Pressing the "Update to …" button initiates a firmware upgrade.

Clicking the button takes you to the "*ISP*" tab of the web interface where a log of the flashing progress is shown:
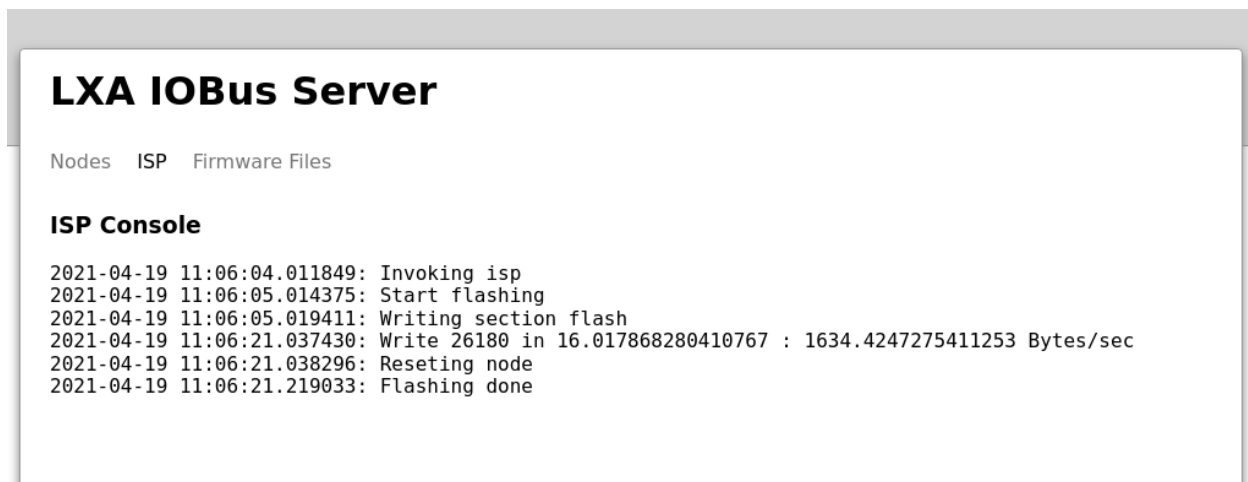

Fig. 3: A successful firmware flashing process terminates with the log message "Flashing done".

Once the flashing is compled you can return to the node information by selecting the "Nodes" tab at the top.

4.3 Firmware Upgrades using the danger-zone button

The `lxa-iobus-server` allows you to flash arbitrary files into the firmware section of any node. As it is generally a bad idea to flash arbitrary firmware into a device this feature is disabled by default.

> **Warning:** With this option you can damage your IOBus devices.

If you intend to use this feature (e.g. to flash a beta-firmware or if you want to deploy your own firmware) you have to use the `--allow-custom-firmware` switch on the command line, e.g.:

```
$ lxa-iobus-server --allow-custom-firmware --firmware-directory firmware/ can0
```

The additional command line switch `--firmware directory <dir>` allows to specify the directory in which uploaded firmware files are stored. If you omit this switch the default directory `firmware/` in the project root is used.

With the `--allow-custom-firmware` switch enabled two new features are available:

- The *Firmware Files* view now contains the option to upload and delete custom firmware files.

- Every node view now has the option to select an arbitrary file to flash.

To flash an arbitrary file first upload the binary using the *Firmware Files* view:

Fig. 4: The *Firmware Files* view. The files listed under *Upstream Firmware files* have been shipped with the server. The files listed under *Local Firmware Files* have been uploaded by the user.

New files can be uploaded using the *Browse* and *Upload* -buttons. Here a file called `candleLight_fw.bin` has been uploaded by the user.

Afterwards this file can be flashed to an arbitrary node in the *Nodes* view:

Select the correct file and start the transfer using the *Flash* -button.

LXA IOBus Server

Nodes ISP Firmware Files

Back

Name	Ethernet-Mux-00003.00023
Address	00000507.00000001.00000004.00000017
Serial	00003.00023
Driver	EthernetMuxDriver
Vendor	Linux Automation GmbH
Device Name	ethmux-S01 Ethernet Mux
Hardware Version	ethmux-S01-R04
Software Version	ethmux-S01 0.3.0 (e679059 @ 2021-04-09 12:35:20) with rustc 1.51.0 (2fd73fabe 2021-03-23)

Locator

Locator

Outputs

SW

Inputs

SW_IN	0
SW_EXT	0

ADCs

AIN0	0.000
VIN	12.189

Flash

Be careful! This dialog lets you flash whatever you want, without checking if the given file is valid or not.
You can brick your device here.
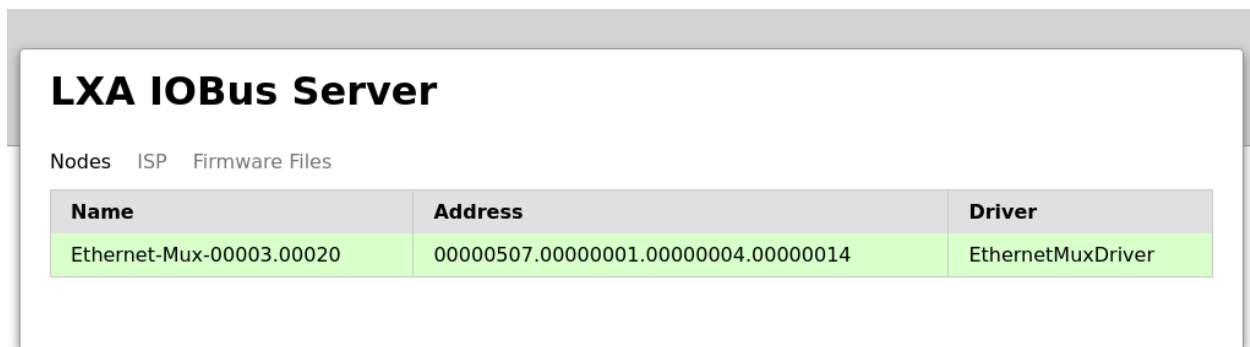
upstream/ptxtac-S03_CAN_GPIO.bin ⌄   Flash

Fig. 5: The section *Flash* for this nodes lists all firmware files available.

USING THE WEB-INTERFACE

Note: Make sure you have installed the control software as described in *Getting Started*.

Once started the server should start enumerating devices connected to the bus. Visit the IOBus Server web interface at http://localhost:8080/ for a list of detected IOBus devices:

LXA IOBus Server

Nodes   ISP   Firmware Files

Name	Address	Driver
Ethernet-Mux-00003.00020	00000507.00000001.00000004.00000014	EthernetMuxDriver

Fig. 1: List of nodes in the IOBus server web interface

More options to control a particular node are available by clicking on a line in the list:

Depending on the type of node different inputs and output are available. Read the manual for the specific node for more information on specific options.

The following fiels are available for every node:

- **Name:** The logical name for this device. This name is used to identify this node on the *REST* interface.

- **Address:** CANopen LSS address for this device.

- **Serial:** Serial number of the device.

- **ADC VIN:** This ADC channel shows the current supply voltage on the IOBus. This voltage should be between 9 and 13V.

- **Locator:** Toggling the Locator LED, that can be used to find a particular device in a lab, is done by clicking the *Locator* button in the interface. The *Locator* indicator can also be used in the opposite direction, as pushing the locator button on the Ethernet-Mux also toggles the state of the on-board LED and the one shown in the web interface:

Fig. 2: The IOBus server node control interface

Fig. 3: List of nodes. Device "00005.00009" has an active Locator.

USING THE REST-API

The actions available through the web interface can alternatively be performed programmatically using the *REST API* provided by the server:

```
# Get a list of available nodes:
$ curl http://localhost:8080/nodes/
{"code": 0, "error_message": "", "result": ["Ethernet-Mux-00003.00020"]}

# Get a list of pins on a device:
$ curl http://localhost:8080/nodes/Ethernet-Mux-00003.00020/pins/
{"code": 0, "error_message": "", "result": ["SW", "SW_IN", "SW_EXT", "AIN0", "VIN"]}

# Get the current status of a pin:
$ curl http://localhost:8080/nodes/Ethernet-Mux-00003.00020/pins/SW/
{"code": 0, "error_message": "", "result": 0}

# Set the status of a pin:
$ curl -d "value=0" -X POST http://localhost:8080/nodes/Ethernet-Mux-00003.00020/pins/SW/
{"code": 0, "error_message": "", "result": null}

# Toggle the Locator LED:
$ curl -X POST http://localhost:8080/nodes/Ethernet-Mux-00003.00020/toggle-locator/
{"code": 0, "error_message": "", "result": null}
```

TROUBLESHOOTING

This section lists common problems and possible solutions. If you experience other problems or would like to add a solution for a problem feel free to open an issue in our Github project or send us an email to info@linux-automation.com.

For assistance you can also join our IRC channel `#lxa` on libera.chat (bridged to the Matrix channel #lxa:matrix.org).

7.1 Bitrate-Intolerant CAN Bus

Problem: The host-side CAN-interface sends an error-frame for every CAN packet sent by the Ethernet-Mux.

The CAN-Bus protocol is designed to allow bitrate offsets of a few percent between bus nodes. This is especially relevant when a bus contains nodes without precise crystal-based clock sources. Synchronization is performed on the receiving side of a CAN-frame by monitoring the actual and expected timing of bit transitions seen on the bus, and adjusting the bit-sampling of subsequent bits accordingly.

The generation of CAN-timings is based on a base clock, that is sub-divided using counters, to determine the sample points for reception and the signal transition points for sending. These counter timings make use of units of time called time quanta `tq`, on Linux these time quanta are given in nanoseconds.

One parameter that is specified in terms of time quanta is the synchronization jump width (`sjw`), a parameter determining the maximum amount of bitrate synchronization performed during reception of a CAN-frame. Currently SocketCAN initializes every device with a synchronization jump width (`sjw`) of 1 time quantum.

As the length of a time quantum `tq` varies widely between different CAN-controllers this results in maximum amount of bitrate-synchronization performed by default also varying widely between CAN-controllers. On some CAN-controllers the amount of synchronization allowed by the default setup is not sufficient to use LXA IOBus devices, leading to frames being rejected by the CAN-controller.

Solution: Use a `sjw` relative the other bit-timings instead of a fixed value of 1.

LXA IOBus devices are tested at a `sjw` of 5% of one bit-time. To determine the current bit-timings the `can0` interface should first be configured to the desired bitrate of 100 kbit/s, e.g. by using systemd-networkd. The resulting bit timings are calculated automatically by the Linux kernel and can then be displayed using the `ip` command:

```
$ ip --details link show can0
5: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP mode DEFAULT group␣
→default qlen 10
  link/can  promiscuity 0 minmtu 0 maxmtu 0
  can state ERROR-PASSIVE (berr-counter tx 128 rx 0) restart-ms 100
    bitrate 100000 sample-point 0.875
    tq 50 prop-seg 87 phase-seg1 87 phase-seg2 25 sjw 1
    peak_canfd: tseg1 1..256 tseg2 1..128 sjw 1..128 brp 1..1024 brp-inc 1
    peak_canfd: dtseg1 1..32 dtseg2 1..16 dsjw 1..16 dbrp 1..1024 dbrp-inc 1
    clock 80000000 numtxqueues 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
```

Shown in line 6 are the timing-parameters `tq`, `prop-seg`, `phase-seg1`, `phase-seg2` and `sjw`. One bit-time consists of `1 + prop-seg + phase-seg1 + phase-seg2` time quanta. The `sjw` should thus be adjusted to a value of `sjw = 0.05 * (1 + prop-seg + phase-seg1 + phase-seg2) = 10`.

The interface can be re-configured accordingly using the command:

```
$ ip link set can0 type can tq 50 prop-seg 87 phase-seg1 87 phase-seg2 25 sjw 10
```

Note: All other values but `sjw` are copied from the status output above.

CONTRIBUTING

Thank you for thinking about contributing to LXA IOBus Server!

Changes should be submitted via a Github pull request.

8.1 Developers Certificate of Origin

This project uses the Developer's Certificate of Origin 1.1 with the same process as used for the Linux kernel:

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Then you just add a line (using `git commit -s`) saying:

Signed-off-by: Random J Developer <random@developer.example.org>

using your real name (sorry, no pseudonyms or anonymous contributions).

NINE

LIST OF ABBREVIATIONS

API

Application Programming Interface. An interface between software components.

ISP

In System Programmer A tool that allows uploading new firmware to a device without requiring disassembly or other invasive actions.

LSS

Layer Setting Services. A CANopen protocol that allows the configuration of individual node ids and communication bitrates.

REST

Representational State Transfer. A paradigm for the design of *API*s. The `lxa-iobus-server` provides a REST API on top of HTTP.

INDICES AND TABLES

- genindex
- modindex
- search