IN3032UG: Ethernet Debugger User Guide

Website: https://intona.eu

Contents

1 Introduction **4**
- 1.1 Features . . . 4
- 1.2 Requirements . . . 4
- 1.3 Restrictions . . . 4
- 1.4 Firmware Changelog . . . 5
  - 1.4.1 Firmware 1.06 . . . 5
  - 1.4.2 Firmware 1.00 . . . 5
- 1.5 Host Tool Changelog . . . 5
  - 1.5.1 Host tool git master . . . 5
  - 1.5.2 Host tool v1.2 . . . 6
  - 1.5.3 Host tool v1.1 (686fe4f) . . . 6
  - 1.5.4 Host tool v1 (f5eed9c) . . . 6

2 Hardware Setup **7**
- 2.1 LED Meaning . . . 7
  - 2.1.1 Port LEDs . . . 7
  - 2.1.2 Main LED . . . 7

3 Software Installation **8**
- 3.1 Linux . . . 8
- 3.2 macOS . . . 8
- 3.3 Windows . . . 8
- 3.4 Verifying Device Access . . . 9
- 3.5 Wireshark extcap Setup . . . 9
  - 3.5.1 Linux, macOS . . . 9
  - 3.5.2 macOS (app bundle) . . . 10
  - 3.5.3 Windows . . . 10
- 3.6 Firmware Update . . . 10
  - 3.6.1 Unix-like . . . 10
  - 3.6.2 Windows . . . 11

4 Capturing **12**
- 4.1 Wireshark extcap . . . 12
  - 4.1.1 Capturing Options . . . 12
  - 4.1.2 Wireshark extcap Toolbar . . . 12
- 4.2 Directly Starting Wireshark from Host Tool . . . 12
- 4.3 Statistics . . . 13
- 4.4 Capturing to a File . . . 13
- 4.5 Selecting the Device . . . 13
- 4.6 Configuring the Buffer Size . . . 13

5 Other Features **14**
- 5.1 PoE Passthrough . . . 14
- 5.2 PTP Timestamps . . . 14
- 5.3 Supported Command Line Options . . . 14
- 5.4 Interactive Command Line . . . 15
- 5.5 Scripting . . . 15
- 5.6 Supported Commands . . . 15
- 5.7 IPC Interface . . . 16
  - 5.7.1 Windows . . . 16
  - 5.7.2 UNIX (Linux/Mac) . . . 16
  - 5.7.3 Protocol . . . 17
- 5.8 Identify Function . . . 17
- 5.9 Latency Tester . . . 18

5.9.1 Introduction . . . 18
5.9.2 Instructions . . . 18
5.10 Packet Injection . . . 21
5.11 Blocking Ports . . . 22
5.12 Packet Disruption . . . 22
5.13 MDIO Access . . . 22
5.13.1 Raw MDIO access . . . 22
5.13.2 Changing PHY Ethernet Speed . . . 23
5.14 Device Settings . . . 23

6 Known Problems **23**

7 Further Readings **24**
7.1 White Papers . . . 24
7.1.1 Ethernet Debugger Timing . . . 24
7.2 Statements . . . 24
7.2.1 Letter of Volatility . . . 24

1 Introduction

The Intona Ethernet Debugger is a device to capture packets between two Gigabit Ethernet devices. It provides two ethernet ports, and each port forwards all traffic to the other port, as well as to a PC connected via USB. The intended purpose is low level debugging of anything above the ethernet physical layer, mainly using Wireshark and similar protocol analyzers. It helps when developing your own protocols layered on top of ethernet, developing your own MAC, or just for observing what is going on on your network.

1.1 Features

This device can log complete ethernet packets as received by the PHY. There is no processing of captured packets – preamble, SFD, and FCS are all left intact. Packets with incorrect CRC sums are not discarded. Ethernet packets which violate the specification are captured as far it is possible. Some normally invisible low level details are explicitly logged, such as interpacket gaps and CRC errors. Jumbo frames (ethernet packets longer than 1500 bytes) are supported and fully captured up to 16KB size.

Capture output is directly streamed to the PC. There is no kernel device driver. The device is accessed through a libusb userspace driver. You do not necessarily need elevated privileges. Installing the device will not destabilize your system. In particular, the device is not exposed as network device. This has the advantage that your OS will not mess with it. Neither will it attempt to drop or filter packets received through it, nor will it attempt to send random packets to it (ARP etc.). The latter would show up in Wireshark, and confuse your development efforts.

Capture can be directly started from Wireshark (if installed correctly). The userspace driver also provides a command line interface, which can be used to access advanced feature. An IPC interface is provided for use cases like scripting.

The debugger can block packets in one or both directions, corrupt packets, inject new packets. This is interesting for development and security research. (For example, you can test resilience of your ethernet connected device or software against random packet drops, test its behavior on flooding, or implement a network stack fuzzer.)

There are many other features. See Other features section.

1.2 Requirements

The software works on Windows, Linux, and macOS. We provide an installer for Windows. Windows 10 64 bit is required, but Windows 7 may work as well. For macOS, a homebrew tap is provided[1]. For Linux, source code and build instructions are provided[2], which should work on any Linux distro.

USB 3.0 or later host and cable are recommended. USB 2.0 may work in low bandwidth scenarios. Using an USB hub, and/or connecting multiple USB devices to an USB hub/host may reduce the maximum bandwidth at which capture is possible without capture overflows.

1.3 Restrictions

Ethernet is intercepted by putting two PHYs between the two ports. There is no direct connection between the ethernet TX/RX wires of the ports. Each PHY negotiates the ethernet connection separately. No link can be established without USB power.

[1] https://github.com/intona/homebrew-ethernet-debugger
[2] https://github.com/intona/ethernet-debugger#build-instructions

Old firmware (before 1.06) requires the user to set the ethernet speed manually if the devices connected to the debugger's ports negotiated different ethernet standards (for example 100 MBit vs. 1 Gb).

The PC needs to be fast to capture at full speed. Capturing in real time with maximum ethernet bandwidth directly to Wireshark or a slow hard disk may not be possible. (This is due to host performance problems outside of our control.) Packet buffer overflows should be expected when operating near maximum bandwidth. There is no hardware packet filter.

1.4 Firmware Changelog

The firmware version is reported in the bcdDevice field in the USB device descriptor, or can be determined by using the "hw_info" command in the host tool.

1.4.1 Firmware 1.06

Updating to host tool v1.2 is recommended.

Bug fixes

- Fix device becoming unable to capture packets on FIFO overflows

Potential incompatibilities and problems

- This is compatible down to host software v1, but host software v1.2 is recommended; there may be strange behavior with host software v1 in certain cases
- The device accesses some MDIO registers automatically now, instead of leaving the host software in full control

New features

- Add autospeed mode (do not require user to manually adjust each PHY's speed if they have negotiated different speeds)
- The autospeed setting is persistently stored on the device
- Improved packet timestamp accuracy

1.4.2 Firmware 1.00

- Initial release.

1.5 Host Tool Changelog

The host tool's version can be determined with the "hw_info" command, or simply running "nose --version".

Note: you can also look at the public git repository's commit log.

1.5.1 Host tool git master

Unreleased, but publicly available. (Also picked up by Homebrew formulae.)

May be referenced as host tool v1.3 in this document.

Bug fixes

- Correct doubled injector packet count reported by hw_info command

New features

- Latency tester feature

- Tab completion (if built with readline; dysfunctional on Windows)
- New options: --cmd, --run, --exit-on, --exit-timeout

1.5.2 Host tool v1.2

Updating to firmware 1.06 is recommended.

Potential incompatibilities and problems

- Firmware update is now invoked differently
- Changes to inject and disrupt command parameters

Bug fixes

- In Wireshark extcap mode, terminate properly if no packets were captured

New features

- Full support for firmware 1.06
- Slightly improved link status reporting
- Extend capabilities of inject and disrupt commands
- Change USB device name format and include device address (allows distinguishing multiple devices connected to chained hubs)
- Accept device serial number as device name and report it via extcap to Wireshark
- Add quoting/escaping and named arguments to command parser
- Perform more parameter validation in command/option parser
- Unify PHY/port names to A/B/AB/none (a mix of conventions was used before)

1.5.3 Host tool v1.1 (686fe4f)

- Add ”speed” convenience command
- Improve capture stats log behavior
- Note that this used to be untagged in the public git repository

1.5.4 Host tool v1 (f5eed9c)

- Initial release
- Note that this used to be untagged in the public git repository

2 Hardware Setup

The ethernet debugger needs to be powered via USB to forward any packets. Without power, communication between the two ports is blocked. Make sure USB cable and host port are at least USB 3.0 compliant. If you use USB hubs or USB isolators, make sure they all support USB 3.0. USB 2.0 will work, but will provide degraded functionality, as USB 2.0 bandwidth is lower than that of Gigabit Ethernet.

Attach the ethernet cables to the ports. Make sure you are breathing regularly. As soon as both port LEDs indicate a connection, and the negotiated ethernet speed matches between both ports, communication is possible.

2.1 LED Meaning

2.1.1 Port LEDs

The LED in use indicates the ethernet speed mode. If packets are sent or received, the corresponding LEDs will blink.

Speed	LED L	LED R
1000 MBit	off	on
100 MBit	on	on
10 MBit	on	off
no link	off	off

Note that it's possible for one port to have a link, while the other port does not. They also can have mismatched speed. In both cases, capture will not work. 10 MBit mode is not supported.

2.1.2 Main LED

Normally, the main LED should be blue. It will blink if capturing is in progress. The following states exist:

LED state	Meaning
Blue	USB super speed connection is up
Green	USB high speed connection is up (slow, but functional)
Cyan	USB power only (will not work)
Blue blinking	capturing at USB super speed is in progress
Green blinking	capturing at USB high speed is in progress (will drop packets)
Blue/green blinking	the blink_led command is being used (only for a short moment)
Red	initial bootloader failed (probably flash problem)
Red blinking	bootloader failed (probably flash problem)
Off	low level bootloader/firmware crash, or no USB power
Red/yellow blinking	fatal higher level firmware error
Red/blue or Red/green blinking	firmware update failed; running factory firmware version

When you experience problems, it is useful to describe the LED state exactly in support requests, even if the observed variant is not listed above.

3 Software Installation

The software consists of a host tool, called ”nose”. It performs the following roles:

- userspace driver for the device
- Wireshark integration via Wireshark extcap
- command line tool for explicit access

3.1 Linux

No binaries or packages are provided. You can build it from the public git repository. Binary builds or packages for popular Linux distributions may be provided if there is enough demand.

- build the host tool from the git repository (see https://github.com/intona/ethernet-debugger#build-instructions)
- create a symlink for Wireshark extcap (see Wireshark section for details)

You may need to install an udev rule to get access to the USB device as normal user:

```
sudo cp udev.rules /etc/udev/rules.d/50-intona-ethernet-debugger.rules
sudo udevadm trigger
```

3.2 macOS

No binaries are provided. Hence the software is provided as source code, you can automatically download and build it using Homebrew.

```
brew install --HEAD intona/ethernet-debugger/nose
```

Homebrew is a 3rd party project for installing freely available software on macOS. See https://brew.sh/ for details and how to install Homebrew itself.

You need to setup Wireshark extcap manually. See below.

We decided to not provide binaries for macOS because it is quite impossible to deliver unified, stable executables that will work on various releases of both software API and CPU types. Think on the platform change to ARM. This is no issue when compiling from sources using Homebrew.

3.3 Windows

- make sure that Wireshark is installed before you proceed
- double-click the installer
- press next a lot of times

It does not matter whether the device is connected during installation. In fact, the installer does not try to access the device at all.

You can reinstall any time. Updating Wireshark might remove the Ethernet Debugger Wireshark integration. Reinstalling the Ethernet Debugger will restore it.

3.4 Verifying Device Access

A simple way to verify whether the software works is by simply running the host tool. It is a command line program. On Windows, double-clicking **nose.exe** will open a terminal window, while on Unix, you need to open a terminal window manually, and then run **nose** in it.

If the installation succeeded, and the device is connected, you should see the following:

```
Device 2:3:7 opened.
PHY A: link=down speed=0MBit
PHY B: link=down speed=0MBit
Warning: no link.
```

The example above has the device on USB bus 2, port 3, device address 7.

If the device could not be found or accessed, the following is shown:

```
No devices found.
```

You can stop the host tool by closing the terminal or by entering the ”exit” command.

3.5 Wireshark extcap Setup

Wireshark is the recommended way to use the Ethernet Debugger. It is 3rd party software and not developed by Intona. Download and install it from Wireshark’s website: https://www.wireshark.org/download.html

Normally, the Windows installation procedure installs our host tool as a Wireshark ”extcap”. In short, ”extcap” allows external programs (such as our host tool) to provide a capturing source. See the Wireshark extcap section for details. If the host tool is not correctly installed as extcap source, you will not be able to start capture from the Wireshark GUI (but other methods of capturing will still work.)

The host tool supports extcap directly via special command line parameters. It must be located in the ”extcap” sub directory within the Wireshark installation directory or the user’s Wireshark configuration directory. The paths depend on the operating system and the Wireshark installation location.

You can confirm whether it’s installed correctly by opening the Wireshark about dialog, and switching to the ”Plugins” tab. There should be an entry named ”nose”.

Old Wireshark versions
The non-global/user-specific extcap paths below require at least Wireshark 3.1.1. Older releases support global paths only.

3.5.1 Linux, macOS

It is recommended to create a symlink to the host tool. The global path is something like /usr/lib/wireshark/extcap/ or /usr/lib/x86_64-linux-gnu/wireshark/extcap/. The exact path depends on the Linux distro. The user-specific path is usually **~/.config/wireshark/extcap/**.

The following should install it locally, assuming ”nose” is already installed:

```
mkdir -p ~/.config/wireshark/extcap/
ln -s `which nose` ~/.config/wireshark/extcap/nose
```

3.5.2 macOS (app bundle)

The following is useful if you want to install the extcap globally, or on older Wireshark versions, assuming "nose" is already installed via Homebrew:

```
ln -s `which nose` /Applications/Wireshark.app/Contents/MacOS/extcap/nose
```

The **/Applications/Wireshark/** part of the path can be different, depending on where exactly Wireshark is installed. Check the Wireshark about dialog ("Folders" tab) if you are unsure.

You need to start Wireshark at least once before you run the above command. Otherwise, macOS may show a security warning, and it won't work.

3.5.3 Windows

Since Windows does not support symlinks properly, it is recommended to create a .bat file in the Wireshark extcap sub-directory. The installer creates a file named **intona-ethernet-debugger.bat** with the following contents (assuming default paths and English locale):

```
"C:\Program Files\Intona\Ethernet Debugger\nose.exe" "%*"
```

This "redirects" all invocations to the actual installation location.

The user-specific path is **C:\Users\USERNAME\AppData\Roaming\Wireshark\extcap**. Replace **USERNAME** with the actual username. You may need to create the last component of the path. The installer does not try to use this.

Updating Wireshark tends to remove and recreate the Wireshark installation directory. This will also remove the **intona-ethernet-debugger.bat** file created by the Ethernet Debugger installer. You could run the installer again to fix this. Manually moving the .bat to the Wireshark user-specific configuration path mentioned above avoids this.

3.6 Firmware Update

Firmware updates may be required to get new features and to apply bug fixes. Normally they are not necessary. Applying such an update must be done explicitly. The update process rewrites the device's flash memory, and should not be interrupted. Make sure the device is connected via USB 3.0, as the update will take a long time with USB 2.x. Firmware downloads are available here[3].

3.6.1 Unix-like

Plug in the device, and ensure it's using USB 3.0. Then run the following command on the terminal:

```
nose --firmware-update Downloads/firmware.dat
```

Where **Downloads/firmware.dat** is the path to the firmware binary you downloaded.

If the firmware file is accepted, and the device is accessible, something like this will appear:

[3] https://www.intona.eu/products/ethernet-debugger#downloads

```
Firmware file: version 1.06
Select firmware update action:

  Choice   Address   Serial           Firmware version
 ------------------------------------------------------
  4        2:7:12    08900037         1.06
 ------------------------------------------------------
  a        Update all devices with outdated firmware
  b        Force update all devices (dangerous)
  c        Do nothing and exit
 ------------------------------------------------------

Enter your choice:
```

If you type 4 followed by the enter key, the device 08900037 will be updated. The tool will exit when the update is finished or aborted. On success, the device restarts on its own, and runs the new firmware immediately.

You can confirm successful update by comparing the device version number (**bcdDevice**) with the version indicated by the update. If the device comes up and blinks red, retry the update, or if the tool fails again, contact support.

If the --device option is provided, the tool will update the given device without asking for confirmation. --firmware-update-all will update all detected devices without asking for confirmation. In both cases (at least with v1.2) the firmware is written only if it's newer than the firmware on the device, unless --firmware-update-force is provided.

Old host tool versions (before release v1.2) do not ask for a choice, but start the update with the first device found without confirmation. It is recommended to download and install the newest host tool before performing a firmware update.

3.6.2 Windows

The same instructions as with Unix apply. You can double-click "firmware-update.bat" in Explorer in the Ethernet Debugger installation folder to avoid constructing a command line. This will use firmware.dat in the same folder. In addition, the installer comes with the latest release of the firmware. Installers for host tool v1.0 (build 53) do not have this yet; download a newer version here[4].

(The host tool never "phones home", and there is no automatic update over internet.)

[4] https://intona.eu/en/products/ethernet-debugger#downloads

4 Capturing

The main purpose of the ethernet debugger is to capture packets. The following methods are available.

4.1 Wireshark extcap

If you open Wireshark, it should display any plugged in Intona Ethernet Debuggers as **`Ethernet Debugger USB (08900037)`** in the list of capture interfaces. The **`08900037`** in the brackets is the serial number (as in the USB device descriptor). (Some versions of Wireshark also show the device address in the format used by the host tool.) Double click this entry, and Wireshark should start capturing. The Ethernet Debugger's main LED will start blinking.

There is no hotplug mechanism for Wireshark extcaps. If you connect or disconnect devices while Wireshark is running, you may need to press F5 or restart Wireshark to update the device list of Ethernet Debugger capture devices.

Note that if the bandwidth utilization is high, the internal FIFO may overrun, leading to lost packets. The host tool adds a packet comment to the first packet after a run of dropped packets.

It may also happen that Wireshark freezes if the amount of data is too large, because the GUI requires a large amount of resources to deal with packet input. (Capturing to disk via the "nose" tool may help reducing packet drop. You can open the capture file with Wireshark afterwards.)

Various error conditions may deadlock Wireshark and the host tool on capture start.

4.1.1 Capturing Options

Wireshark lets you set some Ethernet Debugger specific options before starting capture. Click on the gear-like symbol left of the Ethernet Debugger interface in Wireshark's Capture interface list.

4.1.2 Wireshark extcap Toolbar

The host tool provides a toolbar in Wireshark. This is implemented through the extcap mechanism. It is slightly clunky due to Wireshark restrictions (all GUI code is provided by Wireshark, and not everything can be realized). The toolbar can be shown by enabling it in the Wireshark "View" menu, "Interface Toolbars" sub-menu.

4.2 Directly Starting Wireshark from Host Tool

You may use the **`--wireshark`** option of the host tool to start Wireshark and capturing in one go. It attempts to find the installation path of Wireshark, sets up a named FIFO, and starts a new Wireshark process.

Example

```
nose --wireshark
```

Lifetime of Wireshark process on Unix
Since host tool version v1.2, Wireshark is not terminated anymore if capturing ends or **nose** is exited. Older versions always terminated it due to being in the same process session.

4.3 Statistics

The host tool **--capture-stats** option can be used to enable regular statistic updates on the terminal. The ”set capture-stats true” command can be used to do this at runtime. (You can enter this command on the Wireshark extcap toolbar, for example.)

4.4 Capturing to a File

The host tool **--fifo** option can be used to capture either to a real file on disk, or a named FIFO. The **capture_start** command is similar, and can be used to start capturing via the host tool command line or IPC interface. The format of the output is PcapNG (see https://pcapng.github.io/pcapng/). You may use the third party open source libpcap library to parse such files. If you use an actual FIFO, you can stream in real time.

Note that if you capture to disk, overruns can happen due to waiting on disk I/O. The host tool tries to avoid this by using decoupled memory buffers, but these may be slowly filled up, until a software overrun happens.

Example

```
# Capturing to a file until Ctrl+C is hit, and log capture statistics to stdout.
nose --capture-stats --fifo target_file.pcapng

# Manually starting Wireshark.
# On terminal 1:
mkfifo /tmp/fifo
nose --fifo /tmp/fifo
# On terminal 2:
wireshark -k -i /tmp/fifo
```

4.5 Selecting the Device

If you have multiple Ethernet Debuggers, the **--device** option can be used to pick a specific device. Passing the special value **help** to this option lists all devices that were found.

Multi-Capture
Selecting multiple devices at once is not possible. However, if extcap is correctly installed, you can select multiple capture devices in Wireshark. This will provide a merged view of data coming from multiple devices and host tool instances.

4.6 Configuring the Buffer Size

The **--capture-soft-buffer** and **--capture-usb-buffer** can be used to fine-tune the sizes of the fixed size buffers allocated on the host. Raising them may reduce buffer overruns on the host PC.

5 Other Features

5.1 PoE Passthrough

Both ethernet ports are capable of handling Power over Ethernet (PoE) as specified in IEEE 802.3af, 802.3at and 802.3bt. Power is passed through in both directions without interception.

5.2 PTP Timestamps

The device provides high resolution timestamps for packets. This can help to debug PTP related issues, or any other timing issues. These timestamps are in nanoseconds with 10 ns resolution.

Each PHY has its own FIFO, which may affect accuracy. In addition, device-internal CDC may affect the accuracy. Internally, the timestamps are generated by a 100 MHz clock and are passed to the ethernet PHY's (RGMII) clock domain, which introduces jitter. The timestamps are relative to device start.

The host tool capture output adds a start offset to the timestamps to make them roughly line up with wall clock times. However, this offset is not precise, and there is no time correction. The absolute time of a packet event might be slightly different from real time. The longer the capture is running, the higher the deviation will be.

IN3083WP has additional information.

5.3 Supported Command Line Options

You can list supported options as follows:

```
nose --help
```

Current list of options:

Name	Description
--verbosity 0|1|2	Set verbosity log level: 0 silent, 1 normal/default, 2 verbose messages
--version	Print host software version and exit
--selftest	Run internal self-test. (Requires a loop between the two ports.)
--selftest-serial	Internal.
--wireshark	Start wireshark and dump packets to it. Terminate once done.
--device name	Open this device. (Pass "help" to get a list. "none" prevents automatic opening of a device.)
--firmware-update file	Perform a firmware update using this file.
--firmware-update-all	Update firmware of all devices that have been found. (Since v1.2.)
--firmware-update-force	Update firmware even for devices which have the current or newer firmware version. (Since v1.2.)
--fifo file	Start capture and write to the given file or fifo. (Overwrites the target if it's a file.)
--ipc-connect path	Connect IPC to this socket/named pipe. Terminate on disconnect.
--ipc-server name	Host IPC on this socket/named pipe.
--capture-soft-buffer num	Capture soft buffer (in bytes, accepts kib/mib/gib suffix)
--capture-usb-buffer num	Capture libusb buffer (in bytes, accepts kib/mib/gib suffix)
--capture-stats	Show capture statistics every 1 seconds.
--extcap-*	Various options for use by the Wireshark extcap mechanism.

Name	Description
--capture	Used by Wireshark; ignored by host tool.
--strip-frames	Strip preamble, SFD, and FCS from ethernet frames.
--cmd commands	Run commands on opening. (Must be a single string, separate commands with ;) (Since v1.3.)
--run commands	Run commands after opening. (Syntax like --cmd.) (Since v1.3.)
--exit-on	Control when exactly the tool should exit automatically. (Since v1.3.)
--exit-timeout	Always exit after the given time of seconds has expired. (Since v1.3.)

5.4 Interactive Command Line

The host tool has an interactive command line interface. When starting the host tool without commands, it will wait for commands from the terminal. You can use the ”help” command to list available commands and their parameters. Many advanced features (such as listed below) are accessible only through this interface.

By default, the host tool will read from stdin and accept commands. It will terminate if stdin is closed or returns EOF. If run on the terminal, it offers an interactive command line using libreadline. (If the host tool was built without libreadline, or you can use the rlwrap[5] 3rd party tool to get comfortable line editing and history:

```
rlwrap nose <arguments for nose>
```

5.5 Scripting

The host tool can be used for scripting. The **`--run`** and **`--exit-on`** options (available since host tool v1.3) can be used to run individual host tool commands.

Example

```
nose --run 'inject A --file mypacket.dat' --exit-on always
```

(The **`--exit-on`** is needed to make the tool exit after running the command.)

For more complicated use cases, the IPC interface offers and out-of-process API. (See IPC Interface section.)

5.6 Supported Commands

Name	Description
blink_led	Flash main LED
block_ports	Block or unblock all packets
capture_start	Start capturing to a file or FIFO
capture_stop	Stop current capture
cfg_packet	Send internal command to device
device_close	Close the current device
device_list	List all Ethernet Debuggers connected to this PC
device_open	Open a device

[5] https://github.com/hanslub42/rlwrap

Name	Description
disrupt	Setup packet disruption and port blocking
disrupt_stop	Disable packet disruption and port blocking
exit	Exit the host tool
help	Show all commands
hw_info	Show device information (including firmware version etc.)
inject	Setup packet injector
inject_stop	Disable packet injector
mdio_read	Read a PHY's MDIO register
mdio_write	Write a PHY's MDIO register
reset_device_settings	Restore default settings on the currently opened Ethernet Debugger
set	Set a command line parameter
set_device_phy_wait	Configure PHY speed negotiation delay time
speed	Configure PHY speed negotiation mode
latency_tester_sender	Setup device as latency tester sender (Since v1.3)
latency_tester_receiver	Setup device as latency tester receiver (Since v1.3)

Some commands are described in further detail in the following sections. The **help** command lists parameters accepted by each command.

5.7 IPC Interface

The command line interface is available via IPC (unix domain sockets on UNIX, named pipes on Windows). This may be useful for scripting or creating GUI frontends. For example, you could inject or drop packets under specific situations, or start/stop capturing at specific times

The IPC server created with **--ipc-server** is available as long as the host tool runs. The path, at which the IPC communication channel is available, depends on the OS. (The tool prints the full path when it starts the server.)

The **--ipc-connect** option makes the host tool initiate the IPC connection to the given path. This connection behaves the same as a connection made to a server started with **--ipc-server**. This mode may be more convenient with certain frameworks, for example when the framework makes it easy to start a server.

5.7.1 Windows

The IPC communication channel is a local win32 named pipe (CreateNamedPipeA()). For example, "**nose --ipc-server foo**" will create the named pipe **\\.\pipe\foo**, which can be accessed by other programs in read/write mode. For testing, a third party program such as PuTTY can be used (enter the full named pipe path as a serial device; unfortunately, configuring the terminal to act decently is hard: by default there is no local echo, and CTRL+j must be used to send the required **\n** ASCII 10 line terminator).

With **--ipc-connect**, the argument is a full path, which will be passed to CreateFileA(). In theory it can be anything, as long as access with **FILE_FLAG_OVERLAPPED** works.

5.7.2 UNIX (Linux/Mac)

The IPC communication channel is a Unix domain socket located on **/tmp**. For example, "**nose --ipc-server foo**" will create the socket **/tmp/too.socket**. (If the filesystem entry already exists, it will be

deleted, even if it's a normal file.) For testing, the third party program socat can be used to interactively send commands: "socat /tmp/foo.socket -"

`--ipc-connect` takes a full path. Both sockets are supported (connect() is used) and other types of files (open() is used).

5.7.3 Protocol

You can send commands in text or JSON form (the **`help`** command lists available commands and shows the basic syntax). The protocol uses 1 JSON object per line (in both directions of communication). A line is terminated with **`\n`** (ASCII line feed, byte 10). It is fairly self-describing:

IPC example

```
{"command":"mdio_read", "phy":1,"address":1,"id":1}   // client to host tool
{"id":1,"result":31049,"success":true}                // host tool to client
```

The **`id`** field is an arbitrary integer chosen by the client, and can be used to associate requests with replies. The C-style comments are for illustration, and not part of the protocol.

Log messages (such as output when PHY links change their state) are wrapped into special JSON messages:

Log message example

```
{"type":"log","msg":"PHY 1: link=up speed=1000MBit\n"}
```

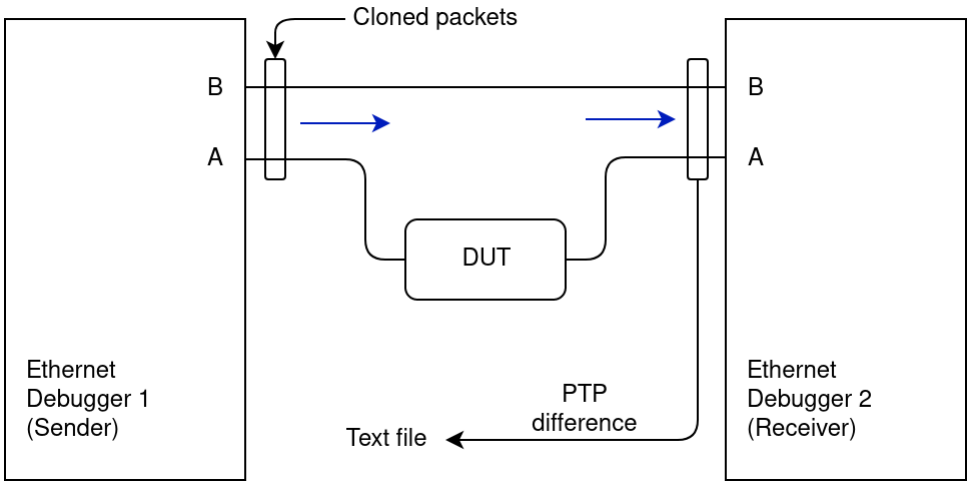
5.8 Identify Function

The **`blink_led`** command will flash the device main LED blue/green for a moment. This is helpful to determine which device is opened on a host tool instance if multiple Ethernet Debuggers are connected to a PC.

5.9 Latency Tester

5.9.1 Introduction

This feature (available since host tool v1.3) involves sending generated ethernet packets from one Ethernet Debugger to a second one, and measuring the delay. This can be used to test the latency if a 3rd device, using a setup like this:



Ethernet Debugger 1 is the sender, Ethernet Debugger 2 is the receiver. The sender generates packets that get sent out of port A and B at the same time. DUT is expected to propagate the packet unchanged to the receiver. The receiver measures the difference in arrival time between port A and B.

The Latency Tester feature takes care of generating/sending the packets and analyzing them on the receiver side. It writes the computed delay time to a text file. The packets use a (hopefully) unused EtherType (0xBEEF). The feature as currently implemented can test at most Layer 2 devices (like ethernet switches). Testing devices that operate on a higher layer (such as IP routers) are not supported, as that would require implementing parts of ARP and IP (though it's theoretically possible).

5.9.2 Instructions

Connect the devices as in the diagram above. On the PC, open two "nose" instances, one for the sender, one for the receiver. Use the device_list command to confirm whether the correct device is opened (change it with the device_open command). Then run the latency_tester_sender command on the sender instance, and latency_tester_receiver on the receiver instance. By default the sender starts sending continuously with 1 packet per 1 ms, with a test sequence that lasts 10 seconds. The receiver ignores the packets until a start packet is detected, and then logs the status every 10 packets (it logs the delay for only the 10th packet, use file output to retrieve all data points).

Sender setup

```
Device serial1 / 4:2:15 opened.
PHY A: link=up speed=1000MBit (master)
PHY B: link=up speed=1000MBit (master)
> device_list
Devices:
 - 'serial1' (4:2:15) [opened]
 - 'serial2' (4:2:16)
Found 2 devices.
> device_open serial2
USB device closed.
Device serial2 / 4:2:16 opened.
Opening succeeded.
PHY A: link=up speed=1000MBit (slave)
PHY B: link=up speed=1000MBit (master)
> latency_tester_sender
Latency tester: new test run (0)
Latency tester: new test run (1)
Latency tester: new test run (2)
```

If continuous sending is not desired, you can also just run ”latency_tester_sender --once”.

Receiver setup

```
Device serial1 / 4:2:15 opened.
PHY A: link=up speed=1000MBit (master)
PHY B: link=up speed=1000MBit (master)
> latency_tester_receiver
Starting capture thread succeeded.
Receiver setup. Listening to incoming packets.
Starting recording sequence...
seq=99 diff=-410
seq=199 diff=-410
seq=299 diff=-410
```

The output gives you an idea whether it actually works. Actual data should be retrieved by using file output. File output can be used with for example:

Receiver file output

```
> latency_tester_receiver --out-file mydata.txt
```

You may want to use an absolute path here (perhaps using drag & drop from a file manager to get the path) if the concept of working directories is too uncomfortable.

The file is opened when a sequence starts, and closed when it ends. If the file already exists when a sequence starts, the file is **not** overwritten. Instead, the sequence is skipped repeatedly until the file is deleted or moved by the user. Example:

Terminal Output of Latency Tester Example Run

```
> latency_tester_receiver --out-file mydata.txt
Starting capture thread succeeded.
Receiver setup. Listening to incoming packets.
File 'mydata.txt' exists, skipping sequence.
File 'mydata.txt' exists, skipping sequence.
Opened file 'mydata.txt' for writing.
Starting recording sequence...
seq=99 diff=410
seq=199 diff=410
seq=299 diff=410
seq=399 diff=410
seq=499 diff=410
seq=599 diff=410
...
seq=9899 diff=420
seq=9999 diff=410
Recording sequence finished.
Problems detected: no
File 'mydata.txt' closed.
File 'mydata.txt' exists, skipping sequence.
File 'mydata.txt' exists, skipping sequence.
```

The file mydata.txt was deleted shortly after the receiver was started. The file was then deleted. The host program picked up the next sequence of test packets and wrote them to mydata.txt. The sequences after that are skipped because mydata.txt still exists. (It will log two skip messages per sequence, because it encounters a first packet from both paths.) The idea behind this behavior is that tests can be repeated without much interaction, simply by renaming the output file (maybe giving it a more descriptive name), without the danger of accidentally overwriting files.

The output file simply contains the differences in receive timestamps in nanoseconds, with 10 ns resolution:

Example Output File (Partial)

```
420
430
410
410
420
410
420
430
410
420
420
420
420
420
420
420
```

These differences should be roughly equivalent to the latency the DUT incurs on ethernet. The values are positive if the latency values on port A are higher than on port b.

See the ”PTP Timestamps” section for details on the quality of the timestamps.

By default, 10000 samples are sent (depending on sender configuration), and the receiver reports ”Problems detected: no” only if all samples were included. If you see errors logged on the terminal, there is probably some sort of problem due to packet drop or corruption.

5.10 Packet Injection

It is possible to manually inject new packets into the ethernet connection. As no actual network interface is provided, OS mechanisms (such as sockets) cannot be used. Using the host tool is required.

The **`inject`** command provides this functionality. It is possible to send arbitrary ethernet packets, including packets that are not spec-compliant. This is not a high speed send path – full ethernet bandwidth cannot be reached. (Although you can instruct the command to repeat packets, in which case it will produce a high bandwidth stream of the same packet being repeated.)

Use **`inject_stop`** to disable the injector again. Use **`hw_info`** to check whether it’s currently enabled.

CRC errors can be injected. Degenerate packets (too short, IPG too low) can be created with the ”**`raw`**” parameter. Low level physical layer coding errors can be generated with the ”**`gen-error`**” parameter (PHY emits symbol error for a specific byte). The IPG can be controlled with the ”**`gap`**” parameter, which will set the distance to other generated packets as well as packets that are normally transferred through the wire.

Since host tool v1.3, packet data can be read from files with ”**`--file filename`**”.

Use ”**`help inject`**” to list all parameters.

Using the command may drop packets coming from the opposite source port. The injection logic will wait until transmission is turned off, then it will inject the packet, and fully drop any other packets from the opposite port.

Example

```
# Inject a packet that starts with a AB:CD:12:34:56 dest. MAC,
# the rest of the packet filled with 0s, on port A (into the
# stream of traffic flowing from B to A):
inject A ABCD123456
```

See **help** output for advanced parameters that control repeating, raw output (your own preamble), and so on.

5.11 Blocking Ports

The **block_ports** command blocks all traffic through the device in a specific direction:

Command	Effect
block_ports A	Block traffic from B→A, unblock A→B
block_ports B	Block traffic from A→B, unblock B→A
block_ports AB	Block all traffic
block_ports none	Unblock all traffic

This command uses the same hardware logic as the **disrupt** command (basically it's just a simpler version of the same command). Using either resets the state set by the other command.

5.12 Packet Disruption

The **disrupt** command can be used to drop or to destroy some or all packets in a certain direction. This is a form of error injection suited for testing reliability of low level protocols. It can either flip a single bit in a packet at a user-chosen byte offset, or discard entire packets.

(Restriction: cannot drop specific packet according to filter-like matching criteria etc.)

Use **disrupt_stop** to disable disruption again. Use **hw_info** to check whether it's currently enabled.

Example

```
# Disrupt a number of packets for a short time:
#  B         on port B (traffic flowing from A to B)
#  true      just drop the packets (instead of creating CRC errors)
#  20        drop 20 packets in total
#  10        drop only every 10th packet (this makes it active for ~200 packets)
disrupt B true 20 10
```

5.13 MDIO Access

5.13.1 Raw MDIO access

The **mdio_read** and **mdio_write** commands provide direct access to the PHY's MDIO registers. Access to register 22 is intercepted/emulated by the firmware.

5.13.2 Changing PHY Ethernet Speed

Ethernet speed is controlled by the PHYs. By default, they are set to use a common speed, and the firmware automatically adjusts the PHYs to use the slowest negotiated speed of either PHY (devices with firmware 1.00 do not support this behavior - you need to set the speed manually).

You can use the **speed** command to force specific speed modes on both PHYs:

Speed	Command
1000 MBit	speed 1000
100 MBit (full duplex)	speed 100
10 MBit (full duplex)	speed 10
Highest common speed mode of both PHY	speed same
Manual control	speed manual

The **same** mode puts the PHYs into auto negotiation mode at first, and if the speed modes mismatch, the faster PHY is forced into a lower speed mode. If a link goes down and up again, the process is repeated. Technically, the process can re-establish the link multiple times, which makes it slower and may be annoying when troubleshooting connected devices. The **set_device_phy_wait** command can be used to set the fixed time after which the state machine assumes the PHY cannot establish a link.

The **manual** mode lets each PHY negotiate the speed independently, which will lead to problems if the device's port A and B PHYs do not negotiate the same speed.

The values set by the "**speed**" and the "**set_device_phy_wait**" commands are stored as permanent settings on the device.

Older host tool versions do not have the "same" mode implemented, and require the user to manually correct speed modes. Even older versions do not even offer the "speed" command, and require using manual mdio_write commands. In addition, the "same" mode requires at least version 1.06 firmware.

5.14 Device Settings

Since firmware 1.06, the device persistently stores some settings on its EEPROM. You can inspect these settings with the "**fw_info**" command. You can clear all persistently stored settings with the "**reset_device_settings**" command. Power cycle the device after running the latter command successfully in order to clear any runtime state (PHY registers, inject/disrupt features).

6 Known Problems

- If no ethernet is connected, or there are no packets on the wire at all, the host tool may not terminate properly, even if Wireshark is terminated. This happens because the host tool will never write to the pipe to Wireshark. The host tool will continue to run in the background, and will block access to the device. In this case, the LED will continue blinking, and you may have to kill the host tool manually. This problem should not occur with host tool v1.2, which has a workaround that terminates it after a one second timeout.

- If extcap is used, and start of capture fails (such as inaccessible device), Wireshark does not terminate gracefully. This seems to be a Wireshark issue.
- If extcap is used, and capturing errors (such as unplugging the Ethernet Debugger device), Wireshark does sometimes not notice that capturing terminated.
- If extcap is used, Wireshark sometimes gets into a mode in which the Wireshark GUI seems to indicate that capturing started correctly, but the host tool is blocked trying to open Wireshark's extcap control pipes. Restarting Wireshark helps (you may need to kill the blocked "nose" process as well).
- 10 MBit mode is not working.

7 Further Readings

7.1 White Papers

7.1.1 Ethernet Debugger Timing

The Ethernet Debugger can be used to intercept and capture traffic between two Ethernet devices. The device is normally not supposed to affect the Ethernet traffic itself. But for technical reasons, the device can affect the traffic by introducing additional latency and jitter compared to a simple Ethernet cable. This effect is minimal, but can matter for real time applications such as PTP. The goal of this document is to provide information about the introduced latency and jitter, as well as actual measurements.

Online-Link[6], PDF-Download[7]

7.2 Statements

7.2.1 Letter of Volatility

This letter describes volatile, non-volatile, and storage media on the specified product. Customers can use this document to comply with security requirements and equipment handling procedures.

Online-Link[8], PDF-Download[9]

Document version: 89 / Jan 13, 2022 10:11

[6] https://intona.eu/doc/in3038wp
[7] https://intona.eu/doc/in3038wp/pdf
[8] https://intona.eu/doc/in3037st
[9] https://intona.eu/doc/in3037st/pdf