Preface

Foreword

This book tries to give you a guide to start your own experiments into the wonderful world of Wireshark development.

Developers who are new to Wireshark often have a hard time getting their development environment up and running. This is especially true for Windows developers, as a lot of the tools and methods used when building Wireshark are much more common in the UNIX world than on Windows.

The first part of this book will describe how to set up the environment needed to develop Wireshark.

The second part of this book will describe how to change the Wireshark source code.

We hope that you find this book useful, and look forward to your comments.

Who should read this document?

The intended audience of this book is anyone going into the development of Wireshark.

This book is not intended to explain the usage of Wireshark in general. Please refer the Wireshark User's Guide about Wireshark usage.

By reading this book, you will learn how to develop Wireshark. It will hopefully guide you around some common problems that frequently appear for new (and sometimes even advanced) developers of Wireshark.

Acknowledgements

The authors would like to thank the whole Wireshark team for their assistance. In particular, the authors would like to thank:

- Gerald Combs, for initiating the Wireshark project.
- Guy Harris, for many helpful hints and his effort in maintaining the various contributions on the mailing lists.
- Frank Singleton from whose README.idl2wrs idl2wrs: Creating dissectors from CORBA IDL files is derived.

The authors would also like to thank the following people for their helpful feedback on this document:
And of course a big thank you to the many, many contributors of the Wireshark development community!

About this document

This book was developed by Ulf Lamping, updated for VS2013 by Graham Bloice, and updated for later versions of Visual Studio by various contributors.

It is written in AsciiDoc.

Where to get the latest copy of this document?

The latest copy of this documentation can always be found at https://www.wireshark.org/docs/wsdg_html_chunked/.

Providing feedback about this document

Should you have any feedback about this document, please send it to the authors through wireshark-dev[AT]wireshark.org.

Typographic Conventions

The following table shows the typographic conventions that are used in this guide.

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italic</td>
<td>File names, folder names, and extensions</td>
<td>C:\Development\wireshark.</td>
</tr>
<tr>
<td>Monospace</td>
<td>Commands, flags, and environment variables</td>
<td>CMake's -G option.</td>
</tr>
<tr>
<td>Bold Monospace</td>
<td>Commands that should be run by the user</td>
<td>Run cmake -G Ninja ...</td>
</tr>
<tr>
<td>Key</td>
<td>Keyboard shortcut</td>
<td>Press Ctrl + Down to move to the next packet.</td>
</tr>
<tr>
<td>Menu</td>
<td>Menu item</td>
<td>Select Go › Next Packet to move to the next packet.</td>
</tr>
</tbody>
</table>
Admonitions

Important and notable items are marked as follows:

**WARNING**

This is a warning
You should pay attention to a warning, otherwise data loss might occur.

**CAUTION**

This is a caution
Act carefully (i.e., exercise care).

**IMPORTANT**

This is important information
RTFM - Read The Fine Manual

**TIP**

This is a tip
Tips are helpful for your everyday work using Wireshark.

**NOTE**

This is a note
A note will point you to common mistakes and things that might not be obvious.

Shell Prompt and Source Code Examples

**Bourne shell, normal user**

```
$ # This is a comment
$ git config --global log.abbrevcommit true
```

**Bourne shell, root user**

```
# # This is a comment
# ninja install
```

**Command Prompt (cmd.exe)**

```
>rem This is a comment
>cd C:\Development
```

**PowerShell**

```
PS$># This is a comment
PS$> choco list -l
```
C Source Code

#include "config.h"

/* This method dissects foos */
static int
dissect_foo_message(tvbuff_t *tvb, packet_info *pinfo _U_, proto_tree *tree _U_, void *
data _U_)
{
    /* TODO: implement your dissecting code */
    return tvb_captured_length(tvb);
}
Wireshark Build Environment

The first part describes how to set up the tools, libraries and sources needed to generate Wireshark and how to do some typical development tasks.
Introduction

What Is Wireshark?

Well, if you want to start Wireshark development, you might already know what Wireshark is doing. If not, please have a look at the [Wireshark User's Guide](#), which will provide a lot of general information about it.

Supported Platforms

Wireshark currently runs on most UNIX-like platforms and various Windows platforms. It requires Qt, GLib, libpcap and some other libraries in order to run.

As Wireshark is developed in a platform independent way and uses libraries (such as the Qt GUI library) which are available for many different platforms, it's thus available on a wide variety of platforms.

If a binary package is not available for your platform, you should download the source and try to build it. Please report your experiences to [wireshark-dev@wireshark.org](mailto:wireshark-dev@wireshark.org).

Binary packages are available for the following platforms along with many others:

**Unix And Unix-like Platforms**

- Apple macOS
- FreeBSD
- HP-UX
- IBM AIX
- NetBSD
- OpenBSD
- Oracle Solaris

**Linux**

- Arch Linux
- Debian GNU/Linux
- Ubuntu
Wireshark supports Windows natively via the **Windows API**. Note that in this documentation and elsewhere we tend to use the terms “Win32”, “Win”, and “Windows” interchangeably to refer to the Windows API. “Win64” refers to the Windows API on 64-bit platforms. Wireshark runs on and can be compiled on the following Windows versions:

- Windows 11 / Windows Server 2022
- Windows 8.1 / Windows Server 2012 R2
- Windows 8 / Windows Server 2012

Development on Windows 7, Server 2008 R2, Vista, Server 2008, and older versions may be possible but is not supported.

Some versions of Windows support **case sensitive directories**. We don’t officially support building or running Wireshark in this environment, but we will accept patches to fix any issues that might arise.

## Development And Maintenance Of Wireshark

Wireshark was initially developed by Gerald Combs. Ongoing development and maintenance of Wireshark is handled by the Wireshark core developers, a loose group of individuals who fix bugs and provide new functionality.

There have also been a large number of people who have contributed protocol dissectors and other improvements to Wireshark, and it is expected that this will continue. You can find a list of the people who have contributed code to Wireshark by checking the About dialog box of Wireshark, or have a look at the [https://www.wireshark.org/about.html#authors](https://www.wireshark.org/about.html#authors) page on the Wireshark web site.

The communication between the developers is usually done through the developer mailing list, which can be joined by anyone interested in the development activities. At the time this document was written, more than 500 persons were subscribed to this mailing list!
It is strongly recommended to join the developer mailing list, if you are going to do any Wireshark development. See Mailing Lists about the different Wireshark mailing lists available.

**Programming Languages Used**

Most of Wireshark is implemented in C. A notable exception is the code in `ui/qt`, which is written in C++.

The typical task for a new Wireshark developer is to extend an existing dissector, or write a new dissector for a specific network protocol. Most dissectors are written in C11, so a good knowledge of C will be sufficient for Wireshark development in almost any case. Dissectors can also be written in Lua, which might be more suitable for your specific needs. As noted above, if you’re going to modify Wireshark’s user interface you will need a knowledge of C++.

Modifying the build system and support tooling might requires knowledge of CMake, Python, PowerShell, Bash, or Perl. Note that these are required to build Wireshark, but not to run it. If Wireshark is installed from a binary package, none of these helper tools are needed on the target system.

**Open Source Software**

Wireshark is an open source software (OSS) project, and is released under the GNU General Public License (GPL). You can freely use Wireshark on any number of computers you like, without worrying about license keys or fees or such. In addition, all source code is freely available under the GPL. Because of that, it is very easy for people to add new protocols to Wireshark, either as plugins, or built into the source, and they often do!

You are welcome to modify Wireshark to suit your own needs, and it would be appreciated if you contribute your improvements back to the Wireshark community.

You gain three benefits by contributing your improvements back to the community:

- Other people who find your contributions useful will appreciate them, and you will know that you have helped people in the same way that the developers of Wireshark have helped you and other people.
- The developers of Wireshark might improve your changes even more, as there’s always room for improvement. Or they may implement some advanced things on top of your code, which can be useful for yourself too.
- The maintainers and developers of Wireshark will maintain your code as well, fixing it when API changes or other changes are made, and generally keeping it in tune with what is happening with Wireshark. So if Wireshark is updated (which is done often), you can get a new Wireshark version from the website and your changes will already be included without any effort for you.

The Wireshark source code and binary packages for some platforms are all available on the
Releases And Distributions

Official Wireshark releases can be found at https://www.wireshark.org/download.html. Minor releases typically happen every six weeks and typically include bug fixes and security updates. Major releases happen about once a year and include new features and new protocol support. Official releases include binary packages for Windows and macOS along with source code.

Binary Distributions

The Wireshark development team would like to make it as easy as possible for people to obtain and use Wireshark. This means that we need to support the software installation systems that different operating systems provide. We currently offer the following types of precompiled packages as part of each official release:

- Windows .exe installer. This is an executable file that installs Wireshark, and optionally Npcap and USBPcap, created using NSIS. It is the most popular installation method on Windows.
- Windows PortableApps .paf.exe file. This is a self-contained package that can be run from anywhere, either standalone or under the PortableApps.com Platform.
- Windows .msi installer. This installs Wireshark using Microsoft's Installer Database, created using the WiX toolset. It does not yet include Npcap or USBPcap and is somewhat experimental.
- macOS .dmg. This is a disk image which includes a drag-installable Wireshark application bundle along with utility packages for installing ChmodBPF and adding Wireshark to your PATH environment variable.

Most Linux and UNIX distributions have their own packaging systems which usually include Wireshark. The Wireshark sources include support for creating the following types of packages:

- Debian .deb files. Packaging assets can be found in the debian directory in the Wireshark sources.
- Red Hat .rpm files. Packaging assets can be found in the packaging/rpm directory in the Wireshark sources.

You can also create your own binary packages. See Binary Packaging for details.

The Source Code Distribution

Wireshark is and will always be open source. You're welcome to download a source tarball, build it, and modify it under the terms of the GPLv2. However, it’s usually much easier to use a binary package if you want to get up and running quickly in a production environment.

Source tarballs are commonly used for building the binary packages for UNIX and UNIX-like platforms. However, if you are going to modify the Wireshark sources, e.g. to contribute changes
back or to develop an in-house version of Wireshark we recommend that you use the latest Git sources. For details about the different ways to get the Wireshark source code see Obtaining The Wireshark Sources.

Before building Wireshark from a source distribution, make sure you have all the tools and libraries required to build. Later chapters describe the required tools and libraries in detail.

Automated Builds (GitLab CI)

The Wireshark development team uses GitLab's continuous integration (CI) system to automatically build Wireshark for each Git merge request and commit. Automated builds provide several useful services:

• Cross-platform testing. Inbound merge requests and commits can be tested on each of our supported platforms, which ensures that a developer on one platform doesn't break the build on other platforms.

• A health indicator for the source code. The CI badges at https://gitlab.com/wireshark/wireshark can quickly tell you how healthy the latest code is. Green is good, red is bad.

• Fast code delivery. After a change is committed to the repository, an installer is usually available within an hour at https://www.wireshark.org/download/automated/. This can be quite helpful for resolving issues, e.g. a bug reporter can easily verify a bugfix by installing a recent build.

• Automated regression tests. We run a comprehensive test suite as part of each build and continuously run fuzz tests that try to crash the dissection engine.

What Do The Automated Builds Do?

GitLab's CI operates by running a series of steps and reporting success or failure. A typical CI job might do the following:

1. Check out Wireshark from the source repository.
2. Build Wireshark.
3. Create a source tarball, binary package, or installer.
4. Run regression tests.

GitLab's CI marks successful jobs with a green checkmark and failed jobs with a red “X”. Jobs provide a link to the corresponding console logfile which provides additional information.

Release packages are built on the following platforms:

• Windows Server 2022 x64, Visual Studio 2022
• Windows 11 Arm64, Visual Studio 2022
• Ubuntu 22.04 x64, gcc
Static code analysis and fuzz tests are run on the following platforms:

- macOS x64, clang
- macOS Arm64, clang

**Reporting problems and getting help**

If you have problems, or need help with Wireshark, there are several places that may be of interest to you (well, beside this guide of course).

**Website**

You will find lots of useful information on the Wireshark homepage at https://www.wireshark.org/.

**Wiki**

The Wireshark Wiki at https://gitlab.com/wireshark/wireshark/-/wikis/ provides a wide range of information related to Wireshark and packet capturing in general. You will find a lot of information not part of this developer's guide. For example, there is an explanation how to capture on a switched network, an ongoing effort to build a protocol reference and a lot more.

And best of all, if you would like to contribute your knowledge on a specific topic (maybe a network protocol you know well), you can edit the Wiki pages by simply using your webbrowser.

**FAQ**

The "Frequently Asked Questions" will list often asked questions and the corresponding answers.

Before sending any mail to the mailing lists below, be sure to read the FAQ, as it will often answer any questions you might have. This will save yourself and others a lot of time. Keep in mind that a lot of people are subscribed to the mailing lists.

You will find the FAQ inside Wireshark by clicking the menu item Help/Contents and selecting the FAQ page in the upcoming dialog.

An online version is available at the Wireshark website: https://www.wireshark.org/faq.html. You might prefer this online version as it's typically more up to date and the HTML format is easier to use.

**Other sources**

If you don't find the information you need inside this book, there are various other sources of
information:

- The file doc/README.developer and all the other README.xxx files in the source code. These are various documentation files on different topics

  Read the README

  README.developer is packed full with all kinds of details relevant to the developer of Wireshark source code. Its companion file README.dissector advises you around common pitfalls, shows you basic layout of dissector code, shows details of the APIs available to the dissector developer, etc.

- The Wireshark source code

- Tool documentation of the various tools used (e.g. manpages of sed, gcc, etc.)

- The different mailing lists. See Mailing Lists

Q&A Site

The Wireshark Q&A site at https://ask.wireshark.org/ offers a resource where questions and answers come together. You have the option to search what questions were asked before and what answers were given by people who knew about the issue. Answers are graded, so you can pick out the best ones easily. If your issue isn't discussed before you can post one yourself.

Mailing Lists

There are several mailing lists available on specific Wireshark topics:

**wireshark-announce**

This mailing list will inform you about new program releases, which usually appear about every 4-8 weeks.

**wireshark-users**

This list is for users of Wireshark. People post questions about building and using Wireshark, others (hopefully) provide answers.

**wireshark-dev**

This list is for Wireshark developers. People post questions about the development of Wireshark, others (hopefully) provide answers. If you want to start developing a protocol dissector, join this list.

**wireshark-bugs**

This list is for Wireshark developers. Every time a change to the bug database occurs, a mail to this mailing list is generated. If you want to be notified about all the changes to the bug database, join this list. Details about the bug database can be found in Bug Database (GitLab Issues).
**wireshark-commits**

This list is for Wireshark developers. Every time a change to the Git repository is checked in, a mail to this mailing list is generated. If you want to be notified about all the changes to the Git repository, join this list. Details about the Git repository can be found in The Wireshark Git repository.

You can subscribe to each of these lists from the Wireshark web site: https://www.wireshark.org/lists/. From there, you can choose which mailing list you want to subscribe to by clicking on the Subscribe/Unsubscribe/Options button under the title of the relevant list. The links to the archives are included on that page as well.

**TIP**

You can search in the list archives to see if someone previously asked the same question and maybe already got an answer. That way you don't have to wait until someone answers your question.

**Bug Database (GitLab Issues)**

The Wireshark community collects bug reports in an issues database at https://gitlab.com/wireshark/wireshark/-/issues. This database is filled with manually filed bug reports, usually after some discussion on wireshark-dev, and automatic bug reports from continuous integration jobs.

**Reporting Problems**

**NOTE**

Before reporting any problems, please make sure you have installed the latest version of Wireshark. Reports on older maintenance releases are usually met with an upgrade request.

If you report problems, provide as much information as possible. In general, just think about what you would need to find that problem, if someone else sends you such a problem report. Also keep in mind that people compile/run Wireshark on a lot of different platforms.

When reporting problems with Wireshark, it is helpful if you supply the following information:

1. The version number of Wireshark and the dependent libraries linked with it, e.g. Qt, GLib, etc. You can obtain this with the command `wireshark -v`.
2. Information about the platform you run Wireshark on.
3. A detailed description of your problem.
4. If you get an error/warning message, copy the text of that message (and also a few lines before and after it, if there are some), so others may find the build step where things go wrong. Please don't give something like: "I get a warning when compiling x" as this won't give any direction to look at.
**NOTE**

*Dont't send large files*

Do not send large files (>100KB) to the mailing lists, just place a note that further data is available on request. Large files will only annoy a lot of people on the list who are not interested in your specific problem. If required, you will be asked for further data by the persons who really can help you.

**WARNING**

*Dont't send confidential information*

If you send captured data to the mailing lists, or add it to your bug report, be sure it doesn't contain any sensitive or confidential information, such as passwords. Visibility of such files can be limited to certain groups in the GitLab Issues database by marking the issue confidential.

**Reporting Crashes on UNIX-like platforms**

When reporting crashes with Wireshark, it is helpful if you supply the traceback information (besides the information mentioned in Reporting Problems).

You can obtain this traceback information with the following commands:

```bash
$ gdb `whereis wireshark | cut -f2 -d: | cut -d' ' -f2` core >& bt.txt
backtrace
^D
$  
```

*Using GDB*

Type the characters in the first line verbatim. Those are back-tics there.

`backtrace` is a `gdb` command. You should enter it verbatim after the first line shown above, but it will not be echoed. The `^D` (Control-D, that is, press the Control key and the D key together) will cause `gdb` to exit. This will leave you with a file called `bt.txt` in the current directory. Include the file with your bug report.

If you do not have `gdb` available, you will have to check out your operating system’s debugger.

You should mail the traceback to `wireshark-dev[AT]wireshark.org` or attach it to your bug report.

**Reporting Crashes on Windows platforms**

You can download Windows debugging symbol files (.pdb) from the following locations:

- 64-bit Windows: [https://www.wireshark.org/download/win64/all-versions/](https://www.wireshark.org/download/win64/all-versions/)

Files are named "Wireshark-pdb-winbits-x.y.z.zip" to match their corresponding "Wireshark-
"winbits-x.y.z.exe" installer packages.
Setup and Build Instructions

UN*X

Build environment setup

The following must be installed in order to build Wireshark:

- a C compiler and a C++ compiler;
- the Flex lexical analyzer;
- Python 3;
- CMake;
- several required libraries.

Either make or Ninja can be used to build Wireshark; at least one of those must be installed.


Perl is required to generate some code and run some code analysis checks.

Some features of Wireshark require additional libraries to be installed. The processes for doing so on various UN*X families is shown here.

There are shell scripts in the tools directory to install the packages and libraries required to build Wireshark. Usage is available with the --help option. root permission is required to run the scripts. The available scripts and their options for a given family of UN*Xes are shown in the section for that family.

Alpine Linux

The setup script is tools/alpine-setup.sh; its options are:

- --install-optional install optional software as well
- --install-all install everything
- [other] other options are passed as-is to apk

Arch Linux and pacman-based systems

The setup script is tools/arch-setup.sh; its options are:

- --install-optional install optional software as well
- --install-test-deps install packages required to run all tests
• --install-all install everything
• [other] other options are passed as-is to pacman

**BSD systems such as FreeBSD, NetBSD, OpenBSD, and DragonFly BSD**

The setup script is `tools/bsd-setup.sh`; its options are:

• --install-optional install optional software as well
• [other] other options are passed as-is to pkg manager

**Debian, and Linux distributions based on Debian, such as Ubuntu**

The setup script is `tools/debian-setup.sh`; its options are:

• --install-optional install optional software as well
• --install-deb-deps install packages required to build the .deb file
• --install-test-deps install packages required to run all tests
• --install-qt5-deps force installation of packages required to use Qt5
• --install-qt6-deps force installation of packages required to use Qt6
• --install-all install everything
• [other] other options are passed as-is to apt

**RPM-based Linux distributions such as Red Hat, Centos, Fedora, and openSUSE**

The setup script is `tools/rpm-setup.sh`; its options are:

• --install-optional install optional software as well
• --install-rpm-deps install packages required to build the .rpm file
• --install-qt5-deps force installation of packages required to use Qt5
• --install-qt6-deps force installation of packages required to use Qt6
• --install-all install everything
• [other] other options are passed as-is to the packet manager

**macOS**

You must first install Xcode.

After installing Xcode, the setup script `tools/macos-setup.sh` will install the rest of the tools and libraries required to build Wireshark, except for Qt 6, as well as the additional tools required to build the documentation and the libraries required for all Wireshark features. If you're using
Homebrew, the script `tools/macos-setup-brew.sh` will install the same tools and libraries from Homebrew.

If you will be building Wireshark with Qt 6, which is the default for Wireshark 4.0 and later, you will also have to install Qt; the `tools/macos-setup.sh` script will not install Qt 6. To install Qt, go to the Download Qt for open source use page, select “macOS” if it’s not already selected, and then select “Qt online installer for macOS”. This will download a .dmg for the installer; launch the installer. It will require that you log into your Qt account; if you don’t have an account, select “Sign up” to create one. The next page will require you to accept the LGPL (Lesser GNU Public License); do so. Continue to the “Installation Folder“ page of the installer screen, and select the “Custom installation“ option. On the “Select Components“ screen of the installer, select, for the desired Qt version, the “macOS” component. For example, at the time of this writing the Qt 6.5.3 “macOS” component is used to build the official packages. The “Qt Debug Information Files” component contains dSYM files which can be used for debugging. You can deselect all of the other the components such as “Qt Charts” or “Android xxxx” as they aren’t required.

Qt 6 needs the "Qt 5 Compatibility Module" to be installed as well. Additionally, the module "Qt Multimedia" may be installed, to support advanced controls for playing back streams in the RTP Player dialog.

**Building**

Before building:

On macOS, you will need to set the Qt installation directory in the environment:

```bash
WIRESHARK_QT6_PREFIX_PATH=~/Qt/6.5.3/macos
export WIRESHARK_QT6_PREFIX_PATH
```

If you want to append a custom string to the package version, run the command

```bash
WIRESHARK_VERSION_EXTRA=-YourExtraVersionInfo
export WIRESHARK_VERSION_EXTRA
```

The recommended (and fastest) way to build Wireshark is with CMake and Ninja. Building with make took nearly 2x time as Ninja in one experiment.

CMake builds are best done in a separate build directory, such as a `build` subdirectory of the top-level source directory.

If that directory is a subdirectory of the top-level source directory, to generate the build files, change to the build directory and enter the following command:

```bash
cmake ..
```
to use make as the build tool or

```bash
cmake -G Ninja ..
```

to use Ninja as the build tool.

If you created the build directory in the same directory that contains the top-level Wireshark source directory, to generate the build files, change to the build directory and enter the following command:

```bash
cmake ../{source directory}
```
to use make as the build tool or

```bash
cmake -G Ninja ../{source directory}
```
to use Ninja as the build tool.

{source directory} is the name of the top-level Wireshark source directory.

If you need to build with a non-standard configuration, you can run

```bash
cmake -LH ../{source directory}
```
to see what options you have.

You can then run Ninja or make to build Wireshark.

```bash
ninja
# or
make
```

Once you have build Wireshark with ninja or make above, you should be able to test it by entering `run/wireshark`.

**Optional: Install**

Install Wireshark in its final destination:

```bash
make install
```
Once you have installed Wireshark with `make install` above, you should be able to run it by entering `wireshark`.

**Optional: Create User’s and Developer’s Guide**

To build the Wireshark User’s Guide and the Wireshark Developer’s Guide, build the `all_guides` target, e.g. `make all_guides` or `ninja all_guides`. Detailed information to build these guides can be found in the file `docbook/README.adoc` in the Wireshark sources.

**Optional: Create an installable or source code package**

You can create packages using the following build targets and commands:

**Source code tarball**

Build the `dist` target.

**deb (Debian) package**

Create a symlink in the top-level source directory to `packaging/debian`, then run `dpkg-buildpackage`.

**RPM package**

Build the `wireshark_rpm` target.

**AppImage package**

Build the `wireshark_appimage` target.

**macOS .dmg package containing an application bundle**

Build the `wireshark_dmg` or `logray_dmg` targets.

Installable packages typically require building Wireshark first.

**Troubleshooting during the build and install on Unix**

A number of errors can occur during the build and installation process. Some hints on solving these are provided here.

If the `cmake` stage fails you will need to find out why. You can check the file `CMakeOutput.log` and `CMakeError.log` in the build directory to find out what failed. The last few lines of this file should help in determining the problem.

The standard problems are that you do not have a required development package on your system or that the development package isn’t new enough. Note that installing a library package isn’t enough. You need to install its development package as well.

If you cannot determine what the problems are, send an email to the `wireshark-dev` mailing list explaining your problem. Include the output from `cmake` and anything else you think is relevant.
such as a trace of the `make` stage.

# Windows: Using Microsoft Visual Studio

A quick setup guide for Windows development with recommended configurations.

### WARNING

Unless you know exactly what you are doing, you should strictly follow the recommendations below. They are known to work and if the build breaks, please re-read this guide carefully.

**Known traps are:**

1. Not using the correct (x64 or x86) version of the Visual Studio command prompt.
2. Not using a supported version of Windows. Please check [here](#) that your installed version is supported and updated.

### Recommended: Install Chocolatey

**Chocolatey** is a native package manager for Windows. There are packages for most of the software listed below. Along with traditional Windows packages it supports the Python Package Index.

Chocolatey tends to install packages into its own path (%ChocolateyInstall%), although packages are free to use their own preferences. You can install Chocolatey packages using the command `choco install` (or its shorthand, `cinst`), e.g.

```bash
rem Flex is required.
choco install -y winflexbison3
rem Git, CMake, Python, etc are also required, but can be installed
rem via their respective installation packages.
choco install -y git cmake python3
```

### Install Microsoft Visual Studio

Download and install **“Microsoft Visual Studio 2022 Community Edition”**. If you prefer you can instead download and install **“Microsoft Visual Studio 2019 Community Edition”**. The examples below are for Visual Studio 2022 but can be adapted for Visual Studio 2019. These are small utilities that download all the other required parts (which are quite large).

Check the checkbox for “Desktop development with C++” and then uncheck all the optional components other than

- “MSVC ... VS 2022 C++” item with the “... build tools (Latest)”
- “Windows 11 SDK”
You can alternatively use Chocolatey to install Visual Studio, using the Visual Studio Community and Native Desktop workload packages. Note that this includes Visual Studio's CMake component.

```
choco install -y visualstudio2022community visualstudio2022-workload-nativedesktop
```

You can use other Microsoft C compiler variants, but VS2022 is used to build the development releases and is the preferred option. It's possible to compile Wireshark with a wide range of Microsoft C compiler variants. For details see Microsoft compiler toolchain (Windows native).

You may have to do this as Administrator.

Compiling with gcc or Clang is not recommended and will certainly not work (at least not without a lot of advanced tweaking). For further details on this topic, see GNU Compiler Toolchain (UNIX And UNIX-like Platforms). This may change in future as releases of Visual Studio add more cross-platform support.

Why is this recommended? While this is a huge download, the Community Editions of Visual Studio are free (as in beer) and include the Visual Studio integrated debugger. Visual Studio 2022 is also used to create official Wireshark builds, so it will likely have fewer development-related problems.

**Install Qt**

The main Wireshark application uses the Qt windowing toolkit. To install Qt, go to the "Download Qt" page, select "Download open source", then "Download Qt Online Installer", and download "Qt Online Installer for Windows". When executing it, sign up or log in, and use Next button to proceed. When asked, select "Custom installation".

In the "Select Components" page, select your desired Qt version. We recommend the latest LTS version, and the stable Windows installers currently ship with Qt 6.5.3. Select the following components:

- MSVC 2019 64-bit
- Qt 5 Compatibility Module
- Qt Debug Information Files (contains PDB files which can be used for debugging)
- Under "Additional Libraries" select "Qt Multimedia" to support advanced controls for playing back streams in the RTP Player dialog
- You can deselect all of the other the components such as "Qt Charts" or "Android xxxx" as they aren't required.

The CMake variable CMAKE_PREFIX_PATH (see [https://doc.qt.io/qt-6/cmake-get-started.html](https://doc.qt.io/qt-6/cmake-get-started.html))
should be set as appropriate for your environment and should point to the Qt installation directory, e.g. `C:\Qt\6.5.3\msvc2019_64` Alternatively you can also use the environment variable `WIRESHARK_QT6_PREFIX_PATH`.

Qt 6 is the default option for building Wireshark, but Wireshark has support for Qt 5.12 and later. To enable Wireshark to build with Qt 5 pass `-DUSE_qt6=OFF` to cmake.

**Install Python**

Get a Python 3 installer from [https://python.org/download/](https://python.org/download/) and install Python. Its installation location varies depending on the options selected in the installer and on the version of Python that you are installing. At the time of this writing the latest version of Python is 3.10, and common installation directories are `C:\[Users][username]\AppData\Local\Programs\Python\Python310`, `C:\Program Files\Python310`, and `C:\Python310`.

Alternatively you can install Python using Chocolatey:

```
choco install -y python3
```

Chocolatey will likely install Python in one of the locations above, or possibly in `C:\[Tools]\Python3`.

**Install Git**

Please note that the following is not required to build Wireshark but can be quite helpful when working with the sources.

Working with the Git source repositories is highly recommended, as described in [Obtaining The Wireshark Sources](#). It is much easier to update a personal source tree (local repository) with Git rather than downloading a zip file and merging new sources into a personal source tree by hand. It also makes first-time setup easy and enables the Wireshark build process to determine your current source code revision.

There are several ways in which Git can be installed. Most packages are available at the URLs below or via Chocolatey. Note that many of the GUI interfaces depend on the command line version.

If installing the Windows version of git select the `Use Git from the Windows Command Prompt` (in chocolatey the `/GitOnlyOnPath` option). Do not select the `Use Git and optional Unix tools from the Windows Command Prompt` option (in chocolatey the `/GitAndUnixToolsOnPath` option).

**The Official Windows Installer**

The official command-line installer is available at [https://git-scm.com/download/win](https://git-scm.com/download/win).

**Git Extensions**

Git Extensions is a native Windows graphical Git client for Windows. You can download the

**TortoiseGit**

TortoiseGit is a native Windows graphical Git similar to TortoiseSVN. You can download the installer from https://tortoisegit.org/download/.

**Command Line client via Chocolatey**

The command line client can be installed (and updated) using Chocolatey:

```bash
choco install -y git
```

**Others**

A list of other GUI interfaces for Git can be found at https://git-scm.com/downloads/guis

**Install CMake**

While CMake is required to build Wireshark, it might have been installed as a component of either Visual Studio or Qt. If that's the case you can skip this step. If you do want or need to install CMake, you can get it from https://cmake.org/download/. Installing CMake into the default location is recommended. Ensure the directory containing cmake.exe is added to your path.

Alternatively you can install it using Chocolatey:

```bash
choco install -y cmake
```

Chocolatey ensures cmake.exe is on your path.

**Install Asciidoctor, Xsltproc, And DocBook**

Asciidoctor can be run directly as a Ruby script or via a Java wrapper (AsciidoctorJ). The JavaScript flavor (Asciidoctor.js) isn't yet supported. It is used in conjunction with Xsltproc and DocBook to generate the documentation you’re reading and the User's Guide.

You can install AsciidoctorJ, Xsltproc, and DocBook using Chocolatey. AsciidoctorJ requires a Java runtime and there are many to choose from. Chocolatey doesn't support alternative package dependencies at the present time, including dependencies on Java. As a result, installing the asciidoctorj package won’t automatically install a Java runtime — you must install one separately.

```bash
choco install -y <your favorite Java runtime>
choco install -y asciidoctorj xsltproc docbook-bundle
```
Chocolatey ensures that asciidocj.exe and xsltproc.exe is on your path and that xsltproc uses the DocBook catalog.

**Install winflexbison**

Get the winFlexBison installer from https://sourceforge.net/projects/winflexbison/ and install into the default location. Ensure the directory containing win_flex.exe is on your path.

Alternatively you can install Winflexbison using Chocolatey:

```
choco install -y winflexbison3
```

Chocolatey ensures win_flex.exe is on your path.

**Optional: Install Perl**

If needed you can get a Perl installer from http://strawberryperl.com/ or https://www.activestate.com/ and install Perl into the default location.

Alternatively you can install Perl using Chocolatey:

```
choco install -y strawberryperl
# ...or...
choco install -y activeperl
```

**Install and Prepare Sources**

**TIP**

*Make sure everything works*

It’s a good idea to make sure Wireshark compiles and runs at least once before you start hacking the Wireshark sources for your own project. This example uses Git Extensions but any other Git client should work as well.

**Download sources** Download Wireshark sources into C:\Development\wireshark using either the command line or Git Extensions:

Using the command line:

```
cd C:\Development
git clone https://gitlab.com/wireshark/wireshark.git
```

Using Git extensions:

1. Open the Git Extensions application. By default Git Extensions will show a validation checklist
at startup. If anything needs to be fixed do so now. You can bring up the checklist at any time via 
Tools › Settings.

2. In the main screen select Clone repository. Fill in the following:

Repository to clone: https://gitlab.com/wireshark/wireshark.git

Destination: Your top-level development directory, e.g. C:\Development.

Subdirectory to create: Anything you’d like. Usually wireshark.

<table>
<thead>
<tr>
<th>TIP</th>
<th>Check your paths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Make sure your repository path doesn’t contain spaces.</td>
</tr>
</tbody>
</table>

3. Click the [Clone] button. Git Extensions should start cloning the Wireshark repository.

Open a Visual Studio Command Prompt

From the Start Menu (or Start Screen), navigate to the “Visual Studio 2022” folder and choose the Command Prompt appropriate for the build you wish to make, e.g. “x64 Native Tools Command Prompt for VS 2022” for a 64-bit version. Depending on your version of Windows the Command Prompt list might be directly under “Visual Studio 2022” or you might have to dig for it under multiple folders, e.g. Visual Studio 2022 › Visual Studio Tools › Windows Desktop Command Prompts.

You can set up a build environment in your own command prompt by running the appropriate vcvarsARCHITECTURE.bat command. See Use the Microsoft C++ toolset from the command line for details.

<table>
<thead>
<tr>
<th>TIP</th>
<th>Pin the items to the Task Bar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pin the Command Prompt you use to the Task Bar for easy access.</td>
</tr>
</tbody>
</table>

All subsequent operations take place in this Command Prompt window.

1. Set environment variables to control the build.

Set the following environment variables, using paths and values suitable for your installation:
rem Let CMake determine the library download directory name under
rem WIRESHARK_BASE_DIR or set it explicitly by using WIRESHARK_LIB_DIR.
rem Set *one* of these.
set WIRESHARK_BASE_DIR=C:\Development
rem set WIRESHARK_LIB_DIR=c:\wireshark-x64-libs
rem Set the Qt installation directory
set WIRESHARK_QT6_PREFIX_PATH=C:\Qt\6.5.3\msvc2019_64
rem Append a custom string to the package version. Optional.
set WIRESHARK_VERSION_EXTRA=-YourExtraVersionInfo

Setting these variables could be added to a batch file to be run after you open the Visual Studio Tools Command Prompt.

TIP Use of Qt's LTS branch

It is generally recommended to use a LTS ("long term support") version for Qt. The current LTS version for Qt 6 is 6.5.3.

2. Create and change to the correct build directory. CMake is best used in an out-of-tree build configuration where the build is done in a separate directory from the source tree, leaving the source tree in a pristine state. 64 and 32 bit builds require a separate build directory. Create (if required) and change to the appropriate build directory.

mkdir C:\Development\wsbuild64
cd C:\Development\wsbuild64

to create and jump into the build directory.

The build directory can be deleted at any time and the build files regenerated as detailed in Generate the build files.

Generate the build files

CMake is used to process the CMakeLists.txt files in the source tree and produce build files appropriate for your system.

You can generate Visual Studio solution files to build either from within Visual Studio, or from the command line with MSBuild. CMake can also generate other build types but they aren’t supported.

The initial generation step is only required the first time a build directory is created. Subsequent builds will regenerate the build files as required.

If you’ve closed the Visual Studio Command Prompt prepare it again.

To generate the build files enter the following at the Visual Studio command prompt:
cmake -G "Visual Studio 17 2022" -A x64 ..\wireshark

Adjusting the path to the Wireshark source tree as required. To use a different generator modify the 
-G parameter. `cmake -G` lists all the CMake supported generators, but only Visual Studio is 
supported for Wireshark builds. 32-bit builds are no longer supported.

The CMake generation process will download the required 3rd party libraries (apart from Qt) as 
required, then test each library for usability before generating the build files.

At the end of the CMake generation process the following should be displayed:

```
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Development/wsbuild64
```

If you get any other output, there is an issue in your environment that must be rectified before 
building. Check the parameters passed to CMake, especially the -G option and the path to the 
Wireshark sources and the environment variables `WIRESHARK_BASE_DIR` and `CMAKE_PREFIX_PATH`.

Build Wireshark

Now it's time to build Wireshark!

1. If you've closed the Visual Studio Command Prompt prepare it again.
2. Run

   ```
   msbuild /m /p:Configuration=RelWithDebInfo Wireshark.sln
   ```

   to build Wireshark.
3. Wait for Wireshark to compile. This will take a while, and there will be a lot of text output in the 
   command prompt window
4. Run `C:\Development\wsbuild64\run\RelWithDebInfo\Wireshark.exe` and make sure it starts.
5. Open Help › About. If it shows your "private" program version, e.g.: Version 4.3.0-
   myprotocol123 congratulations! You have compiled your own version of Wireshark!

You may also open the Wireshark solution file (`Wireshark.sln`) in the Visual Studio IDE and build 
there.

   **TIP** If compilation fails for suspicious reasons after you changed some source files try to 
   clean the build files by running `msbuild /m /p:Configuration=RelWithDebInfo
   Wireshark.sln /t:Clean` and then building the solution again.
The build files produced by CMake will regenerate themselves if required by changes in the source tree.

**Debug Environment Setup**

You can debug using the Visual Studio Debugger or WinDbg. See the section on using the [Debugger Tools](#).

**Optional: Create User’s and Developer’s Guide**

To build the Wireshark User’s Guide and the Wireshark Developer’s Guide, build the `all_guides` target, e.g. `msbuild all_guides.vcxproj`. Detailed information to build these guides can be found in the file `docbook/README.adoc` in the Wireshark sources.

**Optional: Create a Wireshark Installer**

Note: You should have successfully built Wireshark before doing the following.

If you want to build your own `Wireshark-4.3.0-myprotocol123-x64.exe`, you’ll need NSIS. You can download it from [http://nsis.sourceforge.net](http://nsis.sourceforge.net).

Note that the 32-bit version of NSIS will work for both 64-bit and 32-bit versions of Wireshark. NSIS version 3 is required.

If you’ve closed the Visual Studio Command Prompt prepare it again. Run

```bash
msbuild /m /p:Configuration=RelWithDebInfo wireshark_nsis_prep.vcxproj
msbuild /m /p:Configuration=RelWithDebInfo wireshark_nsis.vcxproj
```

To build a Wireshark installer. If you sign your executables you should do so between the “wireshark_nsis_prep” and “wireshark_nsis” steps. To sign your installer you should place the signing batch script on the path. It must be named "sign-wireshark.bat". It should be autodetected by CMake, to always require signing set the `-DENABLE_SIGNED_NSIS=On` CMake option.

Run

```bash
packaging
sis\wireshark-{wireshark-version}-myprotocol123-x64.exe
```

to test your new installer. It’s a good idea to test on a different machine than the developer machine.

**Windows: Using MinGW-w64 with MSYS2**

MSYS2 comes with different environments/subsystems and the first thing you have to decide is
which one to use. The differences among the environments are mainly environment variables, default compilers/linkers, architecture, system libraries used etc. If you are unsure, go with UCRT64.

**Building from source**

1. Open the shell for the selected 64-bit environment.
2. Download the Wireshark source code using Git, if you haven’t done so already, and cd into that directory.
3. Install needed dependencies:

   ```bash
   tools/msys2-setup.sh --install-all
   ```

4. Build using CMake + Ninja:

   ```bash
   mkdir build && cd build
   # Ninja generator is the default
   cmake -DENABLE_CCACHE=On -DFETCH_lua=Yes ..
   ninja
   ninja test       # optional, to run the test suite
   ninja install    # optional, install to the MSYS2 shell path
   ```

The application should be launched using the same shell.

**Building an .exe installer**

1. Follow the instructions above to compile Wireshark from source.
2. Build the NSIS installer target.

   ```bash
   ninja wireshark_nsis_prep
   ninja wireshark_nsis
   ```

If successful the installer can be found in `${CMAKE_BINARY_DIR}/packaging/nsis`.

Alternatively you can also use the PKGBUILD included in the Wireshark source distribution to compile Wireshark into a binary package that can be installed using `pacman`.

**Comparison with MSVC toolchain**

The official Wireshark Windows installer is compiled using Microsoft Visual Studio (MSVC). Currently the MSYS2 build has the following limitations compared to the build using MSVC:
• Lua does not have custom UTF-8 patches.
• The Event Tracing for Windows (ETW) extcap cannot be compiled using MinGW-w64.
• Enhanced Kerberos dissection with decryption is not available.
• AirPcap is not supported.

Windows: Cross-compilation using Linux

It is possible to compile Wireshark for Microsoft Windows using Linux and MinGW. This way developers can deploy Wireshark on Windows systems without requiring a Windows host machine. Building for Windows using a Linux host is also easier for devs already familiar with Linux, the build itself is faster and it uses a very mature C/C++ compiler (GCC) and debugger (GDB).

Using Fedora Linux

Fedora Linux provides the best out-of-the-box support for MinGW cross-compilation. Fedora is what the project uses to test the build and it's what we recommend. While any other reasonably modern Linux distribution can be used, that will make the process more time consuming and involve some trial and error to setup.

The build instructions on Fedora follow the familiar recipe for building Wireshark using Linux.

Building from source

1. Install needed dependencies:

   ```
   tools/mingw-rpm-setup.sh --install-all
   ```

2. Build using CMake + Ninja:

   ```
   mkdir build && cd build
   mingw64-cmake -G Ninja -DENABLE_CCACHE=Yes -DFETCH_lua=Yes ..
   ninja
   ```

   Note that currently it is not possible to run the test-suite when cross-compiling.

3. Build the NSIS installer

   ```
   ninja wireshark_nsis_prep
   ninja wireshark_nsis
   ```

If successful the installer can be found in `CMAKE_BINARY_DIR/packaging/nsis`.
Notes and comparison with MSVC builds

- Only the MSVCRT C library for Microsoft Windows can be used. Support for the UCRT (Universal C Runtime) library on Fedora Linux is in the initial stages of deployment and not ready for prime-time (at the time of this writing).
- Some optional dependencies are missing from Fedora repositories and must be compiled from source if desired. An up-to-date complete list can be found in the bug tracker (issue 19108).
- Lua does not have custom UTF-8 patches.
- The Event Tracing for Windows (ETW) extcap cannot be compiled using MinGW-w64.
- Enhanced Kerberos dissection with decryption is not available.
- AirPcap is not supported.

Using Arch Linux

Arch Linux has good support for MinGW using packages from the AUR. Note that the mingw-w64 AUR packages sometimes break. If that happens you may be required to fix it or skip the package until it is fixed by the maintainer, if it's an optional dependency. You may also want to consider using an unofficial user repository (such as the ownstuff repository) to provide pre-compiled packages. This will greatly simplify the initial setup and subsequent upgrades.

CAUTION
AUR packages and unofficial user repositories are user-produced content. These packages are completely unofficial and have not been thoroughly vetted. It is your decision whether to trust their maintainers and you take full responsibility for choosing to use them.

You will need to install an AUR helper. This guide assumes paru is being used.

1. Install required dependencies from official repositories:

   ```
   pacman -S mingw-w64 nsis lemon qt6-tools ccache
   ```

2. Install required dependencies from the AUR:

   ```
   paru -S mingw-w64-cmake
   paru -S mingw-w64-glib2
   paru -S mingw-w64-libgcrypt
   paru -S mingw-w64-c-ares
   paru -S mingw-w64-speexdsp
   paru -S mingw-w64-libpcap
   ```

3. Install Qt6:
4. Install optional dependencies:

```
paru -S mingw-w64-gnutls
paru -S mingw-w64-lz4
paru -S mingw-w64-snappy
paru -S mingw-w64-opus
paru -S mingw-w64-opencore-amr
paru -S mingw-w64-libxml2
paru -S mingw-w64-libnghttp2
paru -S mingw-w64-libssh
paru -S mingw-w64-minizip
```

Search the AUR for other dependencies not listed above.

5. Build Wireshark using CMake + Ninja. From the directory containing the Wireshark source tree run:

```
mkdir build && cd build
x86_64-w64-mingw32-cmake -G Ninja -DENABLE_CCACHE=Yes -DFETCH_lua=Yes \
-DMINGW_SYSROOT=/usr/x86_64-w64-mingw32 ..
ninja
```

This will automatically download and build Lua as a static library.

To reconfigure the CMake build you may to do it explicitly by running `x86_64-w64-mingw32-cmake .` in the build directory, instead of letting `ninja` do it for you automatically.

6. Build the NSIS installer

```
ninja wireshark_nsis_prep
ninja wireshark_nsis
```

If everything goes well the installer can be found in `$CMAKE_BINARY_DIR/packaging/nsis`.

The same notes as the build using Fedora apply.
Work with the Wireshark sources

Introduction

This chapter will explain how to work with the Wireshark source code. It will show you how to:

- Get the source
- Compile it on your machine
- Submit changes for inclusion in the official release

This chapter will not explain the source file contents in detail, such as where to find specific functionality. This is done in Source overview.

The Wireshark Git repository

Git is used to keep track of the changes made to the Wireshark source code. The official repository is hosted at GitLab, and incoming changes are evaluated and reviewed there. For more information on GitLab see their documentation.

Why Git?

Git is a fast, flexible way of managing source code. It allows large scale distributed development and ensures data integrity.

Why GitLab?

GitLab makes it easy to contribute. You can make changes locally and push them to your own work area at gitlab.com, or if your change is minor you can make changes entirely within your web browser.

Historical trivia: GitLab is the fourth iteration of our source code repository and code review system.

Wireshark originally used Concurrent Versions System (CVS) and migrated to Subversion in July 2004. We migrated from Subversion to Git and Gerrit in January 2014, and from Gerrit to GitLab in August 2020.

Using Wireshark’s GitLab project you can:

- Keep your private sources up to date with very little effort.
- Receive notifications about code reviews and issues.
- Get the source files from any previous release (or any other point in time).
- Browse and search the source code using a web interface.
- See which person changed a specific piece of code.
Git Naming Conventions

Like most revision control systems, Git uses branching to manage different copies of the source code and allow parallel development. Wireshark uses the following branch naming conventions:

`master`.
Main feature development and odd-numbered development releases.

`release-x.y, master-x.y`.
Stable release maintenance. For example, release-3.4 is used to manage the 3.4.x official releases.

Tags for major releases and release candidates consist of a “v” followed by a version number such as “v3.2.1” or “v3.2.3rc0”. Major releases additionally have a tag prefixed with “wireshark-” followed by a version number, such as “wireshark-3.2.0”.

Browsing And Searching The Source Code

If you need a quick look at the Wireshark source code you can browse the repository files in GitLab at

https://gitlab.com/wireshark/wireshark/-/tree/master

You can view commit logs, branches, and tags, find files and search the repository contents. You can also download individual files.

Obtaining The Wireshark Sources

There are two primary ways to obtain Wireshark's source code: Git and compressed .tar archives. Each is described in more detail below. We recommend using Git for day to day development, particularly if you wish to contribute changes back to the project. The age mentioned in the following sections indicates the age of the most recent change in that set of the sources.

Git Over SSH Or HTTPS

This method is strongly recommended for day to day development.

You can use a Git client to download the source code from Wireshark's code review system. Anyone can clone from the anonymous HTTP git URL:

https://gitlab.com/wireshark/wireshark.git

If you have a GitLab account you can also clone using SSH:

git@gitlab.com:wireshark/wireshark.git

If wish to make changes to Wireshark you must create a GitLab account, create a fork of the official
Wireshark repository, update your fork, and create a merge request. See [Contribute Your Changes](#) for details.

The following example shows how to get up and running on the command line. See [Git client](#) for information on installing and configuring graphical Git clients.

1. Now on to the command line. First, make sure `git` works:

   ```
   $ git --version
   ```

2. If this is your first time using Git, make sure your username and email address are configured. This is particularly important if you plan on uploading changes:

   ```
   $ git config --global user.name "Henry Perry"
   $ git config --global user.email henry.perry@example.com
   ```

3. Next, clone the Wireshark repository:

   ```
   # If you have a GitLab account, you can use the SSH URL:
   $ git clone -o upstream git@gitlab.com:wireshark/wireshark.git
   # If you don't you can use the HTTPS URL:
   $ git clone -o upstream https://gitlab.com/wireshark/wireshark.git
   # You can speed up cloning in either case by adding --shallow-since=1year or --depth=5000.
   ```

   The clone only has to be done once. This will copy all the sources (including directories) from the server to your machine and check out the latest version.

   The `-o upstream` flag uses the origin name “upstream” for the repository instead of the default “origin” as described in the [GitLab documentation](#).

   Cloning may take some time depending on the speed of your internet connection.

   The `--shallow-since=1year` option limits cloned commits to the last 1 year.

   The `--depth=5000` option limits cloned commits to the last 5000.

**Development Snapshots**

This method is useful for one-off builds or if Git is inaccessible (e.g. because of a restrictive firewall).

Our GitLab CI configuration automatically generates development packages, including source packages. They can be found at [https://www.wireshark.org/download/automated/src/](https://www.wireshark.org/download/automated/src/). Packages are
available for recent commits in the master branch and each release branch.

**Official Source Releases**

This method is recommended for building downstream release packages.

The official source releases can be found at [https://www.wireshark.org/download.html](https://www.wireshark.org/download.html). You should use these sources if you want to build Wireshark on your platform based on an official release with minimal or no changes, such as Linux distribution packages.

**Update Your Wireshark Sources**

After you've obtained the Wireshark sources for the first time, you might want to keep them in sync with the sources at the upstream Git repository.

**TIP**

*) Take a look at the recent commits first

As development evolves, the Wireshark sources are compilable most of the time — but not always. You should take a look at [https://gitlab.com/wireshark/wireshark/-/commits/master](https://gitlab.com/wireshark/wireshark/-/commits/master) before fetching or pulling to make sure the builds are in good shape.

**Update Using Git**

From time to time you will likely want to synchronize your master branch with the upstream repository. You can do so by running:

```
$ git pull --rebase upstream master
```

**Build Wireshark**

The sources contain several documentation files. It's a good idea to read these files first. After obtaining the sources, tools and libraries, the first place to look at is `doc/README.developer`. Inside you will find the latest information for Wireshark development for all supported platforms.

**TIP**

*) Build Wireshark before changing anything

It is a very good idea to first test your complete build environment (including running and debugging Wireshark) before making any changes to the source code (unless otherwise noted).

Building Wireshark for the first time depends on your platform.

**Building on Unix**

Follow the build procedure in `UN*X` to build Wireshark.
**Windows Native**

Follow the build procedure in [Windows: Using Microsoft Visual Studio](#) to build Wireshark.

After the build process has successfully finished, you should find a `Wireshark.exe` and some other files in the `run\RelWithDebInfo` directory.

**Build Type**

CMake can compile Wireshark for several different build types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Compiler Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelWithDebInfo</td>
<td>-O2 -g -DNDEBUG</td>
<td>Build with optimizations and generate debug symbols. Disables assertions and disables debug level logs</td>
</tr>
<tr>
<td>Debug</td>
<td>-g -DWS_DEBUG</td>
<td>For development, no optimization. Enables assertions and debug level logs</td>
</tr>
<tr>
<td>Release</td>
<td>-O3 -DNDEBUG</td>
<td>Optimized for speed, no debug symbols or debug level logs or assertions</td>
</tr>
<tr>
<td>MinSizeRel</td>
<td>-Os -DNDEBUG</td>
<td>Optimized for size, no debug symbols or debug level logs or assertions</td>
</tr>
</tbody>
</table>

The default is `RelWithDebInfo`, which provides a good compromise of some optimization (-O2) along with including debug symbols (-g) for release builds. For normal development coding you probably want to be using `Debug` build type or set `-DENABLE_DEBUG=On`, to enable full logging capabilities and debug code.

CMake will automatically add the -DNDEBUG option to certain build types. This macro is used to disable assertions but it can be overruled using ENABLE_ASSERT, which can be used to unconditionally enable assertions if defined.

To change the build type, set the CMake variable `CMAKE_BUILD_TYPE`, e.g.:

```bash
$ cmake .. -DCMAKE_BUILD_TYPE=Debug
```

or on Windows,
> msbuild /m /p:Configuration=Debug Wireshark.sln

Run Your Version Of Wireshark

**TIP**

_Beware of multiple Wiresharks_

An already installed Wireshark may interfere with your newly generated version in various ways. If you have any problems getting your Wireshark running the first time, it might be a good idea to remove the previously installed version first.

Unix-Like Platforms

After a successful build you can run Wireshark right from the _run_ directory. There's no need to install it first.

`$ ./run/wireshark`

There's no need to run Wireshark as root user, but depending on your platform you might not be able to capture. Running Wireshark this way can be helpful since debugging output will be displayed in your terminal. You can also change Wireshark's behavior by setting various environment variables. See the **ENVIRONMENT VARIABLES** section of the Wireshark man page for more details.

On macOS, Wireshark is built as an application bundle (`run/Wireshark.app`) by default, and `run/wireshark` will be a wrapper script that runs `Wireshark.app/Contents/MacOS/Wireshark`. Along with running `./run/wireshark` as shown above you can also run it on the command line with `open run/Wireshark.app`.

Windows Native

By default the CMake-generated Visual C++ project places all of the files necessary to run Wireshark in the subdirectory `run\RelWithDebInfo`. As with the Unix-like build described above, you can run Wireshark from the build directory without installing it first.

>` \run\RelWithDebInfo\Wireshark`

Debug Your Version Of Wireshark

Optimization can make debugging a bit more difficult, e.g. by changing the execution order of statements. To disable optimization, set the **build type** to **Debug**.

Full debug logs can be invaluable to investigate any issues with the code. By default debug level logs
are only enabled with **Debug** build type. You can enable full debug logs and extra debugging code by configuring the ENABLE_DEBUG CMake option. This in turn will define the macro symbol `WS_DEBUG` and enable the full range of debugging code in Wireshark.

There is an additional CMake knob called `ENABLE_DEBUG_UTF_8` that can be used to control specifically the extra validation Wireshark performs internally for invalid UTF-8 encodings in internal strings, which should never happen and can be somewhat expensive to check during normal usage.

Conversely, the **Release** or **MinSizeRel** build types can be used to optimize further for speed or size, but do not include debug symbols for use with debuggers, and completely disable lower level logging and asserts, optimizing away the code path. Ensure that you have not built with one of those types before attempting debugging.

**Wireshark Logging**

Wireshark has a flexible logging system to assist in development and troubleshooting. Logging configuration takes into account what, when and where to output diagnostic messages.

- The 'what generates log messages' is defined through logging domain(s).
- The 'when it generates log messages' is defined through the logging level.
- The 'where it outputs log messages' is defined through the output channel(s).

The details to configure and use the logging system are explained in the following sections.

**Logging Domains**

Any part of Wireshark can be assigned a logging domain. This is already done for most of the internals of Wireshark, e.g., "Main", "Capture", "Epan", "GUI". The domains are defined in the `ws_log_defs.h` header but dissectors should define their own logging domain. Any string can be used as ID for a logging domain.

**Logging Levels**

The following logging levels are defined from highest to lowest:

- error
- critical
- warning
- message
- info
- debug
- noisy
By default logging output is generated for logging level "message" and above. If the logging level is lowered or raised all log output generated at or above this level is sent to the log output. Note that if the build type is not set to Debug then by default all log output for the logging levels "debug" and "noisy" will be optimized away by the compiler and cannot be emitted to the log output, regardless of the logging settings. To enable debug logging for all build types, set the CMake variable -DENABLE_DEBUG=ON.

There is also a special "echo" logging level used exclusively for temporary debugging print outs (usually via the WS_DEBUG_HERE macro).

Logging Output

By default logging output is sent to stderr. In addition to that it is possible to configure a log file. This collects all log output to the file, besides the normal output streams. The output can then be read in a text editor or used with other text processing tools.

A program can also register its own log writer when the standard facilities are insufficient or special handling is required.

Configure Logging

Logging can be configured through either environment variables or command line parameters.

The following environment variables and command line parameters are used by the logging system:

WIRESHARK_LOG_DOMAIN, WIRESHARK_LOG_DOMAINS, or --log-domain, --log-domains
This is a filter for the domain(s) which are to generate log messages.

WIRESHARK_LOG_LEVEL, or --log-level
This is the level (below critical) for which log messages are to be generated. This is used for all configured domains.

WIRESHARK_LOG_DEBUG, or --log-debug
These domain(s) will generate debug level log messages regardless of the log level and log domains configured.

WIRESHARK_LOG_NOISY, or --log_noisy
These domain(s) will generate noisy level log messages regardless of the log level and log domains configured.

Multiple domains can be concatenated using commas or semicolons. The match can be inverted by prefixing the domain(s) list with an exclamation mark.
Traps Set By Logging

Sometimes it can be helpful to abort the program right after a log message of a certain level or a certain domain is output.

The following environment variables are used to configure a trap by the logging system:

**WIRESHARK_LOG_FATAL**, or **--log_fatal**

This is the level for which log messages are fatal. This can either be "critical" or "warning" level.

**WIRESHARK_LOG_FATAL_DOMAIN**, **WIRESHARK_LOG_FATAL_DOMAINS**, or **--log-fatal-domain**, **--log-fatal-domains**

These are the domain(s) where output of a log message is fatal. This is less commonly used than the fatal log level setting above.

Logging APIs

The logging API can be found in `wsutil/wslog.h`.

To use the logging API for your code add the definition of the ID of your logging domain right after including `config.h`. For example:

```c
/* My code doing something awesome */
#include "config.h"
#define WS_LOG_DOMAIN "MyCode"

#include <wireshark.h>
...
```

Populate your code with the applicable function calls to generate log messages when enabled. The following convenience macros are provided:

- `ws_error()`
- `ws_critical()`
- `ws_warning()`
- `ws_message()`
- `ws_info()`
- `ws_debug()`
- `ws_noisy()`

All these take `printf()` style parameters. There is also a `WS_DEBUG_HERE` macro that is always active and outputs to a special "echo" domain for temporary debug print outs. `WS_DEBUG_HERE` should be
used for development purposes only and not appear in final delivery of the code.

**Unix-Like Platforms**

You can debug using command-line debuggers such as gdb, dbx, or lldb. If you prefer a graphic debugger, you can use an IDE or debugging frontend such as Qt Creator, CLion, or Eclipse.

Additional traps can be set on Wireshark, see [Traps Set By Logging](#)

**Memory Safety andLeaks**

Wireshark's wmem memory management framework makes it easy to allocate memory in pools with a certain scope that is freed automatically at a certain point (such as the end of dissecting a packet or when closing a file), even if a dissector raises an exception after allocating the memory. Memory in a pool is also freed collectively, which can be considerably faster than calling `free()` individually on each individual allocation. Proper use of wmem makes a dissector faster and less prone to memory leaks with unexpected data, which happens frequently with capture files.

However, wmem's block allocation can obscure issues that memory checkers might otherwise catch. Fortunately, the `WIRESHARK_DEBUG_WMEM_OVERRIDE` environment variable can be set at runtime to instruct wmem to use a specific memory allocator for all allocations, some of which are more compatible with memory checkers:

- **simple** - Uses `malloc()` only, no block allocation, compatible with Valgrind
- **strict** - Finds invalid memory via canaries and scrubbing freed memory
- **block** - Standard block allocator for file and epan scopes
- **block_fast** - Block allocator for short-lived scope, e.g. packet, (`free()` is a no-op)

The **simple** allocator produces the most accurate results with tools like Valgrind and can be enabled as follows:

```
$ export WIRESHARK_DEBUG_WMEM_OVERRIDE=simple
```

Wireshark uses GLib's GSlice memory allocator, either indirectly via wmem or via various GLib API calls. GLib provides a `G_SLICE` environment variable that can be set to `always-malloc` (similar to `simple`) or `debug-blocks` (similar to `strict`). See [https://developer-old.gnome.org/glib/stable/glib-running.html](https://developer-old.gnome.org/glib/stable/glib-running.html) for details. The C libraries on FreeBSD, Linux, and macOS also support memory allocation debugging via various environment variables. You can enable many of them by running `source tools/debug-alloc.env` in a POSIX shell.

If you're encountering memory safety bugs, you might want to build with Address Sanitizer (ASAN) so that Wireshark will immediately alert you to any detected issues. It works with GCC or Clang, provided that the appropriate libraries are installed.
$ cmake .. -G Ninja -DENABLE_ASAN=1
$ source ../tools/debug-alloc.env
$ ./run/tshark ...

TIP

ASAN slows things down by a factor of 2 (or more), so having a different build directory for an ASAN build can be useful.

ASAN will catch more errors when run with either the simple or strict wmem allocator than with the defaults. (It is more compatible with the strict allocator and the analogous GSlice debug-blocks option than Valgrind is.)

For additional instrumentation, ASAN supports a number of options.

For further investigating memory leaks, the following can be useful:

```bash
# This slows things down a lot more but results in more precise backtraces,  
# especially when calling third party libraries (such as the C++ standard  
# library):
$ export ASAN_OPTIONS=fast_unwind_on_malloc=0
# This causes LeakSanitizer to print the addresses of leaked objects for  
# inspection in a debugger:
$ export LSAN_OPTIONS=report_objects=1
```

LeakSanitizer and AddressSanitizer can detect issues in third-party libraries that you cannot do anything about. For example, internal Qt library calls to the fontconfig library can produce leaks. To ignore them, create a suppressions file with an appropriate entry, e.g. leak:libfontconfig.

If you are just interested in memory safety checking, but not memory leak debugging, disable the included LeakSanitizer with:

```bash
$ export ASAN_OPTIONS=detect_leaks=0
```

Windows Native

You can debug using the Visual Studio Debugger or WinDbg. See the section on using the Debugger Tools.

Make Changes To The Wireshark Sources

There are several reasons why you might want to change Wireshark’s sources:

- Add support for a new protocol (i.e., add a new dissector)
• Change or extend an existing dissector
• Fix a bug
• Implement a glorious new feature

Wireshark’s developers work on a variety of different platforms and use a variety of different development environments. Although we don’t enforce or recommend a particular environment, your editor should support EditorConfig in order to make sure you pick up the correct indentation style for any files that you might edit.

The internal structure of the Wireshark sources are described in Wireshark Development.

**TIP**

Ask the wireshark-dev[AT]wireshark.org mailing list before you start a new development task.

If you have an idea what you want to add or change it’s a good idea to contact the developer mailing list (see Mailing Lists) and explain your idea. Someone else might already be working on the same topic, so a duplicated effort can be reduced. Someone might also give you tips that should be thought about (like side effects that are sometimes very hard to see).

Contribute Your Changes

If you have finished changing the Wireshark sources to suit your needs, you might want to contribute your changes back to the Wireshark community. You gain the following benefits by contributing your improvements:

*It's the right thing to do.*

Other people who find your contributions useful will appreciate them, and you will know that you have helped people in the same way that the developers of Wireshark have helped you.

*You get free enhancements.*

By making your code public, other developers have a chance to make improvements, as there’s always room for improvements. In addition someone may implement advanced features on top of your code, which can be useful for yourself too.

You save time and effort. The maintainers and developers of Wireshark will maintain your code as well, updating it when API changes or other changes are made, and generally keeping it in tune with what is happening with Wireshark. So if Wireshark is updated (which is done often), you can get a new Wireshark version from the website and your changes will already be included without any effort for you.

There’s no direct way to push changes to the main repository. Only a few people are authorised to actually make changes to the source code (check-in changed files). If you want to submit your changes, you should upload them to the code review system at https://gitlab.com/wireshark/wireshark/-/merge_requests. This requires you to set up git as described at Git Over SSH Or HTTPS.
Creating Merge Requests

GitLab uses a forking workflow, which looks like this:

![GitLab Workflow Diagram](image)

**Figure 1. GitLab Workflow**

In the diagram above, your fork can be created by pressing the “Fork” button at https://gitlab.com/wireshark/wireshark. Your local repository can be created as described in Git Over SSH Or HTTPS. You only need to do this once. You should pull from the main repository on a regular basis in order to ensure that your sources are current. You should push any time you want to make a merge request or otherwise make your code public. The “Pull”, “Push”, and “Merge Request” parts of the workflow are important, so let’s look at them in more detail.

First, you need to set up your environment. For the steps below we’ll pretend that your username is “henry.perry”.

1. Sign in to https://gitlab.com/wireshark/wireshark by clicking “Sign in / Register” in the upper right corner of the web page and following the login instructions.
2. Add an SSH key to your account as described in the GitLab documentation.
3. Make sure you have a clone of the main repository as described in Git Over SSH Or HTTPS.
4. Create your own personal fork of the Wireshark project by pressing the “Fork” button at https://gitlab.com/wireshark/wireshark.

**WARNING** If you want to make merge requests you must keep your fork public. Making it private will disassociate it from the main Wireshark repository.

5. Add a remote for your personal repository. The main repository remote is named “upstream”, so we’ll name this one “downstream”.

   ```bash
   $ git remote add downstream git@gitlab.com:henry.perry/wireshark.git
   ```

6. Double-check your remotes:
Before you begin it’s a good idea to synchronize your local repository with the main repository. This is the **Pull** part of the workflow. You should do this periodically in order to stay up to date and avoid merge conflicts later on.

1. Fetch and optionally apply the latest changes.

   ```
   # Fetch changes from upstream and apply them to the current branch...
   $ git pull --rebase upstream master
   # ...or fetch changes and leave the current branch alone
   $ git fetch upstream
   ```

Now you're ready to create a merge request (the **Push** and **Merge Request** parts of the workflow above).

1. First, create a branch for your change:

   ```
   $ git checkout -b my-glorious-new-feature upstream/master
   ```

2. Write some code! See [Some Tips For A Good Patch](#) and [Code Requirements](#) for details.

3. Commit your changes. See [Writing a Good Commit Message](#) for details.

   ```
   $ git commit -a
   ```

4. Push your changes to your personal repository.

   ```
   $ git push downstream HEAD
   ```

5. Go to [https://gitlab.com/wireshark/wireshark/-/merge_requests](https://gitlab.com/wireshark/wireshark/-/merge_requests). You should see a “Create merge request” button. Press it.

6. In the merge request page, make sure “Allow commits from members who can merge to the target branch” is selected so that core developers can rebase your change. You might want to select “Delete source branch when merge request is accepted” as well. Click the “Submit merge request” button.
Updating Merge Requests

At this point various automated tests will be run and someone will review your change. If you need to make changes you can do so by force-pushing it to the same branch in your personal repository.

1. Push your changes to your personal repository.

```bash
# First, make sure you're on the right branch.
$ git status
On branch my-glorious-new-feature
```

2. Update your code.

3. Push your changes to your personal repository.

```bash
# Modify the current commit and force-push...
$ git commit --amend ...
$ git push downstream +HEAD
# ...or keep the current commit as-is add another commit on top of it
$ git commit ...
$ git push downstream HEAD
```

The + sign is shorthand for forcing the push (-f).

Some Tips For A Good Patch

Some tips that will make the merging of your changes into Git much more likely (and you want exactly that, don’t you?):

*Use the latest Git sources.*

It’s a good idea to work with the same sources that are used by the other developers. This usually makes it much easier to apply your patch. For information about the different ways to get the sources, see [Obtaining The Wireshark Sources](#).

*Update your sources just before making a patch.*

For the same reasons as the previous point.

*Inspect your patch carefully.*

Run `git diff` or `git show` as appropriate and make sure you aren’t adding, removing, or omitting anything you shouldn’t.

*Give your branch a brief but descriptive name.*

Short, specific names such as `snowcone-machine-protocol` are preferred.

*Don’t put unrelated things into one large change.*
Merge requests should be limited in scope. For example, updates to the Snowcone Machine Protocol dissector and the Coloring Rules dialog box should be in separate merge requests.

In general, making it easier to understand and apply your patch by one of the maintainers will make it much more likely (and faster) that it will actually be applied.

Thank you in advance for your patience.

Wireshark is a volunteer effort. As a result, we can’t guarantee a quick turnaround time.

Preview the final product.

Wireshark’s GitLab CI jobs are disabled by default for forks, but if you need to test any CI jobs you can do so under the “Pipelines” section in your repository. For example, if your change might affect Debian (apt) packaging you can run the “build:debian-stable” job.

Writing a Good Commit Message

When running `git commit`, you will be prompted to describe your change. Here are some guidelines on how to make that message more useful to other people (and to scripts that may try to parse it):

Provide a brief description (under 60 characters or so) of the change in the first line.
If the change is specific to a single protocol, start this line with the abbreviated name of the protocol and a colon. If the change is not yet complete prefix the line with “WIP:” to inform this change not to be submitted yet. This be removed when the change is ready to be merged.

Insert a single blank line after the first line.
This is required by various formatting tools and helpful to humans.

Provide a detailed description of the change in the lines that follow.
Break paragraphs where needed. Limit each line to 80 characters.

You can also reference and close issues in a commit message by prefixing the issue number with a number sign. For example, “closes #5” will close issue number 5.

Putting all that together, we get the following example:

```
MIPv6: Fix dissection of Service Selection Identifier

APN field is not encoded as a dotted string so the first character is not a length. Closes #10323.
```

Code Requirements

To ensure Wireshark’s code quality and to reduce friction in the code review process, there are some things you should consider before submitting a patch:
Follow the Wireshark source code style guide.

Wireshark runs on many platforms, and can be compiled with a number of different compilers. It's easy to write code that compiles on your machine, but doesn't compile elsewhere. The guidelines at Coding Style describe the techniques and APIs that you can use to write high-quality, portable, and maintainable code in our environment.

Submit dissectors as built-in whenever possible.

Developing a new dissector as a plugin can make compiling and testing quicker, but it's usually best to convert it to built-in before submitting for review. This reduces the number of files that must be installed with Wireshark and ensures your dissector will be available on all platforms.

Dissectors vary, so this is not a hard-and-fast rule. Most dissectors are single C modules that can easily be put into “the big pile.” Some (most notably ASN.1 dissectors) are generated using templates and configuration files. Others are split across multiple source files and are often more suitable to be placed in a separate plugin directory.

Ensure that the Wireshark Git Pre-Commit Hook is in the repository.

In your local repository directory, there will be a .git/hooks/ directory, with sample git hooks for running automatic actions before and after git commands. You can also optionally install other hooks that you find useful.

In particular, the pre-commit hook will run every time you commit a change and can be used to automatically check for various errors in your code. The sample git pre-commit hook simply detects whitespace errors such as mixed tabs and spaces. To install it just remove the .sample suffix from the existing pre-commit.sample file.

Wireshark provides a custom pre-commit hook which does additional Wireshark-specific API and formatting checks, but it might return false positives. If you want to install it, copy the pre-commit file from the tools directory (cp ./tools/pre-commit .git/hooks/) and make sure it is executable or it will not be run.

If the pre-commit hook is preventing you from committing what you believe is a valid change, you can run git commit --no-verify to skip running the hooks. Warning: using --no-verify avoids the commit-msg hook, and thus if you have setup this hook it will not run.

Additionally, if your system supports symbolic links, as all UNIX-like platforms do, you can use them instead of copying files. Running ln -s ../../tools/pre-commit .git/hooks creates a symbolic link that will make the hook to be up-to-date with the current master.

Choose a compatible license.

Wireshark is released under the GPL version 2 or later, and it is strongly recommended that incoming code use that license. If that is not possible, it must use a compatible license. The following licenses are currently allowed:

- BSD 1, 2, 3 clause
- ISC
Fuzz test your changes.

Fuzz testing is a very effective way of finding dissector related bugs. In our case fuzzing involves making random changes to capture files and feeding them to TShark in order to try to make it crash or hang. There are tools available to automatically do this on any number of input files. See https://gitlab.com/wireshark/wireshark/-/wikis/FuzzTesting for details.

Backporting A Change

When a bug is fixed in the master branch it’s sometimes desirable or necessary to backport the fix to a release branch. You can do this in Git by cherry-picking the change from one branch to another. Suppose you want to backport change 1ab2c3d4 from the master branch to master-3.2. You can do so as follows:

```shell
# Create a new topic branch for the backport.
$ git checkout -b backport-1ab2c3d4 upstream/master-3.2

# Cherry-pick the change. Include a "cherry picked from..." line.
$ git cherry-pick -x 1ab2c3d4

# If there are conflicts, fix them.

# Compile and test the change.
$ ninja
$ ...

# OPTIONAL: Add entries to docbook/release-notes.adoc.
$EDITOR docbook/release-notes.adoc

# If you made any changes, update your commit.
git commit --amend -a

# Push the change to your working repository.
git push downstream HEAD
```

You can also cherry-pick changes in the GitLab web UI.
Binary Packaging

Delivering binary packages makes it much easier for the end-users to install Wireshark on their target system. This section will explain how the binary packages are made.

Packaging Guidelines

The following guidelines should be followed by anyone creating and distributing third-party Wireshark packages or redistributing official Wireshark packages.

Spelling And Capitalization

Wireshark is spelled with a capital “W”, and with everything else lower case. “WireShark” in particular is incorrect.

Main URL

The official Wireshark project URL is https://www.wireshark.org/.

Download URLs

Official packages are distributed on the main web server (www.wireshark.org) and a number of download mirrors. The canonical locations for packages are in the all_versions subdirectories on each server.

For example, if your packaging system links to or downloads the source tarball and you want to download from 1.na.dl.wireshark.org, use

https://1.na.dl.wireshark.org/download/src/all-versions/wireshark-4.3.0.tar.xz

instead of

https://1.na.dl.wireshark.org/download/src/wireshark-4.3.0.tar.xz

Staying Current

Wireshark releases are announced on the wireshark-announce mailing list. A PAD file is also published at https://www.wireshark.org/wireshark-pad.xml which contains the current stable version and release date.

Artwork

Logo and icon artwork can be found in the image directory in the distribution. This is available online at

https://gitlab.com/wireshark/wireshark/-/tree/master/image
Licensing

Wireshark is released under the GNU General Public License version 2 or later. Make sure you and your package comply with this license.

Trademarks

Wireshark and the “fin” logo are registered trademarks of the Wireshark Foundation. Make sure you and your package comply with trademark law.

Privileges

All function calls that require elevated privileges are in dumpcap.

WIRESHARK CONTAINS OVER THREE MILLION LINES OF SOURCE CODE. DO NOT RUN THEM AS ROOT.

Warnings are displayed when Wireshark and TShark are run as root.

There are two configure-time options on non-Windows systems that affect the privileges a normal user needs to capture traffic and list interfaces:

- **DDUMPCAP_INSTALL_OPTION=capabilities**
  
  Install dumpcap with cap_net_admin and cap_net_raw capabilities. Linux only.

- **DDUMPCAP_INSTALL_OPTION=suid**
  
  Install dumpcap setuid root.

These are necessary for non-root users to be able to capture on most systems, e.g. on Linux or FreeBSD if the user doesn't have permissions to access /dev/bpf*. Setcap installation is preferred over setuid on Linux. If `-DDUMPCAP_INSTALL_OPTION=capabilities` is used it will override any setuid settings.

The **-DENABLE_CAP** option is only useful when dumpcap is installed setuid. If it is enabled dumpcap will try to drop any setuid privileges it may have while retaining the `CAP_NET_ADMIN` and `CAP_NET_RAW` capabilities. It is enabled by default, if the Linux capabilities library (on which it depends) is found.

Note that enabling setcap or setuid installation allows packet capture for ALL users on your system. If this is not desired, you can restrict dumpcap execution to a specific group or user. The following two examples show how to restrict access using setcap and setuid respectively:
Customization

Custom version information can be added by running tools/make-version.py. If your package contains significant changes we recommend that you use this to differentiate it from official Wireshark releases.

```
tools/make-version.py --set-release --untagged-version-extra=-{vcsinfo}-FooCorp
--tagged-version-extra=-FooCorp
```

See tools/make-version.py for details.

The Git version corresponding to each release is in version.h. It’s defined as a string. If you need a numeric definition, let us know.

If you have a question not addressed here, please contact wireshark-dev[AT]wireshark.org.

Debian: .deb Packages

The Debian Package is built using dpkg-buildpackage, based on information found in the source tree under packaging/debian. You must create a top-level symbolic link to packaging/debian before building. See https://www.debian.org/doc/manuals/maint-guide/build.en.html for a more in-depth discussion of the build process.

In the wireshark directory, type:

```
ln -snf packaging/debian
export DEB_BUILD_OPTIONS="nocheck"
dpkg-buildpackage -b -us -uc -jauto
```

to build the Debian Package.

Red Hat: .rpm Packages

You can build an RPM package using the wireshark_rpm target. If you are building from a git
checkout, the package version is derived from the current git HEAD. If you are building from source extracted from a tarball created with git archive (such as those downloaded from http://www.wireshark.org/download.html), you must place the original tarball into your build directory.

The package is built using rpmbuild, which comes as standard on many flavours of Linux, including Red Hat, Fedora, and openSUSE. The process creates a clean build environment in ${CMAKE_BINARY_DIR}/packaging/rpm/BUILD each time the RPM is built. The settings that control the build are in ${CMAKE_SOURCE_DIR}/packaging/rpm/wireshark.spec.in. The generated SPEC file contains CMake flags and other settings for the RPM build environment. Many of these come from the parent CMake environment. Notable ones are:

- **prefix** is set to CMAKE_INSTALL_PREFIX. By default this is /usr/local. Pass -DCMAKE_INSTALL_PREFIX=/usr to create a package that installs into /usr.
- Whether or not to create the “wireshark-qt” package (-DBUILD_wireshark).
- Lua, c-ares, nghttp2, and other library support (-DENABLE_…).
- Building with Ninja (-G Ninja).

In your build directory, type:

```
ninja wireshark_rpm
# ...or, if you're using GNU make...
make wireshark_rpm
```

to build the binary and source RPMs. When it is finished there will be a message stating where the built RPM can be found.

**TIP**

*This might take a while*

This creates a tarball, extracts it, compiles Wireshark, and constructs a package. This can take quite a long time. You can speed up the process by using Ninja. If you're using GNU make you can add the following to your ~/.rpmmacros file to enable parallel builds:

```
%_smp_mflags -j %(grep -c processor /proc/cpuinfo)
```

Building the RPM package requires quite a few packages and libraries including GLib, gcc, flex, Asciidoctor, and Qt development tools such as uic and moc. The required Qt packages can usually be obtained by installing the qt5-devel package. For a complete list of build requirements, look for the “BuildRequires” lines in packaging.rpm/wireshark.spec.in.
macOS: .dmg Packages

The macOS Package is built using macOS packaging tools, based on information found in the source tree under packaging/macossx. It requires Asciidoctor and dmgbuild.

In your build directory, type:

```
ninja wireshark_dmg logray_dmg # (Modify as needed)
# ...or, if you're using GNU make...
make wireshark_dmg logray_dmg # (Modify as needed)
```

to build the macOS Packages.

Windows: NSIS .exe Installer

The Nullsoft Install System is a free installer generator for Windows systems. Instructions on installing it can be found in Windows: NSIS (Optional). NSIS is script based. You can find the main Wireshark installer generation script at packaging/nis/wireshark.nsi.

When building with CMake you must first build the `wireshark_nsis_prep` target, followed by the `wireshark_nsis` target, e.g.

```
> msbuild /m /p:Configuration=RelWithDebInfo wireshark_nsis_prep.vcxproj
> msbuild /m /p:Configuration=RelWithDebInfo wireshark_nsis.vcxproj
```

Splitting the packaging projects in this way allows for code signing.

- **TIP**
  
  This might take a while

  Please be patient while the package is compressed. It might take some time, even on fast machines.

If everything went well, you will now find something like: `wireshark-setup-4.3.0.exe` in the `packaging/nsis` directory in your build directory.

Windows: PortableApps .paf.exe Package

PortableApps.com is an environment that lets users run popular applications from portable media such as flash drives and cloud drive services.

- Install the PortableApps.com Platform. Install for “all users”, which will place it in `C:\PortableApps`.
- Add the following apps:
  - PortableApps.com Installer
When building with CMake you must first build the `wireshark_nsis_prep` target (which takes care of general packaging dependencies), followed by the `wireshark_portableapps` target, e.g.

```
> msbuild /m /p:Configuration=RelWithDebInfo wireshark_nsis_prep.vcxproj
> msbuild /m /p:Configuration=RelWithDebInfo wireshark_portableapps.vcxproj
```

TIP

*This might take a while*

Please be patient while the package is compressed. It might take some time, even on fast machines.

If everything went well, you will now find something like: `WiresharkPortable64_4.3.0.paf.exe` in the `packaging/portableapps` directory.

## Mime Types

Wireshark uses various mime-types for dragging dropping as well as file formats. This chapter gives an overview over all the mime types being used, as well as the data format in which data has to be provided for each individual mimetype.

If not otherwise stated, the data is encoded as a JSON Object.

### Display Filter

**MimeType:** `application/vnd.wireshark.displayfilter`

Display filters are being dragged and dropped by utilizing this mime type.

```json
{
  "filter": "udp.port == 8080",
  "field": "udp.port",
  "description": "UDP Port"
}
```

### Coloring Rules

**MimeType:** `application/vnd.wireshark.coloringrules`

Coloring Rules are being used for dragging and dropping color rules inside the coloring rules dialog.
Filter List

MimeType: application/vnd.wireshark.filterlist

*Internal Use only* - used on the filter list for moving entries within the list

Column List

MimeType: application/vnd.wireshark.columnlist

*Internal Use only* - used on the column list for moving entries within the list
Tool Reference

Introduction

This chapter will provide you with information about the various tools needed for Wireshark development. None of the tools mentioned in this chapter are needed to run Wireshark. They are only needed to build it.

Most of these tools have their roots on UNIX or UNIX-like platforms such as Linux, but Windows ports are also available. Therefore the tools are available in different "flavours":

- UNIX and UNIX-like platforms: The tools should be commonly available on the supported UNIX and UNIX-like platforms. Cygwin is unsupported.

- Windows native: Some tools are available as native Windows tools, no special emulation is required. Many of these tools can be installed (and updated) using Chocolatey, a Windows package manager similar to the Linux package managers apt-get or yum.

WARNING

Follow the directions

Unless you know exactly what you are doing, you should strictly follow the recommendations given in Setup and Build Instructions.

The following sections give a very brief description of what a particular tool is doing, how it is used in the Wireshark project and how it can be installed and tested.

Documentation for these tools is outside the scope of this document. If you need further information on using a specific tool you should find lots of useful information on the web, as these tools are commonly used. You can also get help for the UNIX based tools with toolname --help or the man page via man toolname.

You will find explanations of the tool usage for some of the specific development tasks in Work with the Wireshark sources.

Chocolatey

Chocolatey is a Windows package manager that can be used to install (and update) many of the packages required for Wireshark development. Chocolatey can be obtained from the website or from a Command Prompt:

C:\@powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object net.webclient).DownloadString(_https://chocolatey.org/install.ps1_))" & & SET PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin

or a Powershell prompt:
Chocolatey sometimes installs packages in unexpected locations. Python is a notable example. While it's typically installed in a top-level directory, e.g. C:\Python37 or in %PROGRAMFILES%, e.g. C:\Program Files\Python37, Chocolatey tends to install it under C:\ProgramData\chocolatey or C:\Tools. If you want to avoid this behavior you'll probably want to install Python using the packages from python.org.

Other package managers for Windows include the Windows Package Manager (winget) and Scoop. As of January 2022 neither option provides all of the packages we require, but that might change in the future.

**CMake**

Wireshark's build environment can be configured using CMake on various UNIX-like platforms, including Linux, macOS, and *BSD, and on Windows. CMake is designed to support out-of-tree builds - so much so that in-tree builds do not work properly in all cases. Along with being cross-platform, CMake supports many build tools and environments including traditional make, Ninja, and MSBuild.

Building with CMake typically includes creating a build directory and specifying a generator, aka a build tool. For example, to build Wireshark using Ninja in the directory `wireshark-ninja` you might run the following commands:

```
# Starting from your Wireshark source directory, create a build directory
# alongside it.
$ cd ..
$ mkdir wireshark-ninja
$ cd wireshark-ninja
# Assumes your source directory is named "wireshark".
$ cmake -G Ninja ../wireshark
$ ninja (or cmake --build .)
```

Using CMake on Windows is described further in Generate the build files.

Along with specifying a generator with the `-G` flag you can set variables using the `-D` flag. Useful variables and generators include the following:

**-DBUILD_wireshark=OFF**

Don't build the Wireshark GUI application. Each command line utility has its own BUILD_xxx flag as well. For example, you can use -DBUILD_mmdbresolve=OFF to disable mmdbresolve.
-DENABLE_CCACHE=ON
   Build using the ccache compiler cache.

-DCAP=OFF
   Disable the POSIX capabilities check

-DCMAKE_BUILD_TYPE=Debug
   Enable debugging symbols

-DCARES_INCLUDE_DIR=/your/custom/cares/include,
-DCARES_LIBRARY=/your/custom/cares/lib/libcares.so
   Let you set the path to a locally-compiled version of c-ares. Most optional libraries have xxx_INCLUDE_DIR and xxx_LIB flags that let you control their discovery.

-DCMAKE_OSX_DEPLOYMENT_TARGET=10.12
   Specify the minimum macOS version for Wireshark and each command line utility. Note that this doesn't affect the minimum target for third-party libraries. For example, if you're building for macOS 10.12 you'll need to install Qt 5.14 or earlier and ensure that other libraries support macOS 10.12, for example by running tools/macos-setup.sh -t 10.12.

-DENABLE_APPLICATION_BUNDLE=OFF
   Disable building an application bundle (Wireshark.app) on macOS

You can list all build variables (with help) by running cmake -LH [options] ../<Name_of_WS_source_dir>. This lists the cache of build variables after the cmake run. To only view the current cache, add option -N.

Depending on your needs, it might be useful to save your CMake configuration options in a file outside your build directory. CMake supports this via its presets option. For example, adding the following to CMakeUserPresets.json would let you build using Ninja in the build directory, enable ccache, and set a custom Qt directory by running cmake --preset mydev:
After running cmake, you can always run `make help` to see a list of all possible make targets.

Note that CMake honors user umask for creating directories as of now. You should set the umask explicitly before running the `install` target.

CMake links:

The home page of the CMake project: [https://cmake.org/](https://cmake.org/)

Official documentation: [https://cmake.org/documentation/](https://cmake.org/documentation/)

About CMake in general and why KDE4 uses it: [https://lwn.net/Articles/188693/](https://lwn.net/Articles/188693/)


Frequently Asked Questions: [https://gitlab.kitware.com/cmake/community/wikis/FAQ](https://gitlab.kitware.com/cmake/community/wikis/FAQ)

**GNU Compiler Toolchain (UNIX And UNIX-like Platforms)**

**gcc (GNU Compiler Collection)**

The GCC C compiler is available for most UNIX and UNIX-like operating systems.

If GCC isn't already installed or available as a package for your platform, you can get it at: [https://gcc.gnu.org/](https://gcc.gnu.org/).

After correct installation, typing at the bash command line prompt:
$ gcc --version

should result in something like

gcc (Ubuntu 4.9.1-16ubuntu6) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Your version string may vary, of course.

**gdb (GNU Project Debugger)**

GDB is the debugger for the GCC compiler. It is available for many (if not all) UNIX-like platforms.

If you don’t like debugging using the command line, many [GUI frontends for it available](https://www.gnu.org/software/gdb/gdb.html), including Qt Creator, CLion, and Eclipse.

If gdb isn’t already installed or available as a package for your platform, you can get it at: [https://www.gnu.org/software/gdb/gdb.html](https://www.gnu.org/software/gdb/gdb.html).

After correct installation:

$ gdb --version

should result in something like:

GNU gdb (GDB) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Your version string may vary, of course.

**make (GNU Make)**

*NOTE*  
*GNU make isn’t supported either for Windows*

GNU Make is available for most of the UNIX-like platforms.

If GNU Make isn’t already installed or available as a package for your platform, you can get it at: [https://www.gnu.org/software/make/](https://www.gnu.org/software/make/).
After correct installation:

```
$ make --version
```

should result in something like:

```
GNU Make 4.0
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2013 Free Software Foundation, Inc.
Licence GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Your version string may vary, of course.

**Ninja**

Ninja is an alternative to make, and is available for many of the UNIX-like platforms. It runs builds faster than make does.

It is designed to have its build files generated by tools such as CMake; to generate build files for Ninja, run CMake with the `-G Ninja` flag.

If Ninja isn’t already installed, see the list of suggestions for Ninja packages at: [https://github.com/ninja-build/ninja/wiki/Pre-built-Ninja-packages](https://github.com/ninja-build/ninja/wiki/Pre-built-Ninja-packages).

If Ninja isn’t already installed and isn’t available as a package for your platform, you can get it from: [https://ninja-build.org](https://ninja-build.org). You can download the source code or binaries for Linux, macOS, and Windows (we have not tested Ninja on Windows).

**Microsoft compiler toolchain (Windows native)**

To compile Wireshark on Windows using the Microsoft C/C++ compiler (MSVC), you’ll need:

1. C compiler (`cl.exe`)
2. Assembler (`ml.exe` for 32-bit targets and `ml64.exe` for 64-bit targets)
3. Linker (`link.exe`)
4. Resource Compiler (`rc.exe`)
5. C runtime headers and libraries (e.g. `stdio.h`, `vcruntime140.lib`)
6. Windows platform headers and libraries (e.g. `windows.h`, `WS2_32.lib`)
Official Toolchain Packages And Alternatives

Official releases are or were built with the following Visual C++ versions:

- Wireshark 4.2.x: Microsoft Visual C++ 2022.
- Wireshark 4.0.x: Microsoft Visual C++ 2022.
- Wireshark 2.4.x: Microsoft Visual C++ 2015.
- Wireshark 2.2.x: Microsoft Visual C++ 2013.
- Wireshark 1.10.x: Microsoft Visual C++ 2010 SP1.
- Wireshark 1.8.x: Microsoft Visual C++ 2010 SP1.
- Wireshark 1.6.x: Microsoft Visual C++ 2008 SP1.
- Wireshark 1.4.x: Microsoft Visual C++ 2008 SP1.
- Wireshark 1.2.x: Microsoft Visual C++ 2008 SP1.
- Wireshark 1.0.x and earlier: Microsoft Visual C++ 6.0.

Using the release compilers is recommended for Wireshark development work.

“Community” editions of Visual Studio such as “Visual Studio Community 2022” can be used to compile Wireshark but any PortableApps packages you create with them might require the installation of a separate Visual C++ Redistributable package on any machine on which the PortableApps package is to be used. See Visual C++ Runtime “Redistributable” Files below for more details.

However, you might already have a different Microsoft C++ compiler installed. It should be possible to use any of the following with the considerations listed. You will need to sign up for a Visual Studio Dev Essentials account if you don't have a Visual Studio (MSDN) subscription. The older versions can be downloaded from https://visualstudio.microsoft.com/vs/older-downloads/.

Visual C++ 2022 Community Edition

IDE + Debugger?

Yes
**SDK required for 64-bit builds?**

No

CMake Generator: **Visual Studio 17**

You can use Chocolatey to install Visual Studio, e.g:

```
PS:\> choco install visualstudiocommunity2022 visualstudio2022-workload-nativedesktop
```

If you wish to build Arm64 executables you must install the following components:

**Microsoft.VisualStudio.Component.VC.Tools.ARM64**

MSVC v143 - VS 2022 C++ ARM64/ARM64EC build tools (Latest)

**Microsoft.VisualStudio.Component.VC.Runtimes.ARM64.Spectre**

MSVC v143 - VS 2022 C++ ARM64/ARM64EC Spectre-mitigated libs (Latest)

**cl.exe (C Compiler)**

The following table gives an overview of the possible Microsoft toolchain variants and their specific C compiler versions ordered by release date.

<table>
<thead>
<tr>
<th>Compiler Package</th>
<th>VC++</th>
<th>_MSC_VER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio 2022 (17.4.2)</td>
<td>14.34</td>
<td>1934</td>
</tr>
</tbody>
</table>

A description of **_MSC_VER** and **_MSC_FULL_VER**, and their relation to Visual Studio and compiler versions, can be found at [Microsoft-specific predefined macros](#).

Information on the VC++ version can be found in the file `wsutil/version_info.c`.

After correct installation of the toolchain, typing at the Visual Studio Command line prompt (cmd.exe):

```
> cl
```

should result in something like:

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.23.28106.4 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

However, the version string may vary.
link.exe (Linker)

After correct installation, typing at the Visual Studio Command line prompt (cmd.exe):

```
> link
```

should result in something like:

```
Microsoft (R) Incremental Linker Version 14.23.28106.4
Copyright (C) Microsoft Corporation. All rights reserved.

usage: LINK [options] [files] [@commandfile]
... 
```

However, the version string may vary.

Documentation on recent versions of the linker can be found at Microsoft Docs

Visual C++ Runtime “Redistributable” Files

Please note: The following is not legal advice. Ask your preferred lawyer instead. It’s the authors view and this view might be wrong.

Wireshark and its libraries depend on POSIX functions such as fopen() and malloc(). On Windows, these functions are provided by the Microsoft Visual C++ C Runtime (CRT). There are many different versions of the CRT and Visual C++ 2015 and later use the Universal CRT (UCRT).

The Universal CRT comes standard with Windows 10 and is installed as part of Windows Update on earlier versions of Windows. The Wireshark .exe installers include redistributables (vc_redist.x64.exe or vc_redist.x86.exe) which ensure that the Universal CRT is installed and up to date.

NOTE  
Make sure you’re allowed to distribute this file

The files to redistribute must be mentioned in the redist.txt file of the compiler package. Otherwise it can’t be legally redistributed by third parties like us.

The following Microsoft Docs link is recommended for the interested reader:

Redistributing Visual C++ Files

In all cases where vc_redist.x64.exe or vc_redist.x86.exe is downloaded it should be downloaded to the directory into which the support libraries for Wireshark have been downloaded and installed.
This directory is specified by the `WIRESHARK_BASE_DIR` or `WIRESHARK_LIB_DIR` environment variables. It need not, and should not, be run after being downloaded.

**Windows Platform SDK**

The Windows Platform SDK (PSDK) or Windows SDK is a free (as in beer) download and contains platform specific headers and libraries (e.g. `windows.h`, `WSock32.lib`, etc.). As new Windows features evolve in time, updated SDKs become available that include new and updated APIs.

When you purchase a commercial Visual Studio or use the Community Edition, it will include an SDK.

**Documentation Toolchain**

Wireshark’s documentation is split across two directories. The `doc` directory contains man pages written in Asciidoctor markup. The `docbook` directory contains the User’s Guide, Developer’s Guide, and the release notes, which are also written in Asciidoctor markup. The split is for historical reasons (described below), and the documentation will likely be consolidated into one directory in the future.

Our various output formats are generated using the following tools. Intermediate formats are in *italics*.

**Man page roff**

Asciidoctor

**Man page HTML**

Asciidoctor

**Guide HTML**

Asciidoctor → *DocBook XML* → `xsltproc + DocBook XSL`

**Guide PDF**

Asciidoctor

**Release notes HTML**

Asciidoctor

**Release notes text**

Asciidoctor → *HTML* → `html2text.py`

**Asciidoctor**

Asciidoctor comes in several flavors: a Ruby gem (Asciidoctor), a Java bundle (AsciidoctorJ), and transpiled JavaScript (Asciidoctor.js). The Ruby and Java flavors can be used to build Wireshark’s
documentation, but the JavaScript flavor doesn’t support all of the features that we require.

The guides and release notes were originally written in DocBook (hence the directory name). They were later converted to AsciiDoc and then migrated to Asciidoctor. The man pages were originally in Perl’s POD (Plain Old Documentation) format and were later converted to Asciidoctor. We use Asciidoctor’s modern (>= 1.5.0) syntax.

PDF output requires Asciidoctor’s PDF backend. It is included with AsciidoctorJ but not with Asciidoctor.

**DocBook XML and XSL**

Converting from DocBook to HTML requires the DocBook DTD ([http://www.sagehill.net/docbookxsl/ToolsSetup.html](http://www.sagehill.net/docbookxsl/ToolsSetup.html)) and DocBook stylesheets ([http://www.sagehill.net/docbookxsl/InstallStylesheets.html](http://www.sagehill.net/docbookxsl/InstallStylesheets.html)). These are available via installable packages on most Linux distributions, Chocolatey, and Homebrew.

**xsltproc**

**xsltproc** converts DocBook XML to various formats based on XSL stylesheets. It either ships as part of the operating system or is available via an installable package on most Linux distributions, Chocolatey, and Homebrew.

**Debugger**

Using a good debugger can save you a lot of development time.

The debugger you use must match the C compiler Wireshark was compiled with, otherwise the debugger will simply fail or you will only see a lot of garbage.

**Visual Studio Integrated Debugger**

You can use the integrated debugger of Visual Studio if your toolchain includes it. Open the solution in your build directory and build and debug as normal with a Visual Studio solution.

To set the correct paths for Visual Studio when running Wireshark under the debugger, add the build output directory to the path before opening Visual Studio from the same command prompt, e.g.

```
C:\Development\wsbuild64>set PATH="%PATH%;C:\Development\wsbuild64\run\RelwithDebInfo"
C:\Development\wsbuild64>wireshark.sln
```

for PowerShell use
When Visual Studio has finished loading the solution, set the executable to be run in the debugger, e.g. Executables\Wireshark, by right clicking it in the Solution Explorer window and selecting "Set as StartUp Project". Also set the Solution Configuration (usually RelWithDebInfo) from the droplist on the toolbar.

**NOTE**

Currently Visual Studio regards a command line build as incomplete, so will report that some items need to be built when starting the debugger. These can either be rebuilt or ignored as you wish.

The normal build is an optimised release version so debugging can be a bit difficult as variables are optimised out into registers and the execution order of statements can jump around.

If you require a non-optimised version, then build using a debug configuration.

**Debugging Tools For Windows**

You can also use the Microsoft Debugging Tools for Windows toolkit, which is a standalone GUI debugger. Although it’s not that comfortable compared to debugging with the Visual Studio integrated debugger it can be helpful if you have to debug on a machine where an integrated debugger is not available.

You can get it free of charge from Microsoft in several ways, see the [Debugging tools for Windows](###) page.

You can also use Chocolatey to install WinDbg:

```
PS:\> choco install windbg
```

To debug Wireshark using WinDbg, open the built copy of Wireshark using the File → Open Executable... menu, i.e. C:\Development\wsbuild64\run\RelWithDebInfo\Wireshark.exe. To set a breakpoint open the required source file using the File → Open Source File... menu and then click on the required line and press F9. To run the program, press F5.

If you require a non-optimised version, then build using a debug configuration, e.g. `msbuild /m /p:Configuration=Debug Wireshark.sln`. The build products will be found in C:\Development\wsbuild64\run\Debug\.

**bash**

The bash shell is needed to run several shell scripts.
Unix

Bash (the GNU Bourne-Again SHell) is available for most UNIX and UNIX-like platforms. If it isn’t already installed or available as a package for your platform, you can get it at https://www.gnu.org/software/bash/bash.html.

After correct installation, typing at the bash command line prompt:

```bash
$ bash --version
```

should result in something like:

```
GNU bash, version 4.4.12(1)-release (x86_64-pc-linux-gnu)  
Copyright (C) 2016 Free Software Foundation, Inc.
```

Your version string will likely vary.

Python

Python is an interpreted programming language. It is used to generate some source files, documentation, testing and other tasks. Python 3.6 and later is required. Python 2 is no longer supported.

Python is either included or available as a package on most UNIX-like platforms. Windows packages and source are available at https://python.org/download/.

You can also use Chocolatey to install Python:

```
PS:\> choco install python3
```

Chocolatey installs Python into C:\Python37 by default. You can verify your Python version by running

```bash
$ python3 --version
```

on UNIX-like platforms and
rem Official package
C:> cd python35
C:Python35> python --version

rem Chocolatey
C:> cd \tools\python3
C:\tools\python3> python --version

on Windows. You should see something like

Python 3.5.1

Your version string may vary of course.

**Flex**

Flex is a lexical analyzer generator used for Wireshark’s display filters, some file formats, and other features.

**Unix**

Flex is available for most UNIX and UNIX-like platforms. See the next section for native Windows options.

If GNU flex isn't already installed or available as a package for your platform you can get it at https://www.gnu.org/software/flex/.

After correct installation running the following

```bash
$ flex --version
```

should result in something like:

```
flex version 2.5.4
```

Your version string may vary.

**Windows**

A native Windows version of flex is available in the winflexbison3 Chocolatey package. Note that the executable is named *win flex*.
Native packages are available from other sources such as GnuWin. They aren’t officially supported but should work.

**Git client**

The Wireshark project uses its own Git repository to keep track of all the changes done to the source code. Details about the usage of Git in the Wireshark project can be found in The Wireshark Git repository.

If you want to work with the source code and are planning to commit your changes back to the Wireshark community, it is recommended to use a Git client to get the latest source files. For detailed information about the different ways to obtain the Wireshark sources, see Obtaining The Wireshark Sources.

You will find more instructions in Git Over SSH Or HTTPS on how to use the Git client.

**Unix**

Git is available for most UNIX and UNIX-like platforms. If Git isn’t already installed or available as a package for your platform, you can get it at: https://git-scm.com/.

After correct installation, typing at the bash command line prompt:

```
$ git --version
```

should result in something like:

```
git version 2.14.1
```

Your version will likely be different.

**Windows**

The Git command line tools for Windows can be found at https://git-scm.com/download/win and can also be installed using Chocolatey:

```
PS:\> choco install git
```

After correct installation, typing at the command line prompt (cmd.exe):
> git --version

should result in something like:

```
$ git --version
git version 2.16.1.windows.1
```

However, the version string may vary.

**Git Powershell Extensions (Optional)**

A useful tool for command line git on Windows is PoshGit. Posghit provides git command completion and alters the prompt to indicate the local working copy status. You can install it using Chocolatey:

```
PS:\> choco install poshgit
```

**Git GUI Client (Optional)**

Along with the traditional command-line client, several GUI clients are available for a number of platforms. See [https://git-scm.com/downloads/guis](https://git-scm.com/downloads/guis) for details.

**Perl (Optional)**

Perl is an interpreted programming language. Perl is used to convert various text files into usable source code and for various source code checks. Perl version 5.6 and above should work fine.

**Unix**

Perl is available for most UNIX and UNIX-like platforms. If it isn't already installed or available as a package for your platform, you can get it at [https://www.perl.org/](https://www.perl.org/).

After correct installation, typing at the bash command line prompt:

```
$ perl --version
```

should result in something like:
This is perl 5, version 26, subversion 0 (v5.26.0) built for x86_64-linux-gnu-thread-multi
(with 62 registered patches, see perl -V for more detail)

Copyright 1987-2017, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on this system using "man perl" or "perldoc perl". If you have access to the Internet, point your browser at http://www.perl.org/, the Perl Home Page.

However, the version string may vary.

**Windows**

A native Windows Perl package can be obtained from Strawberry Perl or Active State. The installation should be straightforward.

You may also use Chocolatey to install either package:

```
PS:\> choco install strawberryperl
```

or

```
PS:\> choco install activeperl
```

After correct installation, typing at the command line prompt (cmd.exe):

```
> perl -v
```

should result in something like:
This is perl, v5.8.0 built for MSWin32-x86-multi-thread
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2002, Larry Wall

Binary build 805 provided by ActiveState Corp. http://www.ActiveState.com
Built 18:08:02 Feb  4 2003

However, the version string may vary.

**patch (Optional)**

The patch utility is used to merge a diff file into your own source tree. This tool is only needed, if you want to apply a patch (diff file) from someone else (probably from the developer mailing list) to try out in your own private source tree.

It most cases you may not need the patch tool installed. Git should handle patches for you.

**Unix**

Patch is available for most UNIX and UNIX-like platforms. If GNU patch isn't already installed or available as a package for your platform, you can get it at https://www.gnu.org/software/patch/
patch.html.

After correct installation, typing at the bash command line prompt:

```
$ patch --version
```

should result in something like:

```
patch 2.5.8
Copyright (C) 1988 Larry Wall
Copyright (C) 2002 Free Software Foundation, Inc.

This program comes with NO WARRANTY, to the extent permitted by law.
You may redistribute copies of this program
under the terms of the GNU General Public License.
For more information about these matters, see the file named COPYING.

written by Larry Wall and Paul Eggert
```

However, the version string may vary.
**Windows**

The Windows native Git tools provide patch. A native Windows patch package can be obtained from [http://gnuwin32.sourceforge.net/](http://gnuwin32.sourceforge.net/). The installation should be straightforward.

**Windows: NSIS (Optional)**

The NSIS (Nullsoft Scriptable Install System) is used to generate `Wireshark-4.3.0-x64.exe` from all the files needed to be installed, including all required DLLs, plugins, and supporting files.

To install it, download the latest released version from [https://nsis.sourceforge.net](https://nsis.sourceforge.net). NSIS v3 is required. You can also install it using Chocolatey:

```
PS$> choco install nsis
```

You can find more instructions on using NSIS in [Windows: NSIS .exe Installer](#).

**Windows: WiX Toolset (Optional)**

The WiX Toolset can be used to generate Windows Installer (.msi) packages. You can download it from the [WiX web site](https://wixtoolset.org) or install it using Chocolatey:

```
PS$> choco install wixtoolset
```

This also requires the Visual C++ redistributable merge modules, which can be installed by selecting “Individual Components → C++ 2022 Redistributable MSMs” or “…2019 Redistributable MSMs” as appropriate for your compiler in the Visual Studio installer.

Wireshark’s .msi packaging is currently experimental and the generated packages may be incomplete.

**Windows: PortableApps (Optional)**

The PortableApps.com Installer is used to generate `WiresharkPortable64_4.3.0.paf.exe` from all the files needed to be installed, including all required DLLs, plugins, and supporting files.

To install it, do the following:

- Download the latest PortableApps.com Platform release from [https://portableapps.com/](https://portableapps.com/).
- Install the following applications in the PortableApps.com environment:
  - PortableApps.com Installer
  - PortableApps.com Launcher
You can find more instructions on using the PortableApps.com Installer in Windows: PortableApps .paf.exe Package.
Introduction

Like most applications, Wireshark depends on libraries provided by your operating system and by third parties, including the C runtime library, GLib, libpcap, and Qt. While running Wireshark only requires the libraries themselves, building it requires header files, import libraries, and related resources.

Binary libraries are available in different formats and are specific to the target operating system, platform, and compiler. They can be compiled by hand, but are most often installed as pre-built packages.

On most Linux systems, the required binary and development libraries can be installed using your package manager. We provide setup scripts that will install the required packages for most distributions. See Build environment setup for details.

On macOS, you can install pre-built packages using a third party package manager such as Homebrew or MacPorts. As with Linux, we provide tools/macos-setup-brew.sh, which will install the required Homebrew packages. We also provide tools/macos-setup.sh, which will download, build, and install required packages. Note that tools/macos-setup.sh installs into /usr/local by default; you can change this with the -p flag.

Windows doesn’t have a good library package manager at the present time, so we provide our own pre-built libraries. They can be installed using tools/win-setup.ps1 and are automatically installed when you run CMake. With the exception of Qt, all libraries required to build Wireshark on Windows are available for download at https://dev-libs.wireshark.org/windows/. See Windows Automated Library Download for details.

Windows Automated Library Download

The required libraries (apart from Qt) are automatically downloaded as part of the CMake generation step, and subsequently as required when libraries are updated.

The libraries are downloaded into the directory indicated by the environment variable WIRESHARK_BASE_DIR, this must be set appropriately for your environment. The libraries are downloaded and extracted into WIRESHARK_BASE_DIR\wireshark-x64-libs or WIRESHARK_BASE_DIR\wireshark-arm64-libs depending on your target platform.

You may also set the library directory to a custom value with the environment variable WIRESHARK_LIB_DIR, but you may run into problems if you switch between major versions or target platforms.
Qt

The Qt library is used to build the UI for Wireshark and is used to provide a platform independent UI. Wireshark can be built with Qt 5.12 or later, but looks for Qt 6 as the default version.

To enable builds with Qt 5.x, the command-line option `-DUSE_qt6=OFF` has to be set for cmake.

For more information on the Qt libraries, see The Qt Application Framework.

Unix

Most Linux distributions provide Qt and its development libraries as standard packages. The required libraries and tools will likely be split across several packages. For example, building on Ubuntu requires `qt6-tools-dev`, `qt6-tools-dev-tools`, `libqt6svg6-dev`, `qt6-multimedia-dev`, and possibly others.

The Qt Project provides an installation tool for macOS, similar to Windows. It is available at https://www.qt.io/download-open-source/#section-2.

Windows

Qt 6 must be installed manually from the Qt installers page https://www.qt.io/download-open-source/#section-2 using the version of Qt appropriate for your compiler.

The CMake variable CMAKE_PREFIX_PATH (see https://doc.qt.io/qt-6/cmake-get-started.html) should be set to your Qt installation directory, e.g. `C:\Qt\6.5.3\msvc2019_64`. Alternatively you can also use the environment variable `WIRESHARK_QT6_PREFIX_PATH`.

GLib And Supporting Libraries

The GLib library is used as a basic platform abstraction library and can be used in both CLI and GUI applications. For a detailed description about GLib see The GLib library.

GLib depends on GNU libiconv, GNU gettext, and other libraries. You will typically not come into contact with these while doing Wireshark development. Wireshark’s build system check for and require both GLib and its dependencies.

Unix

The GLib library is available for most Linux distributions and UNIX flavors. If it isn't already installed and isn't available as a package for your platform, you can get it at https://wiki.gnome.org/Projects/GLib.

Windows

GLib is part of our vcpkg-export bundles and is available at https://dev-libs.wireshark.org/windows/
c-ares

C-Ares is used for asynchronous DNS resolution and lets us resolve names with a minimal performance impact.

Unix

If this library isn't already installed or available as a package for your platform, you can get it at https://c-ares.org/.

Windows

C-Ares is built using vcpkg and is available at https://dev-libs.wireshark.org/windows/packages/.

SMI (Optional)

LibSMI is used for MIB and PIB parsing and for OID resolution.

Unix

If this library isn't already installed or available as a package for your platform, you can get it at https://www.ibr.cs.tu-bs.de/projects/libsmi/.

Windows


zlib (Optional)

zlib is designed to be a free, general-purpose, legally unencumbered — that is, not covered by any patents — lossless data-compression library for use on virtually any computer hardware and operating system.

— The zlib web site, https://www.zlib.net/

Unix

This library is almost certain to be installed on your system. If it isn't or you don't want to use the default library you can download it from https://www.zlib.net/.
**Windows**

zlib is part of our vcpkg-export bundles and is available at [https://dev-libs.wireshark.org/windows/packages/](https://dev-libs.wireshark.org/windows/packages/).

### libpcap or Npcap (Optional, But Strongly Recommended)

Libpcap and Npcap provide the packet capture capabilities that are central to Wireshark's core functionality.

**Unix: libpcap**

If this library isn't already installed or available as a package for your platform, you can get it at [https://www.tcpdump.org/](https://www.tcpdump.org/).

**Windows: Npcap**

The Windows build environment compiles and links against a libpcap SDK built using vcpkg and includes the Npcap packet capture driver with the .exe installer. Both are automatically downloaded by CMake.

You can download the Npcap Windows packet capture library manually from [https://npcap.com/](https://npcap.com/).

---

**WARNING**

_Npcap has its own license with its own restrictions_

Insecure.Com LLC, aka “The Nmap Project” has granted the Wireshark Foundation the right to include Npcap with the installers that we distribute from wireshark.org. If you wish to distribute your own Wireshark installer or any other package that includes Npcap you must comply with the Npcap license and may be required to purchase a redistribution license. Please see [https://npcap.com/](https://npcap.com/) for more details.

### GnuTLS (Optional)

The GNU Transport Layer Security Library is used to enable TLS decryption using an RSA private key.

**Unix**

If this library isn't already installed or available as a package for your platform, you can get it at [https://gnutls.org/](https://gnutls.org/).

**Windows**
We provide packages cross-compiled using MinGW32 at https://dev-libs.wireshark.org/windows/packages/.

**Gcrypt**

The Gcrypt Library is a low-level cryptographic library that provides support for many ciphers and message authentication codes, such as DES, 3DES, AES, Blowfish, SHA-1, SHA-256, and others.

**Unix**

If this library isn’t already installed or available as a package for your platform, you can get it at https://directory.fsf.org/wiki/Libgcrypt.

**Windows**

We provide packages for Windows at https://dev-libs.wireshark.org/windows/packages/.

**Kerberos (Optional)**

The Kerberos library is used to dissect Kerberos, sealed DCERPC and secure LDAP protocols.

**Unix**

If this library isn’t already installed or available as a package for your platform, you can get it at https://web.mit.edu/Kerberos/dist/.

**Windows**

We provide packages for Windows at https://dev-libs.wireshark.org/windows/packages/.

**Lua (Optional)**

The Lua library is used to add scripting support to Wireshark. Wireshark 4.2.x and earlier support Lua versions 5.1 and 5.2. Recent versions of Wireshark have added support for Lua 5.3 and 5.4 as well.

**Unix**

If this library isn’t already installed or available as a package for your platform, you can get it at https://www.lua.org/download.html.

**Windows**

We provide packages for Windows, patched for UTF-8 support, at https://dev-libs.wireshark.org/
MaxMindDB (Optional)

MaxMind Inc. publishes a set of IP geolocation databases and related open source libraries. They can be used to map IP addresses to geographical locations and other information.

If libmaxminddb library isn’t already installed or available as a package for your platform, you can get it at https://github.com/maxmind/libmaxminddb.

We provide packages for Windows at https://dev-libs.wireshark.org/windows/packages/.

WinSparkle (Optional)

WinSparkle is an easy-to-use software update library for Windows developers.

Windows

We provide copies of the WinSparkle package at https://dev-libs.wireshark.org/windows/packages/.
Wireshark Development

Wireshark Development
The second part describes how the Wireshark sources are structured and how to change the sources such as adding a new dissector.
Introduction

Source overview

Wireshark consists of the following major parts:

- Packet dissection - in the `/epan/dissectors` and `/plugins/epan/*` directories
- Capture file I/O - using Wireshark's own wiretap library
- Capture - using the libpcap and Npcap libraries, in `dumpcap.c` and the `/capture` directory
- User interface - using Qt and associated libraries
- Utilities - miscellaneous helper code
- Help - using an external web browser and text output

Coding Style

The coding style guides for Wireshark can be found in the “Portability” section of the file `doc/README.developer`.

The GLib library

GLib is used as a basic platform abstraction library. It doesn’t provide any direct GUI functionality.

To quote the GLib Reference Manual:

GLib provides the core application building blocks for libraries and applications written in C. It provides the core object system used in GNOME, the main loop implementation, and a large set of utility functions for strings and common data structures.

How Wireshark Works

Introduction

This chapter will give you a short overview of how Wireshark works.

Overview

The following will give you a simplified overview of Wireshark's function blocks:
The function blocks in more detail:

**GUI**

Handling of all user input/output (all windows, dialogs and such). Source code can be found in
the ui/qt directory.

Core

Main "glue code" that holds the other blocks together. Source code can be found in the root directory.

Epan

Enhanced Packet ANalyzer — the packet analyzing engine. Source code can be found in the epan directory. Epan provides the following APIs:

- Protocol Tree. Dissection information for an individual packet.
- Dissectors. The various protocol dissectors in epan/dissectors.
- Dissector Plugins - Support for implementing dissectors as separate modules. Source code can be found in plugins.
- Display Filters - The display filter engine at epan/dfilter.

Wiretap

The wiretap library is used to read and write capture files in libpcap, pcapng, and many other file formats. Source code is in the wiretap directory.

Capture

The interface to the capture engine. Source code is in the root directory.

Dumpcap

The capture engine itself. This is the only part that executes with elevated privileges. Source code is in the root directory.

Npcap and libpcap

These are external libraries that provide packet capture and filtering support on different platforms. The filtering in Npcap and libpcap works at a much lower level than Wireshark's display filters and uses a significantly different mechanism. That's why there are different display and capture filter syntaxes.

Capturing packets

Capturing takes packets from a network adapter and saves them to a file on your hard disk.

Since raw network adapter access requires elevated privileges, these functions are isolated to the dumpcap program. Placing the capture functionality into dumpcap allows the rest of the code (dissectors, user interface, etc.) to run with normal user privileges.

To hide all the low-level machine dependent details from Wireshark, the libpcap and Npcap (see libpcap or Npcap (Optional, But Strongly Recommended)) libraries are used. These libraries provide a general purpose interface to capture packets and are used by a wide variety of applications.
Capture Files

Wireshark can read and write capture files in its natural file formats, pcapng and pcap, which are used by many other network capturing tools, such as tcpdump. Additionally, Wireshark supports reading and writing packet capture files in formats used by other network capture tools. This support is implemented in Wireshark's wiretap library, which provides a general purpose interface for reading and writing packet capture formats and supports more than twenty packet capture formats.

Dissect packets

Wireshark dissects packets in what it calls 'two-pass' dissection.

Wireshark performs a first pass of dissecting all packets as they are loaded from the file. All packets are dissected sequentially and this information is used to populate Wireshark's packet list pane and to build state and other information needed when displaying the packet.

Wireshark later performs 'second pass' ad-hoc dissections on the packets that it needs data from. This enables Wireshark to fill in fields that require future knowledge, like the 'response in frame #' fields, and correctly calculate reassembly frame dependencies.

For example, Wireshark will perform an ad-hoc dissection when a user selects a packet (to display the packet details), calculates a statistic (so all values are computed), or performs another action that requires packet data. However, because Wireshark may only dissect the packets that are needed, there is no guarantee that Wireshark will dissect all packets again, nor is there any guarantee as to the order that the packets will be dissected after the first pass.
Packet Capture

This chapter needs to be reviewed and extended.

Adding A New Capture Type To Libpcap

For this discussion, let’s assume you’re working with libpcap 1.0 or later. You probably don’t want to work with a version older than 1.0, even if whatever OS you’re using happens to include libpcap - older versions are not as friendly towards adding support for devices other than standard network interfaces.

First, read the libpcap documentation on writing a new libpcap module

(It’s currently incomplete, but I’ll be finishing it up over time. If you have contributions, feel free to submit pull requests for it.)

If you had to introduce one or more new DLT_* values, you will also have to add support in Wireshark for those DLT_* values to wiretap/pcap-common.c, which might mean adding one or more WTAP_ENCAP types to wtap.h and to the encap_table[] table in wiretap/wtap.c. You’d then have to write a dissector or dissectors for the link-layer protocols or protocols and have them register themselves with the wtap_encap dissector table, with the appropriate WTAP_ENCAP values by calling dissector_add_uint().

Adding Capture Interfaces And Log Sources Using Extcap

The extcap interface is a versatile plugin interface that allows external binaries to act as capture interfaces directly in Wireshark. It is used in scenarios, where the source of the capture is not a traditional capture model (live capture from an interface, from a pipe, from a file, etc). The typical example is connecting esoteric hardware of some kind to the main Wireshark app.

Without extcap, a capture can always be achieved by directly writing to a capture file:

Bash example for traditional capture with a capture file.

```
$ the-esoteric-binary --the-strange-flag --interface=stream1 --file dumpfile.pcap &
$ wireshark dumpfile.pcap
```

but the extcap interface allows for such a connection to be easily established and configured using the Wireshark GUI.

The extcap subsystem is made of multiple extcap binaries that are automatically called by the GUI
in a row. In the following chapters we will refer to them as “the extcaps”.

Extcaps may be any binary or script within the extcap directory. Please note, that scripts need to be executable without prefacing a script interpreter before the call.

**IMPORTANT**

**Windows Users** Because of restrictions directly calling the script may not always work. In such a case, a batch file may be provided, which then in turn executes the script. Please refer to [Execute A Script-based Extcap On Windows](#) for more information.

When Wireshark launches an extcap, it automatically adds its installation path (normally `C:\Program Files\Wireshark\`) to the DLL search path so that the extcap library dependencies can be found (it is not designed to be launched by hand). This is done on purpose. There should only be extcap programs (executables, Python scripts, etc.) in the extcap folder to reduce the startup time and not have Wireshark trying to execute other file types.

**Extcap Command Line Interface**

The actual capture is run after a setup process that can be done manually by the user or automatically by the GUI. All the steps performed are done for every extcap.

Let’s go through those steps.

**Query For Available Interfaces**

In the first step the extcap is queried for its interfaces.

```
$ extcapbin --extcap-interfaces
```

This call must print the existing interfaces for this extcap and must return 0. The output must conform to the grammar specified for extcap, and it is specified in the doc/extcap.4 generated man page (in the build dir).

Wireshark 2.9 and later also pass `--extcap-version=x.x`, which provides the calling Wireshark's major and minor version. This can be used to change behavior depending on the Wireshark version in question.

**Example call for interface query**

```
$ extcap_example.py --extcap-interfaces --extcap-version=4.0
extcap {version=1.0}{help=Some help url}
interface {value=example1}{display=Example interface 1 for extcap}
interface {value=example2}{display=Example interface 2 for extcap}
```

The version for the extcap sentence (which may exist as many times as is needed, but only the last
one will be used) will be used for displaying the version information of the extcap interface in the
about dialog of Wireshark.

The value for each interface will be used in subsequent calls as the interface name <iface>.

Using the help argument, an interface may provide a generic help URL for the extcap utility.

**Ask For DLTs For Each Interface**

Next, the extcap binary is queried for all valid DLTs for all the interfaces returned by step 1.

```
$ extcap_example.py --extcap-dlts --extcap-interface <iface>
```

This call must print the valid DLTs for the interface specified. This call is made for all the interfaces
and must return 0.

*Example for the DLT query*

```
$ extcap_example.py --extcap-interface IFACE --extcap-dlts
dlt {number=147}{name=USER1}{display=Demo Implementation for Extcap}
```

A binary or script which neither provides an interface list or a DLT list will not show up in the
extcap interfaces list.

**The Extcap Configuration Interface**

The extcap binary is next asked for the configuration of each specific interface

```
$ extcap_example.py --extcap-interface <iface> --extcap-config
```

Each interface can have custom options that are valid for this interface only. Those config options
are specified on the command line when running the actual capture. To allow an end-user to
specify certain options, such options may be provided using the extcap config argument.

To share which options are available for an interface, the extcap responds to the command `--extcap-config`, which shows all the available options (aka additional command line options).

Those options are used to build a configuration dialog for the interface.
Example for interface options

```
$ extcap_example.py --extcap-interface <iface> --extcap-config
arg {number=0}{call=--delay}{display=Time delay}{tooltip=Time delay between packages}{type=integer}{range=1,15}{required=true}
arg {number=1}{call=--message}{display=Message}{tooltip=Package message content}{placeholder=Please enter a message here ...}{type=string}
arg {number=2}{call=--verify}{display=Verify}{tooltip=Verify package content}{type=boolflag}
arg {number=3}{call=--remote}{display=Remote Channel}{tooltip=Remote Channel Selector}{type=selector}
arg {number=4}{call=--server}{display=IP address for log server}{type=string}{validation=\b(?:(?:25\[0-5]|2\[0-4]\[0-9]|[01]?\[0-9]\[0-9\])\.){3}(?:25\[0-5]|2\[0-4]\[0-9]|[01]?\[0-9]\[0-9\])\b}
value {arg=3}{value=if1}{display=Remote1}{default=true}
value {arg=3}{value=if2}{display=Remote2}{default=false}
```

Now the user can click on the options and change them. They are sent to the extcap when the capture is launched.

There are several kind of options available:

**File**
A path to a file displayed as a text entry and file selector.

**Flag**
A boolean value displayed as a checkbox. `boolflag` for instance expects the option to be present resulting in the corresponding entry set to true or false.

**Selection**
A set of fixed values displayed as a combobox, radio group, or selection list. Selections can be presented multiple times in the command line. Subsequent value items must be provided in the config list.

**Timestamp**
A time value displayed as a date/time editor.

**Value**
A text or numeric value displayed as an entry box. Values are passed as a single value via the command-line call.

The Extcap Capture Process

Once the interfaces are listed and configuration is customized by the user the capture can be started.

```
$ extcap_example.py --extcap-interface <iface> [params] --capture [--extcap-capture-filter <cfilter>]
    --fifo FIFO
```
To run the capture, the extcap must implement the `--capture`, `--extcap-capture-filter` and `--fifo` options.

They are automatically added by Wireshark, which opens the fifo for reading. All the other options are automatically added to run the capture. The extcap interface is used like all other interfaces (meaning that capture on multiple interfaces, as well as stopping and restarting the capture is supported).

**Execute A Script-based Extcap On Windows**

Although Windows will run batch and PowerShell scripts directly, other scripting languages require extra effort. In most cases this involves creating a wrapper script which runs the appropriate interpreter. For example, in order to run a Python-based extcap, you can create `scriptname.bat` inside your extcap folder with the following content:

```
@echo off
C:\Windows\py.exe C:\Path\to\my\extcap.py %*
```

**Extcap Arguments**

The extcap interface provides the possibility for generating a GUI dialog to set and adapt settings for the extcap binary.

All options must provide a number, by which they are identified. No number may be provided twice. All options must present the elements `call` and `display`, with `call` specifying the argument’s name on the command line and `display` specifying the name in the GUI.

Additionally `tooltip` and `placeholder` may be provided, which will give the user information about what to enter into this field.

These options do have types, for which the following types are being supported:

- `integer, unsigned, long, double`  
  This provides a field for entering a numeric value of the given data type. A `default` value may be provided, as well as a `range`.

```
arg {number=0}{call=--delay}{display=Time delay}{tooltip=Time delay between packages}{type=integer}{range=1,15}{default=0}
```
**string**

This provides a field for entering a text value.

```
arg {number=1}{call=--server}{display=IP Address}{tooltip=IP Address for log server}{type=string}{validation=\b(?:\:25\[0-5]\|2\[0-4]\[0-9]\|[01]\[0-9]\[0-9]\|[0-9]\|0-9])\b{3}(?:25\[0-5]\|2\[0-4]\[0-9]\|0-9][0-9]\|0-9))\b}
```

**validation** allows to provide a regular expression string, which is used to check the user input for validity beyond normal data type or range checks. Back-slashes must be escaped (as in \b for \b)

**password**

Lets the user provide a masked string to the capture. Password strings are not saved with other capture settings.

```
arg {number=0}{call=--password}{display=The user password}{tooltip=The password for the connection}{type=password}
```

**boolean, boolflag**

This provides the possibility to set a true/false value. **boolflag** values will only appear in the command line if set to true, otherwise they will not be added to the command-line call for the extcap interface.

```
arg {number=2}{call=--verify}{display=Verify}{tooltip=Verify package content}{type=boolflag}
```
**fileselect**

Lets the user provide a file path. If `mustexist=true` is provided, the GUI shows the user a dialog for selecting a file. When `mustexist=false` is used, the GUI shows the user a file dialog for saving a file.

```
arg {number=3}{call=--logfile}{display=Logfile}{tooltip=A file for log messages}{type=fileselect}{mustexist=false}
```

**selector, editselector, radio, multicheck**

Option fields where the user may choose from one or more options. If `parent` is provided for the value items, the option fields for `multicheck` and `selector` are presented in a tree-like structure. `selector` and `radio` values must present a default value, which will be the value provided to the extcap binary for this argument. `editselector` option fields let the user select from a list of items or enter a custom value.

```
arg {number=3}{call=--remote}{display=Remote Channel}{tooltip=Remote Channel Selector}{type=selector}
value {arg=3}{value=if1}{display=Remote1}{default=true}
value {arg=3}{value=if2}{display=Remote2}{default=false}
```

**Reload A Selector**

A selector may be reloaded from the configuration dialog of the extcap application within Wireshark. With the reload argument (defaults to false), the entry can be marked as reloadable.

```
arg {number=3}{call=--remote}{display=Remote Channel}{tooltip=Remote Channel Selector}{type=selector}{reload=true}{placeholder=Load interfaces...}
```

After this has been defined, the user will get a button displayed in the configuration dialog for this
extcap application, with the text "Load interfaces..." in this case, and a generic "Reload" text if no text has been provided.

The extcap utility is then called again with all filled out arguments and the additional parameter **--extcap-reload-option <option_name>**. It is expected to return a value section for this option, as it would during normal configuration. The provided option list is then presented as the selection, a previous selected option will be reselected if applicable.

**Validation Of Arguments**

Arguments may be set with `{required=true}` which enforces a value being provided, before a capture can be started using the extcap options dialog. This is not being checked, if the extcap is started via a simple double-click. The necessary fields are marked for the customer, to ensure a visibility for the end customer of the required argument.

Additionally text and number arguments may also be checked using a regular expression, which is provided using the validation attribute (see example above). The syntax for such a check is the same as for Qt RegExp classes. This feature is only active in the Qt version of Wireshark.

**Toolbar Controls**

An extcap utility can provide configuration for controls to use in an interface toolbar. These controls are bidirectional and can be used to control the extcap utility while capturing.

This is useful in scenarios where configuration can be done based on findings in the capture process, setting temporary values or give other inputs without restarting the current capture.

*Example of interface definition with toolbar controls*

```bash
$ extcap_example.py --extcap-interfaces
extcap {version=1.0}{display=Example extcap interface}
interface {value=example1}{display=Example interface 1 for extcap}
interface {value=example2}{display=Example interface 2 for extcap}
control {number=0}{type=string}{display=Message}{tooltip=Package message content. Must start with a capital letter.}{validation=[A-Z]+}{required=true}
control {number=1}{type=selector}{display=Time delay}{tooltip=Time delay between packages}
control {number=2}{type=boolean}{display=Verify}{default=true}{tooltip=Verify package content}
control {number=3}{type=button}{display=Turn on}{tooltip=Turn on or off}
control {number=4}{type=button}{role=logger}{display=Log}{tooltip=Show capture log}
value {control=1}{value=1}{display=1 sec}{default=true}
value {control=1}{value=2}{display=2 sec}{default=true}
```

All controls will be presented as GUI elements in a toolbar specific to the extcap utility. The extcap must not rely on using those controls (they are optional) because of other capturing tools not using GUI (e.g. tshark, tfshark).
Controls

The controls are similar to the *arguments*, but without the *call* element. All controls may be given a default value at startup and most can be changed during capture, both by the extcap and the user (depending on the type of control).

All controls must provide a *number*, by which they are identified. No *number* may be provided twice. All options must present the elements *type* and *display*, where *type* provides the type of control to add to the toolbar and *display* providing the name in the GUI.

Additionally *tooltip* and *placeholder* may be provided, which will give the user information about what to enter into this field.

All controls, except from the logger, help and restore buttons, may be disabled (and enabled) in GUI by the extcap during capture. This can be because of set-once operations, or operations which takes some time to complete.

All control values which are changed by the user (not equal to the default value) will be sent to the extcap utility when starting a capture. The extcap utility may choose to discard initial values and set new values, depending on implementation.

These *types* are defined as controls:

- **boolean**  
  This provides a checkbox which lets the user set a true/false value.

  The extcap utility can set a default value at startup, and can change (set) and receive value changes while capturing. When starting a capture the GUI will send the value if different from the default value.

  The payload is one byte with binary value 0 or 1.

  Valid Commands: Set value, Enable, Disable.
button

This provides a button with different roles:

control

This button will send a signal when pressed. This is the default if no role is configured. The button is only enabled when capturing.

The extcap utility can set the button text at startup, and can change (set) the button text and receive button press signals while capturing. The button is disabled and the button text is restored to the default text when not capturing.

The payload is either the button text or empty (signal).

Valid Commands: Set value, Enable, Disable.

logger

This provides a logger mechanism where the extcap utility can send log entries to be presented in a log window. This communication is unidirectional.

The payload is the log entry, and should be ended with a newline. Maximum length is 65535 bytes.

Valid Commands: Set log entry, Add log entry.

The Set command will clear the log before adding the entry.

help

This button opens the help page, if configured. This role has no controls and will not be used in communication.

Valid Commands: None.

restore

This button will restore all control values to default. This role has no controls and will not be used in communication. The button is only enabled when not capturing.

Valid Commands: None.
**selector**  This provides a combo box with fixed values which can be selected.

The extcap utility can set default values at startup, and add and remove values and receive change in value selection while capturing. When starting a capture the GUI will send the value if different from the default value.

The payload is a string with the value, and optionally a string with a display value if this is different from the value. This two string values are separated by a null character.

Valid Commands: Set selected value, Add value, Remove value, Enable, Disable.

If value is empty the Remove command will remove all entries.

**string**  This provides a text edit line with the possibility to set a string or any value which can be represented in a string (integer, float, date, etc.).

The extcap utility can set a default string value at startup, and can change (set) and receive value changes while capturing. When starting a capture the GUI will send the value if different from the default value.

The payload is a string with the value. Maximum length is 32767 bytes.

Valid Commands for control: Set value, Enable, Disable.

The element VALIDATION allows to provide a regular expression string, which is used to check the user input for validity beyond normal data type or range checks. Back-slashes must be escaped (as in \b for \b).

**Messages**

In addition to the controls it’s possible to send a single message from the extcap utility to the user. This message can be put in the status bar or displayed in a information, warning or error dialog which must be accepted by the user. This message does not use the NUMBER argument so this can have any value.

**Control Protocol**

The protocol used to communicate over the control pipes has a fixed size header of 6 bytes and a payload with 0 - 65535 bytes.

*Table 3. Control packet:*

<table>
<thead>
<tr>
<th>Sync Pipe Indication (1 byte)</th>
<th>Message Length (3 bytes network order)</th>
<th>Control Number (1 byte)</th>
</tr>
</thead>
</table>

101
Command (1 byte)

Payload
(0 - 65535 bytes)

**Sync Pipe Indication**

The common sync pipe indication. This protocol uses the value “T”.

**Message Length**

Payload length + 2 bytes for control number and command.

**Control Number**

Unique number to identify the control. This number also gives the order of the controls in the interface toolbar.

**Table 4. Commands and application for controls**

<table>
<thead>
<tr>
<th>Command Byte</th>
<th>Command Name</th>
<th>Control type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initialized</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>Set</td>
<td>boolean / button / logger / selector / string</td>
</tr>
<tr>
<td>2</td>
<td>Add</td>
<td>logger / selector</td>
</tr>
<tr>
<td>3</td>
<td>Remove</td>
<td>selector</td>
</tr>
<tr>
<td>4</td>
<td>Enable</td>
<td>boolean / button / selector / string</td>
</tr>
<tr>
<td>5</td>
<td>Disable</td>
<td>boolean / button / selector / string</td>
</tr>
<tr>
<td>6</td>
<td>Statusbar message</td>
<td>none</td>
</tr>
<tr>
<td>7</td>
<td>Information message</td>
<td>none</td>
</tr>
<tr>
<td>8</td>
<td>Warning message</td>
<td>none</td>
</tr>
<tr>
<td>9</td>
<td>Error message</td>
<td>none</td>
</tr>
</tbody>
</table>

The **Initialized** command will be sent from the GUI to the extcap utility when all user changed control values are sent after starting a capture. This is an indication that the GUI is ready to receive control values.

The GUI will only send **Initialized** and **Set** commands. The extcap utility shall not send the **Initialized** command.

Messages with unknown control number or command will be silently ignored.
Packet Dissection

How packet dissection works

Each dissector decodes its part of the protocol and then hands off decoding to subsequent dissectors for an encapsulated protocol.

Every dissection starts with the Frame dissector which dissects the details of the capture file itself (e.g. timestamps). From there it passes the data on to the lowest-level data dissector, e.g. the Ethernet dissector for the Ethernet header. The payload is then passed on to the next dissector (e.g. IP) and so on. At each stage, details of the packet are decoded and displayed.

Dissectors can either be built-in to Wireshark or written as a self-registering plugin (a shared library or DLL). There is little difference in having your dissector as either a plugin or built-in. You have limited function access through the ABI exposed by functions declared as WS_DLL_PUBLIC.

The big benefit of writing a dissector as a plugin is that rebuilding a plugin is much faster than rebuilding wireshark after editing a built-in dissector. As such, starting with a plugin often makes initial development quicker, while the finished code may make more sense as a built-in dissector.

NOTE

Read README.dissector

The file doc/README.dissector contains detailed information about writing a dissector. In many cases it is more up to date than this document.

Adding a basic dissector

Let’s step through adding a basic dissector. We’ll start with the made up “foo” protocol. It consists of the following basic items.

- A packet type - 8 bits. Possible values: 1 - initialisation, 2 - terminate, 3 - data.
- A set of flags stored in 8 bits. 0x01 - start packet, 0x02 - end packet, 0x04 - priority packet.
- A sequence number - 16 bits.
- An IPv4 address.

Setting up the dissector

The first decision you need to make is if this dissector will be a built-in dissector and included in the main program, or a plugin.

Plugins are easier to write initially, so let’s start with that. With a little care, the plugin can be converted into a built-in dissector.
Let's go through this a bit at a time. First we have some boilerplate include files. These will be pretty constant to start with.

Then a `#define` for the UDP port that carries `foo` traffic.

Next we have `proto_foo`, an int that stores our protocol handle and is initialised to `-1`. This handle will be set when the dissector is registered within the main program. It's good practice to make all variables and functions that aren't exported static to minimize name space pollution. This normally isn't a problem unless your dissector gets so big that it spans multiple files.

Now that we have the basics in place to interact with the main program, we'll start with two protocol dissector setup functions: `proto_register_XXX` and `proto_reg_handoff_XXX`.

Each protocol must have a register function with the form "proto_register_XXX". This function is used to register the protocol in Wireshark. The code to call the register routines is generated automatically and is called when Wireshark starts. In this example, the function is named `proto_register_foo`.

`proto_register_foo` calls `proto_register_protocol()`, which takes a `name`, `short name`, and `filter_name`. The name and short name are used in the "Preferences" and "Enabled protocols" dialogs and the documentation's generated field name list. The `filter_name` is used as the display filter name. `proto_register_protocol()` returns a protocol handle, which can be used to refer to the protocol and obtain a handle to the protocol's dissector.

Next we need a handoff routine.
Dissector Handoff.

```c
void
proto_reg_handoff_foo(void)
{
    static dissector_handle_t foo_handle;

    foo_handle = create_dissector_handle(dissect_foo, proto_foo);
    dissector_add_uint("udp.port", FOO_PORT, foo_handle);
}
```

A handoff routine associates a protocol handler with the protocol's traffic. It consists of two major steps: The first step is to create a dissector handle, which is a handle associated with the protocol and the function called to do the actual dissecting. The second step is to register the dissector handle so that traffic associated with the protocol calls the dissector.

In this example, `proto_reg_handoff_foo()` calls `create_dissector_handle()` to obtain a dissector handle for the foo protocol. It then uses `dissector_add_uint()` to associate traffic on UDP port FOO_PORT (1234) with the foo protocol, so that Wireshark will call `dissect_foo()` when it receives UDP traffic on port 1234.

Wireshark's dissector convention is to put `proto_register_foo()` and `proto_reg_handoff_foo()` as the last two functions in the dissector source.

The next step is to write the dissecting function, `dissect_foo()`. We'll start with a basic placeholder.

Dissection.

```c
static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree _U_, void *data _U_)
{
    col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear the info column */
    col_clear(pinfo->cinfo,COL_INFO);

    return tvb_captured_length(tvb);
}
```

dissect_foo() is called to dissect the packets presented to it. The packet data is held in a special buffer referenced here as `tvb`. The packet_info structure contains general data about the protocol and we can update information here. The tree parameter is where the detail dissection takes place. Note that the _U_ following tree and data signals to the compiler that the parameters are unused, so that the compiler does not print a warning.

For now we'll do the minimum we can get away with. `col_set_str()` is used to set Wireshark's Protocol column to "FOO" so everyone can see it's being recognised. The only other thing we do is to
clear out any data in the INFO column if it's being displayed.

At this point we have a basic dissector ready to compile and install. The dissector doesn't do anything other than identify the protocol and label it. Here is the dissector's complete code:

*Complete packet-foo.c:*

```c
#include "config.h"
#include <epan/packet.h>

#define FOO_PORT 1234

static int proto_foo;

static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree _U_, void *data _U_)
{
    col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear the info column */
    col_clear(pinfo->cinfo,COL_INFO);

    return tvb_captured_length(tvb);
}

void
proto_register_foo(void)
{
    proto_foo = proto_register_protocol ( 
        "FOO Protocol", /* name         */
        "FOO",            /* short_name   */
        "foo"            /* filter_name  */
    );
}

void
proto_reg_handoff_foo(void)
{
    static dissector_handle_t foo_handle;

    foo_handle = create_dissector_handle(dissect_foo, proto_foo);
    dissector_add_uint("udp.port", FOO_PORT, foo_handle);
}
```

To compile this dissector and create a plugin a few support files are required, besides the dissector source in `packet-foo.c`:

- *CMakeLists.txt* - Contains the CMake file and version info for this plugin.
Samples of these files are available in the gryphon plugin directory (plugins/epan/gryphon). If you copy the files from the gryphon plugin, CMakeLists.txt will need to be updated with the correct plugin name, version info, and the relevant files to compile.

In the main top-level source directory, copy CMakeListsCustom.txt.example to CMakeListsCustom.txt and add the path of your plugin to the list in CUSTOM_PLUGIN_SRC_DIR.

Compile the dissector to a DLL or shared library and either run Wireshark from the build directory as detailed in Run Your Version Of Wireshark or copy the plugin binary into the plugin directory of your Wireshark installation and run that.

**Dissecting the protocol’s details**

Now that we have our basic dissector up and running, let’s do something with it. The simplest thing to start with is labeling the payload. We can label the payload by building a subtree to decode our results into. This subtree will hold all the protocol’s details and helps keep things looking nice in the detailed display.

We add the new subtree with proto_tree_add_item(), as is depicted below:

```
Plugin Packet Dissection.

static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear out stuff in the info column */
    col_clear(pinfo->cinfo,COL_INFO);

    proto_item *ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, ENC_NA);

    return tvb_captured_length(tvb);
}
```

As the FOO protocol does not encapsulate another protocol, we consume all of the tvb’s data, from 0 to the end (-1).

The final parameter specifies the “encoding” and is set to ENC_NA (“not applicable”), as the protocol doesn’t specifically use big endian (ENC_BIG_ENDIAN) or little endian (ENC_LITTLE_ENDIAN).

After adding the call to proto_tree_add_item(), there should be a label FOO in the protocol’s detailed display. Selecting this label will highlight the remaining contents of the packet.

Now let’s go to the next step and add some protocol dissection. To do this we’ll need to construct
tables to define which fields will be present in the packet and to store the opened/closed state of the subtree. We'll add these statically allocated arrays to the beginning of the file and name them hf_register_info (‘hf’ is short for ‘header field’) and ett. The arrays will then be registered after the call to proto_register_protocol() by calling proto_register_field_array() and proto_register_subtree_array():

Registering data structures.

```c
static int hf_foo_pdu_type;
static int ett_foo;

/* ... */

void
proto_register_foo(void)
{
    static hf_register_info hf[] = {
        { &hf_foo_pdu_type,
            { "FOO PDU Type", "foo.type",
              FT_UINT8, BASE_DEC,
              NULL, 0x0,
              NULL, HFILL }
        }
    };

    /* Setup protocol subtree array */
    static int *ett[] = {
        &ett_foo
    };

    proto_foo = proto_register_protocol (  
        "FOO Protocol", /* name       */  
        "FOO",         /* short_name */  
        "foo"          /* filter_name*/  
    );

    proto_register_field_array(proto_foo, hf, array_length(hf));
    proto_register_subtree_array(ett, array_length(ett));
}
```

As you can see, a field foo.type was defined inside the array of header fields.

Now we can dissect the FOO PDU Type (referenced as foo.type) field in dissect_foo() by adding the FOO Protocol’s subtree with proto_item_add_subtree() and then calling proto_tree_add_item() to add the field:
Dissector starting to dissect the packets.

As mentioned earlier, the foo protocol begins with an 8-bit packet type which can have three possible values: 1 - initialisation, 2 - terminate, 3 - data. Here's how we can add the packet details:

The `proto_item_add_subtree()` call has added a child node to the protocol tree which is where we will do our detail dissection. The expansion of this node is controlled by the `ett_foo` variable. This remembers if the node should be expanded or not as you move between packets. All subsequent dissection will be added to this tree, as you can see from the next call. A call to `proto_tree_add_item()` in the `foo_tree`, this time using the `hf_foo_pdu_type` to control the formatting of the item. The pdu type is one byte of data, starting at 0. We assume it is in network order (also called big endian), so that is why we use `ENC_BIG_ENDIAN`. For a 1-byte quantity, there is no order issue, but it is good practice to make this the same as any multibyte fields that may be present, and as we will see in the next section, this particular protocol uses network order.

If we look in detail at the `hf_foo_pdu_type` declaration in the static array we can see the details of the definition.

```
static hf_register_info hf[] = {
    { &hf_foo_pdu_type,
        { "FOO PDU Type", "foo.type",
            FT_UINT8, BASE_DEC,
            NULL, 0x0,
            NULL, HFILL }
    }
};
```

- `hf_foo_pdu_type` - The node's index.
- `FOO PDU Type` - The item's label.
- `foo.type` - The item's abbreviated name, for use in the display filter (e.g., `foo.type=1`).
- `FT_UINT8` - The item's type: An 8bit unsigned integer. This tallies with our call above where we tell it to only look at one byte.
- `BASE_DEC` - For an integer type, this tells it to be printed as a decimal number. It could be hexadecimal (BASE_HEX) or octal (BASE_OCT) if that made more sense.

We’ll ignore the rest of the structure for now.

If you install this plugin and try it out, you’ll see something that begins to look useful.

Now let’s finish off dissecting the simple protocol. We need to add a few more variables to the
hfarray, and a couple more procedure calls.

Wrapping up the packet dissection.

```c
static int hf_foo_flags;
static int hf_foo_sequenceno;
static int hf_foo_initialip;
...

static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    int offset = 0;

    ...
    proto_item *ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, ENC_NA);
    proto_tree *foo_tree = proto_item_add_subtree(ti, ett_foo);
    proto_tree_add_item(foo_tree, hf_foo_pdu_type, tvb, offset, 1, ENC_BIG_ENDIAN);
    offset += 1;
    proto_tree_add_item(foo_tree, hf_foo_flags, tvb, offset, 1, ENC_BIG_ENDIAN);
    offset += 1;
    proto_tree_add_item(foo_tree, hf_foo_sequenceno, tvb, offset, 2, ENC_BIG_ENDIAN);
    offset += 2;
    proto_tree_add_item(foo_tree, hf_foo_initialip, tvb, offset, 4, ENC_BIG_ENDIAN);
    offset += 4;
    ...

    return tvb_captured_length(tvb);
}

void
proto_register_foo(void) {
    ...
    {
        &hf_foo_flags,
            { "FOO PDU Flags", "foo.flags",
            FT_UINT8, BASE_HEX,
            NULL, 0x0,
            NULL, HFILL }
    },
    {
        &hf_foo_sequenceno,
            { "FOO PDU Sequence Number", "foo.seqn",
            FT_UINT16, BASE_DEC,
            NULL, 0x0,
            NULL, HFILL }
    },
    {
        &hf_foo_initialip,
```
This dissects all the bits of this simple hypothetical protocol. We've introduced a new variable offset into the mix to help keep track of where we are in the packet dissection. With these extra bits in place, the whole protocol is now dissected.

**Improving the dissection information**

We can certainly improve the display of the protocol with a bit of extra data. The first step is to add some text labels. Let's start by labeling the packet types. There is some useful support for this sort of thing by adding a couple of extra things. First we add a simple table of type to name.

*Naming the packet types.*

```c
static const value_string packettypenames[] = {
    { 1, "Initialise" },
    { 2, "Terminate" },
    { 3, "Data" },
    { 0, NULL }
};
```

This is a handy data structure that can be used to look up a name for a value. There are routines to directly access this lookup table, but we don't need to do that, as the support code already has that added in. We just have to give these details to the appropriate part of the data, using the VALS macro.

*Adding Names to the protocol.*

```c
{ &hf_foo_pdu_type,
    { "FOO PDU Type", "foo.type",
      FT_UINT8, BASE_DEC,
      VALS(packettypenames), 0x0,
      NULL, HFILL }
}
```

This helps in deciphering the packets, and we can do a similar thing for the flags structure. For this we need to add some more data to the table though.
Adding Flags to the protocol.

```c
#define FOO_START_FLAG      0x01
#define FOO_END_FLAG        0x02
#define FOO_PRIORITY_FLAG   0x04

static int hf_foo_startflag;
static int hf_foo_endflag;
static int hf_foo_priorityflag;

static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    ...
    ...
    static int* const bits[] = {
        &hf_foo_startflag,
        &hf_foo_endflag,
        &hf_foo_priorityflag,
        NULL
    };

    proto_tree_add_bitmask(foo_tree, tvb, offset, hf_foo_flags, ett_foo, bits,
                        ENC_BIG_ENDIAN);
    offset += 1;
    ...
    ...
    return tvb_captured_length(tvb);
}

void
proto_register_foo(void) {
    ...
    ...
    { &hf_foo_startflag,
        { "FOO PDU Start Flags", "foo.flags.start",
          FT_BOOLEAN, 8,
          NULL, FOO_START_FLAG,
          NULL, HFILL } }
    },
    { &hf_foo_endflag,
        { "FOO PDU End Flags", "foo.flags.end",
          FT_BOOLEAN, 8,
          NULL, FOO_END_FLAG,
          NULL, HFILL } }
    },
    { &hf_foo_priorityflag,
        { "FOO PDU Priority Flags", "foo.flags.priority",
          FT_BOOLEAN, 8,
          NULL, FOO_PRIORITY_FLAG,
          NULL, HFILL } }
    };
```
Some things to note here. For the flags, as each bit is a different flag, we use the type `FT_BOOLEAN`, as the flag is either on or off. Second, we include the flag mask in the 7th field of the data, which allows the system to mask the relevant bit. We've also changed the 5th field to 8, to indicate that we are looking at an 8 bit quantity when the flags are extracted. Then finally we add the extra constructs to the dissection routine.

This is starting to look fairly full featured now, but there are a couple of other things we can do to make things look even more pretty. At the moment our dissection shows the packets as "Foo Protocol" which whilst correct is a little uninformative. We can enhance this by adding a little more detail. First, let's get hold of the actual value of the protocol type. We can use the handy function `tvb_get_guint8()` to do this. With this value in hand, there are a couple of things we can do. First we can set the INFO column of the non-detailed view to show what sort of PDU it is - which is extremely helpful when looking at protocol traces. Second, we can also display this information in the dissection window.
So here, after grabbing the value of the first 8 bits, we use it with one of the built-in utility routines `val_to_str()`, to lookup the value. If the value isn't found we provide a fallback which just prints the value in hex. We use this twice, once in the INFO field of the columns — if it's displayed, and similarly we append this data to the base of our dissecting tree.

**How to add an expert item**

A dissector showing the protocol fields and interpretation of their values is very informative. It can be even more helpful if the dissector can draw your attention to fields where something noteworthy can be seen. This can be something as simple as the start flag of a session, or something more severe as an invalid value.

Here we take our dissector for FOO and add an expert item for the sequence number being zero (assuming that’s a noteworthy thing for this protocol).
# include <epan/expert.h>

static expert_field ei_foo_seqn_zero;

/* ... */

void
proto_register_foo(void)
{
    /* ... */
    expert_module_t* expert_foo;

    /* ... */
    static ei_register_info ei[] = {
    {
        &ei_foo_seqn_zero,
        { "foo.seqn_zero", PI_SEQUENCE, PI_CHAT,
          "Sequence number is zero", EXPFILL }
    }
    
};

    /* ... */
    expert_foo = expert_register_protocol(proto_foo);
    expert_register_field_array(expert_foo, ei, array_length(ei));
}

Let's go through this step by step. The data structures and functions needed for expert items are found in epan/expert.h, so we have to include that file.

Next we have to allocate an expert_field structure for every type of expert item we would like to add to the dissection. This structure is initialised with EI_INIT.

Now we have to register with the protocol we are providing expert info for. Since we already have a function to register our protocol, we add the expert info registration there too. This is done by calling expert_register_protocol() with the handle for the protocol we received earlier in this function.

Next we need to register an array of definitions of expert items that we would like to add to the dissection. This array, not unlike the array of header fields before, contains all the data the dissection engine needs to create and handle the expert items.

The expert item definition consists of a pointer to the expert_field structure we defined before and a structure with data elements of the expert item itself.

• "foo.seqn_zero" - The expert items display filter
**PI_SEQUENCE** - The group to which the expert item belongs

**PI_CHAT** - The severity of the expert item

"Sequence number is zero" - The text string added to the dissection

We'll ignore the rest of the structure for now.

To keep an overview of lots of expert items it helps to categorize them into groups. Currently there are several types of groups defined, e.g. **checksum**, **sequence**, **protocol**, etc. All these are defined in the epan/proto.h header file.

Not every noteworthy field value is of equal severity. The start of a session is nice to know, while an invalid value may be significant error in the protocol. To differentiate between these severities the expert item is assigned one of them: **comment**, **chat**, **note**, **warn** or **error**. Try to choose the lowest one which is suitable. The topic you're currently working on seems probably more important than it will look like in a few weeks.

With the expert item array setup, we add this to the dissection engine with a call to `expert_register_field_array()`.

Now that all information of the expert item is defined and registered it's time to actually add the expert item to the dissection.

**Expert item use.**

```c
static int dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    uint32_t sequenceno = 0xFFFF;

    /* ... */

    ti = proto_tree_add_item_ret_uint(foo_tree, hf_foo_sequenceno,
                                       tvb, offset, 2, ENC_BIG_ENDIAN, &sequenceno);
    if (sequenceno == 0) {
        expert_add_info(pinfo, ti, &ei_foo_seqn_zero);
    }
    /* ... */
}
```

There’s been a slight alteration to the function used to add the sequence number dissection. First the proto_item created by the function is saved in previously defined variable `ti`, and the actual value of the field is stored in the variable `sequenceno`. We can now use the value of this field to determine whether to add the expert item.

Adding the expert item is simply done by calling `expert_add_info()` with reference to the
packet_info structure, the proto item ti to add the expert item to and the previously defined and registered expert item information.

How to handle transformed data

Some protocols do clever things with data. They might possibly encrypt the data, or compress data, or part of it. If you know how these steps are taken it is possible to reverse them within the dissector.

As encryption can be tricky, let’s consider the case of compression. These techniques can also work for other transformations of data, where some step is required before the data can be examined.

What basically needs to happen here, is to identify the data that needs conversion, take that data and transform it into a new stream, and then call a dissector on it. Often this needs to be done “on-the-fly” based on clues in the packet. Sometimes this needs to be used in conjunction with other techniques, such as packet reassembly. The following shows a technique to achieve this effect.

Decompressing data packets for dissection.

```c
uint8_t flags = tvb_get_guint8(tvb, offset);
offset ++;
if (flags & FLAG_COMPRESSED) { /* the remainder of the packet is compressed */
    uint16_t orig_size = tvb_get_ntohs(tvb, offset);
    unsigned char *decompressed_buffer = (unsigned char*)wmem_alloc(pinfo->pool, orig_size);
    offset += 2;
    decompress_packet(tvb_get_ptr(tvb, offset, -1),
                     tvb_captured_length_remaining(tvb, offset),
                     decompressed_buffer, orig_size);
    /* Now re-setup the tvb buffer to have the new data */
    next_tvb = tvb_new_child_real_data(tvb, decompressed_buffer, orig_size, orig_size);
    add_new_data_source(pinfo, next_tvb, "Decompressed Data");
    } else {
    next_tvb = tvb_new_subset_remaining(tvb, offset);
    }
    offset = 0;
    /* process next_tvb from here on */
```

The first steps here are to recognise the compression. In this case a flag byte alerts us to the fact the remainder of the packet is compressed. Next we retrieve the original size of the packet, which in this case is conveniently within the protocol. If it’s not, it may be part of the compression routine to work it out for you, in which case the logic would be different.

So armed with the size, a buffer is allocated to receive the uncompressed data using wmem_alloc() in pinfo→pool memory, and the packet is decompressed into it. The tvb_get_ptr() function is useful to
get a pointer to the raw data of the packet from the offset onwards. In this case the decompression
routine also needs to know the length, which is given by the tvb_captured_length_remaining() function.

Next we build a new tvb buffer from this data, using the tvb_new_child_real_data() call. This data is
a child of our original data, so calling this function also acknowledges that. No need to call
	vb_set_free_cb() as the pinfo→pool was used (the memory block will be automatically freed when
the pinfo pool lifetime expires). Finally we add this tvb as a new data source, so that the detailed
display can show the decompressed bytes as well as the original.

After this has been set up the remainder of the dissector can dissect the buffer next tvb, as it's a
new buffer the offset needs to be 0 as we start again from the beginning of this buffer. To make the
rest of the dissector work regardless of whether compression was involved or not, in the case that
compression was not signaled, we use tvb_new_subset_remaining() to deliver us a new buffer based
on the old one but starting at the current offset, and extending to the end. This makes dissecting the
packet from this point on exactly the same regardless of compression.

How to reassemble split packets

Some protocols have times when they have to split a large packet across multiple other packets. In
this case the dissection can't be carried out correctly until you have all the data. The first packet
doesn't have enough data, and the subsequent packets don't have the expect format. To dissect
these packets you need to wait until all the parts have arrived and then start the dissection.

The following sections will guide you through two common cases. For a description of all possible
functions, structures and parameters, see epan/reassemble.h.

How to reassemble split UDP packets

As an example, let's examine a protocol that is layered on top of UDP that splits up its own data
stream. If a packet is bigger than some given size, it will be split into chunks, and somehow signaled
within its protocol.

To deal with such streams, we need several things to trigger from. We need to know that this packet
is part of a multi-packet sequence. We need to know how many packets are in the sequence. We
also need to know when we have all the packets.

For this example we'll assume there is a simple in-protocol signaling mechanism to give details. A
flag byte that signals the presence of a multi-packet sequence and also the last packet, followed by
an ID of the sequence and a packet sequence number.
We start by saving the fragmented state of this packet, so we can restore it later. Next comes some protocol specific stuff, to dig the fragment data out of the stream if it's present. Having decided it is present, we let the function `fragment_add_seq_check()` do its work. We need to provide this with a certain amount of parameters:

- The `msg_reassembly_table` table is for bookkeeping and is described later.
- The tvb buffer we are dissecting.
- The offset where the partial packet starts.
- The provided packet info.
- The sequence number of the fragment stream. There may be several streams of fragments in flight, and this is used to key the relevant one to be used for reassembly.
• Optional additional data for identifying the fragment. Can be set to NULL (as is done in the example) for most dissectors.

• msg_num is the packet number within the sequence.

• The length here is specified as the rest of the tvb as we want the rest of the packet data.

• Finally a parameter that signals if this is the last fragment or not. This might be a flag as in this case, or there may be a counter in the protocol.

Reassembling fragments part 2

```c
new_tvb = process_reassembled_data(tvb, offset, pinfo, 
    "Reassembled Message", frag_msg, &msg_frag_items, 
    NULL, msg_tree);

if (frag_msg) { /* Reassembled */
    col_append_str(pinfo->cinfo, COL_INFO, 
        " (Message Reassembled)");
} else { /* Not last packet of reassembled Short Message */
    col_append_fstr(pinfo->cinfo, COL_INFO, 
        " (Message fragment %u)", msg_num);
}

if (new_tvb) { /* take it all */
    next_tvb = new_tvb;
} else { /* make a new subset */
    next_tvb = tvb_new_subset_remaining(tvb, offset);
}
else { /* Not fragmented */
    next_tvb = tvb_new_subset_remaining(tvb, offset);
}

......

pinfo->fragmented = save_fragmented;
```

Having passed the fragment data to the reassembly handler, we can now check if we have the whole message. If there is enough information, this routine will return the newly reassembled data buffer.

After that, we add a couple of informative messages to the display to show that this is part of a sequence. Then a bit of manipulation of the buffers and the dissection can proceed. Normally you will probably not bother dissecting further unless the fragments have been reassembled as there won’t be much to find. Sometimes the first packet in the sequence can be partially decoded though if you wish.

Now the mysterious data we passed into the fragment_add_seq_check().
First a `reassembly_table` structure is declared and initialised in the protocol initialisation routine. The second parameter specifies the functions that should be used for identifying fragments. We will use `addresses_ports_reassembly_table_functions` in order to identify fragments by the given sequence number (`msg_seqid`), the source and destination addresses and ports from the packet.

Following that, a `fragment_items` structure is allocated and filled in with a series of ett items, hf data items, and a string tag. The ett and hf values should be included in the relevant tables like all the other variables your protocol may use. The hf variables need to be placed in the structure something like the following. Of course the names may need to be adjusted.
static hf_register_info hf[] = {
    ... {
        &hf_msg_fragments,
        { "Message fragments", "msg.fragments", 
          FT_NONE, BASE_NONE, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment,
        { "Message fragment", "msg.fragment", 
          FT_FRAMENUM, BASE_NONE, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment_overlap,
        { "Message fragment overlap", "msg.fragment.overlap", 
          FT_BOOLEAN, 0, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment_overlap_conflicts,
        { "Message fragment overlapping with conflicting data", 
          "msg.fragment.overlap.conflicts", 
          FT_BOOLEAN, 0, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment_multiple_tails,
        { "Message has multiple tail fragments", 
          "msg.fragment.multiple_tails", 
          FT_BOOLEAN, 0, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment_too_long_fragment,
        { "Message fragment too long", "msg.fragment.too_long_fragment", 
          FT_BOOLEAN, 0, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment_error,
        { "Message defragmentation error", "msg.fragment.error", 
          FT_FRAMENUM, BASE_NONE, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_fragment_count,
        { "Message fragment count", "msg.fragment.count", 
          FT_UINT32, BASE_DEC, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_reassembled_in,
        { "Reassembled in", "msg.reassembled.in", 
          FT_FRAMENUM, BASE_NONE, NULL, 0x00, NULL, HFILL } },
    {
        &hf_msg_reassembled_length,
        { "Reassembled length", "msg.reassembled.length", 
          FT_UINT32, BASE_DEC, NULL, 0x00, NULL, HFILL } },
    ...
These hf variables are used internally within the reassembly routines to make useful links, and to add data to the dissection. It produces links from one packet to another, such as a partial packet having a link to the fully reassembled packet. Likewise there are back pointers to the individual packets from the reassembled one. The other variables are used for flagging up errors.

**How to reassemble split TCP Packets**

A dissector gets a `tvbuff_t` pointer which holds the payload of a TCP packet. This payload contains the header and data of your application layer protocol.

When dissecting an application layer protocol you cannot assume that each TCP packet contains exactly one application layer message. One application layer message can be split into several TCP packets.

You also cannot assume that a TCP packet contains only one application layer message and that the message header is at the start of your TCP payload. More than one messages can be transmitted in one TCP packet, so that a message can start at an arbitrary position.

This sounds complicated, but there is a simple solution. `tcp_dissect_pdus()` does all this tcp packet reassembling for you. This function is implemented in `epan/dissectors/packet-tcp.h`. 

```c
static int *ett[] =
{
  ...
  &ett_msg_fragment,
  &ett_msg_fragments
  ...
```
Reassembling TCP fragments

```c
#include "config.h"
#include <epan/packet.h>
#include <epan/prefs.h>
#include "packet-tcp.h"
...
#define FRAME_HEADER_LEN 8

/* This method dissects fully reassembled messages */
static int
dissect_foo_message(tvbuff_t *tvb, packet_info *pinfo _U_, proto_tree *tree _U_, void *data _U_)
{
    /* TODO: implement your dissecting code */
    return tvb_captured_length(tvb);
}

/* determine PDU length of protocol foo */
static unsigned
get_foo_message_len(packet_info *pinfo _U_, tvbuff_t *tvb, int offset, void *data _U_)
{
    /* TODO: change this to your needs */
    return (unsigned)tvb_get_ntohl(tvb, offset+4); /* e.g. length is at offset 4 */
}

/* The main dissecting routine */
static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data)
{
    tcp_dissect_pdus(tvb, pinfo, tree, true, FRAME_HEADER_LEN,
                     get_foo_message_len, dissect_foo_message, data);
    return tvb_captured_length(tvb);
}
...
```

As you can see this is really simple. Just call `tcp_dissect_pdus()` in your main dissection routine and move your message parsing code into another function. This function gets called whenever a message has been reassembled.

The parameters tvb, pinfo, tree and data are just handed over to `tcp_dissect_pdus()`. The 4th parameter is a flag to indicate if the data should be reassembled or not. This could be set according to a dissector preference as well. Parameter 5 indicates how much data has at least to be available.
to be able to determine the length of the foo message. Parameter 6 is a function pointer to a method that returns this length. It gets called when at least the number of bytes given in the previous parameter is available. Parameter 7 is a function pointer to your real message dissector. Parameter 8 is the data passed in from parent dissector.

Protocols which need more data before the message length can be determined can return zero. Other values smaller than the fixed length will result in an exception.

How to tap protocols

Adding a Tap interface to a protocol allows it to do some useful things. In particular you can produce protocol statistics from the tap interface.

A tap is basically a way of allowing other items to see what’s happening as a protocol is dissected. A tap is registered with the main program, and then called on each dissection. Some arbitrary protocol specific data is provided with the routine that can be used.

To create a tap, you first need to register a tap. A tap is registered with an integer handle, and registered with the routine register_tap(). This takes a string name with which to find it again.

Initialising a tap

```c
#include <epan/packet.h>
#include <epan/tap.h>

static int foo_tap;

struct FooTap {
    int packet_type;
    int priority;
    ...
};

void proto_register_foo(void) {
    ...
    foo_tap = register_tap("foo");
}
```

Whilst you can program a tap without protocol specific data, it is generally not very useful. Therefore it’s a good idea to declare a structure that can be passed through the tap. This needs to be a static structure as it will be used after the dissection routine has returned. It’s generally best to pick out some generic parts of the protocol you are dissecting into the tap data. A packet type, a priority or a status code maybe. The structure really needs to be included in a header file so that it can be included by other components that want to listen in to the tap.

Once you have these defined, it’s simply a case of populating the protocol specific structure and
then calling `tap_queue_packet`, probably as the last part of the dissector.

Calling a protocol tap

```c
static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    ...
    fooinfo = wmem_alloc(pinfo->pool, sizeof(struct FooTap));
    fooinfo->packet_type = tvb_get_guint8(tvb, 0);
    fooinfo->priority = tvb_get_ntohs(tvb, 8);
    ...
    tap_queue_packet(foo_tap, pinfo, fooinfo);
    ...
    return tvb_captured_length(tvb);
}
```

This now enables those interested parties to listen in on the details of this protocol conversation.

**How to produce protocol stats**

Given that you have a tap interface for the protocol, you can use this to produce some interesting statistics (well presumably interesting!) from protocol traces.

This can be done in a separate plugin, or in the same plugin that is doing the dissection. The latter scheme is better, as the tap and stats module typically rely on sharing protocol specific data, which might get out of step between two different plugins.

Here is a mechanism to produce statistics from the above TAP interface.

**Initialising a stats interface**

```c
#include <epan/stats_tree.h>

/* register all http trees */
static void register_foo_stat_trees(void) {
    stats_tree_register_plugin("foo", "foo", "Foo" STATS_TREE_MENU_SEPARATOR "Packet Types", 0,
        foo_stats_tree_packet, foo_stats_tree_init, NULL);
}

WS_DLL_PUBLIC_DEF void plugin_register_tap_listener(void)
{
    register_foo_stat_trees();
}
```
Working from the bottom up, first the plugin interface entry point is defined, `plugin_register_tap_listener()`. This simply calls the initialisation function `register_foo_stat_trees()`.

This in turn calls the `stats_tree_register_plugin()` function, which takes three strings, an integer, and three callback functions.

1. This is the tap name that is registered.
2. An abbreviation of the stats name.
3. The name of the stats module. A “/” character can be used to make sub menus.
4. Flags for per-packet callback
5. The function that will called to generate the stats.
6. A function that can be called to initialise the stats data.
7. A function that will be called to clean up the stats data.

In this case we only need the first two functions, as there is nothing specific to clean up.

Initialising a stats session

```c
static const uint8_t* st_str_packets = "Total Packets";
static const uint8_t* st_str_packet_types = "FOO Packet Types";
static int st_node_packets = -1;
static int st_node_packet_types = -1;

static void foo_stats_tree_init(stats_tree* st)
{
    st_node_packets = stats_tree_create_node(st, st_str_packets, 0, STAT_DT_INT, true);
    st_node_packet_types = stats_tree_create_pivot(st, st_str_packet_types, st_node_packets);
}
```

In this case we create a new tree node, to handle the total packets, and as a child of that we create a pivot table to handle the stats about different packet types.
Generating the stats

```c
static tap_packet_status foo_stats_tree_packet(stats_tree* st, packet_info* pinfo,
    epan_dissect_t* edt, const void* p, tap_flags_t flags)
{
    struct FooTap *pi = (struct FooTap *)p;
    tick_stat_node(st, st_str_packets, 0, false);
    stats_tree_tick_pivot(st, st_node_packet_types,
        val_to_str(pi->packet_type, packettypenames, "Unknown packet type (%d"));
    return TAP_PACKET_REDRAW;
}
```

In this case the processing of the stats is quite simple. First we call the `tick_stat_node` for the `st_str_packets` packet node, to count packets. Then a call to `stats_tree_tick_pivot()` on the `st_node_packet_types` subtree allows us to record statistics by packet type.

How to use conversations

Some info about how to use conversations in a dissector can be found in the file `doc/README.dissector`, chapter 2.2.

**idl2wrs: Creating dissectors from CORBA IDL files**

Many of Wireshark’s dissectors are automatically generated. This section shows how to generate one from a CORBA IDL file.

**What is it?**

As you have probably guessed from the name, `idl2wrs` takes a user specified IDL file and attempts to build a dissector that can decode the IDL traffic over GIOP. The resulting file is “C” code, that should compile okay as a Wireshark dissector.

`idl2wrs` parses the data struct given to it by the `omniidl` compiler, and using the GIOP API available in `packet-giop.[ch]`, generates `get_CDR_xxx` calls to decode the CORBA traffic on the wire.

It consists of 4 main files.

**README.idl2wrs**

This document

**wireshark_be.py**

The main compiler backend

**wireshark_gen.py**

A helper class, that generates the C code.
idl2wrs
A simple shell script wrapper that the end user should use to generate the dissector from the IDL file(s).

Why do this?

It is important to understand what CORBA traffic looks like over GIOP/IIOP, and to help build a tool that can assist in troubleshooting CORBA interworking. This was especially the case after seeing a lot of discussions about how particular IDL types are represented inside an octet stream.

I have also had comments/feedback that this tool would be good for say a CORBA class when teaching students what CORBA traffic looks like “on the wire”.

It is also COOL to work on a great Open Source project such as the case with “Wireshark” (https://www.wireshark.org/).

How to use idl2wrs

To use the idl2wrs to generate Wireshark dissectors, you need the following:

• Python must be installed. See https://python.org/
• omniidl from the omniORB package must be available. See http://omniorb.sourceforge.net/
• Of course you need Wireshark installed to compile the code and tweak it if required. idl2wrs is part of the standard Wireshark distribution

To use idl2wrs to generate an Wireshark dissector from an idl file use the following procedure:

• To write the C code to stdout.

  $ idl2wrs <your_file.idl>

  e.g.:

  $ idl2wrs echo.idl

• To write to a file, just redirect the output.

  $ idl2wrs echo.idl > packet-test-idl.c

You may wish to comment out the register_giop_user_module() code and that will leave you with heuristic dissection.

If you don’t want to use the shell script wrapper, then try steps 3 or 4 instead.
• To write the C code to stdout.

```
$ omniidl -p ./ -b wireshark_be <your file.idl>
```

e.g.:

```
$ omniidl -p ./ -b wireshark_be echo.idl
```

• To write to a file, just redirect the output.

```
$ omniidl -p ./ -b wireshark_be echo.idl > packet-test-idl.c
```

You may wish to comment out the `register_giop_user_module()` code and that will leave you with heuristic dissection.

• Copy the resulting C code to subdirectory epan/dissectors/ inside your Wireshark source directory.

```
$ cp packet-test-idl.c /dir/where/wireshark/lives/epan/dissectors/
```

The new dissector has to be added to CMakeLists.txt in the same directory. Look for the declaration `DISSECTOR_SRC` and add the new dissector there. For example,

```
DISSECTOR_SRC = \
    ${CMAKE_CURRENT_SOURCE_DIR}/packet-2dparityfec.c \n    ${CMAKE_CURRENT_SOURCE_DIR}/packet-3com-njack.c \n    ...
```

becomes

```
DISSECTOR_SRC = \n    ${CMAKE_CURRENT_SOURCE_DIR}/packet-test-idl.c \n    ${CMAKE_CURRENT_SOURCE_DIR}/packet-2dparityfec.c \n    ${CMAKE_CURRENT_SOURCE_DIR}/packet-3com-njack.c \n    ...
```

For the next steps, go up to the top of your Wireshark source directory.

• Create a build dir
$ mkdir build && cd build

• Run cmake

$ cmake ..

• Build the code

$ make

• Good Luck !!

**TODO**

• Exception code not generated (yet), but can be added manually.
• Enums not converted to symbolic values (yet), but can be added manually.
• Add command line options etc
• More I am sure :-)

**Limitations**

See the TODO list inside *packet-giop.c*

**Notes**

The `-p ./` option passed to omniidl indicates that the `wireshark_be.py` and `wireshark_gen.py` are residing in the current directory. This may need tweaking if you place these files somewhere else.

If it complains about being unable to find some modules (e.g. `tempfile.py`), you may want to check if `PYTHONPATH` is set correctly.
Lua Support in Wireshark

Introduction

Lua is a powerful light-weight programming language designed for extending applications. Wireshark contains an embedded Lua interpreter which can be used to write dissectors, taps, and capture file readers and writers. Wireshark versions 4.2.x and earlier support Lua 5.1 and 5.2, and newer versions support Lua 5.3, and 5.4. All versions support the Lua BitOp library.

If Lua is enabled, Wireshark will first try to load a file named init.lua from the global plugins directory. and then from the user’s personal plugins directory. Then all files ending with .lua are loaded from the global plugins directory. Then all files ending with .lua in the personal Lua plugin's directory.

Whether or not Lua scripts are enabled can be controlled via the enable_lua variable. Lua scripts are enabled by default. To disable Lua scripts, set the enable_lua variable to false. Wireshark 2.6 and earlier enabled or disabled Lua scripts using the variable disable_lua (deprecated). If both enable_lua and disable_lua are present, disable_lua is ignored.

Example for init.lua

```lua
-- Set enable_lua to false to disable Lua support.
enable_lua = true

if not enable_lua then
    return
end

-- If false and Wireshark was started as (setuid) root, then the user
-- will not be able to execute custom Lua scripts from the personal
-- configuration directory, the -Xlua_script command line option or
-- the Lua Evaluate menu option in the GUI.
-- Note: Not checked on Windows. running_superuser is always false.
run_user_scripts_when_superuser = true

The command line option -X lua_script:file.lua can also be used to load specific Lua scripts.

The Lua code is executed after all protocol dissectors are initialized and before reading any file.

Wireshark for Windows uses a modified Lua runtime (lua-unicode) to support Unicode (UTF-8) filesystem paths. This brings consistency with other platforms (for example, Linux and macOS).

Example: Creating a Menu with Lua

The code below adds a menu "Lua Dialog Test" under the Tools menu. When selected, it opens a
dialog prompting the user for input and then opens a text window with the output.

```lua
-- Define the menu entry's callback
local function dialog_menu()
    local function dialog_func(person, eyes, hair)
        local window = TextWindow.new("Person Info");
        local message = string.format("Person %s with %s eyes and %s hair.", person, eyes, hair);
        window:set(message);
    end
    new_dialog("Dialog Test", dialog_func, "A Person", "Eyes", "Hair")
end

-- Create the menu entry
register_menu("Lua Dialog Test", dialog_menu, MENU_TOOLS_UNSORTED)

-- Notify the user that the menu was created
if gui_enabled() then
    local splash = TextWindow.new("Hello!");
    splash:set("Wireshark has been enhanced with a useless feature.\n")
    splash:append("Go to 'Tools->Lua Dialog Test' and check it out!")
end

Example: Dissector written in Lua

local p_multi = Proto("multi", "MultiProto");

local vs_protos = {
    [2] = "mtp2",
    [3] = "mtp3",
    [5] = "h248",
    [6] = "ranap",
    [7] = "rnsap",
    [8] = "nbap"
}

local f_proto = ProtoField.uint8("multi.protocol", "Protocol", base.DEC, vs_protos)
local f_dir = ProtoField.uint8("multi.direction", "Direction", base.DEC, { [1] = "incoming", [0] = "outgoing"})
local f_text = ProtoField.string("multi.text", "Text")

p_multi.fields = { f_proto, f_dir, f_text }
local data_dis = Dissector.get("data")

local protos = {
    [2] = Dissector.get("mtp2"),
    [3] = Dissector.get("mtp3"),
    [4] = Dissector.get("alcap"),
    [5] = Dissector.get("h248"),
    [6] = Dissector.get("ranap"),
    [7] = Dissector.get("rnsap"),
    [8] = Dissector.get("nbap"),
    [9] = Dissector.get("rrc"),
    [10] = DissectorTable.get("sctp.ppi").get_dissector(3), -- m3ua
    [11] = DissectorTable.get("ip.proto").get_dissector(132), -- sctp
}

function p_multi.dissector(buf, pkt, tree)
    local subtree = tree:add(p_multi, buf(0,2))
    subtree:add(f_proto, buf(0,1))
    subtree:add(f_dir, buf(1,1))

    local proto_id = buf(0,1):uint()
    local dissector = protos[proto_id]

    if dissector ~= nil then
        -- Dissector was found, invoke subdissector with a new Tvb,
        -- created from the current buffer (skipping first two bytes).
        dissector:call(buf(2):tvb(), pkt, tree)
    elseif proto_id < 2 then
        subtree:add(f_text, buf(2))
        -- pkt.cols.info:set(buf(2, buf:len() - 3):string())
    else
        -- fallback dissector that just shows the raw data.
        data_dis:call(buf(2):tvb(), pkt, tree)
    end
end

local wtap_encap_table = DissectorTable.get("wtap_encap")
local udp_encap_table = DissectorTable.get("udp.port")

wtap_encap_table:add(wtap.USER15, p_multi)
wtap_encap_table:add(wtap.USER12, p_multi)
udp_encap_table:add(7555, p_multi)
-- This program will register a menu that will open a window with a count of occurrences
-- of every address in the capture

default function menuable_tap()
    -- Declare the window we will use
    local tw = TextWindow.new("Address Counter")

    -- This will contain a hash of counters of appearances of a certain address
    local ips = {}

    -- this is our tap
    local tap = Listener.new();

    local function remove()
        -- this way we remove the listener that otherwise will remain running indefinitely
        tap:remove();
    end

    -- we tell the window to call the remove() function when closed
    tw:set_atclose(remove)

    -- this function will be called once for each packet
    function tap.packet(pinfo,tvb)
        local src = ips[tostring(pinfo.src)] or 0
        local dst = ips[tostring(pinfo.dst)] or 0

        ips[tostring(pinfo.src)] = src + 1
        ips[tostring(pinfo.dst)] = dst + 1
    end

    -- this function will be called once every few seconds to update our window
    function tap.draw(t)
        tw:clear()
        for ip,num in pairs(ips) do
            tw:append(ip .. "\t" .. num .. "\n");
        end
    end

    -- this function will be called whenever a reset is needed
    -- e.g. when reloading the capture file
    function tap.reset()
        tw:clear()
        ips = {}
    end
end

-- Ensure that all existing packets are processed.
retap_packets()
end

-- using this function we register our function
-- to be called when the user selects the Tools->Test->Packets menu
register_menu("Test/Packets", menuable_tap, MENU_TOOLS_UNSORTED)
Wireshark’s Lua API Reference Manual

This Part of the User Guide describes the Wireshark specific functions in the embedded Lua.

Classes group certain functionality, the following notational conventions are used:

- `Class.function()` represents a class method (named `function`) on class `Class`, taking no arguments.
- `Class.function(a)` represents a class method taking one argument.
- `Class.function(...)` represents a class method taking a variable number of arguments.
- `class:method()` represents an instance method (named `method`) on an instance of class `Class`, taking no arguments. Note the lowercase notation in the documentation to clarify an instance.
- `class.prop` represents a property `prop` on the instance of class `Class`.

Trying to access a non-existing property, function or method currently gives an error, but do not rely on it as the behavior may change in the future.

Utility Functions

Global Functions

`get_version()`

Gets the Wireshark version as a string.

**Returns**

The version string, e.g. "3.2.5".

`set_plugin_info(table)`

Set a Lua table with meta-data about the plugin, such as version.

The passed-in Lua table entries need to be keyed/indexed by the following:

- "version" with a string value identifying the plugin version (required)
- "description" with a string value describing the plugin (optional)
- "author" with a string value of the author's name(s) (optional)
- "repository" with a string value of a URL to a repository (optional)

Not all of the above key entries need to be in the table. The 'version' entry is required, however. The others are not currently used for anything, but might be in the future and thus using them might be useful. Table entries keyed by other strings are ignored, and do not cause an error.
Example

```lua
local my_info = {
    version = "1.0.1",
    author = "Jane Doe",
    repository = "https://github.com/octocat/Spoon-Knife"
}

set_plugin_info(my_info)
```

Arguments

table

The Lua table of information.

format_date(timestamp)

Formats an absolute timestamp into a human readable date.

Arguments

timestamp

A timestamp value to convert.

Returns

A string with the formatted date

format_time(timestamp)

Formats a relative timestamp in a human readable time.

Arguments

timestamp

A timestamp value to convert.

Returns

A string with the formatted time

get_preference(preference)

Get a preference value.

Arguments
**preference**  
The name of the preference.

**Returns**  
The preference value, or nil if not found.

**set_preference(preference, value)**  
Set a preference value.

**Arguments**  

**preference**  
The name of the preference.

**value**  
The preference value to set.

**Returns**  
true if changed, false if unchanged or nil if not found.

**reset_preference(preference)**  
Reset a preference to default value.

**Arguments**  

**preference**  
The name of the preference.

**Returns**  
true if valid preference

**apply_preferences()**  
Write preferences to file and apply changes.

**report_failure(text)**  
Reports a failure to the user.

**Arguments**
loadfile(filename)

Loads a Lua file and compiles it into a Lua chunk, similar to the standard `loadfile` but searches additional directories. The search order is the current directory, followed by the user's personal configuration directory, and finally the global configuration directory.

Example

```
local loaded_chunk = assert(loadfile('foo.lua'))
loaded_chunk()
-- ok to call foo at this point
foo(1,2)
```

Arguments

filename

Name of the file to be loaded. If the file does not exist in the current directory, the user and system directories are searched.

dofile(filename)

Loads a Lua file and executes it as a Lua chunk, similar to the standard `dofile` but searches additional directories. The search order is the current directory, followed by the user's personal configuration directory, and finally the global configuration directory.

Arguments

filename

Name of the file to be run. If the file does not exist in the current directory, the user and system directories are searched.

register_stat_cmd_arg(argument, [action])

Register a function to handle a \texttt{-z} option

Arguments
argument

The name of the option argument.

action (optional)

The function to be called when the command is invoked.

GUI Support

ProgDlg

Creates and manages a modal progress bar. This is intended to be used with coroutines, where a main UI thread controls the progress bar dialog while a background coroutine (worker thread) yields to the main thread between steps. The main thread checks the status of the [Cancel] button and if it's not set, returns control to the coroutine.

Figure 3. A progress bar in action

The legacy (GTK+) user interface displayed this as a separate dialog, hence the “Dlg” suffix. The Qt user interface shows a progress bar inside the main status bar.

ProgDlg.new([title], [task])

Creates and displays a new ProgDlg progress bar with a [Cancel] button and optional title. It is highly recommended that you wrap code that uses a ProgDlg instance because it does not automatically close itself upon encountering an error. Requires a GUI.

Example

```lua
if not gui_enabled() then return end

local p = ProgDlg.new("Constructing", "tacos")

-- We have to wrap the ProgDlg code in a pcall in case some unexpected
-- error occurs.
local ok, errmsg = pcall(function()
    local co = coroutine.create(
        function()
            local limit = 100000
            for i=1,limit do
                print("co", i)
                coroutine.yield(i/limit, "step ".i.." of
                "..limit)
            end
        end
    end
end
```
-- Whenever coroutine yields, check the status of the cancel button to determine when to break. Wait up to 20 sec for coroutine to finish.
local start_time = os.time()
while coroutine.status(co) ~= 'dead' do
    local elapsed = os.time() - start_time

    -- Quit if cancel button pressed or 20 seconds elapsed
    if p:stopped() or elapsed > 20 then
        break
    end

    local res, val, val2 = coroutine.resume(co)
    if not res or res == false then
        if val then
            debug(val)
        end
        print('coroutine error')
        break
    end

    -- show progress in progress dialog
    p:update(val, val2)
end

p:close()
if not ok and errmsg then
    report_failure(errmsg)
end

Arguments

title (optional)
    Title of the progress bar. Defaults to "Progress".

task (optional)
    Optional task name, which will be appended to the title. Defaults to the empty string (""").

Returns

The newly created ProgDlg object.
progdlg:update(progress, [task])
Sets the progress dialog's progress bar position based on percentage done.

Arguments

progress
Progress value, e.g. 0.75. Value must be between 0.0 and 1.0 inclusive.

task (optional)
Task name. Currently ignored. Defaults to empty string ("").

Errors

• GUI not available
• Cannot be called for something not a ProgDlg
• Progress value out of range (must be between 0.0 and 1.0)

progdlg:stopped()
Checks whether the user has pressed the [Cancel] button.

Returns

Boolean true if the user has asked to stop the operation, false otherwise.

progdlg:close()
Hides the progress bar.

Returns

A string specifying whether the Progress Dialog has stopped or not.

Errors

• GUI not available

TextWindow

Creates and manages a text window. The text can be read-only or editable, and buttons can be added below the text.
TextWindow.new([title])

Creates a new TextWindow text window and displays it. Requires a GUI.

Example

```lua
if not gui_enabled() then return end

-- create new text window and initialize its text
local win = TextWindow.new("Log")
win:set("Hello world!"

-- add buttons to clear text window and to enable editing
win:add_button("Clear", function() win:clear() end)
win:add_button("Enable edit", function() win:set_editable(true) end)

-- add button to change text to uppercase
win:add_button("Uppercase", function()
    local text = win:get_text()
    if text ~= "" then
        win:set(string.upper(text))
    end
end)

-- print "closing" to stdout when the user closes the text window
win:set_atclose(function() print("closing") end)
```

Arguments
title (optional)
Title of the new window. Optional. Defaults to "Untitled Window".

Returns
The newly created TextWindow object.

Errors
• GUI not available

textwindow:set_atclose(action)
Set the function that will be called when the text window closes.

Arguments
action
A Lua function to be executed when the user closes the text window.

Returns
The TextWindow object.

Errors
• GUI not available

textwindow:set(text)
Sets the text to be displayed.

Arguments
text
The text to be displayed.

Returns
The TextWindow object.

Errors
• GUI not available

textwindow:append(text)
Appends text to the current window contents.
Arguments
text
   The text to be appended.

Returns
The TextWindow object.

Errors
• GUI not available

textwindow:prepend(text)
Prepends text to the current window contents.

Arguments
text
   The text to be prepended.

Returns
The TextWindow object.

Errors
• GUI not available

textwindow:clear()
Erases all of the text in the window.

Returns
The TextWindow object.

Errors
• GUI not available

textwindow:get_text()
Get the text of the window.

Returns
The TextWindow's text.

Errors

- GUI not available

textwindow:close()

Close the window.

Errors

- GUI not available

textwindow:set_editable(editable)

Make this text window editable.

Arguments

editable (optional)

true to make the text editable, false otherwise. Defaults to true.

Returns

The TextWindow object.

Errors

- GUI not available

textwindow:add_button(label, function)

Adds a button with an action handler to the text window.

Arguments

label

The button label.

function

The Lua function to be called when the button is pressed.

Returns

The TextWindow object.

Errors
Global Functions

gui_enabled()

Checks if we're running inside a GUI (i.e. Wireshark) or not.

Returns

Boolean true if a GUI is available, false if it isn’t.

register_menu(name, action, [group])

Register a menu item in one of the main menus. Requires a GUI.

Arguments

name

The name of the menu item. Use slashes to separate submenus. (e.g. Lua Scripts › My Fancy Statistics). (string)

action

The function to be called when the menu item is invoked. The function must take no arguments and return nothing.

group (optional)

Where to place the item in the menu hierarchy. If omitted, defaults to MENU_STAT_GENERIC. Valid packet (Wireshark) items are:

- MENU_PACKET_ANALYZE_UNSORTED: Analyze
- MENU_PACKET_STAT_UNSORTED: Statistics
- MENU_STAT_GENERIC: Statistics, first section
- MENU_STAT_CONVERSATION_LIST: Statistics › Conversation List
- MENU_STAT_ENDPOINT_LIST: Statistics › Endpoint List
- MENU_STAT_RESPONSE_TIME: Statistics › Service Response Time
- MENU_STAT_RSERPOOL = Statistics › Reliable Server Pooling (RSerPool)
- MENU_STAT_TELEPHONY: Telephony
- MENU_STAT_TELEPHONY_ANSI: Telephony › ANSI
- MENU_STAT_TELEPHONY_GSM: Telephony › GSM
- MENU_STAT_TELEPHONY_3GPP_UU: Telephony › 3GPP Uu
Valid log (Logray) items are:

- MENU_LOG_ANALYZE_UNSORTED: Analyze
- MENU_LOG_STAT_UNSORTED = 16

The following are deprecated and shouldn’t be used in new code:

- MENU_ANALYZE_UNSORTED, superseded by MENU_PACKET_ANALYZE_UNSORTED
- MENU_ANALYZE_CONVERSATION, superseded by MENU_ANALYZE_CONVERSATION_FILTER
- MENU_STAT_CONVERSATION, superseded by MENU_STAT_CONVERSATION_LIST
- MENU_STAT_ENDPOINT, superseded by MENU_STAT_ENDPOINT_LIST
- MENU_STAT_RESPONSE, superseded by MENU_STAT_RESPONSE_TIME
- MENU_STAT_UNSORTED, superseded by MENU_PACKET_STAT_UNSORTED

```
register_packet_menu(name, action, [required_fields])
```

Register a menu item in the packet list.

**Arguments**

**name**

The name of the menu item. Use slashes to separate submenus. (e.g. level1/level2/name). (string)

**action**

The function to be called when the menu item is invoked. The function must take a variable number of arguments and return nothing. The arguments will be FieldInfo objects, one for each field present in the selected packet.

**required_fields (optional)**

A comma-separated list of packet fields (e.g., http.host,dns.qry.name) which all must be present for the menu to be displayed. If omitted, the packet menu will be displayed for all packets.

```
new_dialog(title, action, ...)
```

Displays a dialog, prompting for input. The dialog includes an [OK] button and [Cancel] button. Requires a GUI.
Example

```lua
if not gui_enabled() then return end

-- Prompt for IP and port and then print them to stdout
local label_ip = "IP address"
local label_port = "Port"
local function print_ip(ip, port)
    print(label_ip, ip)
    print(label_port, port)
end
new_dialog("Enter IP address", print_ip, label_ip, label_port)

-- Prompt for 4 numbers and then print their product to stdout
new_dialog(
    "Enter 4 numbers",
    function (a, b, c, d) print(a * b * c * d) end,
    "a", "b", "c", "d"
)
```

Arguments

**title**

The title of the dialog.

**action**

Action to be performed when the user presses [OK].

... Strings to be used as labels of the dialog's fields. Each string creates a new labeled field. The first field is required. Instead of a string it is possible to provide tables with fields 'name' and 'value' of type string. Then the created dialog's field will be labeled with the content of name and prefilled with the content of value.

Errors

- GUI not available
- At least one field required
retap_packets()
Rescans all packets and runs each tap listener without reconstructing the display.

copy_to_clipboard(text)
Copy a string into the clipboard. Requires a GUI.

Arguments
text
The string to be copied into the clipboard.

open_capture_file(filename, filter)
Open and display a capture file. Requires a GUI.

Arguments
filename
The name of the file to be opened.

filter
The display filter to be applied once the file is opened.

get_filter()
Get the main filter text.

set_filter(text)
Set the main filter text.

Arguments
text
The filter’s text.

get_color_filter_slot(row)
Gets the current packet coloring rule (by index) for the current session. Wireshark reserves 10 slots for these coloring rules. Requires a GUI.

Arguments
row
The index (1-10) of the desired color filter value in the temporary coloring rules list.
Table 5. Default background colors

<table>
<thead>
<tr>
<th>Index</th>
<th>RGB (hex)</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ffc0c0</td>
<td>pink 1</td>
</tr>
<tr>
<td>2</td>
<td>ffc0ff</td>
<td>pink 2</td>
</tr>
<tr>
<td>3</td>
<td>e0c0e0</td>
<td>purple 1</td>
</tr>
<tr>
<td>4</td>
<td>c0c0ff</td>
<td>purple 2</td>
</tr>
<tr>
<td>5</td>
<td>c0e0e0</td>
<td>green 1</td>
</tr>
<tr>
<td>6</td>
<td>c0ffff</td>
<td>green 2</td>
</tr>
<tr>
<td>7</td>
<td>c0fffc0</td>
<td>green 3</td>
</tr>
<tr>
<td>8</td>
<td>ffffc0</td>
<td>yellow 1</td>
</tr>
<tr>
<td>9</td>
<td>e0e0c0</td>
<td>yellow 2</td>
</tr>
<tr>
<td>10</td>
<td>e0e0e0</td>
<td>gray</td>
</tr>
</tbody>
</table>

set_color_filter_slot(row, text)

Sets a packet coloring rule (by index) for the current session. Wireshark reserves 10 slots for these coloring rules. Requires a GUI.

Arguments

row

The index (1-10) of the desired color in the temporary coloring rules list. The default foreground is black and the default backgrounds are listed below.

Table 6. Default background colors

<table>
<thead>
<tr>
<th>Index</th>
<th>RGB (hex)</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ffc0c0</td>
<td>pink 1</td>
</tr>
<tr>
<td>2</td>
<td>ffc0ff</td>
<td>pink 2</td>
</tr>
<tr>
<td>3</td>
<td>e0c0e0</td>
<td>purple 1</td>
</tr>
<tr>
<td>4</td>
<td>c0c0ff</td>
<td>purple 2</td>
</tr>
<tr>
<td>5</td>
<td>c0e0e0</td>
<td>green 1</td>
</tr>
<tr>
<td>6</td>
<td>c0ffff</td>
<td>green 2</td>
</tr>
<tr>
<td>7</td>
<td>c0fffc0</td>
<td>green 3</td>
</tr>
<tr>
<td>8</td>
<td>ffffc0</td>
<td>yellow 1</td>
</tr>
<tr>
<td>9</td>
<td>e0e0c0</td>
<td>yellow 2</td>
</tr>
<tr>
<td>10</td>
<td>e0e0e0</td>
<td>gray</td>
</tr>
</tbody>
</table>
The color list can be set from the command line using two unofficial preferences: `gui.colorized_frame.bg` and `gui.colorized_frame.fg`, which require 10 hex RGB codes (6 hex digits each), e.g.

```
wireshark -o
gui.colorized_frame.bg:${RGB0},${RGB1},${RGB2},${RGB3},${RGB4},${RGB5},${RGB6},${RGB7},
_${RGB8},${RGB9}
```

For example, this command yields the same results as the table above (and with all foregrounds set to black):

```
wireshark -o
gui.colorized_frame.bg:ffc0c0,ffc0ff,e0c0e0,c0c0ff,c0fffc0,fff0c0,e00000 -o
gui.colorized_frame.fg:000000,000000,000000,000000,000000,000000,000000,000000,000000
```

text

The **display filter** for selecting packets to be colorized.

**apply_filter()**

Apply the filter in the main filter box. Requires a GUI.

**WARNING**

Avoid calling this from within a dissector function or else an infinite loop can occur if it causes the dissector to be called again. This function is best used in a button callback (from a dialog or text window) or menu callback.

**reload()**

Reload the current capture file. Deprecated. Use `reload_packets()` instead.

**reload_packets()**

Reload the current capture file. Requires a GUI.

**WARNING**

Avoid calling this from within a dissector function or else an infinite loop can occur if it causes the dissector to be called again. This function is best used in a button callback (from a dialog or text window) or menu callback.

**redissect_packets()**

Redissect all packets in the current capture file. Requires a GUI.
WARNING
Avoid calling this from within a dissector function or else an infinite loop can occur if it causes the dissector to be called again. This function is best used in a button callback (from a dialog or text window) or menu callback.

reload_lua_plugins()

Reload all Lua plugins.

browser_open_url(url)

Opens an URL in a web browser. Requires a GUI.

WARNING
Do not pass an untrusted URL to this function.

Do not pass an untrusted URL to this function.

It will be passed to the system’s URL handler, which might execute malicious code, switch on your Bluetooth-connected foghorn, or any of a number of unexpected or harmful things.

Arguments
url
The url.

Arguments
filename
The file name.

Functions For New Protocols And Dissectors

The classes and functions in this chapter allow Lua scripts to create new protocols for Wireshark. Proto protocol objects can have Pref preferences, ProtoField fields for filterable values that can be displayed in a details view tree, functions for dissecting the new protocol, and so on.
The dissection function can be hooked into existing protocol tables through `DissectorTable` so that the new protocol dissector function gets called by that protocol, and the new dissector can itself call on other, already existing protocol dissectors by retrieving and calling the `Dissector` object. A `Proto` dissector can also be used as a post-dissector, at the end of every frame's dissection, or as a heuristic dissector.

**Dissector**

A reference to a dissector, used to call a dissector against a packet or a part of it.

**Dissector:get(name)**

Obtains a dissector reference by name.

**Arguments**

- **name**
  
  The name of the dissector.

**Returns**

The `Dissector` reference if found, otherwise `nil`.

**Dissector:list()**

Gets a Lua array table of all registered Dissector names.

*Note: This is an expensive operation, and should only be used for troubleshooting.*

**Returns**

The array table of registered dissector names.

**dissector:call(tvb, pinfo, tree)**

Calls a dissector against a given packet (or part of it).

**Arguments**

- **tvb**
  
  The buffer to dissect.

- **pinfo**
  
  The packet info.

- **tree**
  
  The tree on which to add the protocol items.
Returns

Number of bytes dissected. Note that some dissectors always return number of bytes in incoming buffer, so be aware.

dissector:__call(tvb, pinfo, tree)

Calls a dissector against a given packet (or part of it).

Arguments

tvb
  The buffer to dissect.

pinfo
  The packet info.

tree
  The tree on which to add the protocol items.

dissector:__tostring()

Gets the Dissector's description.

Returns

A string of the Dissector's description.

DissectorTable

A table of subdissectors of a particular protocol (e.g. TCP subdissectors like http, smtp, sip are added to table "tcp.port").

Useful to add more dissectors to a table so that they appear in the “Decode As...” dialog.

DissectorTable.new(tablename, [uiname], [type], [base], [proto])

Creates a new DissectorTable for your dissector's use.

Arguments

tablename
  The short name of the table. Use lower-case alphanumeric, dot, and/or underscores (e.g., "ansi_map.tele_id" or "udp.port").

uiname (optional)
  The name of the table in the user interface. Defaults to the name given in tablename, but can be
any string.

**type (optional)**

One of `ftypes.UINT8`, `ftypes.UINT16`, `ftypes.UINT24`, `ftypes.UINT32`, `ftypes.STRING`, `ftypes.NONE`, or `ftypes.GUID`. Defaults to `ftypes.UINT32`.

**base (optional)**

One of `base.NONE`, `base.DEC`, `base.HEX`, `base.OCT`, `base.DEC_HEX` or `base.HEX_DEC`. Defaults to `base.DEC`.

**proto (optional)**

The `Proto` object that uses this dissector table.

**Returns**

The newly created DissectorTable.

`DissectorTable.heuristic_new(tablename, [uiname], proto)`

Creates a new heuristic `DissectorTable` for your dissector's use. Returns true if table was created successfully.

Since: 4.2.0

**Arguments**

**tablename**

The short name of the table. Use lower-case alphanumeric, dot, and/or underscores.

**uiname (optional)**

The name of the table in the user interface. Defaults to the name given in `tablename`, but can be any string.

**proto**

The `Proto` object that uses this dissector table.

`DissectorTable.list()`

Gets a Lua array table of all DissectorTable names - i.e., the string names you can use for the first argument to DissectorTable.get().

Note: This is an expensive operation, and should only be used for troubleshooting.

**Returns**

The array table of registered DissectorTable names.
**DissectorTable.heuristic_list()**

Gets a Lua array table of all heuristic list names - i.e., the string names you can use for the first argument in Proto:register_heuristic().

Note: This is an expensive operation, and should only be used for troubleshooting.

**Returns**

The array table of registered heuristic list names

**DissectorTable.try_heuristics(listname, tvb, pinfo, tree)**

Try all the dissectors in a given heuristic dissector table.

**Arguments**

listname

The name of the heuristic dissector.

tvb

The buffer to dissect.

pinfo

The packet info.

tree

The tree on which to add the protocol items.

**Returns**

True if the packet was recognized by the sub-dissector (stop dissection here).

**DissectorTable.get(tablename)**

Obtain a reference to an existing dissector table.

**Arguments**

tablename

The short name of the table.

**Returns**

The **DissectorTable** reference if found, otherwise **nil**.
dissectortable:add(pattern, dissector)

Add a Proto with a dissector function or a Dissector object to the dissector table.

Arguments

pattern
  The pattern to match (either an integer, a integer range or a string depending on the table's type).

dissector
  The dissector to add (either a Proto or a Dissector).

dissectortable:set(pattern, dissector)

Clear all existing dissectors from a table and add a new dissector or a range of new dissectors.

Arguments

pattern
  The pattern to match (either an integer, a integer range or a string depending on the table's type).

dissector
  The dissector to add (either a Proto or a Dissector).

dissectortable:remove(pattern, dissector)

Remove a dissector or a range of dissectors from a table.

Arguments

pattern
  The pattern to match (either an integer, a integer range or a string depending on the table's type).

dissector
  The dissector to remove (either a Proto or a Dissector).

dissectortable:remove_all(dissector)

Remove all dissectors from a table.

Arguments

dissector
  The dissector to remove (either a Proto or a Dissector).
dissectortable:try(pattern, tvb, pinfo, tree)

Try to call a dissector from a table.

**Arguments**

**pattern**

The pattern to be matched (either an integer or a string depending on the table’s type).

**tvb**

The Tvb to dissect.

**pinfo**

The packet’s Pinfo.

**tree**

The TreeItem on which to add the protocol items.

**Returns**

Number of bytes dissected. Note that some dissectors always return number of bytes in incoming buffer, so be aware.

dissectortable:get_dissector(pattern)

Try to obtain a dissector from a table.

**Arguments**

**pattern**

The pattern to be matched (either an integer or a string depending on the table’s type).

**Returns**

The Dissector handle if found, otherwise nil

dissectortable:add_for_decode_as(proto)

Add the given Proto to the “Decode as...” list for this DissectorTable. The passed-in Proto object’s dissector() function is used for dissecting.

**Arguments**

**proto**

The Proto to add.
dissectortable:__tostring()

Gets some debug information about the DissectorTable.

**Returns**

A string of debug information about the DissectorTable.

**Pref**

A preference of a Proto.

**Pref.bool(label, default, description)**

Creates a boolean preference to be added to a Proto.prefs Lua table.

**Example**

```lua
-- create a Boolean preference named "bar" for Foo Protocol
-- (assuming Foo doesn't already have a preference named "bar")
proto_foo.prefs.bar = Pref.bool( "Bar", true, "Baz and all the rest" )
```

**Arguments**

**label**

The Label (text in the right side of the preference input) for this preference.

**default**

The default value for this preference.

**description**

A description of this preference.

**Pref.uint(label, default, description)**

Creates an (unsigned) integer preference to be added to a Proto.prefs Lua table.

**Arguments**

**label**

The Label (text in the right side of the preference input) for this preference.

**default**

The default value for this preference.
description
  A description of what this preference is.

Pref.string(label, default, description)

Creates a string preference to be added to a Proto.prefs Lua table.

Arguments

label
  The Label (text in the right side of the preference input) for this preference.

default
  The default value for this preference.

description
  A description of what this preference is.

Pref.enum(label, default, description, enum, radio)

Creates an enum preference to be added to a Proto.prefs Lua table.

Example:
local OUTPUT_OFF = 0
local OUTPUT_DEBUG = 1
local OUTPUT_INFO = 2
local OUTPUT_WARN = 3
local OUTPUT_ERROR = 4

local output_tab = {
    {1, "Off", OUTPUT_OFF},
    {2, "Debug", OUTPUT_DEBUG},
    {3, "Information", OUTPUT_INFO},
    {4, "Warning", OUTPUT_WARN},
    {5, "Error", OUTPUT_ERROR},
}

-- Create enum preference that shows as Combo Box under
-- Foo Protocol's preferences
proto_foo.prefs.outputlevel = Pref.enum(
    "Output Level",                 -- label
    OUTPUT_INFO,                    -- default value
    "Verbosity of log output",      -- description
    output_tab,                     -- enum table
    false                           -- show as combo box
)

-- Then, we can query the value of the selected preference.
-- This line prints "Output Level: 3" assuming the selected
-- output level is _INFO.
dump( "Output Level: " .. proto_foo.prefs.outputlevel )

Arguments

label
The Label (text in the right side of the preference input) for this preference.

default
The default value for this preference.

description
A description of what this preference is.

enum
An enum Lua table.

radio
Radio button (true) or Combobox (false).
Pref.range(label, default, description, max)

Creates a range (numeric text entry) preference to be added to a Proto.prefs Lua table.

Arguments

label
The Label (text in the right side of the preference input) for this preference.

default
The default value for this preference, e.g., "53", "10-30", or "10-30,53,55,100-120".

description
A description of what this preference is.

max
The maximum value.

Pref.statictext(label, description)

Creates a static text string to be added to a Proto.prefs Lua table.

Arguments

label
The static text.

description
The static text description.

Prefs

The table of preferences of a protocol.

prefs:__newindex(name, pref)

Creates a new preference.

Arguments

name
The abbreviation of this preference.

default
A valid but still unassigned Pref object.
Errors

- Unknown Pref type

`prefs:_index(name)`

Get the value of a preference setting.

Example

```lua
-- print the value of Foo's preference named "bar"
debug( "bar = " .. proto_foo.prefs.bar )
```

Arguments

`name`

The abbreviation of this preference.

Returns

The current value of the preference.

Errors

- Unknown Pref type

Proto

A new protocol in Wireshark. Protocols have several uses. The main one is to dissect a protocol, but they can also be dummies used to register preferences for other purposes.

`Proto.new(name, description)`

Creates a new `Proto` object.

Arguments

`name`

The name of the protocol.

`description`

A Long Text description of the protocol (usually lowercase).

Returns

The newly created `Proto` object.
proto:__call(name, description)

Creates a Proto object.

Arguments

name
The name of the protocol.

description
A Long Text description of the protocol (usually lowercase).

Returns

The new Proto object.

proto:register_heuristic(listname, func)

Registers a heuristic dissector function for this Proto protocol, for the given heuristic list name.

When later called, the passed-in function will be given:

1. A Tvb object
2. A Pinfo object
3. A TreeItem object

The function must return true if the payload is for it, else false.

The function should perform as much verification as possible to ensure the payload is for it, and dissect the packet (including setting TreeItem info and such) only if the payload is for it, before returning true or false.

Since version 1.99.1, this function also accepts a Dissector object as the second argument, to allow re-using the same Lua code as the function proto.dissector(...). In this case, the Dissector must return a Lua number of the number of bytes consumed/parsed: if 0 is returned, it will be treated the same as a false return for the heuristic; if a positive or negative number is returned, then the it will be treated the same as a true return for the heuristic, meaning the packet is for this protocol and no other heuristic will be tried.

Arguments

listname
The heuristic list name this function is a heuristic for (e.g., "udp" or "infiniband.payload").

func
A Lua function that will be invoked for heuristic dissection.
proto.dissector
Mode: Retrieve or assign.
The protocol’s dissector, a function you define.
When later called, the function will be given:

1. A Tvb object
2. A Pinfo object
3. A TreeItem object

proto.prefs
Mode: Retrieve only.
The preferences of this dissector.

proto.prefs_changed
Mode: Assign only.
The preferences changed routine of this dissector, a Lua function you define.
The function is called when the protocol's preferences are changed. It is passed no arguments.

proto.init
Mode: Assign only.
The init routine of this dissector, a function you define.
The init function is called when the a new capture file is opened or when the open capture file is closed. It is passed no arguments.

proto.name
Mode: Retrieve only.
The name given to this dissector.

proto.description
Mode: Retrieve only.
The description given to this dissector.
proto.fields

Mode: Retrieve or assign.

The ProtoField's Lua table of this dissector.

proto.experts

Mode: Retrieve or assign.

The expert info Lua table of this Proto.

ProtoExpert

A Protocol expert info field, to be used when adding items to the dissection tree.

ProtoExpert.new(abbr, text, group, severity)

Creates a new ProtoExpert object to be used for a protocol's expert information notices.

Arguments

abbr

Filter name of the expert info field (the string that is used in filters).

text

The default text of the expert field.

group

Expert group type: one of: expert.group.CHECKSUM, expert.group.SEQUENCE, expert.group.RESPONSE_CODE, expert.group.REQUEST_CODE, expert.group.UNDECODED, expert.group.REASSEMBLE, expert.group.MALFORMED, expert.group.DEBUG, expert.group.PROTOCOL, expert.group.SECURITY, expert.group.COMMENTS_GROUP, expert.group.DECRYPTION, expert.group.ASSUMPTION, expert.group.DEPRECATED, expert.group.RECEIVE, or expert.groupINTERFACE.

severity

Expert severity type: one of: expert.severity.COMMENT, expert.severity.CHAT, expert.severity.NOTE, expert.severity.WARN, or expert.severity.ERROR.

Returns

The newly created ProtoExpert object.

protoexpert:_tostring()

Returns a string with debugging information about a ProtoExpert object.
ProtoField

A Protocol field (to be used when adding items to the dissection tree).

ProtoField.new(name, abbr, type, [valuestring], [base], [mask], [description])

Creates a new ProtoField object to be used for a protocol field.

Arguments

name
  Actual name of the field (the string that appears in the tree).

abbr
  Filter name of the field (the string that is used in filters).

type
  Field Type: one of: ftypes.BOOLEAN, ftypes.CHAR, ftypes.UINT8, ftypes.UINT16, ftypes.UINT24, ftypes.UINT32, ftypes.UINT64, ftypes.INT8, ftypes.INT16, ftypes.INT24, ftypes.INT32, ftypes.INT64, ftypes.FLOAT, ftypes.DOUBLE, ftypes.ABSOLUTE_TIME, ftypes.RELATIVE_TIME, ftypes.STRING, ftypes.STRINGZ, ftypes.UINT_STRING, ftypes.ETHER, ftypes.BYTES, ftypes.UINT_BYTES, ftypes.IPv4, ftypes.IPv6, ftypes.IPXNET, ftypes.FRAMENUM, ftypes.PCRE, ftypes.GUID, ftypes.OID, ftypes.PROTOCOL, ftypes.REL_OID, ftypes.SYSTEM_ID, ftypes.EUI64 or ftypes.NONE.

valuestring (optional)
  A table containing the text that corresponds to the values, or a table containing tables of range string values that corresponds to the values ({min, max, "string")} if the base is base.RANGE_STRING, or a table containing unit name for the values if base is base.UNIT_STRING, or one of frametype.NONE, frametype.REQUEST, frametype.RESPONSE, frametype.ACK or frametype.DUP_ACK if field type is ftypes.FRAMENUM.

base (optional)
  The representation, one of: base.NONE, base.DEC, base.HEX, base.OCT, base.DEC_HEX, base.HEX_DEC, base.UNIT_STRING or base.RANGE_STRING.

mask (optional)
  The bitmask to be used.

description (optional)
  The description of the field.

Returns

The newly created ProtoField object.
ProtoField.char(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of an 8-bit ASCII character.

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.NONE, base.HEX, base.OCT or base.RANGE_STRING.

valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, "string"}) if the base is base.RANGE_STRING.

mask (optional)
Integer mask of this field.

description (optional)
Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.uint8(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of an unsigned 8-bit integer (i.e., a byte).

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base.HEX or base.OCT, base.DEC_HEX, base.HEX_DEC, base.UNIT_STRING or base.RANGE_STRING.
valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, “string”}) if the base is `base.RANGE_STRING`, or a table containing the unit name for the values if base is `base.UNIT_STRING`.

mask (optional)
Integer, String or UInt64 mask of this field.

description (optional)
Description of the field.

Returns
A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

`ProtoField.uint16(abbr, [name], [base], [valuestring], [mask], [description])`
Creates a `ProtoField` of an unsigned 16-bit integer.

Arguments
abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of `base.DEC`, `base.HEX`, `base.OCT`, `base.DEC_HEX`, `base.HEX_DEC`, `base.UNIT_STRING` or `base.RANGE_STRING`.

valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, “string”}) if the base is `base.RANGE_STRING`, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)
Integer, String or UInt64 mask of this field.

description (optional)
Description of the field.

Returns
A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.
ProtoField.uint24(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of an unsigned 24-bit integer.

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base.HEX, base.OCT, base.DEC_HEX, base.HEX_DEC, base.UNIT_STRING, or base.RANGE_STRING.

valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, "string"}) if the base is base.RANGE_STRING, or a table containing the unit name for the values if base is base.UNIT_STRING.

mask (optional)
Integer, String or UInt64 mask of this field.

description (optional)
Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.uint32(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of an unsigned 32-bit integer.

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base.HEX, base.OCT, base.DEC_HEX, base.HEX_DEC, base.UNIT_STRING, or base.RANGE_STRING.
valuestring (optional)

A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, “string”)} if the base is base.RANGE_STRING, or a table containing the unit name for the values if base is base.UNIT_STRING.

mask (optional)

Integer, String or UInt64 mask of this field.

description (optional)

Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.uint64(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of an unsigned 64-bit integer.

Arguments

abbr

Abbreviated name of the field (the string used in filters).

name (optional)

Actual name of the field (the string that appears in the tree).

base (optional)

One of base.DEC, base.HEX, base.OCT, base.DEC_HEX, base.HEX_DEC, base.UNIT_STRING, or base.RANGE_STRING.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, “string”)} if the base is base.RANGE_STRING, or a table containing the unit name for the values if base is base.UNIT_STRING.

mask (optional)

Integer, String or UInt64 mask of this field.

description (optional)

Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.
ProtoField.int8(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of a signed 8-bit integer (i.e., a byte).

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base.UNIT_STRING, or base.RANGE_STRING.

valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values (min, max, "string") if the base is base.RANGE_STRING, or a table containing unit name for the values if base is base.UNIT_STRING.

mask (optional)
Integer, String or UInt64 mask of this field.

description (optional)
Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.int16(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of a signed 16-bit integer.

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base.UNIT_STRING, or base.RANGE_STRING.
valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, "string"}) if the base is base.RANGE_STRING, or a table containing unit name for the values if base is base.UNIT_STRING.

description (optional)
Description of the field.

Returns
A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.int24(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of a signed 24-bit integer.

Arguments
abbr
Abbreviated name of the field (the string used in filters).

description (optional)
Description of the field.

Returns
A ProtoField object to be added to a table set to the Proto.fields attribute.
ProtoField.int32(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of a signed 32-bit integer.

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base_UNIT_STRING, or base.RANGE_STRING.

valuestring (optional)
A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, "string"}) if the base is base.RANGE_STRING, or a table containing unit name for the values if base is base_UNIT_STRING.

mask (optional)
Integer, String or UInt64 mask of this field.

description (optional)
Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.int64(abbr, [name], [base], [valuestring], [mask], [description])

Creates a ProtoField of a signed 64-bit integer.

Arguments

abbr
Abbreviated name of the field (the string used in filters).

name (optional)
Actual name of the field (the string that appears in the tree).

base (optional)
One of base.DEC, base_UNIT_STRING, or base.RANGE_STRING.
valuestring (optional)

A table containing the text that corresponds to the values, or a table containing tables of range string values that correspond to the values ({min, max, "string"}) if the base is base.RANGE_STRING, or a table containing unit name for the values if base is base.UNIT_STRING.

mask (optional)

Integer, String or UInt64 mask of this field.

description (optional)

Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.framenum(abbr, [name], [base], [frametype], [mask], [description])

Creates a ProtoField for a frame number (for hyperlinks between frames).

Arguments

abbr

Abbreviated name of the field (the string used in filters).

name (optional)

Actual name of the field (the string that appears in the tree).

base (optional)

Only base.NONE is supported for framenum.

frametype (optional)

One of frametype.NONE, frametype.REQUEST, frametype.RESPONSE, frametype.ACK or frametype.DUP_ACK.

mask (optional)

Integer, String or UInt64 mask of this field, which must be 0 for framenum.

description (optional)

Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.
ProtoField.bool(abbr, [name], [display], [valuestring], [mask], [description])

Creates a ProtoField for a boolean true/false value.

Arguments

abbr
  Abbreviated name of the field (the string used in filters).

name (optional)
  Actual name of the field (the string that appears in the tree).

display (optional)
  How wide the parent bitfield is (base.NONE is used for NULL-value).

valuestring (optional)
  A table containing the text that corresponds to the values.

mask (optional)
  Integer, String or UInt64 mask of this field.

description (optional)
  Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.absolute_time(abbr, [name], [base], [description])

Creates a ProtoField of a time_t structure value.

Arguments

abbr
  Abbreviated name of the field (the string used in filters).

name (optional)
  Actual name of the field (the string that appears in the tree).

base (optional)
  One of base.LOCAL, base.UTC or base.DOY.UTC.

description (optional)
  Description of the field.
Returns

A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

`ProtoField.relative_time(abbr, [name], [description])`

Creates a `ProtoField` of a time_t structure value.

Arguments

`abbr`
Abbreviated name of the field (the string used in filters).

`name (optional)`
Actual name of the field (the string that appears in the tree).

`description (optional)`
Description of the field.

Returns

A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

`ProtoField.float(abbr, [name], [valuestring], [description])`

Creates a `ProtoField` of a floating point number (4 bytes).

Arguments

`abbr`
Abbreviated name of the field (the string used in filters).

`name (optional)`
Actual name of the field (the string that appears in the tree).

`valuestring (optional)`
A table containing unit name for the values.

`description (optional)`
Description of the field.

Returns

A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.
ProtoField.double(abbr, [name], [valuestring], [description])

Creates a `ProtoField` of a double-precision floating point (8 bytes).

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).

**valuestring (optional)**
A table containing unit name for the values.

**description (optional)**
Description of the field.

**Returns**
A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

ProtoField.string(abbr, [name], [display], [description])

Creates a `ProtoField` of a string value.

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).

**display (optional)**
One of `base.ASCII` or `base.UNICODE`.

**description (optional)**
Description of the field.

**Returns**
A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

ProtoField.stringz(abbr, [name], [display], [description])

Creates a `ProtoField` of a zero-terminated string value.
Arguments

abbr
   Abbreviated name of the field (the string used in filters).

name (optional)
   Actual name of the field (the string that appears in the tree).

display (optional)
   One of base.ASCII or base.UNICODE.

description (optional)
   Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.bytes(abbr, [name], [display], [description])

Creates a ProtoField for an arbitrary number of bytes.

Arguments

abbr
   Abbreviated name of the field (the string used in filters).

name (optional)
   Actual name of the field (the string that appears in the tree).

display (optional)
   One of base.NONE, base.DOT, base.DASH, base.COLON or base.SPACE.

description (optional)
   Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.ubyte(abbr, [name], [display], [description])

Creates a ProtoField for an arbitrary number of unsigned bytes.

Arguments
**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).

**display (optional)**
One of `base.NONE`, `base.DOT`, `base.DASH`, `base.COLON` or `base.SPACE`.

**description (optional)**
Description of the field.

**Returns**
A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

`ProtoField.none(abbr, [name], [description])`
Creates a `ProtoField` of an unstructured type.

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).

**description (optional)**
Description of the field.

**Returns**
A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

`ProtoField.ipv4(abbr, [name], [description])`
Creates a `ProtoField` of an IPv4 address (4 bytes).

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).
description (optional)
   Description of the field.

Returns
A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.ipv6(abbr, [name], [description])
Creates a ProtoField of an IPv6 address (16 bytes).

Arguments
abbr
   Abbreviated name of the field (the string used in filters).

name (optional)
   Actual name of the field (the string that appears in the tree).

description (optional)
   Description of the field.

Returns
A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.ether(abbr, [name], [description])
Creates a ProtoField of an Ethernet address (6 bytes).

Arguments
abbr
   Abbreviated name of the field (the string used in filters).

name (optional)
   Actual name of the field (the string that appears in the tree).

description (optional)
   Description of the field.

Returns
A ProtoField object to be added to a table set to the Proto.fields attribute.
ProtoField.guid(abbr, [name], [description])

Creates a ProtoField for a Globally Unique IDentifier (GUID).

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).

**description (optional)**
Description of the field.

**Returns**

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.oid(abbr, [name], [description])

Creates a ProtoField for an ASN.1 Organizational IDentified (OID).

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).

**name (optional)**
Actual name of the field (the string that appears in the tree).

**description (optional)**
Description of the field.

**Returns**

A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.protocol(abbr, [name], [description])

Creates a ProtoField for a sub-protocol.

**Arguments**

**abbr**
Abbreviated name of the field (the string used in filters).
name (optional)
   Actual name of the field (the string that appears in the tree).

description (optional)
   Description of the field.

Returns
   A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.rel_oid(abbr, [name], [description])

Creates a ProtoField for an ASN.1 Relative-OID.

Arguments

abbr
   Abbreviated name of the field (the string used in filters).

name (optional)
   Actual name of the field (the string that appears in the tree).

description (optional)
   Description of the field.

Returns
   A ProtoField object to be added to a table set to the Proto.fields attribute.

ProtoField.systemid(abbr, [name], [description])

Creates a ProtoField for an OSI System ID.

Arguments

abbr
   Abbreviated name of the field (the string used in filters).

name (optional)
   Actual name of the field (the string that appears in the tree).

description (optional)
   Description of the field.

Returns
   A ProtoField object to be added to a table set to the Proto.fields attribute.
ProtoField.eui64(abbr, [name], [description])

Creates a ProtoField for an EUI64.

Arguments

abbr
    Abbreviated name of the field (the string used in filters).

name (optional)
    Actual name of the field (the string that appears in the tree).

description (optional)
    Description of the field.

Returns

A ProtoField object to be added to a table set to the Proto.fields attribute.

protofield:_tostring()

Returns a string with info about a protofield (for debugging purposes).

protofield.type

Mode: Retrieve only.

The type of the field.

Since: 4.3.0

protofield.abbr

Mode: Retrieve only.

The abbreviated name of the field.

Since: 4.3.0

protofield.name

Mode: Retrieve only.

The actual name of the field.

Since: 4.3.0
**protofield.base**

Mode: Retrieve only.

The base of the field.

Since: 4.3.0

**protofield.valuestring**

Mode: Retrieve only.

The valuestring of the field.

Since: 4.3.0

**protofield.mask**

Mode: Retrieve only.

The mask of the field.

Since: 4.3.0

**protofield.description**

Mode: Retrieve only.

The description of this field.

Since: 4.3.0

### Global Functions

**register_postdissector(proto, [allfields])**

Make a Proto protocol (with a dissector function) a post-dissector. It will be called for every frame after dissection.

**Arguments**

**proto**

The protocol to be used as post-dissector.

**allfields (optional)**

Whether to generate all fields. Note: This impacts performance (default=false).
dissect_tcp_pdus(tvb, tree, min_header_size, get_len_func, dissect_func, [desegment])

Make the TCP-layer invoke the given Lua dissection function for each PDU in the TCP segment, of the length returned by the given get_len_func function.

This function is useful for protocols that run over TCP and that are either a fixed length always, or have a minimum size and have a length field encoded within that minimum portion that identifies their full length. For such protocols, their protocol dissector function can invoke this dissect_tcp_pdus() function to make it easier to handle dissecting their protocol’s messages (i.e., their protocol data unit (PDU)). This function should not be used for protocols whose PDU length cannot be determined from a fixed minimum portion, such as HTTP or Telnet.

Arguments

tvb
The Tvb buffer to dissect PDUs from.

tree
The Tvb buffer to dissect PDUs from.

min_header_size
The number of bytes in the fixed-length part of the PDU.

get_len_func
A Lua function that will be called for each PDU, to determine the full length of the PDU. The called function will be given (1) the Tvb object of the whole Tvb (possibly reassembled), (2) the Pinfo object, and (3) an offset number of the index of the first byte of the PDU (i.e., its first header byte). The Lua function must return a Lua number of the full length of the PDU.

dissect_func
A Lua function that will be called for each PDU, to dissect the PDU. The called function will be given (1) the Tvb object of the PDU’s Tvb (possibly reassembled), (2) the Pinfo object, and (3) the TreeItem object. The Lua function must return a Lua number of the number of bytes read/handled, which would typically be the Tvb:len().

desegment (optional)
Whether to reassemble PDUs crossing TCP segment boundaries or not. (default=true)

Obtaining Dissection Data

Field
A Field extractor to obtain field values. A Field object can only be created outside of the callback functions of dissectors, post-dissectors, heuristic-dissectors, and taps.
Once created, it is used **inside** the callback functions, to generate a `FieldInfo` object.

### Field.new(fieldname)

Create a Field extractor.

**Arguments**

**fieldname**

The filter name of the field (e.g. ip.addr)

**Returns**

The field extractor

**Errors**

- A Field extractor must be defined before Taps or Dissectors get called

### Field.list()

Gets a Lua array table of all registered field filter names.

**NOTE**

This is an expensive operation, and should only be used for troubleshooting.

**Returns**

The array table of field filter names

### field:__call()

Obtain all values (see `FieldInfo`) for this field.

**Returns**

All the values of this field

**Errors**

- Fields cannot be used outside dissectors or taps

### field:__tostring()

Obtain a string with the field filter name.

### field.name

Mode: Retrieve only.
The filter name of this field, or nil.

field.display
Mode: Retrieve only.

The full display name of this field, or nil.

field.type
Mode: Retrieve only.

The ftype of this field, or nil.

FieldInfo
An extracted Field from dissected packet data. A FieldInfo object can only be used within the callback functions of dissectors, post-dissectors, heuristic-dissectors, and taps.

A FieldInfo can be called on either existing Wireshark fields by using either Field.new() or Field() before-hand, or it can be called on new fields created by Lua from a ProtoField.

fieldinfo:_len()
Obtain the Length of the field

fieldinfo:_unm()
Obtain the Offset of the field

fieldinfo:_call()
Obtain the Value of the field.

Previous to 1.11.4, this function retrieved the value for most field types, but for ftypes.UINT_BYTES it retrieved the ByteArray of the field's entire TvbRange. In other words, it returned a ByteArray that included the leading length byte(s), instead of just the value bytes. That was a bug, and has been changed in 1.11.4. Furthermore, it retrieved an ftypes.GUID as a ByteArray, which is also incorrect.

If you wish to still get a ByteArray of the TvbRange, use fieldinfo.range to get the TvbRange, and then use tvbrange:bytes() to convert it to a ByteArray.

fieldinfo:_tostring()
The string representation of the field.
**fieldinfo:_eq()**
Checks whether lhs is within rhs.

**fieldinfo:_le()**
Checks whether the end byte of lhs is before the end of rhs.

**Errors**
- Data source must be the same for both fields

**fieldinfo:_lt()**
Checks whether the end byte of lhs is before the beginning of rhs.

**Errors**
- Data source must be the same for both fields

**fieldinfo.len**
Mode: Retrieve only.
The length of this field.

**fieldinfo.offset**
Mode: Retrieve only.
The offset of this field.

**fieldinfo.value**
Mode: Retrieve only.
The value of this field.

**fieldinfo.label**
Mode: Retrieve only.
The string representing this field.

**fieldinfo.display**
Mode: Retrieve only.
The string display of this field as seen in GUI.
**fieldinfo.type**

Mode: Retrieve only.

The internal field type, a number which matches one of the ftype values.

**fieldinfo.source**

Mode: Retrieve only.

The source Tvb object the FieldInfo is derived from, or nil if there is none.

**fieldinfo.range**

Mode: Retrieve only.

The TvbRange covering the bytes of this field in a Tvb or nil if there is none.

**fieldinfo.generated**

Mode: Retrieve only.

Whether this field was marked as generated (boolean).

**fieldinfo.hidden**

Mode: Retrieve only.

Whether this field was marked as hidden (boolean).

**fieldinfo.is_url**

Mode: Retrieve only.

Whether this field was marked as being a URL (boolean).

**fieldinfo.little_endian**

Mode: Retrieve only.

Whether this field is little-endian encoded (boolean).

**fieldinfo.big_endian**

Mode: Retrieve only.

Whether this field is big-endian encoded (boolean).
fieldinfo.name
Mode: Retrieve only.
The filter name of this field.

Global Functions

all_field_infos()
Obtain all fields from the current tree. Note this only gets whatever fields the underlying dissectors have filled in for this packet at this time - there may be fields applicable to the packet that simply aren't being filled in because at this time they're not needed for anything. This function only gets what the C-side code has currently populated, not the full list.

Errors
- Cannot be called outside a listener or dissector

Obtaining Packet Information

Address
Represents an address.

Address.ip(hostname)
Creates an Address Object representing an IPv4 address.

Arguments
hostname
The address or name of the IP host.

Returns
The Address object.

Address.ipv6(hostname)
Creates an Address Object representing an IPv6 address.

Arguments
hostname
The address or name of the IP host.
Returns
The Address object

Address.ether(eth)
Creates an Address Object representing an Ethernet address.

Arguments
eth
The Ethernet address.

Returns
The Address object.

address:.tostring()

Returns
The string representing the address.

address:.eq()
Compares two Addresses.

address:.le()
Compares two Addresses.

address:.lt()
Compares two Addresses.

Column
A Column in the packet list.

column:.tostring()

Returns
The column’s string text (in parenthesis if not available).

column:.clear()
Clears a Column.
column:set(text)
Sets the text of a Column.

Arguments

  text
  The text to which to set the Column.

column:append(text)
Appends text to a Column.

Arguments

  text
  The text to append to the Column.

column:prepend(text)
Prepends text to a Column.

Arguments

  text
  The text to prepend to the Column.

column:fence()
Sets Column text fence, to prevent overwriting.

column:clear_fence()
Clear Column text fence.

Columns
The Columns of the packet list.

columns:_tostring()

Returns
The string "Columns". This has no real use aside from debugging.

columns:_newindex(column, text)
Sets the text of a specific column. Some columns cannot be modified, and no error is raised if
attempted. The columns that are known to allow modification are "info" and "protocol".

Arguments

column

The name of the column to set. Valid values are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Frame number</td>
</tr>
<tr>
<td>abs_time</td>
<td>Absolute timestamp</td>
</tr>
<tr>
<td>utc_time</td>
<td>UTC timestamp</td>
</tr>
<tr>
<td>cls_time</td>
<td>CLS timestamp</td>
</tr>
<tr>
<td>rel_time</td>
<td>Relative timestamp</td>
</tr>
<tr>
<td>date</td>
<td>Absolute date and time</td>
</tr>
<tr>
<td>date_doy</td>
<td>Absolute year, day of year, and time</td>
</tr>
<tr>
<td>utc_date</td>
<td>UTC date and time</td>
</tr>
<tr>
<td>utc_date_doy</td>
<td>UTC year, day of year, and time</td>
</tr>
<tr>
<td>delta_time</td>
<td>Delta time from previous packet</td>
</tr>
<tr>
<td>delta_time_displayed</td>
<td>Delta time from previous displayed packet</td>
</tr>
<tr>
<td>src</td>
<td>Source address</td>
</tr>
<tr>
<td>src_res</td>
<td>Resolved source address</td>
</tr>
<tr>
<td>src_unres</td>
<td>Numeric source address</td>
</tr>
<tr>
<td>dl_src</td>
<td>Source data link address</td>
</tr>
<tr>
<td>dl_src_res</td>
<td>Resolved source data link address</td>
</tr>
<tr>
<td>dl_src_unres</td>
<td>Numeric source data link address</td>
</tr>
<tr>
<td>net_src</td>
<td>Source network address</td>
</tr>
<tr>
<td>net_src_res</td>
<td>Resolved source network address</td>
</tr>
<tr>
<td>net_src_unres</td>
<td>Numeric source network address</td>
</tr>
<tr>
<td>dst</td>
<td>Destination address</td>
</tr>
<tr>
<td>dst_res</td>
<td>Resolve destination address</td>
</tr>
<tr>
<td>dst_unres</td>
<td>Numeric destination address</td>
</tr>
<tr>
<td>dl_dst</td>
<td>Destination data link address</td>
</tr>
<tr>
<td>dl_dst_res</td>
<td>Resolved destination data link address</td>
</tr>
<tr>
<td>dl_dst_unres</td>
<td>Numeric destination data link address</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>net_dst</td>
<td>Destination network address</td>
</tr>
<tr>
<td>net_dst_res</td>
<td>Resolved destination network address</td>
</tr>
<tr>
<td>net_dst_unres</td>
<td>Numeric destination network address</td>
</tr>
<tr>
<td>src_port</td>
<td>Source port</td>
</tr>
<tr>
<td>src_port_res</td>
<td>Resolved source port</td>
</tr>
<tr>
<td>src_port_unres</td>
<td>Numeric source port</td>
</tr>
<tr>
<td>dst_port</td>
<td>Destination port</td>
</tr>
<tr>
<td>dst_port_res</td>
<td>Resolved destination port</td>
</tr>
<tr>
<td>dst_port_unres</td>
<td>Numeric destination port</td>
</tr>
<tr>
<td>protocol</td>
<td>Protocol name</td>
</tr>
<tr>
<td>info</td>
<td>General packet information</td>
</tr>
<tr>
<td>packet_len</td>
<td>Packet length</td>
</tr>
<tr>
<td>cumulative_bytes</td>
<td>Cumulative bytes in the capture</td>
</tr>
<tr>
<td>direction</td>
<td>Packet direction</td>
</tr>
<tr>
<td>vsan</td>
<td>Virtual SAN</td>
</tr>
<tr>
<td>tx_rate</td>
<td>Transmit rate</td>
</tr>
<tr>
<td>rssi</td>
<td>RSSI value</td>
</tr>
<tr>
<td>dce_call</td>
<td>DCE call</td>
</tr>
</tbody>
</table>

**Example**

```python
pinfo.cols[\'info\'] = 'foo bar'
```

— syntactic sugar (equivalent to above) `pinfo.cols.info = 'foo bar'`

**text**

 The text for the column.

**columns:__index()**

Get a specific `Column`.

**NSTime**

NSTime represents a nstime_t. This is an object with seconds and nanoseconds.
NSTime.new([seconds], [nseconds])

Creates a new NSTime object.

Arguments

seconds (optional)
    Seconds.

nseconds (optional)
    Nano seconds.

Returns

The new NSTime object.

nstime:__call([seconds], [nseconds])

Creates a NSTime object.

Arguments

seconds (optional)
    Seconds.

nseconds (optional)
    Nanoseconds.

Returns

The new NSTime object.

nstime:tonumber()

Returns a Lua number of the NSTime representing seconds from epoch.

Returns

The Lua number.

nstime:__tostring()

Returns

The string representing the nstime.
nstime: __add()
Calculates the sum of two NSTimes.

nstime: __sub()
Calculates the diff of two NSTimes.

nstime: __unm()
Calculates the negative NSTime.

nstime: __eq()
Compares two NSTimes.

nstime: __le()
Compares two NSTimes.

nstime: __lt()
Compares two NSTimes.

nstime.secs
Mode: Retrieve or assign.
The NSTime seconds.

nstime.nsecs
Mode: Retrieve or assign.
The NSTime nano seconds.

Pinfo
Packet information.

pinfo.visited
Mode: Retrieve only.
Whether this packet has been already visited.

pinfo.number
Mode: Retrieve only.
The number of this packet in the current file.

\textbf{pinfo.len}

Mode: Retrieve only.

The length of the frame.

\textbf{pinfo.caplen}

Mode: Retrieve only.

The captured length of the frame.

\textbf{pinfo.abs_ts}

Mode: Retrieve only.

When the packet was captured.

\textbf{pinfo.rel_ts}

Mode: Retrieve only.

Number of seconds passed since beginning of capture.

\textbf{pinfo.delta_ts}

Mode: Retrieve only.

Number of seconds passed since the last captured packet.

\textbf{pinfo.delta_dis_ts}

Mode: Retrieve only.

Number of seconds passed since the last displayed packet.

\textbf{pinfo.curr_proto}

Mode: Retrieve only.

Which Protocol are we dissecting.

\textbf{pinfo.can_desegment}

Mode: Retrieve or assign.

Set if this segment could be desegmented.
\textbf{pinfo.desegment\_len}  
Mode: Retrieve or assign.  
Estimated number of additional bytes required for completing the PDU.

\textbf{pinfo.desegment\_offset}  
Mode: Retrieve or assign.  
Offset in the tvbuff at which the dissector will continue processing when next called.

\textbf{pinfo.fragme\_nted}  
Mode: Retrieve only.  
If the protocol is only a fragment.

\textbf{pinfo.in\_error\_pkt}  
Mode: Retrieve or assign.  
If we’re inside an error packet.

\textbf{pinfo.match\_uint}  
Mode: Retrieve only.  
Matched uint for calling subdissector from table.

\textbf{pinfo.match\_string}  
Mode: Retrieve only.  
Matched string for calling subdissector from table.

\textbf{pinfo.port\_type}  
Mode: Retrieve or assign.  
Type of Port of .src\_port and .dst\_port.

\textbf{pinfo.src\_port}  
Mode: Retrieve or assign.  
Source Port of this Packet.
**pinfo.dst_port**
Mode: Retrieve or assign.
Destination Port of this Packet.

**pinfo.dl_src**
Mode: Retrieve or assign.
Data Link Source Address of this Packet.

**pinfo.dl_dst**
Mode: Retrieve or assign.
Data Link Destination Address of this Packet.

**pinfo.net_src**
Mode: Retrieve or assign.
Network Layer Source Address of this Packet.

**pinfo.net_dst**
Mode: Retrieve or assign.
Network Layer Destination Address of this Packet.

**pinfo.src**
Mode: Retrieve or assign.
Source Address of this Packet.

**pinfo.dst**
Mode: Retrieve or assign.
Destination Address of this Packet.

**pinfo.p2p_dir**
Mode: Retrieve or assign.
Direction of this Packet. (incoming / outgoing)
**pinfo.match**

Mode: Retrieve only.

Port/Data we are matching.

**pinfo.columns**

Mode: Retrieve only.

Access to the packet list columns.

**pinfo.cols**

Mode: Retrieve only.

Access to the packet list columns (equivalent to pinfo.columns).

**pinfo.private**

Mode: Retrieve only.

Access to the private table entries.

**pinfo.hi**

Mode: Retrieve or assign.

Higher Address of this Packet.

**pinfo.lo**

Mode: Retrieve only.

Lower Address of this Packet.

**pinfo.conversation**

Mode: Assign only.

Sets the packet conversation to the given Proto object.

**PrivateTable**

PrivateTable represents the pinfo→private_table.

**privatetable:_tostring()**

Gets debugging type information about the private table.
Returns
A string with all keys in the table, mostly for debugging.

Functions For Handling Packet Data

ByteArray

ByteArray.new([hexbytes], [separator])

Creates a new ByteArray object.

Starting in version 1.11.3, if the second argument is a boolean true, then the first argument is treated as a raw Lua string of bytes to use, instead of a hexadecimal string.

Example

```
local empty = ByteArray.new()
local b1 = ByteArray.new("a1 b2 c3 d4")
local b2 = ByteArray.new("112233")
```

Arguments

hexbytes (optional)

A string consisting of hexadecimal bytes like "00 B1 A2" or "1a2b3c4d".

separator (optional)

A string separator between hex bytes/words (default=" "), or if the boolean value true is used, then the first argument is treated as raw binary data

Returns

The new ByteArray object.

bytearray:__concat(first, second)

Concatenate two ByteArrays.

Arguments

first

First array.

second

Second array.
Returns
The new composite ByteArray.

bytearray:__eq(first, second)
Compares two ByteArray values.

Arguments
first
First array.

second
Second array.

bytearray:prepend(prepped)
Prepend a ByteArray to this ByteArray.

Arguments
prepped
ByteArray to be prepended.

bytearray:append(appended)
Append a ByteArray to this ByteArray.

Arguments
appended
ByteArray to be appended.

bytearray:set_size(size)
Sets the size of a ByteArray, either truncating it or filling it with zeros.

Arguments
size
New size of the array.

Errors
• ByteArray size must be non-negative
bytearray:set_index(index, value)

Sets the value of an index of a ByteArray.

Arguments

index
  The position of the byte to be set.

value
  The char value to set [0-255].

bytearray:get_index(index)

Get the value of a byte in a ByteArray.

Arguments

index
  The position of the byte to get.

Returns

The value [0-255] of the byte.

bytearray:le_int([offset], [length])

Read a little endian encoded signed integer in a ByteArray beginning at given offset with given length.

Since: 4.2.0

Arguments

offset (optional)
  The position of the first byte. Default is 0, or the first byte.

length (optional)
  The length of the integer. Default is -1, or the remaining bytes in the ByteArray.

Returns

The value of the little endian encoded signed integer beginning at given offset with given length.

bytearray:le_int64([offset], [length])

Read a little endian encoded 64 bit signed integer in a ByteArray beginning at given offset with
given length.
Since: 4.2.0

Arguments

offset (optional)
The position of the first byte. Default is 0, or the first byte.

length (optional)
The length of the integer. Default is -1, or the remaining bytes in the ByteArray.

Returns
The value of the little endian encoded 64 bit signed integer as a Int64 object beginning at given offset with given length.

bytearray:le_uint([offset], [length])
Read a little endian encoded unsigned integer in a ByteArray beginning at given offset with given length.
Since: 4.2.0

Arguments

offset (optional)
The position of the first byte. Default is 0, or the first byte.

length (optional)
The length of the integer. Default is -1, or the remaining bytes in the ByteArray.

Returns
The value of the little endian encoded unsigned integer beginning at given offset with given length.

bytearray:le_uint64([offset], [length])
Read a little endian encoded 64 bit unsigned integer in a ByteArray beginning at given offset with given length.
Since: 4.2.0

Arguments

offset (optional)
The position of the first byte. Default is 0, or the first byte.
length (optional)
The length of the integer. Default is -1, or the remaining bytes in the ByteArray.

Returns
The value of the little endian encoded 64 bit unsigned integer as a UInt64 object beginning at given offset with given length.

bytearray:int([offset], [length])
Read a big endian encoded signed integer in a ByteArray beginning at given offset with given length.
Since: 4.2.0

Arguments
offset (optional)
The position of the first byte. Default is 0, or the first byte.

length (optional)
The length of the integer. Default is -1, or the remaining bytes in the ByteArray.

Returns
The value of the big endian encoded 32 bit signed integer beginning at given offset with given length.

bytearray:int64([offset], [length])
Read a big endian encoded 64 bit signed integer in a ByteArray beginning at given offset with given length.
Since: 4.2.0

Arguments
offset (optional)
The position of the first byte. Default is 0, or the first byte.

length (optional)
The length of the integer. Default is -1, or the remaining bytes in the ByteArray.

Returns
The value of the big endian encoded 64 bit signed integer as a Int64 object beginning at given offset and given length.
**bytearray:uint([offset], [length])**

Read a big endian encoded unsigned integer in a *ByteArray* beginning at given offset with given length.

Since: 4.2.0

**Arguments**

**offset (optional)**

The position of the first byte. Default is 0, or the first byte.

**length (optional)**

The length of the integer. Default is -1, or the remaining bytes in the *ByteArray*.

**Returns**

The value of the big endian encoded 32 bit unsigned integer beginning at given offset with given length.

**bytearray:uint64([offset], [length])**

Read a big endian encoded 64 bit unsigned integer in a *ByteArray* beginning at given offset with given length.

Since: 4.2.0

**Arguments**

**offset (optional)**

The position of the first byte. Default is 0, or the first byte.

**length (optional)**

The length of the integer. Default is -1, or the remaining bytes in the *ByteArray*.

**Returns**

The value of the big endian encoded 64 bit unsigned integer as a *UInt64* object beginning at given offset with given length.

**bytearray:len()**

Obtain the length of a *ByteArray*.

**Returns**

The length of the *ByteArray*. 
bytearray:subset(offset, length)

Obtain a segment of a ByteArray, as a new ByteArray.

Arguments

offset
The position of the first byte (0=first).

length
The length of the segment.

Returns

A ByteArray containing the requested segment.

bytearray:base64_decode()

Obtain a Base64 decoded ByteArray.

Returns

The created ByteArray.

bytearray:raw([offset], [length])

Obtain a Lua string of the binary bytes in a ByteArray.

Arguments

offset (optional)
The position of the first byte (default=0/first).

length (optional)
The length of the segment to get (default=all).

Returns

A Lua string of the binary bytes in the ByteArray.

bytearray:tohex([lowercase], [separator])

Obtain a Lua string of the bytes in a ByteArray as hex-ascii, with given separator.

Arguments

lowercase (optional)
True to use lower-case hex characters (default=false).
separator (optional)
A string separator to insert between hex bytes (default=nil).

Returns
A hex-ascii string representation of the ByteArray.

bytearray:__tostring()

Obtain a Lua string containing the bytes in a ByteArray so that it can be used in display filters (e.g. "01FE456789AB").

Returns
A hex-ascii string representation of the ByteArray.

bytearray:tvb(name)

Creates a new Tvb from a ByteArray. The Tvb will be added to the current frame.

Example

```lua
function proto_foo.dissector(buf, pinfo, tree)
    -- Create a new tab named "My Tvb" and add some data to it
    local b = ByteArray.new("11223344")
    local tvb = ByteArray.tvb(b, "My Tvb")

    -- Create a tree item that, when clicked, automatically shows the tab we just created
    tree:add( tvb(1,2), "Foo" )
end
```

Arguments

name
The name to be given to the new data source.

Returns
The created Tvb.

Tvb

A Tvb represents the packet's buffer. It is passed as an argument to listeners and dissectors, and can be used to extract information (via TvbRange) from the packet's data.
To create a `TvbRange` the `Tvb` must be called with offset and length as optional arguments; the offset defaults to 0 and the length to `tvb:captured_len()`.

**WARNING**
Tvbs are usable only by the current listener or dissector call and are destroyed as soon as the listener or dissector returns, so references to them are unusable once the function has returned.

`tvb:__tostring()`

Convert the bytes of a `Tvb` into a string. This is primarily useful for debugging purposes since the string will be truncated if it is too long.

**Returns**
The string.

`tvb:reported_len()`

Obtain the reported length (length on the network) of a `Tvb`.

**Returns**
The reported length of the `Tvb`.

`tvb:captured_len()`

Obtain the captured length (amount saved in the capture process) of a `Tvb`.

**Returns**
The captured length of the `Tvb`.

`tvb:len()`

Obtain the captured length (amount saved in the capture process) of a `Tvb`. Same as `captured_len`; kept only for backwards compatibility.

**Returns**
The captured length of the `Tvb`.

`tvb:reported_length_remaining([offset])`

Obtain the reported (not captured) length of packet data to end of a `Tvb` or 0 if the offset is beyond the end of the `Tvb`.

**Arguments**
offset (optional)
   The offset (in octets) from the beginning of the Tvb. Defaults to 0.

Returns
The remaining reported length of the Tvb.

tvb:bytes([offset], [length])
Obtain a ByteArray from a Tvb.

Arguments
offset (optional)
   The offset (in octets) from the beginning of the Tvb. Defaults to 0.

length (optional)
   The length (in octets) of the range. Defaults to until the end of the Tvb.

Returns
The ByteArray object or nil.

tvb:offset()
Returns the raw offset (from the beginning of the source Tvb) of a sub Tvb.

Returns
The raw offset of the Tvb.

tvb:__call()
Equivalent to tvb:range(...)

tvb:range([offset], [length])
Creates a TvbRange from this Tvb.

Arguments
offset (optional)
   The offset (in octets) from the beginning of the Tvb. Defaults to 0.

length (optional)
   The length (in octets) of the range. Defaults to -1, which specifies the remaining bytes in the Tvb.
Returns
The TvbRange

tvb:raw([offset], [length])
Obtain a Lua string of the binary bytes in a Tvb.

Arguments
offset (optional)
The position of the first byte. Default is 0, or the first byte.

length (optional)
The length of the segment to get. Default is -1, or the remaining bytes in the Tvb.

Returns
A Lua string of the binary bytes in the Tvb.

tvb:_eq()
Checks whether contents of two Tvbs are equal.

TvbRange
A TvbRange represents a usable range of a Tvb and is used to extract data from the Tvb that generated it.

TvbRanges are created by calling a Tvb (e.g. ‘tvb(offset,length)’). A length of -1, which is the default, means to use the bytes up to the end of the Tvb. If the TvbRange span is outside the Tvb’s range the creation will cause a runtime error.

tvbrange:tvb()
Creates a new Tvb from a TvbRange.

tvbrange:uint()
Get a Big Endian (network order) unsigned integer from a TvbRange. The range must be 1-4 octets long.

Returns
The unsigned integer value.
**tvbrange:le_uint()**

Get a Little Endian unsigned integer from a `TvbRange`. The range must be 1-4 octets long.

**Returns**

The unsigned integer value

---

**tvbrange:uint64()**

Get a Big Endian (network order) unsigned 64 bit integer from a `TvbRange`, as a `UInt64` object. The range must be 1-8 octets long.

**Returns**

The `UInt64` object.

---

**tvbrange:le_uint64()**

Get a Little Endian unsigned 64 bit integer from a `TvbRange`, as a `UInt64` object. The range must be 1-8 octets long.

**Returns**

The `UInt64` object.

---

**tvbrange:int()**

Get a Big Endian (network order) signed integer from a `TvbRange`. The range must be 1-4 octets long.

**Returns**

The signed integer value.

---

**tvbrange:le_int()**

Get a Little Endian signed integer from a `TvbRange`. The range must be 1-4 octets long.

**Returns**

The signed integer value.

---

**tvbrange:int64()**

Get a Big Endian (network order) signed 64 bit integer from a `TvbRange`, as an `Int64` object. The range must be 1-8 octets long.

**Returns**
The **Int64** object.

### tvbrange:le_int64()

Get a Little Endian signed 64 bit integer from a **Tvbrange**, as an **Int64** object. The range must be 1-8 octets long.

**Returns**

The **Int64** object.

### tvbrange:float()

Get a Big Endian (network order) floating point number from a **Tvbrange**. The range must be 4 or 8 octets long.

**Returns**

The floating point value.

### tvbrange:le_float()

Get a Little Endian floating point number from a **Tvbrange**. The range must be 4 or 8 octets long.

**Returns**

The floating point value.

### tvbrange:ipv4()

Get an IPv4 Address from a **Tvbrange**, as an **Address** object.

**Returns**

The IPv4 **Address** object.

### tvbrange:le_ipv4()

Get an Little Endian IPv4 Address from a **Tvbrange**, as an **Address** object.

**Returns**

The IPv4 **Address** object.

### tvbrange:ipv6()

Get an IPv6 Address from a **Tvbrange**, as an **Address** object.
Returns
The IPv6 Address object.

tvrange:ether()
Get an Ethernet Address from a TvRange, as an Address object.

Returns
The Ethernet Address object.

Errors
• The range must be 6 bytes long

tvrange:nstime([encoding])
Obtain a time_t structure from a TvRange, as an NSTime object.

Arguments
encoding (optional)
   An optional ENC_* encoding value to use

Returns
The NSTime object and number of bytes used, or nil on failure.

Errors
• The range must be 4 or 8 bytes long

tvrange:le_nstime()
Obtain a nstime from a TvRange, as an NSTime object.

Returns
The NSTime object.

Errors
• The range must be 4 or 8 bytes long

tvrange:string([encoding])
Obtain a string from a TvRange.
Arguments

encoding (optional)
   The encoding to use. Defaults to ENC_ASCII.

Returns

A string containing all bytes in the TVbRange including all zeroes (e.g., "a\000bc\000").

tvrange:ustring()

Obtain a Big Endian (network order) UTF-16 encoded string from a TVbRange.

Returns

A string containing all bytes in the TVbRange including all zeroes (e.g., "a\000bc\000").

tvrange:le_ustring()

Obtain a Little Endian UTF-16 encoded string from a TVbRange.

Returns

A string containing all bytes in the TVbRange including all zeroes (e.g., "a\000bc\000").

tvrange:stringz([encoding])

Obtain a zero terminated string from a TVbRange.

Arguments

encoding (optional)
   The encoding to use. Defaults to ENC_ASCII.

Returns

The string containing all bytes in the TVbRange up to the first terminating zero.

tvbrange:strsize([encoding])

Find the size of a zero terminated string from a TVbRange. The size of the string includes the terminating zero.

Arguments

encoding (optional)
   The encoding to use. Defaults to ENC_ASCII.
Returns
Length of the zero terminated string.

tvbrange:ustringz()
Obtain a Big Endian (network order) UTF-16 encoded zero terminated string from a TvbRange.

Returns
Two return values: the zero terminated string, and the length.

tvbrange:le_ustringz()
Obtain a Little Endian UTF-16 encoded zero terminated string from a TvbRange

Returns
Two return values: the zero terminated string, and the length.

tvbrange:bytes([encoding])
Obtain a ByteArray from a TvbRange.

Starting in 1.11.4, this function also takes an optional encoding argument, which can be set to ENC_STR_HEX to decode a hex-string from the TvbRange into the returned ByteArray. The encoding can be bitwise-or'ed with one or more separator encodings, such as ENC_SEP_COLON, to allow separators to occur between each pair of hex characters.

The return value also now returns the number of bytes used as a second return value.

On failure or error, nil is returned for both return values.

NOTE
The encoding type of the hex string should also be set, for example ENC_ASCII or ENC_UTF_8, along with ENC_STR_HEX.

Arguments
encoding (optional)
An optional ENC_* encoding value to use

Returns
The ByteArray object or nil, and number of bytes consumed or nil.

tvbrange:bitfield([position], [length])
Get a bitfield from a TvbRange.
Arguments

position (optional)
The bit offset (MSB 0 bit numbering) from the beginning of the TvbRange. Defaults to 0.

length (optional)
The length in bits of the field. Defaults to 1.

Returns
The bitfield value
tvbrange:range([offset], [length])
Creates a sub-TvbRange from this TvbRange.

Arguments

offset (optional)
The offset (in octets) from the beginning of the TvbRange. Defaults to 0.

length (optional)
The length (in octets) of the range. Defaults to until the end of the TvbRange.

Returns
The TvbRange.
tvbrange:uncompress(name)
Obtain an uncompressed TvbRange from a TvbRange

Arguments

name
The name to be given to the new data-source.

Returns
The TvbRange.
tvbrange:len()
Obtain the length of a TvbRange.
Obtain the offset in a \texttt{TvbRange}.

\texttt{tvbrange:raw([offset], [length])}

Obtain a Lua string of the binary bytes in a \texttt{TvbRange}.

**Arguments**

\texttt{offset (optional)}

The position of the first byte within the range. Default is 0, or first byte.

\texttt{length (optional)}

The length of the segment to get. Default is -1, or the remaining bytes in the range.

**Returns**

A Lua string of the binary bytes in the \texttt{TvbRange}.

\texttt{tvbrange:__eq()}

Checks whether the contents of two \texttt{TvbRanges} are equal.

\texttt{tvbrange:__tostring()}

Converts the \texttt{TvbRange} into a string. The string can be truncated, so this is primarily useful for debugging or in cases where truncation is preferred, e.g. "67:89:AB:...".

**Returns**

A Lua hex string of the \texttt{TvbRange} truncated to 24 bytes.

## Adding Information To The Dissection Tree

**TreeItem**

\texttt{TreeItem}s represent information in the packet details pane of Wireshark, and the packet details view of TShark. A \texttt{TreeItem} represents a node in the tree, which might also be a subtree and have a list of children. The children of a subtree have zero or more siblings which are other children of the same \texttt{TreeItem} subtree.

During dissection, heuristic-dissection, and post-dissection, a root \texttt{TreeItem} is passed to dissectors as the third argument of the function callback (e.g., \texttt{myproto.dissector(tvbuf, pktinfo, root)}).

In some cases the tree is not truly added to, in order to improve performance. For example for packets not currently displayed/selected in Wireshark’s visible window pane, or if TShark isn’t...
invoked with the `-V` switch. However the "add" type `TreeItem` functions can still be called, and still return `TreeItem` objects - but the info isn't really added to the tree. Therefore you do not typically need to worry about whether there's a real tree or not. If, for some reason, you need to know it, you can use the `TreeItem.visible` attribute getter to retrieve the state.

**treeitem:add_packet_field(protofield, [tvbrange], encoding, [label])**

Adds a new child tree for the given `ProtoField` object to this tree item, returning the new child `TreeItem`.

Unlike `TreeItem:add()` and `TreeItem:add_le()`, the `ProtoField` argument is not optional, and cannot be a `Proto` object. Instead, this function always uses the `ProtoField` to determine the type of field to extract from the passed-in `Tvbrange`, highlighting the relevant bytes in the Packet Bytes pane of the GUI (if there is a GUI), etc. If no `Tvbrange` is given, no bytes are highlighted and the field's value cannot be determined; the `ProtoField` must have been defined/created not to have a length in such a case, or an error will occur. For backwards-compatibility reasons the `encoding` argument, however, must still be given.

Unlike `TreeItem:add()` and `TreeItem:add_le()`, this function performs both big-endian and little-endian decoding, by setting the `encoding` argument to be `ENC_BIG_ENDIAN` or `ENC_LITTLE_ENDIAN`.

The signature of this function:

```
tree_item:add_packet_field(proto_field [,tvbrange], encoding, ...)```

In Wireshark version 1.11.3, this function was changed to return more than just the new child `TreeItem`. The child is the first return value, so that function chaining will still work as before; but it now also returns more information. The second return is the value of the extracted field (i.e., a number, `UInt64`, `Address`, etc.). The third return is the offset where data should be read next. This is useful when the length of the field is not known in advance. The additional return values may be null if the field type is not well supported in the Lua API.

Another new feature added to this function in Wireshark version 1.11.3 is the ability to extract native number `ProtoFields` from string encoding in the `Tvbrange`, for ASCII-based and similar string encodings. For example, a `ProtoField` of type `ftypes.UINT32` can be extracted from a `Tvbrange` containing the ASCII string "123", and it will correctly decode the ASCII to the number 123, both in the tree as well as for the second return value of this function. To do so, you must set the `encoding` argument of this function to the appropriate string `ENC_*` value, bitwise-or'd with the `ENC_STRING` value. `ENC_STRING` is guaranteed to be a unique bit flag, and thus it can added instead of bitwise-or'ed as well. Only single-byte ASCII digit string encoding types can be used for this, such as `ENC_ASCII` and `ENC_UTF_8`.

For example, assuming the `Tvbrange` named “tvb” contains the string ”123":
local myfield = ProtoField.new("Transaction ID", "myproto.trans_id", ftypes.UINT16)

local child, value = tree:add_packet_field(myfield, tvb:range(0,3), ENC_UTF_8 + ENC_STRING)

Arguments

protofield
The ProtoField field object to add to the tree.

tvbrange (optional)
The TvbRange of bytes in the packet this tree item covers/represents.

encoding
The field's encoding in the TvbRange.

label (optional)
One or more strings to append to the created TreeItem.

Returns
The new child TreeItem, the field's extracted value or nil, and offset or nil.

treeitem:add([protofield], [tvbrange], [value], [label])
Adds a child item to this tree item, returning the new child TreeItem.

If the ProtoField represents a numeric value (int, uint or float), then it's treated as a Big Endian (network order) value.

This function has a complicated form: ‘treeitem:add([protofield], [tvbrange], [value], [label])’, such that if the first argument is a ProtoField or a Proto, the second argument is a TvbRange, and a third argument is given, it's a value; but if the second argument is a non-TvbRange, then it's the value (as opposed to filling that argument with 'nil', which is invalid for this function). If the first argument is a non-ProtoField and a non-Proto then this argument can be either a TvbRange or a label, and the value is not in use.

Example
local proto_foo = Proto("foo", "Foo Protocol")
proto_foo.fields.bytes = ProtoField.bytes("foo.bytes", "Byte array")
proto_foo.fields.u16 = ProtoField.uint16("foo.u16", "Unsigned short", base.HEX)

function proto_foo.dissector(buf, pinfo, tree)
    -- ignore packets less than 4 bytes long
    if buf:len() < 4 then return end

    -- #############################################################
    -- # Assume buf(0,4) == {0x00, 0x01, 0x00, 0x02}
    -- #############################################################
    local t = tree:add( proto_foo, buf() )

    -- Adds a byte array that shows as: "Byte array: 00010002"
    t:add( proto_foo.fields.bytes, buf(0,4) )

    -- Adds a byte array that shows as "Byte array: 313233"
    -- (the ASCII char code of each character in "123")
    t:add( proto_foo.fields.bytes, buf(0,4), "123" )

    -- Adds a tree item that shows as: "Unsigned short: 0x0001"
    t:add( proto_foo.fields.u16, buf(0,2) )

    -- Adds a tree item that shows as: "Unsigned short: 0x0064"
    t:add( proto_foo.fields.u16, buf(0,2), 100 )

    -- Adds a tree item that shows as: "Unsigned short: 0x0064 ( big endian )"
    t:add( proto_foo.fields.u16, buf(1,2), 100, nil, "("", nil, "big", 999, nil, "endian", nil, ")" )

    -- LITTLE ENDIAN: Adds a tree item that shows as: "Unsigned short: 0x0100"
    t:add_le( proto_foo.fields.u16, buf(0,2) )

    -- LITTLE ENDIAN: Adds a tree item that shows as: "Unsigned short: 0x6400"
    t:add_le( proto_foo.fields.u16, buf(0,2), 100 )

    -- LITTLE ENDIAN: Adds a tree item that shows as: "Unsigned short: 0x6400 ( little endian )"
    t:add_le( proto_foo.fields.u16, buf(1,2), 100, nil, "("", nil, "little", 999, nil, "endian", nil, ")" )
end

udp_table = DissectorTable.get("udp.port")
udp_table:add(7777, proto_foo)
Arguments

protofield (optional)
   The ProtoField field or Proto protocol object to add to the tree.

tvbrange (optional)
   The TvbRange of bytes in the packet this tree item covers/represents.

value (optional)
   The field’s value, instead of the ProtoField/Proto one.

label (optional)
   One or more strings to use for the tree item label, instead of the ProtoField/Proto one.

Returns

The new child TreeItem.

treeitem:add_le([protofield], [tvbrange], [value], [label])

Adds a child item to this tree item, returning the new child TreeItem.

If the ProtoField represents a numeric value (int, uint or float), then it’s treated as a Little Endian value.

This function has a complicated form: 'treeitem:add_le([protofield], [tvbrange], [value], [label])', such that if the first argument is a ProtoField or a Proto, the second argument is a TvbRange, and a third argument is given, it’s a value; but if the second argument is a non-TvbRange, then it’s the value (as opposed to filling that argument with 'nil', which is invalid for this function). If the first argument is a non-ProtoField and a non-Proto then this argument can be either a TvbRange or a label, and the value is not in use.

Arguments

protofield (optional)
   The ProtoField field or Proto protocol object to add to the tree.

tvbrange (optional)
   The TvbRange of bytes in the packet this tree item covers/represents.

value (optional)
   The field’s value, instead of the ProtoField/Proto one.

label (optional)
   One or more strings to use for the tree item label, instead of the ProtoField/Proto one.
Returns
The new child TreeItem.

treeitem:set_text(text)
Sets the text of the label.
This used to return nothing, but as of 1.11.3 it returns the same tree item to allow chained calls.

Arguments
text
    The text to be used.

Returns
The same TreeItem.

treeitem:append_text(text)
Appends text to the label.
This used to return nothing, but as of 1.11.3 it returns the same tree item to allow chained calls.

Arguments
text
    The text to be appended.

Returns
The same TreeItem.

treeitem:prepend_text(text)
Prepends text to the label.
This used to return nothing, but as of 1.11.3 it returns the same tree item to allow chained calls.

Arguments
text
    The text to be prepended.

Returns
The same TreeItem.
treeitem:add_expert_info([group], [severity], [text])

Sets the expert flags of the item and adds expert info to the packet.

This function does **not** create a truly filterable expert info for a protocol. Instead you should use TreeItem.add_proto_expert_info().

Note: This function is provided for backwards compatibility only, and should not be used in new Lua code. It may be removed in the future. You should only use TreeItem.add_proto_expert_info().

**Arguments**

**group (optional)**

One of: `PI_CHECKSUM`, `PI_SEQUENCE`, `PI_RESPONSE_CODE`, `PI_REQUEST_CODE`, `PI_UNDECODED`, `PI_REASSEMBLE`, `PI_MALFORMED`, `PI_DEBUG`, `PI_PROTOCOL`, `PI_SECURITY`, `PI_COMMENTS_GROUP`, `PI_DECRYPTION`, `PI_ASSUMPTION`, `PI_DEPRECATED`, `PI_RECEIVE`, `PI_INTERFACE`, or `PI_DISSECTOR_BUG`.

**severity (optional)**

One of: `PI_COMMENT`, `PI_CHAT`, `PI_NOTE`, `PI_WARN`, or `PI_ERROR`.

**text (optional)**

The text for the expert info display.

**Returns**

The same TreeItem.

---

treeitem:add_proto_expert_info(expert, [text])

Sets the expert flags of the tree item and adds expert info to the packet.

**Arguments**

**expert**

The ProtoExpert object to add to the tree.

**text (optional)**

Text for the expert info display (default is to use the registered text).

**Returns**

The same TreeItem.

---

treeitem:add tvb_expert_info(expert, tvb, [text])

Sets the expert flags of the tree item and adds expert info to the packet associated with the tvb or TvbRange bytes in the packet.
Arguments

**expert**

The *ProtoExpert* object to add to the tree.

**tvb**

The *Tvb* or *TvbRange* object bytes to associate the expert info with.

**text (optional)**

Text for the expert info display (default is to use the registered text).

Returns

The same TreeItem.

**treeitem:set_generated([bool])**

Marks the *TreeItem* as a generated field (with data inferred but not contained in the packet).

This used to return nothing, but as of 1.11.3 it returns the same tree item to allow chained calls.

Arguments

**bool (optional)**

A Lua boolean, which if *true* sets the *TreeItem* generated flag, else clears it (default=true)

Returns

The same TreeItem.

**treeitem:set_hidden([bool])**

Marks the *TreeItem* as a hidden field (neither displayed nor used in filters). Deprecated

This used to return nothing, but as of 1.11.3 it returns the same tree item to allow chained calls.

Arguments

**bool (optional)**

A Lua boolean, which if *true* sets the *TreeItem* hidden flag, else clears it. Default is *true*.

Returns

The same TreeItem.

**treeitem:set_len(len)**

Set *TreeItem*’s length inside tvb, after it has already been created.
This used to return nothing, but as of 1.11.3 it returns the same tree item to allow chained calls.

**Arguments**

**len**

The length to be used.

**Returns**

The same TreeItem.

**treeitem:referenced(protofield)**

Checks if a ProtoField or Dissector is referenced by a filter/tap/UI.

If this function returns `false`, it means that the field (or dissector) does not need to be dissected and can be safely skipped. By skipping a field rather than dissecting it, the dissector will usually run faster since Wireshark will not do extra dissection work when it doesn't need the field.

You can use this in conjunction with the TreeItem.visible attribute. This function will always return true when the TreeItem is visible. When it is not visible and the field is not referenced, you can speed up the dissection by not dissecting the field as it is not needed for display or filtering.

This function takes one parameter that can be a ProtoField or Dissector. The Dissector form is useful when you need to decide whether to call a sub-dissector.

**Arguments**

**protofield**

The ProtoField or Dissector to check if referenced.

**Returns**

A boolean indicating if the ProtoField/Dissector is referenced

**treeitem:__tostring()**

Returns string debug information about the TreeItem.

**treeitem.text**

Mode: Retrieve or assign.

Set/get the TreeItem's display string (string).

For the getter, if the TreeItem has no display string, then nil is returned.
**treeitem.visible**

Mode: Retrieve only.

Get the TreeItem's subtree visibility status (boolean).

**treeitem.generated**

Mode: Retrieve or assign.

Set/get the TreeItem's generated state (boolean).

**treeitem.hidden**

Mode: Retrieve or assign.

Set/get TreeItem's hidden state (boolean).

**treeitem.len**

Mode: Retrieve or assign.

Set/get TreeItem's length inside tvb, after it has already been created.

### Post-Dissection Packet Analysis

**Listener**

A Listener is called once for every packet that matches a certain filter or has a certain tap. It can read the tree, the packet's Tvb buffer as well as the tapped data, but it cannot add elements to the tree.

**Listener.new([tap], [filter], [allfields])**

Creates a new Listener tap object.

**Arguments**

**tap (optional)**

The name of this tap. See Listener.list() for a way to print valid listener names.

**filter (optional)**

A display filter to apply to the tap. The tap.packet function will be called for each matching packet. The default is nil, which matches every packet. Example: "m2tp".

**allfields (optional)**

Whether to generate all fields. The default is false. Note: This impacts performance.
Returns
The newly created Listener listener object

Errors
- tap registration error

Listener.list()

Gets a Lua array table of all registered Listener tap names.

Note: This is an expensive operation, and should only be used for troubleshooting.

Example

```lua
-- Print a list of tap listeners to stdout.
for _,tap_name in pairs(Listener.list()) do
    print(tap_name)
end
```

Returns
The array table of registered tap names

listener:remove()

Removes a tap Listener.

listener:__tostring()

Generates a string of debug info for the tap Listener.

listener.packet

Mode: Assign only.

A function that will be called once every packet matches the Listener listener filter.

When later called by Wireshark, the packet function will be given:

1. A Pinfo object
2. A Tvb object
3. A tapinfo table

```lua
function tap.packet(pinfo, tvb, tapinfo) ... end
```
NOTE

\textit{tapinfo} is a table of info based on the \textit{Listener} type, or nil.

See \texttt{epan/wslua/taps} for \texttt{tapinfo} structure definitions.

\texttt{listener.draw}

Mode: Assign only.

A function that will be called once every few seconds to redraw the GUI objects; in TShark this function is called only at the very end of the capture file.

When later called by Wireshark, the \texttt{draw} function will not be given any arguments.

\begin{verbatim}
function tap.draw() ... end
\end{verbatim}

\texttt{listener.reset}

Mode: Assign only.

A function that will be called at the end of the capture run.

When later called by Wireshark, the \texttt{reset} function will not be given any arguments.

\begin{verbatim}
function tap.reset() ... end
\end{verbatim}

\section*{Saving Capture Files}

The classes/functions defined in this module are for using a \texttt{Dumper} object to make Wireshark save a capture file to disk. \texttt{Dumper} represents Wireshark’s built-in file format writers (see the \texttt{wtap_name_to_file_type_subtype} function).

(The \texttt{wtap_filetypes} table is deprecated, and should only be used in code that must run on Wireshark 3.4.3 and earlier 3.4 releases or in Wireshark 3.2.11 and earlier 3.2.x releases.)

To have a Lua script create its own file format writer, see the chapter titled "Custom file format reading/writing".

\textbf{Dumper}

\texttt{Dumper.new(filename, [filetype], [encap])}

Creates a file to write packets. \texttt{Dumper:new_for_current()} will probably be a better choice.

\textbf{Arguments}
**filename**

The name of the capture file to be created.

**filetype (optional)**

The type of the file to be created - a number returned by `wtap_name_to_file_type_subtype()`. (The `wtap_filetypes` table is deprecated, and should only be used in code that must run on Wireshark 3.4.3 and earlier 3.4 releases or in Wireshark 3.2.11 and earlier 3.2.x releases.)

**encap (optional)**

The encapsulation to be used in the file to be created - a number entry from the `wtap_encaps` table.

**Returns**

The newly created Dumper object

**dumper:close()**

Closes a dumper.

**Errors**

- Cannot operate on a closed dumper

**dumper:flush()**

Writes all unsaved data of a dumper to the disk.

**dumper:dump(timestamp, pseudoheader, bytearray)**

Dumps an arbitrary packet. Note: Dumper:dump_current() will fit best in most cases.

**Arguments**

**timestamp**

The absolute timestamp the packet will have.

**pseudoheader**

The `PseudoHeader` to use.

**bytearray**

The data to be saved

**dumper:new_for_current([filetype])**

Creates a capture file using the same encapsulation as the one of the current packet.
Arguments

filetype (optional)

The file type. Defaults to pcap.

Returns

The newly created Dumper Object

Errors

• Cannot be used outside a tap or a dissector

dumper:dump_current()

Dumps the current packet as it is.

Errors

• Cannot be used outside a tap or a dissector

PseudoHeader

A pseudoheader to be used to save captured frames.

PseudoHeader.none()

Creates a "no" pseudoheader.

Returns

A null pseudoheader

PseudoHeader.eth([fcslen])

Creates an ethernet pseudoheader.

Arguments

fcslen (optional)

The fcs length

Returns

The ethernet pseudoheader
PseudoHeader.atm([aal], [vpi], [vci], [channel], [cells], [aal5u2u], [aal5len])

Creates an ATM pseudoheader.

Arguments

aal (optional)
   AAL number

vpi (optional)
   VPI

vci (optional)
   VCI

channel (optional)
   Channel

cells (optional)
   Number of cells in the PDU

aal5u2u (optional)
   AAL5 User to User indicator

aal5len (optional)
   AAL5 Len

Returns

The ATM pseudoheader

PseudoHeader.mtp2([sent], [annexa], [linknum])

Creates an MTP2 PseudoHeader.

Arguments

sent (optional)
   True if the packet is sent, False if received.

annexa (optional)
   True if annex A is used.

linknum (optional)
   Link Number.
The MTP2 pseudoheader

**Wtap Functions For Handling Capture File Types**

**Global Functions**

**wtap_file_type_subtype_description(filetype)**

Get a string describing a capture file type, given a filetype value for that file type.

**Arguments**

`filetype`

The type for which the description is to be fetched - a number returned by `wtap_name_to_file_type_subtype()`.

**Returns**

The description of the file type with that filetype value, or nil if there is no such file type.

**wtap_file_type_subtype_name(filetype)**

Get a string giving the name for a capture file type, given a filetype value for that file type.

**Arguments**

`filetype`

The type for which the name is to be fetched - a number returned by `wtap_name_to_file_type_subtype()`.

**Returns**

The name of the file type with that filetype value, or nil if there is no such file type.

**wtap_name_to_file_type_subtype(name)**

Get a filetype value for a file type, given the name for that file type.

**Arguments**

`name`

The name of a file type.

**Returns**
The filetype value for the file type with that name, or nil if there is no such file type.

**wtap_pcap_file_type_subtype()**

Get the filetype value for pcap files.

**Returns**

The filetype value for pcap files.

**wtap_pcap_nsec_file_type_subtype()**

Get the filetype value for nanosecond-resolution pcap files.

**Returns**

The filetype value for nanosecond-resolution pcap files.

**wtap_pcapng_file_type_subtype()**

Get the filetype value for pcapng files.

**Returns**

The filetype value for pcapng files.

### Custom File Format Reading And Writing

The classes/functions defined in this section allow you to create your own custom Lua-based "capture" file reader, or writer, or both.

**CaptureInfo**

A CaptureInfo object, passed into Lua as an argument by FileHandler callback function read_open(), read(), seek_read(), seq_read_close(), and read_close(). This object represents capture file data and meta-data (data about the capture file) being read into Wireshark/TShark.

This object’s fields can be written-to by Lua during the read-based function callbacks. In other words, when the Lua plugin’s FileHandler.read_open() function is invoked, a CaptureInfo object will be passed in as one of the arguments, and its fields should be written to by your Lua code to tell Wireshark about the capture.

**captureinfo:_tostring()**

Generates a string of debug info for the CaptureInfo

**Returns**
String of debug information.

**captureinfo.encap**

Mode: Retrieve or assign.

The packet encapsulation type for the whole file.

See [wtap_encaps](#) for available types. Set to **wtap_encaps.PER_PACKET** if packets can have different types, then later set **FrameInfo.encap** for each packet during `read()/seek_read()`.

**captureinfo.time_precision**

Mode: Retrieve or assign.

The precision of the packet timestamps in the file.

See [wtap_file_tsprec](#) for available precisions.

**captureinfo.snapshot_length**

Mode: Retrieve or assign.

The maximum packet length that could be recorded.

Setting it to 0 means unknown.

**captureinfo.comment**

Mode: Retrieve or assign.

A string comment for the whole capture file, or nil if there is no comment.

**captureinfo.hardware**

Mode: Retrieve or assign.

A string containing the description of the hardware used to create the capture, or nil if there is no hardware string.

**captureinfo.os**

Mode: Retrieve or assign.

A string containing the name of the operating system used to create the capture, or nil if there is no os string.
**captureinfo.user_app**

Mode: Retrieve or assign.

A string containing the name of the application used to create the capture, or nil if there is no `user_app` string.

**captureinfo.hosts**

Mode: Assign only.

Sets resolved ip-to-hostname information.

The value set must be a Lua table of two key-ed names: `ipv4_addresses` and `ipv6_addresses`. The value of each of these names are themselves array tables, of key-ed tables, such that the inner table has a key `addr` set to the raw 4-byte or 16-byte IP address Lua string and a `name` set to the resolved name.

For example, if the capture file identifies one resolved IPv4 address of 1.2.3.4 to `foo.com`, then you must set `CaptureInfo.hosts` to a table of:

```lua
{ ipv4_addresses = { { addr = "\01\02\03\04", name = "foo.com" } } }
```

Note that either the `ipv4_addresses` or the `ipv6_addresses` table, or both, may be empty or nil.

**captureinfo.private_table**

Mode: Retrieve or assign.

A private Lua value unique to this file.

The `private_table` is a field you set/get with your own Lua table. This is provided so that a Lua script can save per-file reading/writing state, because multiple files can be opened and read at the same time.

For example, if the user issued a reload-file command, or Lua called the `reload()` function, then the current capture file is still open while a new one is being opened, and thus Wireshark will invoke `read_open()` while the previous capture file has not caused `read_close()` to be called; and if the `read_open()` succeeds then `read_close()` will be called right after that for the previous file, rather than the one just opened. Thus the Lua script can use this `private_table` to store a table of values specific to each file, by setting this `private_table` in the `read_open()` function, which it can then later get back inside its `read()`, `seek_read()`, and `read_close()` functions.

**CaptureInfoConst**

A `CaptureInfoConst` object, passed into Lua as an argument to the `FileHandler` callback function `write_open()`.
This object represents capture file data and meta-data (data about the capture file) for the current capture in Wireshark/TShark.

This object's fields are read-from when used by write_open function callback. In other words, when the Lua plugin's FileHandler write_open function is invoked, a CaptureInfoConst object will be passed in as one of the arguments, and its fields should be read from by your Lua code to get data about the capture that needs to be written.

**captureinfoconst:__tostring()**

Generates a string of debug info for the CaptureInfoConst

**Returns**

String of debug information.

**captureinfoconst.type**

Mode: Retrieve only.

The file type.

**captureinfoconst.snapshot_length**

Mode: Retrieve only.

The maximum packet length that is actually recorded (vs. the original length of any given packet on-the-wire). A value of 0 means the snapshot length is unknown or there is no one such length for the whole file.

**captureinfoconst.encap**

Mode: Retrieve only.

The packet encapsulation type for the whole file.

See wtap_encaps for available types. It is set to wtap_encaps.PER_PACKET if packets can have different types, in which case each Frame identifies its type, in FrameInfo.packet_encap.

**captureinfoconst.comment**

Mode: Retrieve or assign.

A comment for the whole capture file, if the wtap_presence_flags.COMMENTS was set in the presence flags; nil if there is no comment.
**captureinfoconst.hardware**

Mode: Retrieve only.

A string containing the description of the hardware used to create the capture, or nil if there is no hardware string.

**captureinfoconst.os**

Mode: Retrieve only.

A string containing the name of the operating system used to create the capture, or nil if there is no os string.

**captureinfoconst.user_app**

Mode: Retrieve only.

A string containing the name of the application used to create the capture, or nil if there is no user_app string.

**captureinfoconst.hosts**

Mode: Retrieve only.

A ip-to-hostname Lua table of two key-ed names: `ipv4_addresses` and `ipv6_addresses`. The value of each of these names are themselves array tables, of key-ed tables, such that the inner table has a key `addr` set to the raw 4-byte or 16-byte IP address Lua string and a `name` set to the resolved name.

For example, if the current capture has one resolved IPv4 address of 1.2.3.4 to `foo.com`, then getting `CaptureInfoConst.hosts` will get a table of:

```
{ ipv4_addresses = { { addr = "\01\02\03\04", name = "foo.com" } }, ipv6_addresses = { } }
```

Note that either the `ipv4_addresses` or the `ipv6_addresses` table, or both, may be empty, however they will not be nil.

**captureinfoconst.private_table**

Mode: Retrieve or assign.

A private Lua value unique to this file.

The `private_table` is a field you set/get with your own Lua table. This is provided so that a Lua script can save per-file reading/writing state, because multiple files can be opened and read at the same time.
For example, if two Lua scripts issue a `Dumper:new_for_current()` call and the current file happens to use your script’s writer, then the Wireshark will invoke `write_open()` while the previous capture file has not had `write_close()` called. Thus the Lua script can use this private table to store a table of values specific to each file, by setting this private_table in the `write_open()` function, which it can then later get back inside its `write()`, and `write_close()` functions.

**File**

A File object, passed into Lua as an argument by FileHandler callback functions (e.g., `read_open`, `read`, `write`, etc.). This behaves similarly to the Lua io library’s file object, returned when calling `io.open()`, except in this case you cannot call `file:close()`, `file:open()`, nor `file:setvbuf()`, since Wireshark/TShark manages the opening and closing of files. You also cannot use the ‘io’ library itself on this object, i.e. you cannot do `io.read(file, 4)`. Instead, use this File with the object-oriented style calling its methods, i.e. `myfile:read(4)`. (see later example)

The purpose of this object is to hide the internal complexity of how Wireshark handles files, and instead provide a Lua interface that is familiar, by mimicking the io library. The reason true/raw io files cannot be used is because Wireshark does many things under the hood, such as compress the file, or write to stdout, or various other things based on configuration/commands.

When a File object is passed in through reading-based callback functions, such as `read_open()`, `read()`, and `read_close()`, then the File object’s `write()` and `flush()` functions are not usable and will raise an error if used.

When a File object is passed in through writing-based callback functions, such as `write_open()`, `write()`, and `write_close()`, then the File object’s `read()` and `lines()` functions are not usable and will raise an error if used.

Note: A File object should never be stored/saved beyond the scope of the callback function it is passed in to.

For example:
function myfilehandler.read_open(file, capture)
    local position = file:seek()

    -- read 24 bytes
    local line = file:read(24)

    -- do stuff

    -- it's not our file type, seek back (unnecessary but just to show it...)
    file:seek("set",position)

    -- return false because it's not our file type
    return false
end

file:read()

Reads from the File, similar to Lua's file:read(). See Lua 5.x ref manual for file:read().

file:seek()

Seeks in the File, similar to Lua's file:seek(). See Lua 5.x ref manual for file:seek().

Returns

The current file cursor position as a number.

file:lines()

Lua iterator function for retrieving ASCII File lines, similar to Lua's file:lines(). See Lua 5.x ref manual for file:lines().

file:write()

Writes to the File, similar to Lua's file:write(). See Lua 5.x ref manual for file:write().

file:__:tostring()

Generates a string of debug info for the File object

Returns

String of debug information.

file.compressed

Mode: Retrieve only.
Whether the File is compressed or not.

See `wtap_encaps` for available types. Set to `wtap_encaps.PER_PACKET` if packets can have different types, then later set `FrameInfo.encap` for each packet during `read()/seek_read()`.

**FileHandler**

A FileHandler object, created by a call to `FileHandler.new(arg1, arg2, ...)`. The FileHandler object lets you create a file-format reader, or writer, or both, by setting your own `read_open/read` or `write_open/write` functions.

**FileHandler.new(description, name, internal_description, type)**

Creates a new FileHandler

**Arguments**

- **description**
  
  A description of the file type, for display purposes only. E.g., "Wireshark - pcapng"

- **name**
  
  The file type name, used to look up the file type in various places. E.g., "pcapng". Note: The name cannot already be in use.

- **internal_description**
  
  Descriptive text about this file format, for internal display purposes only

- **type**
  
  The type of FileHandler, "r"/"w"/"rw" for reader/writer/both, include "m" for magic, "s" for strong heuristic

**Returns**

The newly created FileHandler object

**filehandler:__tostring()**

Generates a string of debug info for the FileHandler

**Returns**

String of debug information.

**filehandler.read_open**

Mode: Assign only.
The Lua function to be called when Wireshark opens a file for reading.

When later called by Wireshark, the Lua function will be given:

1. A **File** object
2. A **CaptureInfo** object

The purpose of the Lua function set to this `read_open` field is to check if the file Wireshark is opening is of its type, for example by checking for magic numbers or trying to parse records in the file, etc. The more can be verified the better, because Wireshark tries all file readers until it finds one that accepts the file, so accepting an incorrect file prevents other file readers from reading their files.

The called Lua function should return true if the file is its type (it accepts it), false if not. The Lua function must also set the File offset position (using `file:seek()`) to where it wants it to be for its first `read()` call.

**filehandler.read**

Mode: Assign only.

The Lua function to be called when Wireshark wants to read a packet from the file.

When later called by Wireshark, the Lua function will be given:

1. A **File** object
2. A **CaptureInfo** object
3. A **FrameInfo** object

The purpose of the Lua function set to this `read` field is to read the next packet from the file, and setting the parsed/read packet into the frame buffer using `FrameInfo.data = foo` or `FrameInfo:read_data(file, frame.captured_length)`.

The called Lua function should return the file offset/position number where the packet begins, or false if it hit an error. The file offset will be saved by Wireshark and passed into the set `seek_read()` Lua function later.

**filehandler.seek_read**

Mode: Assign only.

The Lua function to be called when Wireshark wants to read a packet from the file at the given offset.

When later called by Wireshark, the Lua function will be given:

1. A **File** object
2. A **CaptureInfo** object
3. A FrameInfo object
4. The file offset number previously set by the read() function call

The called Lua function should return true if the read was successful, or false if it hit an error. Since 2.4.0, a number is also acceptable to signal success, this allows for reuse of FileHandler:read:

```lua
local function fh_read(file, capture, frame) ... end
myfilehandler.read = fh_read

function myfilehandler.seek_read(file, capture, frame, offset)
    if not file:seek("set", offset) then
        -- Seeking failed, return failure
        return false
    end

    -- Now try to read one frame
    return fh_read(file, capture, frame)
end
```

Since 3.6.0, it’s possible to omit the FileHandler:seek_read() function to get a default seek_read implementation.

**filehandler.read_close**

Mode: Assign only.

The Lua function to be called when Wireshark wants to close the read file completely.

When later called by Wireshark, the Lua function will be given:

1. A File object
2. A CaptureInfo object

It is not necessary to set this field to a Lua function - FileHandler can be registered without doing so - it is available in case there is memory/state to clear in your script when the file is closed.

**filehandler.seq_read_close**

Mode: Assign only.

The Lua function to be called when Wireshark wants to close the sequentially-read file.

When later called by Wireshark, the Lua function will be given:

1. A File object
2. A CaptureInfo object
It is not necessary to set this field to a Lua function - FileHandler can be registered without doing so - it is available in case there is memory/state to clear in your script when the file is closed for the sequential reading portion. After this point, there will be no more calls to `read()`, only `seek_read()`.

**filehandler.can_write_encap**

Mode: Assign only.

The Lua function to be called when Wireshark wants to write a file, by checking if this file writer can handle the wtap packet encapsulation(s).

When later called by Wireshark, the Lua function will be given a Lua number, which matches one of the encapsulations in the Lua `wtap_encaps` table. This might be the `wtap_encap.PER_PACKET` number, meaning the capture contains multiple encapsulation types, and the file reader should only return true if it can handle multiple encap types in one file. The function will then be called again, once for each encap type in the file, to make sure it can write each one.

If the Lua file writer can write the given type of encapsulation into a file, then it returns the boolean true, else false.

**filehandler.write_open**

Mode: Assign only.

The Lua function to be called when Wireshark opens a file for writing.

When later called by Wireshark, the Lua function will be given:

1. A `File` object
2. A `CaptureInfoConst` object

The purpose of the Lua function set to this `write_open` field is similar to the `read_open` callback function: to initialize things necessary for writing the capture to a file. For example, if the output file format has a file header, then the file header should be written within this `write_open` function.

The called Lua function should return true on success, or false if it hit an error.

Also make sure to set the `FileHandler.write` (and potentially `FileHandler.write_finish`) functions before returning true from this function.

**filehandler.write**

Mode: Assign only.

The Lua function to be called when Wireshark wants to write a packet to the file.

When later called by Wireshark, the Lua function will be given:
1. A **File** object

2. A **CaptureInfoConst** object

3. A **FrameInfoConst** object of the current frame/packet to be written

The purpose of the Lua function set to this `write` field is to write the next packet to the file.

The called Lua function should return true on success, or false if it hit an error.

### `filehandler.write_finish`

**Mode:** Assign only.

The Lua function to be called when Wireshark wants to close the written file.

When later called by Wireshark, the Lua function will be given:

1. A **File** object
2. A **CaptureInfoConst** object

It is not necessary to set this field to a Lua function - **FileHandler** can be registered without doing so - it is available in case there is memory/state to clear in your script when the file is closed.

### `filehandler.type`

**Mode:** Retrieve only.

The internal file type. This is automatically set with a new number when the FileHandler is registered.

### `filehandler.extensions`

**Mode:** Retrieve or assign.

One or more semicolon-separated file extensions that this file type usually uses.

For readers using heuristics to determine file type, Wireshark will try the readers of the file's extension first, before trying other readers. But ultimately Wireshark tries all file readers for any file extension, until it finds one that accepts the file.

(Since 2.6) For writers, the first extension is used to suggest the default file extension.

### `filehandler.writing_must_seek`

**Mode:** Retrieve or assign.

True if the ability to seek is required when writing this file format, else false.

This will be checked by Wireshark when writing out to compressed file formats, because seeking is
not possible with compressed files. Usually a file writer only needs to be able to seek if it needs to go back in the file to change something, such as a block or file length value earlier in the file.

**filehandler.writes_name_resolution**

Mode: Retrieve or assign.

True if the file format supports name resolution records, else false.

**filehandler.supported_comment_types**

Mode: Retrieve or assign.

Set to the bit-wise OR’ed number representing the type of comments the file writer supports writing, based on the numbers in the `wtap_comments` table.

**FrameInfo**

A FrameInfo object, passed into Lua as an argument by FileHandler callback functions (e.g., `read`, `seek_read`, etc.).

This object represents frame data and meta-data (data about the frame/packet) for a given `read`/`seek_read/write`'s frame.

This object's fields are written-to/set when used by read function callbacks, and read-from/get when used by file write function callbacks. In other words, when the Lua plugin’s FileHandler `read`/`seek_read/etc.` functions are invoked, a FrameInfo object will be passed in as one of the arguments, and its fields should be written-to/set based on the frame information read from the file; whereas when the Lua plugin’s `FileHandler.write()` function is invoked, the FrameInfo object passed in should have its fields read-from/get, to write that frame information to the file.

**frameinfo:__tostring()**

Generates a string of debug info for the FrameInfo

**Returns**

String of debug information.

**frameinfo:read_data(file, length)**

Tells Wireshark to read directly from given file into frame data buffer, for length bytes. Returns true if succeeded, else false.

**Arguments**
The File object userdata, provided by Wireshark previously in a reading-based callback.

The number of bytes to read from the file at the current cursor position.

Returns
True if succeeded, else returns false along with the error number and string error description.

A Lua string of the frame buffer's data.

Frameinfo.comment
Mode: Retrieve or assign.
Table of comments in this frame.

Frameinfo.time
Mode: Retrieve or assign.
The packet timestamp as an NSTime object.
Note: Set the FileHandler.time_precision to the appropriate wtap_file_tsprec value as well.

Frameinfo.data
Mode: Retrieve or assign.
The data buffer containing the packet.

 NOTE This cannot be cleared once set.

Frameinfo.rec_type
Mode: Retrieve or assign.
The record type of the packet frame
See wtap_rec_types for values.

Frameinfo.flags
Mode: Retrieve or assign.
The presence flags of the packet frame.
See `wtap_presence_flags` for bit values.

**frameinfo.captured_length**

Mode: Retrieve or assign.

The captured packet length, and thus the length of the buffer passed to the `FrameInfo.data` field.

**frameinfo.original_length**

Mode: Retrieve or assign.

The on-the-wire packet length, which may be longer than the `captured_length`.

**frameinfo.encap**

Mode: Retrieve or assign.

The packet encapsulation type for the frame/packet, if the file supports per-packet types. See `wtap_encaps` for possible packet encapsulation types to use as the value for this field.

**FrameInfoConst**

A constant `FrameInfo` object, passed into Lua as an argument by the FileHandler write callback function. This has similar attributes/properties as `FrameInfo`, but the fields can only be read from, not written to.

**frameinfoconst:__tostring()**

Generates a string of debug info for the FrameInfo

**Returns**

String of debug information.

**frameinfoconst:write_data(file, [length])**

Tells Wireshark to write directly to given file from the frame data buffer, for length bytes. Returns true if succeeded, else false.

**Arguments**

**file**

The File object userdata, provided by Wireshark previously in a writing-based callback.

**length (optional)**

The number of bytes to write to the file at the current cursor position, or all if not supplied.
Returns

True if succeeded, else returns false along with the error number and string error description.

frameinfoconst.comment

Mode: Retrieve only.

The first string comment for the packet, if any; nil if there is no comment.

frameinfoconst.time

Mode: Retrieve only.

The packet timestamp as an NSTime object.

frameinfoconst.data

Mode: Retrieve only.

The data buffer containing the packet.

frameinfoconst.rec_type

Mode: Retrieve only.

The record type of the packet frame - see wtap_presence_flags for values.

frameinfoconst.flags

Mode: Retrieve only.

The presence flags of the packet frame - see wtap_presence_flags for bits.

frameinfoconst.captured_length

Mode: Retrieve only.

The captured packet length, and thus the length of the buffer in the FrameInfoConst.data field.

frameinfoconst.original_length

Mode: Retrieve only.

The on-the-wire packet length, which may be longer than the captured_length.

frameinfoconst.encap

Mode: Retrieve only.
The packet encapsulation type, if the file supports per-packet types.

See `wtap_encaps` for possible packet encapsulation types to use as the value for this field.

**Global Functions**

`register_filehandler(filehandler)`

Register the FileHandler into Wireshark/TShark, so they can read/write this new format. All functions and settings must be complete before calling this registration function. This function cannot be called inside the reading/writing callback functions.

**Arguments**

`filehandler`  
The FileHandler object to be registered

**Returns**

the new type number for this file reader/write

`deregister_filehandler(filehandler)`

Deregister the FileHandler from Wireshark/TShark, so it no longer gets used for reading/writing/display. This function cannot be called inside the reading/writing callback functions.

**Arguments**

`filehandler`  
The FileHandler object to be deregistered

**Directory Handling Functions**

**Dir**

A Directory object, as well as associated functions.

**Dir.make(name)**

Creates a directory.

The created directory is set for permission mode 0755 (octal), meaning it is read+write+execute by owner, but only read+execute by group members and others.

If the directory was created successfully, a boolean `true` is returned. If the directory cannot be made
because it already exists, \texttt{false} is returned. If the directory cannot be made because an error occurred, \texttt{nil} is returned.

**Arguments**

\texttt{name}

The name of the directory, possibly including path.

**Returns**

Boolean \texttt{true} on success, \texttt{false} if the directory already exists, \texttt{nil} on error.

\texttt{Dir.exists(name)}

Returns true if the given directory name exists.

If the directory exists, a boolean \texttt{true} is returned. If the path is a file instead, \texttt{false} is returned. If the path does not exist or an error occurred, \texttt{nil} is returned.

**Arguments**

\texttt{name}

The name of the directory, possibly including path.

**Returns**

Boolean \texttt{true} if the directory exists, \texttt{false} if it's a file, \texttt{nil} on error or not-exist.

\texttt{Dir.remove(name)}

Removes an empty directory.

If the directory was removed successfully, a boolean \texttt{true} is returned. If the directory cannot be removed because it does not exist, \texttt{false} is returned. If the directory cannot be removed because an error occurred, \texttt{nil} is returned.

This function only removes empty directories. To remove a directory regardless, use \texttt{Dir.remove_all()}.

**Arguments**

\texttt{name}

The name of the directory, possibly including path.

**Returns**

Boolean \texttt{true} on success, \texttt{false} if does not exist, \texttt{nil} on error.
Dir.remove_all(name)

Removes an empty or non-empty directory.

If the directory was removed successfully, a boolean `true` is returned. If the directory cannot be removed because it does not exist, `false` is returned. If the directory cannot be removed because an error occurred, `nil` is returned.

**Arguments**

- **name**
  
  The name of the directory, possibly including path.

**Returns**

- Boolean `true` on success, `false` if does not exist, `nil` on error.

Dir.open(pathname, [extension])

Opens a directory and returns a `Dir` object representing the files in the directory.

**Example**

```ruby
-- Print the contents of a directory
for filename in Dir.open('/path/to/dir') do
  print(filename)
end
```

**Arguments**

- **pathname**
  
  The pathname of the directory.

- **extension (optional)**
  
  If given, only files with this extension will be returned.

**Returns**

- The `Dir` object.

Dir.personal_config_path([filename])

Gets the personal configuration directory path, with filename if supplied.

**Arguments**
filename (optional)
   A filename.

**Returns**
The full pathname for a file in the personal configuration directory.

**Dir.global_config_path([filename])**
Gets the global configuration directory path, with filename if supplied.

**Arguments**
filename (optional)
   A filename

**Returns**
The full pathname for a file in Wireshark's configuration directory.

**Dir.personal_plugins_path()**
Gets the personal plugins directory path.

**Returns**
The pathname of the personal plugins directory.

**Dir.global_plugins_path()**
Gets the global plugins directory path.

**Returns**
The pathname of the global plugins directory.

**dir:__call()**
Gets the next file or subdirectory within the directory, or **nil** when done.

**Example**

```lua
-- Open a directory and print the name of the first file or subdirectory
local dir = Dir.open('/path/to/dir')
local first = dir()
print(tostring(file))
```
dir:close()

Closes the directory. Called automatically during garbage collection of a `Dir` object.

## Handling 64-bit Integers

Lua uses one single number representation which can be chosen at compile time and since it is often set to IEEE 754 double precision floating point, one cannot store 64 bit integers with full precision.

Lua numbers are stored as floating point (doubles) internally, not integers; thus while they can represent incredibly large numbers, above $2^{53}$ they lose integral precision — they can’t represent every whole integer value. For example if you set a lua variable to the number 9007199254740992 and tried to increment it by 1, you’d get the same number because it can’t represent 9007199254740993 (only the even number 9007199254740994).

Therefore, in order to count higher than $2^{53}$ in integers, we need a true integer type. The way this is done is with an explicit 'Int64' or 'UInt64' object (i.e., Lua userdata). This object has metamethods for all of the math and comparison operators, so you can handle it like any number variable. For the math operators, it can even be mixed with plain Lua numbers.

For example 'my64num = my64num + 1' will work even if 'my64num' is a `Int64` or `UInt64` object. Note that comparison operators (‘=’, ‘<’, ‘>’, etc.) will not work with plain numbers — only other Int64/UInt64 objects. This is a limitation of Lua itself, in terms of how it handles operator overloading.

### WARNING

Many of the UInt64/Int64 functions accept a Lua number as an argument. You should be very careful to never use Lua numbers bigger than 32 bits (i.e., the number value 4,294,967,295 or the literal 0xFFFFFFFF) for such arguments, because Lua itself does not handle bigger numbers consistently across platforms (32-bit vs. 64-bit systems), and because a Lua number is a C-code double which cannot have more than 53 bits of precision. Instead, use a `Int64` or `UInt64` for the argument.

For example, do this...

```lua
local mynum = UInt64(0x2b89dd1e, 0x3f91df0b)
```

...instead of this:

```lua
-- Bad. Leads to inconsistent results across platforms
local mynum = UInt64(0x3f91df0b2b89dd1e)
```

And do this...
Int64

Int64 represents a 64 bit signed integer.

Note the caveats listed above.

Int64.decode(string, [endian])

Decodes an 8-byte Lua string, using the given endianness, into a new Int64 object.

Arguments

string

The Lua string containing a binary 64-bit integer.

endian (optional)

If set to true then little-endian is used, if false then big-endian; if missing or nil, native host endian.

Returns

The Int64 object created, or nil on failure.

Int64.new([value], [highvalue])

Creates a Int64 object.

Arguments

value (optional)

A number, UInt64, Int64, or string of ASCII digits to assign the value of the new Int64. Default is 0.

highvalue (optional)

If this is a number and the first argument was a number, then the first will be treated as a lower 32 bits, and this is the high-order 32 bit number.

Returns
The new \texttt{Int64} object.

\texttt{Int64.max()}

Creates an \texttt{Int64} of the maximum possible positive value. In other words, this should return an \texttt{Int64} object of the number 9,223,372,036,854,775,807.

\textbf{Returns}

The new \texttt{Int64} object of the maximum value.

\texttt{Int64.min()}

Creates an \texttt{Int64} of the minimum possible negative value. In other words, this should return an \texttt{Int64} object of the number -9,223,372,036,854,775,808.

\textbf{Returns}

The new \texttt{Int64} object of the minimum value.

\texttt{Int64.fromhex(hex)}

Creates an \texttt{Int64} object from the given hexadecimal string.

\textbf{Arguments}

\texttt{hex}

The hex-ASCII Lua string.

\textbf{Returns}

The new \texttt{Int64} object.

\texttt{int64:encode([\text{endian}])}

Encodes the \texttt{Int64} number into an 8-byte Lua string using the given endianness.

\textbf{Arguments}

\texttt{endian (optional)}

If set to true then little-endian is used, if false then big-endian; if missing or \texttt{nil}, native host endian.

\textbf{Returns}

The Lua string.
**int64:__call()**

Creates a Int64 object.

**Returns**

The new Int64 object.

**int64:tonumber()**

Returns a Lua number of the Int64 value. Note that this may lose precision.

**Returns**

The Lua number.

**int64:tohex(numbytes)**

Returns a hexadecimal string of the Int64 value.

**Arguments**

numbytes (optional)

The number of hex chars/nibbles to generate. A negative value generates uppercase. Default is 16.

**Returns**

The string hex.

**int64:higher()**

Returns a Lua number of the higher 32 bits of the Int64 value. A negative Int64 will return a negative Lua number.

**Returns**

The Lua number.

**int64:lower()**

Returns a Lua number of the lower 32 bits of the Int64 value. This will always be positive.

**Returns**

The Lua number.
int64:__tostring()

Converts the Int64 into a string of decimal digits.

Returns

The Lua string.

int64:__unm()

Returns the negative of the Int64 as a new Int64.

Returns

The new Int64.

int64:__add()

Adds two Int64 together and returns a new one. The value may wrapped.

int64:__sub()

Subtracts two Int64 and returns a new one. The value may wrapped.

int64:__mul()

Multiplies two Int64 and returns a new one. The value may truncated.

int64:__div()

Divides two Int64 and returns a new one. Integer divide, no remainder. Trying to divide by zero results in a Lua error.

Returns

The Int64 object.

int64:__mod()

Divides two Int64 and returns a new one of the remainder. Trying to modulo by zero results in a Lua error.

Returns

The Int64 object.

int64:__pow()

The first Int64 is taken to the power of the second Int64, returning a new one. This may truncate the
value.

**Returns**

The `Int64` object.

**int64:__eq()**

Returns `true` if both `Int64` are equal.

**int64:__lt()**

Returns `true` if first `Int64` is less than the second.

**int64:__le()**

Returns `true` if the first `Int64` is less than or equal to the second.

**int64:bnot()**

Returns a `Int64` of the bitwise 'not' operation.

**Returns**

The `Int64` object.

**int64:band()**

Returns a `Int64` of the bitwise 'and' operation with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

**int64:bor()**

Returns a `Int64` of the bitwise 'or' operation, with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

**int64:bxor()**

Returns a `Int64` of the bitwise 'xor' operation, with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

**int64:lshift(numbits)**

Returns a `Int64` of the bitwise logical left-shift operation, by the given number of bits.

**Arguments**
**numbits**

The number of bits to left-shift by.

**Returns**

The `Int64` object.

`int64:rshift(numbits)`

Returns a `Int64` of the bitwise logical right-shift operation, by the given number of bits.

**Arguments**

**numbits**

The number of bits to right-shift by.

**Returns**

The `Int64` object.

`int64:arshift(numbits)`

Returns a `Int64` of the bitwise arithmetic right-shift operation, by the given number of bits.

**Arguments**

**numbits**

The number of bits to right-shift by.

**Returns**

The `Int64` object.

`int64:rol(numbits)`

Returns a `Int64` of the bitwise left rotation operation, by the given number of bits (up to 63).

**Arguments**

**numbits**

The number of bits to roll left by.

**Returns**

The `Int64` object.
**int64:ror(numbits)**

Returns a Int64 of the bitwise right rotation operation, by the given number of bits (up to 63).

**Arguments**

**numbits**

The number of bits to roll right by.

**Returns**

The Int64 object.

**int64:bswap()**

Returns a Int64 of the bytes swapped. This can be used to convert little-endian 64-bit numbers to big-endian 64 bit numbers or vice versa.

**Returns**

The Int64 object.

**UInt64**

**UInt64** represents a 64 bit unsigned integer, similar to Int64.

Note the caveats listed above.

**UInt64.decode(string, [endian])**

Decodes an 8-byte Lua binary string, using given endianness, into a new UInt64 object.

**Arguments**

**string**

The Lua string containing a binary 64-bit integer.

**endian (optional)**

If set to true then little-endian is used, if false then big-endian; if missing or nil, native host endian.

**Returns**

The UInt64 object created, or nil on failure.

**UInt64.new([value], [highvalue])**

Creates a UInt64 object.
Arguments

value (optional)
A number, UInt64, Int64, or string of digits to assign the value of the new UInt64. Default is 0.

highvalue (optional)
If this is a number and the first argument was a number, then the first will be treated as a lower 32 bits, and this is the high-order 32-bit number.

Returns
The new UInt64 object.

UInt64.max()
Creates a UInt64 of the maximum possible value. In other words, this should return an UInt64 object of the number 18,446,744,073,709,551,615.

Returns
The maximum value.

UInt64.min()
Creates a UInt64 of the minimum possible value. In other words, this should return an UInt64 object of the number 0.

Returns
The minimum value.

UInt64.fromhex(hex)
Creates a UInt64 object from the given hex string.

Arguments
hex
The hex-ASCII Lua string.

Returns
The new UInt64 object.

uint64:encode([endian])
Encodes the UInt64 number into an 8-byte Lua binary string, using given endianness.
Arguments

 endian (optional)
    If set to true then little-endian is used, if false then big-endian; if missing or nil, native host endian.

Returns

 The Lua binary string.

uint64:__call()

 Creates a Uint64 object.

Returns

 The new Uint64 object.

uint64:tonumber()

 Returns a Lua number of the Uint64 value. This may lose precision.

Returns

 The Lua number.

uint64:__tostring()

 Converts the Uint64 into a string.

Returns

 The Lua string.

uint64:tohex([numbytes])

 Returns a hex string of the Uint64 value.

Arguments

 numbytes (optional)
    The number of hex-chars/nibbles to generate. Negative means uppercase Default is 16.

Returns

 The string hex.
**uint64:higher()**

Returns a Lua number of the higher 32 bits of the `UInt64` value.

**Returns**

The Lua number.

**uint64:lower()**

Returns a Lua number of the lower 32 bits of the `UInt64` value.

**Returns**

The Lua number.

**uint64:__unm()**

Returns the `UInt64` in a new `UInt64`, since unsigned integers can't be negated.

**Returns**

The `UInt64` object.

**uint64:__add()**

Adds two `UInt64` together and returns a new one. This may wrap the value.

**uint64:__sub()**

Subtracts two `UInt64` and returns a new one. This may wrap the value.

**uint64:__mul()**

Multiplies two `UInt64` and returns a new one. This may truncate the value.

**uint64:__div()**

Divides two `UInt64` and returns a new one. Integer divide, no remainder. Trying to divide by zero results in a Lua error.

**Returns**

The `UInt64` result.

**uint64:__mod()**

Divides two `UInt64` and returns a new one of the remainder. Trying to modulo by zero results in a Lua error.
Returns
The UInt64 result.

uint64:__pow()
The first UInt64 is taken to the power of the second UInt64/number, returning a new one. This may truncate the value.

Returns
The UInt64 object.

uint64:__eq()
Returns true if both UInt64 are equal.

uint64:__lt()
Returns true if first UInt64 is less than the second.

uint64:__le()
Returns true if first UInt64 is less than or equal to the second.

uint64:bnot()
Returns a UInt64 of the bitwise 'not' operation.

Returns
The UInt64 object.

uint64:band()
Returns a UInt64 of the bitwise 'and' operation, with the given number/Int64/UInt64. Note that multiple arguments are allowed.

uint64:bor()
Returns a UInt64 of the bitwise 'or' operation, with the given number/Int64/UInt64. Note that multiple arguments are allowed.

uint64:bxor()
Returns a UInt64 of the bitwise 'xor' operation, with the given number/Int64/UInt64. Note that multiple arguments are allowed.
uint64:lshift(numbits)
Returns a `UInt64` of the bitwise logical left-shift operation, by the given number of bits.

Arguments

numbits
   The number of bits to left-shift by.

Returns

The `UInt64` object.

uint64:rshift(numbits)
Returns a `UInt64` of the bitwise logical right-shift operation, by the given number of bits.

Arguments

numbits
   The number of bits to right-shift by.

Returns

The `UInt64` object.

uint64:arshift(numbits)
Returns a `UInt64` of the bitwise arithmetic right-shift operation, by the given number of bits.

Arguments

numbits
   The number of bits to right-shift by.

Returns

The `UInt64` object.

uint64:rol(numbits)
Returns a `UInt64` of the bitwise left rotation operation, by the given number of bits (up to 63).

Arguments

numbits
   The number of bits to roll left by.
Returns
The UInt64 object.

uint64:ror(numbits)
Returns a UInt64 of the bitwise right rotation operation, by the given number of bits (up to 63).

Arguments
numbits
The number of bits to roll right by.

Returns
The UInt64 object.

uint64:bswap()
Returns a UInt64 of the bytes swapped. This can be used to convert little-endian 64-bit numbers to big-endian 64 bit numbers or vice versa.

Returns
The UInt64 object.

Binary encode/decode support
The Struct class offers basic facilities to convert Lua values to and from C-style structs in binary Lua strings. This is based on Roberto Ierusalimschy's Lua struct library found in http://www.inf.puc-rio.br/~roberto/struct/, with some minor modifications as follows:

- Added support for Int64/UInt64 being packed/unpacked, using 'e'/E'.
- Can handle 'long long' integers (i8 / I8); though they're converted to doubles.
- Can insert/specify padding anywhere in a struct. ('X' eg. when a string is following a union).
- Can report current offset in both pack and unpack ('=').
- Can mask out return values when you only want to calculate sizes or unmarshal pascal-style strings using '{' & '}'.

All but the first of those changes are based on an email from Flemming Madsen, on the lua-users mailing list, which can be found here.

The main functions are Struct.pack, which packs multiple Lua values into a struct-like Lua binary string; and Struct.unpack, which unpacks multiple Lua values from a given struct-like Lua binary string. There are some additional helper functions available as well.
All functions in the Struct library are called as static member functions, not object methods, so they are invoked as "Struct.pack(...)", instead of "object:pack(...)".

The first argument to several of the Struct functions is a format string, which describes the layout of the structure. The format string is a sequence of conversion elements, which respect the current endianness and the current alignment requirements. Initially, the current endianness is the machine's native endianness and the current alignment requirement is 1 (meaning no alignment at all). You can change these settings with appropriate directives in the format string.

The supported elements in the format string are as follows:

- ` ` (empty space) ignored.
- `!n` flag to set the current alignment requirement to 'n' (necessarily a power of 2); an absent 'n' means the machine's native alignment.
- `>` flag to set mode to big endian (i.e., network-order).
- `<` flag to set mode to little endian.
- `x` a padding zero byte with no corresponding Lua value.
- `b` a signed char.
- `B` an unsigned char.
- `h` a signed short (native size).
- `H` an unsigned short (native size).
- `l` a signed long (native size).
- `L` an unsigned long (native size).
- `T` a size_t (native size).
- `in` a signed integer with 'n' bytes. An absent 'n' means the native size of an int.
- `In` like `in` but unsigned.
- `e` signed 8-byte Integer (64-bits, long long), to/from a Int64 object.
- `E` unsigned 8-byte Integer (64-bits, long long), to/from a UInt64 object.
- `f` a float (native size).
- `d` a double (native size).
- `s` a zero-terminated string.
- `cn` a sequence of exactly 'n' chars corresponding to a single Lua string. An absent 'n' means 1. When packing, the given string must have at least 'n' characters (extra characters are discarded).
- `c0` this is like `cn`, except that the 'n' is given by other means: When packing, 'n' is the length of the given string; when unpacking, 'n' is the value of the previous unpacked value (which must be a number). In that case, this previous value is not returned.
• `xn` pad to `n` number of bytes, default 1.
• `Xn` pad to `n` alignment, default MAXALIGN.
• `(' to stop assigning items, and `)` start assigning (padding when packing).
• `=' to return the current position / offset.

NOTE
Using `i`, `I`, `h`, `H`, `l`, `L`, `f`, and `T` is strongly discouraged, as those sizes are system-dependent. Use the explicitly sized variants instead, such as `i4` or `E`.

Unpacking of `i/I` is done to a Lua number, a double-precision floating point, so unpacking a 64-bit field (`i8/I8`) will lose precision. Use `e/E` to unpack into a Wireshark `Int64/UInt64` object instead.

Struct

Struct.pack(format, value)

Returns a string containing the values arg1, arg2, etc. packed/encoded according to the format string.

Arguments

format
  The format string

value
  One or more Lua value(s) to encode, based on the given format.

Returns

The packed binary Lua string, plus any positions due to `=' being used in format.

Struct.unpack(format, struct, [begin])

Unpacks/decodes multiple Lua values from a given struct-like binary Lua string. The number of returned values depends on the format given, plus an additional value of the position where it stopped reading is returned.

Arguments

format
  The format string

struct
  The binary Lua string to unpack
begin (optional)
The position to begin reading from (default=1)

Returns
One or more values based on format, plus the position it stopped unpacking.

Struct.size(format)
Returns the length of a binary string that would be consumed/handled by the given format string.

Arguments
format
  The format string

Returns
The size number

Struct.values(format)
Returns the number of Lua values contained in the given format string. This will be the number of returned values from a call to Struct.unpack() not including the extra return value of offset position. (i.e., Struct.values() does not count that extra return value) This will also be the number of arguments Struct.pack() expects, not including the format string argument.

Arguments
format
  The format string

Returns
The number of values

Struct.tohex(bytestring, [lowercase], [separator])
Converts the passed-in binary string to a hex-ascii string.

Arguments
bytestring
  A Lua string consisting of binary bytes

lowercase (optional)
  True to use lower-case hex characters (default=false).
**separator (optional)**

A string separator to insert between hex bytes (default=nil).

**Returns**

The Lua hex-ascii string

**Struct.fromhex(hexbytes, [separator])**

Converts the passed-in hex-ascii string to a binary string.

**Arguments**

**hexbytes**

A string consisting of hexadecimal bytes like "00 B1 A2" or "1a2b3c4d"

**separator (optional)**

A string separator between hex bytes/words (default none).

**Returns**

The Lua binary string

---

**PCRE2 Regular Expressions**

Lua has its own native *pattern* syntax in the string library, but sometimes a real regex engine is more useful. Wireshark comes with Perl Compatible Regular Expressions version 2 (PCRE2). This engine is exposed into Wireshark’s Lua engine through the well-known Lrexlib library. The module is loaded in the global environment using the "rex_pcre2" table. The manual is available at [https://rrthomas.github.io/lrexlib/manual.html](https://rrthomas.github.io/lrexlib/manual.html).
User Interface

Introduction

Wireshark can be logically separated into the backend (dissecting protocols, file loading and saving, capturing, etc.) and the frontend (the user interface).

The following frontends are currently maintained by the Wireshark development team:

- Wireshark, Qt based
- TShark, console based

This chapter is focused on the Wireshark frontend, and especially on the Qt interface.

The Qt Application Framework

Qt is a cross-platform application development framework. While we mainly use the core (QtCore) and user interface (QtWidgets) modules, it also supports a number of other modules for specialized application development, such as networking (QtNetwork) and web browsing (QtWebKit).

At the time of this writing (September 2016) most of the main Wireshark application has been ported to Qt. The sections below provide an overview of the application and tips for Qt development in our environment.

User Experience Considerations

When creating or modifying Wireshark try to make sure that it will work well on Windows, macOS, and Linux. See Human Interface Reference Documents for details. Additionally, try to keep the following in mind:

**Workflow.** Excessive navigation and gratuitous dialogs should be avoided or reduced. For example, compared to the legacy UI many alert dialogs have been replaced with status bar messages. Statistics dialogs are displayed immediately instead of requiring that options be specified.

**Discoverability and feedback.** Most users don’t like to read documentation and instead prefer to learn an application as they use it. Providing feedback increases your sense of control and awareness, and makes the application more enjoyable to use. Most of the Qt dialogs provide a “hint” area near the bottom which shows useful information. For example, the “Follow Stream” dialog shows the packet corresponding to the text under the mouse. The profile management dialog shows a clickable path to the current profile. The main welcome screen shows live interface traffic. Most dialogs have a context menu that shows keyboard shortcuts.
Qt Creator

Qt Creator is a full-featured IDE and user interface editor. It makes adding new UI features much easier. It doesn’t work well on Windows at the present time, so it’s recommended that you use it on macOS or Linux.

To edit and build Wireshark using Qt Creator, open the top-level CMakeLists.txt within Qt Creator. It should ask you to choose a build location. Do so. It should then ask you to run CMake. Fill in any desired build arguments (e.g. `-D CMAKE_BUILD_TYPE=Debug` or `-D ENABLE_CCACHE=ON`) and click the [Run CMake] button. When that completes select Build › Open Build and Run Kit Selector… and make sure wireshark is selected.

Note that Qt Creator uses output created by CMake’s “CodeBlocks” generator. If you run CMake outside of Qt Creator you should use the “CodeBlocks - Unix Makefiles” generator, otherwise Qt Creator will prompt you to re-run CMake.

Source Code Overview

Wireshark's main entry point is in `ui/qt/main.cpp`. Command-line arguments are processed there and the main application class (WiresharkApplication) instance is created there along with the main window.

The main window along with the rest of the application resides in `ui/qt`. Due to its size the main window code is split into several modules, `main_window.cpp`, `wireshark_main_window.cpp` and `wireshark_main_window_slots.cpp`.

Most of the modules in `ui/qt` are dialogs. Although we follow Qt naming conventions for class names, we follow our own conventions by separating file name components with underscores. For example, ColoringRulesDialog is defined in `coloring_rules_dialog.cpp`, `coloring_rules_dialog.h`, and `coloring_rules_dialog.ui`.

General-purpose dialogs are subclasses of QDialog. Dialogs that rely on the current capture file can subclass WiresharkDialog, which provides methods and members that make it easier to access the capture file and to keep the dialog open when the capture file closes.

Coding Practices and Naming Conventions

Names

The code in `ui/qt` directory uses three APIs: Qt (which uses upper camel case), GLib (which uses `snake_case`), and the Wireshark API (which also uses `snake_case`).

As a general rule, for names, Wireshark’s Qt code:

- uses upper camel case, in which words in the name are not separated by underscores, and the first letter of each word is capitalized, for classes, for example, `PacketList`;
• uses lower camel case, in which words in the name are not separated by underscores, and the first letter of each word other than the first word is capitalized, for methods, for example, `resetColumns`;

• uses snake case, in which words in the name are separated by underscores, and the first letter of the word is not capitalized, for variables, with a trailing underscore used for member variables, for example, `packet_list_model_`.

**Dialogs**

Dialogs that work with capture file information shouldn’t close just because the capture file closes. Subclassing `WiresharkDialog` as described above can make it easier to persist across capture files.

When you create a window with a row of standard “OK” and “Close” buttons at the bottom using Qt Creator you will end up with a subclass of `QDialog`. This is fine for traditional modal dialogs, but many times the “dialog” needs to behave like a `QWindow` instead.

Modal dialogs should be constructed with `QDialog(parent)`. Modeless dialogs (windows) should be constructed with `QDialog(NULL, Qt::Window)`. Other combinations (particularly `QDialog(parent, Qt::Window)`) can lead to odd and inconsistent behavior. Again, subclassing `WiresharkDialog` will take care of this for you.

Most of the dialogs in `ui/qt` share many similarities, including method names, widget names, and behavior. Most dialogs should have the following, although it’s not strictly required:

• An `updateWidgets()` method, which enables and disables widgets depending on the current state and constraints of the dialog. For example, the Coloring Rules dialog disables the `Save` button if the user has entered an invalid display filter.

• A `hintLabel()` widget subclassed from `QLabel` or `ElidedLabel`, placed just above the dialog button box. The hint label provides guidance and feedback to the user.

• A context menu (`ctx_menu_`) for additional actions not present in the button box.

• If the dialog box contains a `QTreeWidget` you might want to add your own `QTreeWidgetItem` subclass with the following methods:

  `drawData()`

  Draws column data with any needed formatting.

  `colData()`

  Returns the data for each column as a `QVariant`. Used for copying as CSV, YAML, etc.

  `operator<()`

  Allows sorting columns based on their raw data.
Strings

Wireshark’s C code and GLib use UTF-8 encoded character arrays. Qt (specifically QString) uses UTF-16. You can convert a char * to a QString using simple assignment. You can convert a QString to a const char * using qUtf8Printable.

If you’re using GLib string functions or plain old C character array idioms in Qt-only code you’re probably doing something wrong, particularly if you’re manually allocating and releasing memory. QStrings are generally much safer and easier to use. They also make translations easier.

If you need to pass strings between Qt and GLib you can use a number of convenience routines which are defined in ui/qt/utils/qt_ui_utils.h.

If you’re calling a function that returns wmem-allocated memory it might make more sense to add a wrapper function to qt_ui_utils than to call wmem_free in your code.

Mixing C and C++

Sometimes we have to call C++ functions from one of Wireshark’s C callbacks and pass C++ objects to or from C. Tap listeners are a common example. The C++ FAQ describes how to do this safely.

Tapping usually involves declaring static methods for callbacks, passing this as the tap data.

Internationalization and Translation

Qt provides a convenient method for translating text: QObject::tr(), usually available as tr().

However, please avoid using tr() for static strings and define them in *.ui files instead. tr() on manually created objects like QMenu are not automatically retranslated and must instead be manually translated using changeEvent() and retranslateUi(). See ui/qt/wireshark_main_window.cpp for an example of this.

**NOTE** If your object life is short and your components are (re)created dynamically then it is ok to use tr().

In most cases you should handle the changeEvent in order to catch QEvent::LanguageChange.

Qt makes translating the Wireshark UI into different languages easy. To add a new translation, do the following:

- Add your translation (ui/qt/wireshark_XX.ts) to ui/qt/CMakeLists.txt
- (Recommended) Add a flag image for your language in resources/languages/XX.svg. Update resources/languages/languages.qrc accordingly.
- Run lupdate ui/qt -ts ui/qt/wireshark_XX.ts to generate/update your translation file.
- Add ui/qt/wireshark_XX.ts to .tx/config.
- Translate with Qt Linguist: linguist ui/qt/wireshark_XX.ts.
• Do a test build and make sure the generated `wireshark_XX.qm` binary file is included.

• Push your translation to GitLab for review. See [Contribute Your Changes](#) for details.

Alternatively, you can put your QM and flag files in the `languages` directory in the Wireshark user configuration directory (either `$XDG_CONFIG_HOME/wireshark/languages/` or `$HOME/.wireshark/languages/` on UNIX).

For more information about Qt Linguist see its manual.

You can also manage translations online with Transifex. Translation resources are organized by type of translation and development branch:

**master**
Qt Linguist resources in the `ui/qt` in the master branch.

**debian**
GNU gettext resources in the `debian` directory in the master branch.

**qt-XY, master-XY**
Qt Linguist resources in the `ui/qt` in the `X.Y` release branch. For example, qt-34 matches the Wireshark 3.2 release branch.

**po-XY, debian-XY**
GNU gettext (.po) resources in the `debian` directory in the `X.Y` release branch. For example, po-34 matches the Wireshark 3.4 release branch.

Each week translations are automatically synchronized with the source code through the following steps:

• Pull changes from Transifex by running `tx pull -f`.

• Run `lupdate` on the ts files.

• Push and commit on GitLab.

• Push changes to Transifex by running `tx push`.

**Colors And Themes**

Qt provides a number of colors via the `QPalette` class. Use this class when you need a standard color provided by the underlying operating system.

Wireshark uses an extended version of the Tango Color Palette for many interface elements that require custom colors. This includes the I/O graphs, sequence diagrams, and RTP streams. Please use this palette (defined in `tango_colors.h` and the `ColorUtils` class) if `QPalette` doesn't meet your needs.

Wireshark supports dark themes (aka “dark mode”) on some platforms. We leverage Qt’s dark
theme support when possible, but have implemented our own support and workarounds in some cases. You can ensure that your code includes proper dark theme support by doing the following:

• You can use a macOS-style template icon by creating a monochrome SVG document with "template" appended to the name, e.g. resources/stock_icons/24x24/edit-find.template.svg.

• Qt draws unvisited links Qt::blue no matter what. You can work around this by using ColorUtils::themelinkBrush() and ColorUtils::themelinkStyle().

• You can catch dark and light mode changes by handling QEvent::ApplicationPaletteChange.

Other Issues and Information

The main window has many QActions which are shared with child widgets. See ui/qt/proto_tree.cpp for an example of this.

To demonstrate the functionality of the plugin interface options, a demonstration plugin exists (pluginifdemo). See doc/README.plugins and plugins/epan/pluginifdemo.

GammaRay lets you inspect the internals of a running Qt application similar to Spy++ on Windows.

Human Interface Reference Documents

Wireshark runs on a number of platforms, primarily Windows, macOS, and Linux. It should conform to the Windows, macOS, GNOME, and KDE human interface guidelines as much as possible. Unfortunately, creating a feature that works well across these platforms can sometimes be a juggling act since the human interface guidelines for each platform often contradict one another. If you run into trouble you can ask the wireshark-dev mailing list as well as the User Experience Stack Exchange listed below.

For further reference, see the following:

• Android Design: https://developer.android.com/design/. Wireshark doesn’t have a mobile frontend (not yet, at least) but there is still useful information here.

• GNOME Human Interface Guidelines: https://developer.gnome.org/hig/

• KDE Human Interface Guidelines: https://hig.kde.org


• Design guidelines for the Windows desktop: https://docs.microsoft.com/en-us/windows/desktop/uxguide/guidelines

• User Experience Stack Exchange: https://ux.stackexchange.com/
Wireshark Tests

The Wireshark sources include a collection of Python scripts that test the features of Wireshark, TShark, Dumpcap, and other programs that accompany Wireshark. These are located in the test directory of the Wireshark source tree.

The command line options of Wireshark and its companion command line tools are numerous. These tests help to ensure that we don’t introduce bugs as Wireshark grows and evolves.

Quick Start

The recommended steps to prepare for and run tests with a UN*X toolchain:

• Install two Python packages, pytest: `pip install pytest pytest-xdist`  
• Build programs (“wireshark”, “tshark”, etc.): `ninja`  
• Build additional programs for the “unittests” suite: `ninja test-programs`  
• Run tests in the build directory: `pytest`

Replace `ninja` by `make` as needed.

If building with Microsoft Visual Studio the analogous steps are:

• Install pytest Python packages: `python -m pip install pytest pytest-xdist`  
• Build programs: `msbuild /m /p:Configuration=RelWithDebInfo Wireshark.sln`  
• Build test-programs: `msbuild /m /p:Configuration=RelWithDebInfo test-programs.vcxproj`  
• Run tests: `python -m pytest`

**TIP**  Depending on your PATH, you may need to run the pytest module as a script from your Python interpreter, e.g, `python -m pytest` or `python3 -m pytest` instead of `pytest`.

The test suite will attempt to test as much as possible and skip tests when its dependencies are not satisfied. For example, packet capture tests require a Loopback interface and capture privileges. To avoid capture tests, pass the `--disable-capture` option.

List available tests with `pytest --collectonly`. Enable verbose output with `pytest --verbose`. For more details, see Listing And Running Tests.

You can also run the "ninja test" target instead of invoking pytest directly. This will automatically build the test programs dependency, so it may be preferred for that reason.

Test suite structure

The following sections describes how the test suite is organized.
Test Coverage And Availability

The testing framework can run programs and check their stdout, stderr, and exit codes. It cannot interact with the Wireshark UI. Tests cover capture, command line options, decryption, file format support and conversion, Lua scripting, and other functionality.

Available tests depend on the libraries with which Wireshark was built. For example, some decryption tests depend on a minimum version of Libgcrypt and Lua tests depend on Lua.

Capture tests depend on the permissions of the user running the test script. We assume that the test user has capture permissions on Windows and macOS and capture tests are enabled by default on those platforms.

**TIP**

Build the "test-capture" target on Linux (using sudo) to set dumpcap permissions and enable capture tests.

If a feature is unavailable, the test will be skipped. For example, if an old version of Libgcrypt is in use, then some decryption tests will be skipped while other tests can still run to completion.

Suites, Cases, and Tests

The test suite uses pytest as a test runner. Tests are organized according to suites, cases, and individual tests. Suites correspond to Python modules that match the pattern “suite_*.py”. Cases correspond to one or more classes in each module, and case class methods matching the pattern "test_" correspond to individual tests. For example, the invalid capture filter test in the TShark capture command line options test case in the command line options suite has the ID “suite_clopts.py::TestTsharkCaptureClopts::test_tshark_invalid_capfilter”.

pytest fixtures

A test has typically additional dependencies, like the path to an executable, the path to a capture file, a configuration directory, the availability of an optional library, and so on.

pytest is a test framework which has full parallelization support (test-level instead of just suite-level), provides nice test reports, and allows modular fixtures.

A fixture is a function decorated with @pytest.fixture and can either call pytest.skip("reason") to skip tests that depend on the fixture, or return/yield a value. Test functions (and other fixture functions) can receive the fixture value by using the name of the fixture function as function parameters. Common fixtures are available in fixtures_ws.py and includes cmd_tshark for the path to the tshark executable and capture_file for a factory function that produces the path to a capture file.
Listing And Running Tests

Tests are run with pytest. Pytest features versus the "unittest" standard library module include finer test selection, full parallelism, nicer test execution summaries, better output in case of failures (containing the contents of variables) and the ability to open the PDB debugger on failing tests.

To get started, install pytest 3.0 or newer and pytest-xdist:

```
# Install required packages on Ubuntu 18.04 or Debian jessie-backports
$ sudo apt install python3-pytest python3-pytest-xdist

# Install required packages on other systems
$ pip install pytest pytest-xdist
```

Run pytest in the Wireshark build directory, Wireshark binaries are assumed to be present in the run subdirectory (or run\RelWithDebInfo on Windows).

```
# Run all tests
$ cd /path/to/wireshark/build
$ pytest

# Run all except capture tests
$ pytest --disable-capture

# Run all tests with "decryption" in its name
$ pytest -k decryption

# Run all tests with an explicit path to the Wireshark executables
$ pytest --program-path /path/to/wireshark/build/run
```

To list tests without actually executing them, use the --collect-only option:

```
# List all tests
$ pytest --collect-only

# List only tests containing both "dfilter" and "tvb"
$ pytest --collect-only -k "dfilter and tvb"
```

The test suite will fail tests when programs are missing. When only a subset of programs are built or when some programs are disabled, then the test suite can be instructed to skip instead of fail tests:
# Run tests when libpcap support is disabled (-DENABLE_PCAP=OFF)
$ pytest --skip-missing-programs dumpcap,rawshark

# Run tests and ignore all tests with missing program dependencies
$ pytest --skip-missing-programs all

To open a Python debugger (PDB) on failing tests, use the `--pdb` option and disable parallelism with the `-n0` option:

# Run decryption tests sequentially and open a debugger on failing tests
$ pytest -n0 --pdb -k decryption

## Adding Or Modifying Tests

Tests must be in a Python module whose name matches “suite_*.py”. The module must contain one or more subclasses with a name starting with "Test" something, for example "class TestDissectionHttp2:". Each test case method whose name starts with “test_” constitutes an individual test.

Success or failure conditions are signalled using regular assertions with the "assert" Python keyword.

Test dependencies (such as programs, directories, or the environment variables) are injected through method parameters. Commonly used fixtures include `cmd_tshark` and `capture_file`.

Processes (tshark, capinfos, etc.) are run using the "subprocess" Python module, or the Wireshark `subprocesstest` module with some convenience functions. Possible functions include `subprocesstest.run()`, `subprocesstest.check_run()` or creating `subprocess.Popen` object if the utility functions are not sufficient for some reason. Usually this is only required if two-way communication is performed with the child process. `subprocesstest.check_run()` is exactly the same as calling `subprocesstest.run()` with `check=True` as an argument, only a bit more expressive.

Check the documentation for the Python subprocess module for a full description of the arguments available to the `subprocesstest.run()` convenience wrapper and the `subprocess.Popen` object.

All of the current tests run one or more of Wireshark's suite of executables and either check their return code or their output. A simple example is “suite_clopts.py::TestBasicClopts::test_existing_file”, which reads a capture file using TShark and checks its exit code.
import subprocess
import pytest

class TestBasicClopts:
    def test_existing_file(self, cmd_tshark, capture_file, test_env):
        subprocess.check_run((cmd_tshark, '-r', capture_file('dhcp.pcap')),
                              env=test_env)

        Output can be checked using assert subprocesstest.grep_output(), assert subprocesstest.count_output() or any other assert statement. subprocesstest.check_run() also asserts that the child process returns the value 0 as exit code.

import subprocess
import pytest

class TestDecrypt80211:
    def test_80211_wpa_psk(self, cmd_tshark, capture_file, test_env):
        tshark_proc = subprocess.run((cmd_tshark, '-o', 'wlan.enable_decryption: TRUE',
                                       '-Tfields',
                                       '-e', 'http.request.uri',
                                       '-r', capture_file('wpa-Induction.pcap.gz'),
                                       '-Y', 'http',
                                       ), capture_output=True, env=test_env)
        assert 'favicon.ico' in tshark_proc.stdout

Tests can be run in parallel. This means that any files you create must be unique for each test. Filenames based on the current test name are generated using fixtures such as "capture_file" and "result_file". By default pytest generates paths in the system's temporary directory and the last three pytest runs are kept. Temporary files from older runs are automatically deleted.

**External Tests**

You can create your own Python test files outside of the Wireshark source tree. To include your tests when running the Wireshark test suite, simply add the directory containing your test files to the `pytest` command line. Note that filenames must match the same conventions as discussed above.

In order for your tests to have access to the Wireshark test fixtures, you will need this line in each test file:

```python
from fixtures_ws import *
```
Custom Fixtures

You may wish to define your own test fixtures— for example, a fixture similar to `capture_file` but which gives the path to a file in your external test directory. Here is an example Python file containing such a fixture. It presumes a subdirectory named `extra_captures` which exists in the same directory, and which contains your extra capture files.

```python
# my_fixtures.py
# To use in your own tests, import like so:
#   from my_fixtures import *

from pathlib import Path
import pytest

@ pytest.fixture(scope='session')
def extra_file():
    def resolver(filename):
        return Path(__file__).parent.joinpath("extra_captures", filename)
    return resolver

NOTE
If you give your fixture the same name as an existing Wireshark fixture, any tests using your fixture library will lose access to the Wireshark fixture of the same name. This can lead to confusing behavior and is not recommended.
Creating ASN.1 Dissectors

The asn2wrs compiler can be used to create a dissector from an ASN.1 specification of a protocol. It is a work in progress but has been used to create a number of dissectors.

It supports:

- ITU-T Recommendation X.681 (07/2002), Information technology - Abstract Syntax Notation One (ASN.1): Information object specification
- ITU-T Recommendation X.682 (07/2002), Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification
- ITU-T Recommendation X.683 (07/2002), Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications

It has inbuilt support for:


About ASN.1

The most useful first step in writing an ASN.1-based dissector is to learn about ASN.1. There are a number of free resources available to help with this. One collection of such resources is maintained on the ASN.1 Consortium’s web site.

ASN.1 Dissector Requirements

The compiler needs 4 input files: an ASN.1 description of a protocol, a .cnf file, and two template files. The ASN.1 specification may have to be edited to work, however work is in progress to at least read all ASN1 specifications. Changing the ASN1 file is being deprecated as this creates problems when updating protocols. The H.248 Binary encoding dissector is a good example of a dissector with relatively small changes.

A complete simple ASN1 UDP-based dissector is also available.

Building An ASN.1-Based Plugin

The usual way to build an ASN.1-based dissector is to put it into the epan/dissectors/asn1 subtree. This works well and is somewhat simpler than building as a plugin, but there are two reasons one might want to build as a plugin:

- To speed development, since only the plugin needs to be recompiled.
• To allow flexibility in deploying an updated plugin, since only the plugin needs to be distributed.

Reasons one might not want to build as a plugin:

• The code is somewhat more complex.
• The CMakeFile is quite a bit more complex.
• Building under the asn1 subtree keeps all such dissectors together.

If you still think you'd like to build your module as a plugin, see Building ASN1 Plugins.

Understanding Error Messages

When running asn2wrs, you could get the following errors:

• A LexToken error (main.ParseError: LexToken(DOT, '.', 71)) means that something is not understood in the ASN1 file, line 71, around the dot (.) - can be the dot itself.
• A ParseError (main.ParseError: LexToken(SEMICOLON, ';', 88)) means that the ';' (SEMICOLON) is not understood. Maybe removing it will work?

Hand-Massaging The ASN.1 File

If a portion of your ASN.1 file is unsupported you can modify it by hand as needed. However, the preferred way to resolve the issue is to report it on the wireshark-dev mailing list or in the issue tracker so that asn2wrs can be improved.

Command Line Syntax
ASN.1 to Wireshark dissector compiler

asn2wrs [-h|?] [-d dbg] [-b] [-p proto] [-c cnf_file] [-e] input_file(s) ...

- h|? : Usage
- b : BER (default is PER)
- u : Unaligned (default is aligned)
- p proto : Protocol name (implies -S). Default is module-name
  from input_file (renamed by #.MODULE if present)
- o name : Output files name core (default is <proto>)
- O dir : Output directory for dissector
- c cnf_file : Conformance file
- I path : Path for conformance file includes
- e : Create conformance file for exported types
- E : Just create conformance file for exported types
- -S : Single output for multiple modules
- s template : Single file output (template is input file
  without .c/.h extension)
- k : Keep intermediate files though single file output is used
- L : Suppress #line directive from .cnf file
- D dir : Directory for input_file(s) (default: '.')
- C : Add check for SIZE constraints
- r prefix : Remove the prefix from type names

input_file(s) : Input ASN.1 file(s)

- d dbg : Debug output, dbg = [l][y][p][s][a][t][c][m][o]
  l - lex
  y - yacc
  p - parsing
  s - internal ASN.1 structure
  a - list of assignments
  t - tables
  c - conformance values
  m - list of compiled modules with dependency
  o - list of output files

Generated Files

Asn2wrs creates the following intermediate files:

- packet-proto-ett.c
- packet-proto-ettarr.c
- packet-proto-fn.c
- packet-proto-hf.c
These files should be included in the template file as described in the conformance file examples. Some are optional.

**Step By Step Instructions**

1. Create a directory for your protocol in the `epan/dissectors/asn1` directory and put your ASN.1 file there.
2. Copy `CMakeLists.txt` from another ASN.1 dissector and edit it to suit your needs.
3. Create a .cnf file either by copying an existing one and editing it or using the empty example above.
4. Create template files either by copying suitable existing ones and editing them or use the examples above, putting your protocol name in the appropriate places.
5. Add your dissector to `epan/dissectors/asn1/CMakeLists.txt`
6. Test generating your dissector by building the `generate_dissector-*proto*` target.
7. Depending on the outcome you may have to edit your .cnf file, ASN.1 file etc...
8. Build Wireshark.

**Hints For Using Asn2wrs**

Asn2wrs does not support all of ASN.1 yet. This means you might need to modify the ASN.1 definition before it will compile. This page lists some tips and tricks that might make your life easier.

**COMPONENTS OF**

Asn2wrs does not support the COMPONENTS OF directive. This means that you will have to modify the asd definition to manually remove all COMPONENTS OF directives. Fortunately this is pretty easy. COMPONENTS OF is a directive in ASN.1 which include all specified fields in the referenced SEQUENCE by those fields as if they had been explicitly specified.

**Example**

Assume you have some definition that looks like this:
Foo ::= SEQUENCE {
    field_1 INTEGER,
    field_2 INTEGER
}

Bar ::= SEQUENCE {
    COMPONENTS OF Foo,
    field_3 INTEGER
}

Since Asn2wrs can not handle COMPONENTS OF you will have to modify the ASN.1 file so that instead Bar will look like this:

Bar ::= SEQUENCE {
    field_1 INTEGER,
    field_2 INTEGER,
    field_3 INTEGER
}

That was pretty easy wasn't it?

**Semicolon Characters**

In some ASN1 you may have semicolon characters like this:

\[
\text{PBAddressString ::= SEQUENCE} \{ \text{extension INTEGER(1), natureOfAddressIndicator INTEGER, numberingPlanInd INTEGER, digits OCTET STRING (SIZE(0..19))} \};
\]

You will have to remove the last semicolon character.

**Parameters**

Parameters will have to be replaced too. Something like this:

\[
\text{AChBillingChargingCharacteristics} \{\text{PARAMETERS-BOUND : bound} \} ::= \text{OCTET STRING (SIZE (minAChBillingChargingLength .. maxAChBillingChargingLength))}
\]

Will have to be replaced with the real values of the parameters:
ANY And Parameterized Types

Asn2wrs can handle the type ANY but not parameterized types. Fortunately this is easy to work around with small changes to the ASN file and some conformance file magic. Assuming you have a construct that looks something like this:

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm ALGORITHM.&id({SupportedAlgorithms}),
    parameters ALGORITHM.&Type({SupportedAlgorithms}{@algorithm}) OPTIONAL
}
```

Which is essentially a structure that takes two fields, one field being an object identifier and the second field that can be just about anything, depending on what object identifier was used. Here we just have to rewrite this SEQUENCE slightly so that it looks like this:

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY OPTIONAL
}
```

The only thing remaining now is to add the actual code to manage the dissection of this structure. We do this by using the `#FN_BODY conformance file directive`, which will replace the function body of a dissector with the contents that you specify in the conformance file. For this one we need a string where we store the OID from AlgorithmIdentifier/algorithm so that we can pick it up and act on later from inside the dissector for AlgorithmIdentifier/parameters. So we have to add something like this:

```
static char algorithm_id[64]; /* 64 chars should be enough? */
```

to the template file. Then we add the following to the conformance file:
This example comes from the X509AF dissector. Please see the code there for more examples on how to do this.

**Tagged Assignments**

There is currently a bug in Asn2wrs that makes it generate incorrect code for the case when tagged assignments are used. The bug is two-fold, first the generated code "forgets" to strip of the actual tag and length, second it fails to specify 'implicit_tag' properly.

A tagged assignment is something that looks like this example from the FTAM asn specification:

```
Degree-Of-Overlap ::= [APPLICATION 30] IMPLICIT INTEGER {
  normal(0), consecutive(1), concurrent(2)
}
```

I.e. an assignment that also specifies a tag value.

Until Asn2wrs is enhanced to handle these constructs you MUST add a workaround for it to the conformance file:
This tells Asn2wrs to not autogenerate any code at all for the Degree-Of-Overlap object instead it should use the code specified here. Note that we do have to specify the implicit_tag value explicitly and we can NOT use the parameter passed to the function from the caller (also due to the bug in Asn2wrs) this is the true parameter in the call to dissect_ber_integer(). We specify true here since the definition of Degree-Of-Overlap was using IMPLICIT tags and would have specified false if it was not.

The code above can be easily cut-n-pasted into the conformance file with the exception of the last line that actually calls the next dissector helper (...dissect_ber_integer... in this case). The easiest way to find out exactly what this final line should look like in the conformance file; just generate the dissector first without this workaround and look at what call was generated. Then put that line in the conformance directive and replace implicit_tag with either true or false depending on whether IMPLICIT is used or not.

**Untagged CHOICES**

Asn2wrs cannot handle untagged CHOICEs within either a SET or a SEQUENCE. For example:
The Asn2wrs SET/SEQUENCE parsing only looks one level deep into the dissection tree and does not have access to class/tags of the elements in the CHOICE.

As with COMPONENTS OF, the solution is to expand the CHOICE in-line within the SET or SEQUENCE, but make sure that each element of the CHOICE is marked as OPTIONAL. For example,

```
MessageTransferEnvelope ::= SET {
    ...  
    built-in BuiltInContentType OPTIONAL,  
    extended ExtendedContentType OPTIONAL  
    ...  
}
```

This isn't an entirely correct ASN.1 definition, but should allow successful parsing.

**Imported Module Name Conflicts**

When importing a module using `#.INCLUDE` in the conformance file, this may introduce a definition from the module which contradicts the definition used in the current ASN.1 file. For example, the X.509 Authentication Framework defines Time as

```
Time ::= CHOICE {utcTime UTCTime,
    generalizedTime GeneralizedTime
}
```

whereas X.411 defines Time as
This can lead to failure to decode the ASN.1 as, in the example, Asn2wrs will be passed the wrong attributes when trying to decode an X.411 time. In order to solve this problem, (if you don't want to globally change the conflicting name within the ASN.1 module), then you must add an appropriate `#.TYPE_ATTR` into the conformance file `before` the `#.INCLUDE` line. For example

```plaintext
#.TYPE_ATTR
Time TYPE = FT_STRING DISPLAY = BASE_NONE STRING = NULL BITMASK = 0
```

**Simple ASN.1-Based Dissector**

The following snippets show the different files that make up a dissector for a “FOO” protocol dissector.
FOO protocol dissector
----------------------

This trivial dissector is an example for the struggling dissector developer (me included)
of how to create a dissector for a protocol that is encapsulated in UDP packets
for a specific port, and the packet data is ASN1 PER encoded.

The thing that took me a while to figure out was that in order to see my packet
dissected on the detail pane, I had to:
1. Tell the compiler which block in the ASN1 definition is a PDU definition by adding
   FOO-MESSAGE under the #.PDU directive in the foo.cnf file
2. Add a call to dissect_FOO_MESSAGE_PDU() function in the dissect_foo() function in
   the
   packet-foo-template.c file.

To build and test it:
1. in foo directory, run make
2. run make copy_files
3. add packet-foo.c and packet-foo.h to epan/dissectors/Makefile.common
4. run top level make

CAVEAT: Makefile.nmake was not tested.

You can take it from here :-)

--00--
-- FOO PROTOCOL

--

FOO-PROTOCOL DEFINITIONS AUTOMATIC TAGS ::= BEGIN

-- General definitions

MessageId ::= INTEGER (0..65535)
FlowId ::= INTEGER (0..65535)

MessageData ::= SEQUENCE {
  name OCTET STRING(SIZE(10)),
  value OCTET STRING(SIZE(10))
}

FOO-MESSAGE ::= SEQUENCE {
  messageId MessageId,
  flowId FlowId,
  messageData MessageData
}

END
foo.cnf

# foo.cnf
# FOO conformation file

#.MODULE_IMPORT
#.EXPORTS
#.PDU
FOO-MESSAGE
#.NO_EMIT
#.TYPE_RENAME
#.FIELD_RENAME
#.END

packet-foo-template.h

/* packet-foo.h
 * Routines for foo packet dissection
 *
 * Wireshark - Network traffic analyzer
 * By Gerald Combs <gerald@wireshark.org>
 * Copyright 1998 Gerald Combs
 *
 * SPDX-License-Identifier: GPL-2.0-or-later
 */

#ifndef PACKET_FOO_H
#define PACKET_FOO_H

#endif  /* PACKET_FOO_H */

packet-foo-template.c

/* packet-foo.c
 * Routines for FOO packet dissection
 *
 * Wireshark - Network traffic analyzer
 * By Gerald Combs <gerald@wireshark.org>
 * Copyright 1998 Gerald Combs
 */
/*
 * SPDX-License-Identifier: GPL-2.0-or-later
 */

#include "config.h"

#include <glib.h>
#include <epan/packet.h>
#include <epan/conversation.h>

#include <stdio.h>
#include <string.h>
#include "packet-per.h"
#include "packet-foo.h"

#define PNAME  "FOO Protocol"
#define PSNAME "FOO"
#define PFNAME "foo"
#define FOO_PORT 5001    /* UDP port */
static dissector_handle_t foo_handle;

void proto_reg_handoff_foo(void);
void proto_register_foo(void);

/* Initialize the protocol and registered fields */
static int proto_foo;
static int global_foo_port = FOO_PORT;

#include "packet-foo-hf.c"

/* Initialize the subtree pointers */
static int ett_foo;

#include "packet-foo-ett.c"

#include "packet-foo-fn.c"

static void
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    proto_item     *foo_item = NULL;
    proto_tree     *foo_tree = NULL;
    int             offset = 0;

    /* make entry in the Protocol column on summary display */
    if (check_col(pinfo->cinfo, COL_PROTOCOL))
        col_set_str(pinfo->cinfo, COL_PROTOCOL, PNAME);
    /*
    *  Note: Dissectors should set the length of the packet, otherwise it
    *  won't be correctly displayed. It's the job of the specific dissectors
    *  to determine the length of the packet, either through information
    *  from the dissector handle or the packet type.
    */

    /* set up protocol info */
    proto_tree�� = proto_foo();
    fo}
/* create the foo protocol tree */
if (tree) {
    foo_item = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, ENC_NA);
    foo_tree = proto_item_add_subtree(foo_item, ett_foo);

    dissect_FOO_MESSAGE_PDU(tvb, pinfo, foo_tree);
}

/*--- proto_register_foo ---------------------------------------------*/
void proto_register_foo(void) {

    /* List of fields */
    static hf_register_info hf[] = {
#include "packet-foo-hfarr.c"
    
    /* List of subtrees */
    static int *ett[] = {
        &ett_foo,
#include "packet-foo-ettarr.c"
    
    /* Register protocol */
    proto_foo = proto_register_protocol(PNAME, PSNAME, PFNAME);
    /* Register fields and subtrees */
    proto_register_field_array(proto_foo, hf, array_length(hf));
    proto_register_subtree_array(ett, array_length(ett));

}

/*--- proto_reg_handoff_foo ----------------------------------------*/
void proto_reg_handoff_foo(void)
{
    static bool inited = false;

    if( !inited ) {

        foo_handle = create_dissector_handle(dissect_foo,
                            proto_foo);
        dissector_add("udp.port", global_foo_port, foo_handle);

        inited = true;
    }
}
CMakeLists.txt

set( PROTOCOL_NAME foo )

set( PROTO_OPT )

set( EXT_ASN_FILE_LIST )

set( ASN_FILE_LIST
    Foo.asn )

set( EXTRA_DIST
    ${ASN_FILE_LIST}
    packet-$(PROTOCOL_NAME)-template.c
    $(PROTOCOL_NAME).cnf )

set( SRC_FILES
    ${EXTRA_DIST}
    ${EXT_ASN_FILE_LIST} )

set( A2W_FLAGS )

ASN2WRS()

Conformance (.cnf) Files

The .cnf file tells the compiler what to do with certain things, such as skipping auto generation for some ASN.1 entries. They can contain the following directives:

#.OPT
    Compiler options.

#.MODULE and #.MODULE_IMPORT
    Assign Wireshark protocol name to ASN.1 module name.

#.INCLUDE
    Include another conformance file.
#.EXPORTS

Export type or information object class.

#.PDU, #.PDU_NEW, #.REGISTER, #.REGISTER_NEW, and #.SYNTAX

Create PDU functions and register them optionally to dissector table.

#.CLASS

Declare or define information object class.

#.ASSIGNED_OBJECT_IDENTIFIER

Declare assigned object identifier.

#.TABLE_HDR, #.TABLE_BODY, and #.TABLE_FTR

User tables.

#.OMIT_ASSIGNMENT, #.NO_OMIT_ASSGN, #.OMIT_ALL_ASSIGNMENTS,
#.OMIT_ASSIGNMENTS_EXCEPT, #.OMIT_ALL_TYPE_ASSIGNMENTS,
#.OMIT_TYPE_ASSIGNMENTS_EXCEPT, #.OMIT_ALL_VALUE_ASSIGNMENTS, and
#.OMIT_VALUE_ASSIGNMENTS_EXCEPT

Ignore assignments from ASN.1 source.

#.NO_EMIT and #.USER_DEFINED

See linked text for info.

#.VIRTUAL_ASSGN, #.SET_TYPE, #.MAKE_ENUM, #.MAKE_DEFINES, and
#.ASSIGN_VALUE_TO_TYPE

Unknown.

#.TYPE_RENAME, #.FIELD_RENAME, and #.TF_RENAME

Type/field renaming

#.IMPORT_TAG, #.TYPE_ATTR, #.FIELD_ATTR

Type attributes

#.FN_HDR, #.FN_BODY, #.FN_FTR, and #.FN_PARS

Type function modification

#.END

End of directive

#.END_OF_CNF

End of conformance file
Example .cnf File

```cnf
#.MODULE IMPORT
InformationFramework x509if

#.INCLUDE ../x509if/x509if_exp.cnf

#.EXPORTS +
ObjectName

#.PDU
ObjectName

#.REGISTER
Certificate B "2.5.4.36" "id-at-userCertificate"

#.SYNTAX
ObjectName [FriendlyName]

#.NO_EMIT ONLY_VALS
# this can be used with: [WITH_VALS|WITHOUT_VALS|ONLY_VALS]
# using NO_EMIT NO_VALS means it won't generate value_string array for it
Type1

#.USER DEFINED
Type1 [WITH_VALS|WITHOUT_VALS|ONLY_VALS]

#.TYPE_RENAME

#.FIELD_RENAME

#.TYPE_ATTR Ss-Code TYPE = FT_UINT16 DISPLAY = BASE_HEX STRINGS = VALS(ssCode_vals)
# This entry will change the hf definition from the auto-generated one for Ss-Code ::= OCTET STRING(SIZE(1))

  { &hf_gsm_map_ss_Code,
    { "ss-Code", "gsm_map.ss_Code",
      FT_BYTES, BASE_HEX, NULL, 0, "", HFILL }},

# to:

  { &hf_gsm_map_ss_Code,
    { "ss-Code", "gsm_map.ss_Code",
      FT_UINT16, BASE_HEX, VALS(ssCode_vals), 0, "", HFILL }},
```

304
In the proto_abbr-template.c file the corresponding value string must be inserted. As an example the following would be included in proto_abbr-template.c to define ssCode_vals:

```c
static const value_string ssCode_vals[] = {
    { 0, "ssCodeString 1" },  /* The string for value 0 */
    { 1, "String 2" },        /* String for value 1 */
    { 5, "String for value 5" }, /* Value String 5 */
    { 0, NULL }                /* Null terminated array */
};
```

Note that the NULL value must be the final entry and that the index values need not be consecutive.

Foo is expressed in different ways depending on where you want to insert your code and the ASN.1 code in question.

- Foo
- Foo/foo
- Foo/_item/foo

For Tagged type use:
Example packet-protocol-template.h File

Example template.h file. Replace all PROTOCOL/protocol references with the name of your protocol.
/* packet-protocol.h
 * Routines for Protocol packet dissection
 *
 * $Id$
 *
 * Wireshark - Network traffic analyzer
 * By Gerald Combs <gerald@wireshark.org>
 * Copyright 1998 Gerald Combs
 *
 * SPDX-License-Identifier: GPL-2.0-or-later
 */

#ifndef PACKET_PROTOCOL_H
#define PACKET_PROTOCOL_H

#include "packet-protocol-exp.h"

#endif /* PACKET_PROTOCOL_H */

Example packet-protocol-template.c File

Example template.c file. Replace all PROTOCOL/protocol references with the name of your protocol.

/* packet-protocol.c
 * Routines for PROTOCOL packet dissection
 *
 * $Id$
 *
 * Wireshark - Network traffic analyzer
 * By Gerald Combs <gerald@wireshark.org>
 * Copyright 1998 Gerald Combs
 *
 * SPDX-License-Identifier: GPL-2.0-or-later
 */

#include "config.h"

#include <glib.h>
#include <epan/packet.h>
#include <epan/conversation.h>

#include <stdio.h>
#include <string.h>

#include "packet-ber.h"
#include "packet-protocol.h"
Conformance File Directive Reference

The following directives can be used in conformance (.cnf) files:
#.END

Some of the other directives in the Asn2wrs conformance file consists of multiple lines. The END directive is used to terminate such a directive. All other directives (except #.INCLUDE) automatically act as an implicit #.END directive which is why you will not see many #.END directives in the conformance files for the dissectors shipped with Wireshark.

#.EXPORTS

This directive in the Asn2wrs conformation file is used to export functions for type decoding from the dissector.

Syntax

```plaintext
#.EXPORTS

TypeName [WITH_VALS|WITHOUT_VALS|ONLY_VALS] [WS_VAR] [NO_PROT_PREFIX]
...
#.END
```

Options:

- WITH_VALS (default): Exports dissection function and value string table if present.
- WITHOUT_VALS: Exports only the dissection function.
- ONLY_VALS: Exports only the value string table.
- WS_VAR and WS_VAR_IMPORT: Used for value string table so as it can be exported from libwireshark.dll.
- NO_PROT_PREFIX: - value string table name does not have protocol prefix

Example

```plaintext
#.EXPORTS
NonStandardParameter
RasMessage                      WITH_VALS WS_VAR
H323-UU-PDU/h323-message-body   ONLY_VALS WS_VAR
#.END
```

#.FN_BODY

Sometimes, like when we have ANY types, we might want to replace whatever function body that Asn2wrs generates with code of our own. This is what this directive allows us to do.
Example: ANY

Asn2wrs can handle the type ANY but we have to help it by adding some small changes to the conformance file. Assuming you have a construct that looks something like this:

```asn1
AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY OPTIONAL
}
```

To handle this we need to specify our own function bodies to both the algorithm and the parameters fields, which we do using the `#.FN_BODY` directive.

This particular example also requires us to keep some state between the two field dissectors, namely the OBJECT IDENTIFIER from algorithm which specifies what the content of parameters is. For this one we need a string where we store the oid from AlgorithmIdentifier/algorithm so that we can pick it up and act on later from inside the dissector for AlgorithmIdentifier/parameters. So we have to add:

```c
static char algorithm_id[64]; /* 64 chars should be enough? */
```

to the template file as a placeholder to remember which OID we picked up. Then we add to the conformance file:

```c
#.FN_BODY AlgorithmIdentifier/algorithmId
    offset = dissect_ber_object_identifier(false, pinfo, tree, tvb, offset,
                                               hf_x509af_algorithm_id, algorithm_id);

#.FN_BODY AlgorithmIdentifier/parameters
    offset=call_ber_oid_callback(algorithm_id, tvb, offset, pinfo, tree);
```

The dissector body we specified for AlgorithmIdentifier/algorithmId here stores the retrieved OID inside the variable algorithm_id we specified.
When we later come to the dissector for AlgorithmIdentifier/parameters we pick this OID up from the static variable and just pass it on to the ber/oid dissector helper.

This example comes from the X509AF dissector. Please see the code there for more examples on how to do this.

#.MODULE_IMPORT And #.INCLUDE

These directive in the Asn2wrs conformation file are used to manage references to external type definitions, i.e. IMPORTS. The examples below are all from the X.509 Authentication Framework
Example ASN

This is an example from the X509AF dissector which amongst other things imports definitions from X.509 InformationFramework:

```
IMPORTS
    Name, ATTRIBUTE, AttributeType, MATCHING-RULE, Attribute
FROM InformationFramework informationFramework
```

Which tells the Asn2wrs compiler that the types 'Name', 'ATTRIBUTE', 'AttributeType', 'MATCHING-RULE' and 'Attribute' are declared inside the external InformationFramework ASN module and that they are referenced from this module. In order for Asn2wrs to generate correct code for the dissection it is necessary to give it some help by telling what kind of types these are, i.e. are they INTEGERS or SEQUENCEs or something else.

In order to be able to access these functions from this module it is important that these types have been declared as #.EXPORTS in the X509 InformationFramework dissector so that they are exported and that we can link to them.

#.MODULE_IMPORT

First we need to tell Asn2wrs which protocol name Wireshark uses for the functions in this external import, so that Asn2wrs can generate suitable function call signatures to these external functions. We do this by adding a directive to the conformance file:

```
#.MODULE_IMPORT
    InformationFramework x509if
```

Where InformationFramework is the ASN name for the module used in the asn IMPORTS declaration and that x509if is the name we use inside Wireshark for this protocol. This tells Asn2wrs that the function name to call to dissect Name would be dissect_x509if_Name(...). Without this knowledge Asn2wrs would not know which function name to generate.

#.INCLUDE

Second, in order for Asn2wrs to generate correct code it also needs to know the BER type and class of these types that are imported, since that would affect how they are to be encoded on the wire. This information about what kind of BER attributes these imported types have are done using the #.INCLUDE directive in the conformance file:

```
#.INCLUDE ../x509if/x509if_exp.cnf
```
See `.EXPORTS` for a description and examples of these types of include files.

### .NO_EMIT And .USER_DEFINED

These two directives in the conformance file for Asn2wrs can be used to suppress generation of dissectors and/or value_strings and similar for a protocol. This is useful when there are types in the asn definition that either Asn2wrs can not handle or if we want to handle the dissection ourself inside the template file to do additional state keeping or things that Asn2wrs does not know how to manage.

These two directives are very similar. The only difference between is that `.NO_EMIT` will suppress emitting the dissector for that function and also any value_strings while `.USER_DEFINED` will emit declarations instead of definitions.

I.e. `.USER_DEFINED` will emit declarations such as `extern const value_string Type_vals[];` and `[static] int dissect_Proto_Type(…);`

Use `.NO_EMIT` if you dont need to call this function at all from anywhere (except from the template itself) and use `.USER_DEFINED` is better if you implement the function inside the template but still want to allow it to be called from other places.

### Syntax

#### .USER_DEFINED

```
#.USER_DEFINED
TypeName [WITH_VALS|WITHOUT_VALS|ONLY_VALS]
...
#.END
```

#### .NO_EMIT

```
#.NO_EMIT
TypeName [WITH_VALS|WITHOUT_VALS|ONLY_VALS]
...
#.END
```

Options:

- **WITH_VALS** (default): Both dissection function and value string table are user defined and not emitted.
- **WITHOUT_VALS**: Only dissection function is user defined and not emitted.
- **ONLY_VALS**: Only value string table is user defined and not emitted.

### .PDU and .PDU_NEW

This directive in the Asn2wrs conformance file will generate a wrapper function around an object dissector. This is useful if there is an object inside the ASN.1 definition that we really want to
register as a protocol dissector or if we want it to have a well known signature.

Function Names

The wrapper functions that are created will all be named and have the following signature:

```c
static void dissect_ProtocolName_ObjectName(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree);
```

Notice that this is exactly the same signature as `dissector_t` which is used by all dissector entry points.

Usage

To get Asn2wrs to generate such wrapper functions you just have to list all objects one by one on the lines following the `.PDU` declaration.

Example

```c
#.PDU
SomeObject
```

This will result in Asn2wrs creating this wrapper function in the packet-foo.c dissector file:

```c
static void dissect_SomeObject_PDU(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree) {
    dissect_foo_SomeObject(false, ...)
}
```

This function can then later be called or referenced from the template file or even exported.

#.REGISTER and #.REGISTER_NEW

This directive in the Asn2wrs conformation file can be used to register a dissector for an object to an OID. This is very useful for X.509 and similar protocols where structures and objects are frequently associated with an OID. In particular, some of the structures here encode an OID in a field and then the content in a different field later, and how that field is to be dissected depends on the previously seen OID.

One such example can be seen in the ASN.1 description for X.509/AuthenticationFramework which has a structure defined such as
AlgorithmIdentifier ::= SEQUENCE {
  algorithm   ALGORITHM.&id({SupportedAlgorithms}),
  parameters  ALGORITHM.&Type({SupportedAlgorithms}{@algorithm}) OPTIONAL
}

Which means that the parameters field in this structure, what this field contains and how it is to be dissected depends entirely upon what OID is stored inside algorithm. A whole bunch of protocols use similar types of constructs. While dissection of this particular structure itself currently has to be hand implemented inside the template (see x509af for examples of how this very structure is handled there). The #.REGISTER option in the conformance file will at least make it easy and painless to attach the actual OID to dissector mappings.

Usage

To get Asn2wrs to generate such automatic registration of OID to dissector mappings just use the #.REGISTER directive in the conformation file.

Example

#.REGISTER
Certificate B "2.5.4.36" "id-at-userCertificate"

Which will generate the extra code to make sure that anytime Wireshark needs to dissect the blob associated to the OID "2.5.4.36" it now knows that that is done by calling the subroutine to dissect a Certificate in the current protocol file. The "id-at-userCertificate" is just a free form text string to make Wireshark print a nice name together with the OID when it presents it in the decode pane. While this can be just about anything you want I would STRONGLY use the name used to this object/oid in the actual ASN.1 definition file.

Include File

During the compilation phase Asn2wrs will put all the extra registration code for this in the include file packet-protocol-dis-tab.c. Make sure that you include this file from the template file or the registration to an OID will never occur. #include "packet-protocol-dis-tab.c" should be included from the proto_reg_handoff_protocol function in the template file.

See Also

The various dissectors we have for X.509 such as the X.509AF which contains several examples of how to use this option. That dissector can also serve as an example on how one would handle structures of the type AlgorithmIdentifier above. Asn2wrs can NOT handle these types of structures so we need to implement them by hand inside the template.
This Document’s License (GPL)

As with the original license and documentation distributed with Wireshark, this document is covered by the GNU General Public License (GNU GPL).

If you haven’t read the GPL before, please do so. It explains all the things that you are allowed to do with this code and documentation.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.
We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software. If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

Finally, any free program is threatened constantly by software
patents. We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary. To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and
modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License. The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language. (Hereinafter, translation is included without limitation in
the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope. The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.
You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.