Wireshark Developer’s Guide

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 Win32 developers, as a lot of the tools and methods used when building Wireshark are much more common in the UNIX world than on Win32.

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:

- XXX - Please give feedback :-)

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 and updated for VS2013 by Graham Bloice

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/ in A4 PDF, US letter PDF, single HTML, and chunked HTML.

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 1. Typographic Conventions

Style	Description	Example
Italic	File names, folder names, and extensions	*C:\Development\wireshark*.
`Monospace`	Commands, flags, and environment variables	CMake’s `-G` option.
`Bold Monospace`	Commands that should be run by the user	Run **`cmake -G Ninja ..`**.
[Button]	Dialog and window buttons	Press **[Launch]** to go to the Moon.
`Key`	Keyboard shortcut	Press `Ctrl`+`Down` to move to the next packet.
Menu	Menu item	Select **Go › Next Packet** to move to the next packet.

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.

NOTE

This is a note

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

TIP *This is a tip*

Tips are helpful for your everyday work using Wireshark.

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

Wireshark Build Environment

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

Introduction

Introduction

This chapter will provide you with information about Wireshark development in general.

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 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[AT]wireshark.org.

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

Unix

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

Linux

- Debian GNU/Linux
- Ubuntu
- Gentoo Linux
- IBM S/390 Linux (Red Hat)
- Mandrake Linux

- PLD Linux
- Red Hat Linux
- Rock Linux
- Slackware Linux
- Suse Linux

Microsoft Windows

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. Wireshark runs on and can be compiled on the following platforms:

- Windows 10 / Windows Server 2016
- Windows 8.1 / Windows Server 2012 R2
- Windows 8 / Windows Server 2012
- Windows 7 / Windows Server 2008 R2

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

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 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 plain ANSI 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, or write a new dissector for a specific network protocol. As (almost) any dissector is written in plain old ANSI C, a good knowledge about ANSI C will be sufficient for Wireshark development in almost any case.

So unless you are going to change the build process of Wireshark itself, you won’t come in touch with any other programming language than ANSI C (such as Perl or Python, which are used only in the Wireshark build process).

Beside the usual tools for developing a program in C (compiler, make, …), the build process uses some additional helper tools (Perl, Python, Sed, …), which are needed for the build process when Wireshark is to be build and installed from the released source packages. 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 download page of the Wireshark website: https://www.wireshark.org/download.html.

Releases and distributions

The officially released files can be found at https://www.wireshark.org/download.html. A new Wireshark version is released after significant changes compared to the last release are completed or a serious security issue is encountered. The typical release schedule is about every 4-8 weeks (although this may vary). There are two kinds of distributions: binary and source; both have their advantages and disadvantages.

Binary distributions

Binary distributions are usually easy to install (as simply starting the appropriate file is usually the only thing to do). They are available for the following systems:

- Windows (.exe file). The typical Windows end user is used to getting a setup.exe file which will install all the required things for him.
- Win32 PAF (.paf.exe file). Another Windows end user method is to get a portable application file which will install all the required things for him.
- Debian (.deb file). A user of a Debian Package Manager (DPKG) based system obtains a .deb file from which the package manager checks the dependencies and installs the software.
- Red Hat (.rpm file). A user of a RPM Package Manager (RPM) based system obtains an .rpm file from which the package manager checks the dependencies and installs the software.
- macOS (.dmg file). The typical macOS end user is used to getting a .dmg file which will install all the required things for him.
- Solaris. A Solaris user obtains a file from which the package manager (PKG) checks the dependencies and installs the software.

However, if you want to start developing with Wireshark, the binary distributions won't be too helpful, as you need the source files, of course.

For details about how to build these binary distributions yourself, e.g. if you need a distribution for a special audience, see Binary packaging.

Source code distributions

It's still common for UNIX developers to give the end user a source tarball and let the user compile it on their target machine (configure, make, make install). However, for different UNIX (Linux) distributions it's becoming more common to release binary packages (e.g. .deb or .rpm files) these days.

You should use the released sources if you want to build Wireshark from source on your platform for productive use. However, if you going to develop changes to the Wireshark sources, it might be better to use the latest GIT sources. For details about the different ways to get the Wireshark source code see Obtain the Wireshark sources.

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

Automated Builds (Buildbot)

The Wireshark Buildbot automatically rebuilds Wireshark on every change of the source code repository and indicates problematic changes. This frees the developers from repeating (and annoying) work, so time can be spent on more interesting tasks.

Advantages

- Recognizing (cross platform) build problems - early. Compilation problems can be narrowed down to a few commits, making a fix much easier.

- "Health status" overview of the sources. A quick look at: https://buildbot.wireshark.org/wireshark-master/ gives a good "feeling" if the sources are currently "well". On the other hand, if all is "red", an update of a personal source tree might better be done later …
- "Up to date" binary packages are available. After a change was committed to the repository, a binary package / installer is usually available within a few hours at: https://www.wireshark.org/download/automated/. This can be quite helpful, e.g. a bug reporter can easily verify a bugfix by installing a recent build.
- Automated regression tests. In particular, the fuzz tests often indicate "real life" problems that are otherwise hard to find.

What does the Buildbot do?

The Buildbot will do the following (to a different degree on the different platforms):

- Check out from the source repository
- Build
- Create binary packages and installers
- Create source packages and run distribution checks
- Run regression tests

Each step is represented at the status page by a rectangle, green if it succeeded or red if it failed. Most steps provide a link to the corresponding console logfile, to get additional information.

The Buildbot runs on a platform collection that represents the different "platform specialties" quite well:

- Windows 8.1 x86 (Win32, little endian, Visual Studio 2013)
- Windows Server 2012 R2 x86-64 (Win64, little endian, Visual Studio 2013)
- Ubuntu x86-64 (Linux, little endian, gcc, Clang)
- macOS x86-64 (BSD, little endian, Clang)

and two buildslaves that run static code analysis to help spot coding issues:

- Visual Studio Code Analysis (Win64, little endian, VS 2013)
- Clang Code Analysis (Linux, little endian, Clang)

Each platform is represented at the status page by a single column, the most recent entries are at the top.

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://wiki.wireshark.org/ 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

NOTE

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

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 (Bugzilla).

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

The archives are searchable

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 (Bugzilla)

The Wireshark community collects bug reports in a Bugzilla database at https://bugs.wireshark.org/. This database is filled with manually filed bug reports, usually after some discussion on wireshark-dev, and automatic bug reports from the Buildbot tools.

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.

Reporting Problems

NOTE

Test with the latest version

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

Don’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

Don’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 Bugzilla database though.

Reporting Crashes on UNIX/Linux 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:

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

NOTE

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:

- 32-bit Windows: https://www.wireshark.org/download/win32/all-versions/
- 64-bit Windows: https://www.wireshark.org/download/win64/all-versions/

Files are named "Wireshark-pdb-win*bits*-*x*.*y*.*z*.zip" to match their corresponding "Wireshark-win*bits*-*x*.*y*.*z*.exe" installer packages.

Quick Setup

UNIX: Installation

All the tools required are usually installed on a UNIX developer machine.

If a tool is not already installed on your system, you can usually install it using the package in your distribution: aptitude, yum, Synaptic, etc.

If an install package is not available or you have a reason not to use it (maybe because it’s simply too old), you can install that tool from source code. The following sections will provide you with the webpage addresses where you can get these sources.

Win32/64: Step-by-Step Guide

A quick setup guide for Win32 and Win64 with recommended configuration.

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 (x86 or x64) version of the Visual Studio command prompt.
2. Not copying/downloading the correct version of vcredist_xYY.exe.

Install Microsoft C compiler and SDK

You need to install, in exactly this order:

1. C compiler: Download and install “Microsoft Visual Studio 2015 Community Edition.” This is a small download that then downloads all the other required parts (which are quite large).

Select the "Custom" install and then uncheck all the optional components other than "Common Tools for Visual C++ 2015" (unless you want to use them for purposes other than Wireshark).

You can use Chocolatey to install Visual Studio, to correctly configure the installation, copy the deployment XML file msvc2015AdminDeployment.xml from the source code tools directory and pass the path the file to the chocolatey install command:

```
PS$>choco install -y VisualStudio2015Community --timeout 0 -package-parameters "--
AdminFile path\to\msvc2015AdminDeployment.xml"
```

You can use other Microsoft C compiler variants, but VS2015 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 only). 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, Visual Studio 2015 Community Edition is the only free (as in beer) versions that includes the Visual Studio integrated debugger. Visual Studio 2015 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 download the **Qt Online Installer for Windows** from the Qt Project "Download Open Source" page and select a component that matches your target system and compiler. For example, the “msvc2015 64-bit” component is used to build the official 64-bit packages. You can deselect all the Qt xxxx (e.g. Qt Charts) components as they aren’t required.

Note that installation of separate Qt components are required for 32 bit and 64 bit builds, e.g. “msvc2015 32-bit” and “msvc2015 64-bit”. The environment variable `QT5_BASE_DIR` should be set as appropriate for your environment and should point to the Qt directory that contains the bin directory, e.g. *C:\Qt\5.9.1\msvc2015_64*

The Qt maintenance tool (*C:\Qt\MaintenanceTool.exe*) can be used to upgrade Qt to newer versions.

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 and Cygwin.

Chocolatey tends to install packages into its own path (%ChocolateyInstall%). In most cases this is OK, but in some instances (Python in particular) this might not be what you want. You can install Chocolatey packages using the command `choco install`.

```
> rem Flex and Bison are required.
> choco install -y winflexbison
> rem Git, CMake, Perl, Python, etc are also required, but can be installed
> rem via their respective installation packages.
> choco install -y git cmake
> rem Choose one of Strawberry...
> choco install -y strawberryperl
> rem ...or ActiveState Perl
> choco install -y activeperl
> rem This will likely install Python in a non-standard location, but
> rem should otherwise work.
> choco install -y python3
```

Optional: Install Cygwin

On 32-bit Windows, download the 32-bit Cygwin installer and start it. On 64-bit Windows, download the 64-bit Cygwin installer and start it.

NOTE

Cygwin is no longer required

In the past the Wireshark development toolchain depended on Cygwin, but it it no longer required. Although you can often use the Cygwin version of a particular tool for Wireshark development that's not always the case.

At the "Select Packages" page, you'll need to select some additional packages which are not installed by default. Navigate to the required Category/Package row and, if the package has a "Skip" item in the "New" column, click on the "Skip" item so it shows a version number for:

- Devel/bison (or install Win flex-bison — see Chocolatey above)
- Devel/flex (or install Win flex-bison — see Chocolatey above)
- Devel/git (recommended, but it's also available via Chocolatey — see the Git discussion below)
- Interpreters/perl
- Utils/patch (only if needed) (may be Devel/patch instead)
- Text/docbook-xml45 (only needed if you're building the documenation)

You might also have to install

- Interpreters/m4

if installing Devel/bison doesn't provide a working version of Bison. If m4 is missing bison will fail.

After clicking the **[Next]** button several times, the setup will then download and install the selected packages (this may take a while).

Alternatively you can install Cygwin and its packages using Chocolatey:

```
PS$>choco install -y cygwin
PS$>choco install -y cyg-get
```

Chocolatey installs Cygwin in *C:\tools\cygwin* by default.

You can directly download packages via `cyg-get`

```
PS$>cyg-get docbook-xml45 [...]
```

Install Python

Get the Python 3.5 or 2.7 installer from http://python.org/download/ and install Python into the default location (*C:\Python35* or *C:\Python27*).

Why is this recommended? Cygwin's */usr/bin/python* is a Cygwin-specific symbolic link which cannot be run from Windows. The native package is faster as well.

Alternatively you can install Python using Chocolatey:

```
PS$>choco install -y python3
```

or

```
PS$>choco install -y python2
```

Chocolatey installs Python in *C:\tools\python3* and *C:\tools\python2* by default.

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 Obtain 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.

Git Extensions

Git Extensions is a native Windows graphical Git client for Windows. You can download the installer from https://github.com/gitextensions/gitextensions/releases/latest.

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:

```
PS$> 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

Get the CMake installer from https://cmake.org/download/ and install CMake into the default location. Ensure the directory containing cmake.exe is added to your path.

Alternatively you can install CMake using Chocolatey:

```
PS$>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). It is used in conjunction with Xsltproc and DocBook to generate the documenation you're reading and the User's Guide.

The easiest way to install them on Windows is via Chocolatey:

```
PS$>choco install -y asciidoctorj xsltproc docbook-bundle
```

Chocolatey ensures that asciidoctorj.exe and xsltproc.exe is on your path and that xsltproc uses the DocBook catalog.

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://code.wireshark.org/review/wireshark
```

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://code.wireshark.org/review/wireshark`

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

   Subdirectory to create: Anything you'd like. Usually *wireshark*.

   TIP *Check your paths*

   Make sure your repository path doesn't contain spaces.

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 2015' folder and choose the Command Prompt appropriate for the build you wish to make, e.g. 'VS2015 x64 Native Tools Command Prompt' for a 64-bit version or 'VS2015 x86 Native Tools Command Prompt' for a 32-bit version. Depending on your version of Windows the Command Prompt list might be directly under 'Visual Studio 2015' or you might have to dig for it under multiple folders, e.g. 'Visual Studio 2015 → Visual Studio Tools → Windows Desktop Command Prompts'.

TIP *Pin the items to the Task Bar*

Pin the Command Prompt you use to the Task Bar for easy access.

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-win64-libs
   > rem Set the Qt installation directory
   > set QT5_BASE_DIR=C:\Qt\5.9.1\msvc2015_64
   > rem Append a custom string to the package version. Optional.
   > set WIRESHARK_VERSION_EXTRA=-YourExtraVersionInfo
   ```

   If your Cygwin installation path is not automatically detected by CMake, you can explicitly specify it with the following environment variable:

```
> rem Chocolatey installs Cygwin in an odd location
> set
WIRESHARK_CYGWIN_INSTALL_PATH=C:\ProgramData\chocolatey\lib\Cygwin\tools\cygwin
```

If you are using a version of Visual Studio earlier than VS2012 then you must set an additional env var, e.g. for VS2010 set the following:

```
> set VisualStudioVersion=10.0
```

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

TIP Qt 5.9 is a "long term support" branch of Qt5. We recommend using it to compile Wireshark on Windows.

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 to the source tree, leaving the source tree in a pristine state. 32 and 64 bit builds require a separate build directory. Create (if required) and change to the appropriate build directory.

```
> mkdir C:\Development\wsbuild32
> cd C:\Development\wsbuild32
```

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 14 2015" ..\wireshark
```

Adjusting the paths as required to Python and the wireshark source tree. 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.

To build an x64 version, the `-G` parameter must have a Win64 suffix, e.g. `-G "Visual Studio 14 2015 Win64"`:

```
> cmake -G "Visual Studio 14 2015 Win64" ..\wireshark
```

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/wsbuild32
```

If you get any other output, there is an issue in your envirnment 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 `QT5_BASE_DIR`.

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\wsbuild32\run\RelWithDebInfo\Wireshark.exe* and make sure it starts.
5. Open **Help › About**. If it shows your "private" program version, e.g.: Version 2.9.0-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

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-win32-2.9.0-myprotocol123.exe*, you'll need NSIS. You can download it from http://nsis.sourceforge.net.

Note that the 32-bit version of NSIS will work for both 32-bit and 64-bit versions of Wireshark. NSIS v3 is required.

Note: If you do not yet have a copy of vcredist_x86.exe or vcredist_x64.exe in *./wireshark-win**XX**-libs* (where ***XX*** is 32 or 64) you will need to download the appropriate file and place it in *./wireshark-win**XX**-libs* before starting this step.

If building an x86 version using a Visual Studio "Express" edition or an x64 version with any edition, then you must have the appropriate vcredist file for your compiler in the support libraries directory (*vcredist_x86.exe* in *wireshark-32-libs* or *vcredist_x64.exe* in *wireshark-win64-libs*).

The files can be located in the Visual Studio install directory for non-Express edition builds, or downloaded from Microsoft for Expresss edition builds.

Note you must use the correct version of vcredist for your compiler, unfortunately they all have the same name (*vcredist_x86.exe* or *vcredist_x64.exe*). You can use Windows Explorer and examine the 'Properties → Details' tab for a vcredist file to determine which compiler version the file is for use with.

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

Run

```
> msbuild /m /p:Configuration=RelWithDebInfo nsis_package_prep.vcxproj
> msbuild /m /p:Configuration=RelWithDebInfo nsis_package.vcxproj
```

to build a Wireshark installer. If you sign your executables you should do so between the "nsis_package_prep" and "nsis_package" steps.

Run

```
> packaging\nsis\wireshark-win64-{wireshark-version}-myprotocol123.exe
```

to test your new installer. It’s a good idea to test on a different machine than the developer machine. Note that if you’ve built an x86 version, the installer name will contain “win32”.

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 code is stored inside Wireshark project's Git repository located at a server at the wireshark.org domain.

Changes to the official repository are managed using the Gerrit code review system. Gerrit makes it easy to test and discuss changes before they are pushed to the main repository. For an overview of Gerrit see the Quick Introduction.

Why Git?

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

Why Gerrit?

Gerrit makes it easy to contribute. You can sign in with any OpenID provider and push your changes. It's usable from both the web and command line and is integrated with many popular tools.

NOTE

Git is our **third** *revision control system*

Wireshark originally used Concurrent Versions System (CVS) and migrated to Subversion in July 2004. The Subversion repository was subsequently migrated to Git in January 2014.

Using Wireshark's Git repository you can:

- Keep your private sources up to date with very little effort
- Get a mail notification when the official source code changes
- Get the source files from any previous release (or any other point in time)
- Have a quick look at the sources using a web interface
- See which person changed a specific piece of code
- and much more

The web interface to the Git repository

If you need a quick look at the Wireshark source code you can browse the most recent file versions in the master branch using Gitweb:

https://code.wireshark.org/review/gitweb?p=wireshark.git;a=tree

You can also view commit logs, branches, tags, and past revisions:

https://code.wireshark.org/review/gitweb?p=wireshark.git

Like most revision control systems, Git uses branching to manage different copies of the source code and allow parallel development. Wireshark uses the following branches for official releases:

- *master*: Main feature development and odd-numbered "feature" releases.
- *master-x.y*: Stable release maintenance. For example, master-1.10 is used to manage the 1.10.x official releases.

Obtain the Wireshark sources

There are several ways to obtain the sources from Wireshark’s Git repository.

TIP

Check out from the master branch using Git.

Using Git is much easier than synchronizing your source tree by hand using any of the snapshot methods mentioned below. Git merges changes into your personal source tree in a very comfortable and quick way. So you can update your source tree several times a day without much effort.

NOTE

Keep your sources up to date

The following ways to retrieve the Wireshark sources are sorted in decreasing source timeliness. If you plan to commit changes you’ve made to the sources, it’s a good idea to keep your private source tree as current as possible.

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

Recommended for development purposes.

Age: a few minutes.

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

- https://code.wireshark.org/review/wireshark

If you create a Gerrit account you can clone from an authenticated URL:

- ssh://your.username@code.wireshark.org:29418/wireshark
- https://your.username@code.wireshark.org/review/wireshark

SSH lets you use Gerrit on the command line. HTTP lets you access the repository in environments that block the Gerrit SSH port (29418). At the time of this writing (early 2014) we recommend that you use the SSH interface. However, this may change as more tools take advantage of Gerrit's HTTP REST API.

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 and Gerrit clients.

1. Sign in to https://code.wireshark.org/review using OpenID (click Register or Sign In in the upper right corner of the web page). Follow the login instructions.
2. In the upper right corner of the web page, click on your account name and select *Settings*.
3. Under *Profile* set a username. This will be the username that you use for SSH access. For the steps below we'll assume that your username is `henry.perry`.
4. Select *SSH Public Keys* and add one or more keys. You will typically upload a key for each computer that you use.
5. Install git-review. This is an installable package in many Linux distributions. You can also install it as a Python package. (This step isn't strictly necessary but it makes working with Gerrit much easier.) To install it from Chocolatey run

   ```
   # Make sure "Scripts" is in our path
   PS$>$env:path += ";C:\tools\python2\Scripts"
   PS$>choco install pip
   PS$>choco install git-review -source python
   ```

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

   ```
   $ git --version
   ```

7. 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
   ```

8. Next, clone the Wireshark master:

   ```
   $ git clone ssh://henry.perry@code.wireshark.org:29418/wireshark
   ```

   The checkout only has to be done once. This will copy all the sources of the latest version (including directories) from the server to your machine. This may take some time depending on the speed of your internet connection.

9. Then set up the git pre-commit hook and the push address:

   ```
   $ cd wireshark
   $ cp tools/pre-commit .git/hooks/
   $ git config --add remote.origin.push HEAD:refs/for/master
   ```

   This will run a few basic checks on commit to make sure that the code does not contain trivial errors. It will also warn if it is out of sync with its master copy in the tools/ directory. The change in the push address is necessary: We have an asymmetric process for pulling and pushing because of gerrit.

10. Initialize git-review.

    ```
    $ git review -s
    ```

    This prepares your local repository for use with Gerrit, including installing the `commit-msg` hook script.

Git web interface

Recommended for informational purposes only, as only individual files can be downloaded.

Age: a few minutes (same as anonymous Git access).

The entire source tree of the Git repository is available via a web interface at https://code.wireshark.org/review/gitweb?p=wireshark.git. You can view each revision of a particular file, as well as diffs between different revisions. You can also download individual files but not entire directories.

Buildbot Snapshots

Recommended for development purposes, if direct Git access isn't possible (e.g. because of a restrictive firewall).

Age: some number of minutes (a bit older than the Git access).

The Buildbot server will automatically start to generate a snapshot of Wireshark's source tree after a source code change is committed. These snapshots can be found at https://www.wireshark.org/download/automated/src/.

If Git access isn't possible, e.g. if the connection to the server isn't possible because of a corporate firewall, the sources can be obtained by downloading the Buildbot snapshots. However, if you are going to maintain your sources in parallel to the "official" sources for some time, it's recommended to use the anonymous (or authenticated) Git access if possible (believe it, it will save you a lot of time).

Released sources

Recommended for building pristine packages.

Age: from days to weeks.

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

The differences between the released sources and the sources in the Git repository will keep on growing until the next release is made. (At the release time, the released and latest Git repository versions are identical again :-).

Update the 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 Buildbot first

As development evolves, the Wireshark sources are compilable most of the time — but not always. You should take a look at https://buildbot.wireshark.org/trunk/waterfall before fetching or pulling to make sure the builds are in good shape.

Update Using Git

After you clone Wireshark's Git repository you can update by running

```
$ git status
$ git pull
```

Depending on your preferences and work habits you might want to run `git pull --rebase` or `git checkout -b my-topic-branch origin/master` instead.

Fetching should only take a few seconds, even on a slow internet connection. It will update your local repository history with changes from the official repository. If you and someone else have changed the same file since the last update, Git will try to merge the changes into your private file (this works remarkably well).

Update Using Source Archives

There are several ways to download the Wireshark source code (as described in Obtain the Wireshark sources), but bringing the changes from the official sources into your personal source tree is identical.

First of all, you will download the new `.tar.xz` file of the official sources the way you did it the first time.

If you haven't changed anything in the sources, you could simply throw away your old sources and reinstall everything just like the first time. But be sure, that you really haven't changed anything. It might be a good idea to simply rename the "old" dir to have it around, just in case you remember later that you really did change something before.

If you have changed your source tree, you have to merge the official changes since the last update into your source tree. You will install the content of the `.tar.xz` file into a new directory and use a good merge tool (e.g. http://winmerge.sourceforge.net/for Win32) to bring your personal source tree in sync with the official sources again.

This method can be problematic and can be much more difficult and error-prone than using Git.

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

The recommended (and fastest) way to build Wireshark is with CMake and Ninja:

```
# 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 .)
```

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

```
$ cmake -LH ../wireshark
```

to see what options you have.

Win32 native

Follow the build procedure in Build Wireshark 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.

Run generated Wireshark

> **TIP**
>
> *Tip!*
>
> 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/Linux

After a successful build you can run Wireshark right from the build directory. Still the program would need to know that it's being run from the build directory and not from its install location. This has an impact on the directories where the program can find the other parts and relevant data files.

In order to run the Wireshark from the build directory set the environment variable `WIRESHARK_RUN_FROM_BUILD_DIRECTORY` and run Wireshark. If your platform is properly setup, your build directory and current working directory are not in your PATH, so the command line to launch Wireshark would be:

```
$ WIRESHARK_RUN_FROM_BUILD_DIRECTORY=1 ./wireshark
```

There's no need to run Wireshark as root user, you just won't be able to capture. When you opt to run Wireshark this way, your terminal output can be informative when things don't work as expected.

Win32 Native

During the build all relevant program files are collected in a subdirectory `run\RelWithDebInfo`. You can run the program from there by launching the Wireshark.exe executable.

Debug Your Generated Wireshark

Unix/Linux

You can debug using command-line debuggers such as gdb, dbx, or lldb. If you prefer a graphic debugger, you can use the Data Display Debugger (ddd).

Additional traps can be set on GLib by setting the `G_DEBUG` environment variable:

```
$ G_DEBUG=fatal_criticals ddd wireshark
```

See http://library.gnome.org/devel/glib/stable/glib-running.html

Win32 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

As the Wireshark developers are working on many different platforms, a lot of editors are used to develop Wireshark (emacs, vi, Microsoft Visual Studio and many, many others). There's no "standard" or "default" development environment.

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

- Add support for a new protocol (a new dissector)
- Change or extend an existing dissector
- Fix a bug
- Implement a glorious new feature

The internal structure of the Wireshark sources will be described in Wireshark Development.

TIP

Ask the wireshark-dev 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 Git 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://code.wireshark.org/review. This requires you to set up git as described at Git over SSH or HTTPS.

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 Obtain 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` and make sure you aren't adding, removing, or omitting anything you shouldn't.
- *Find a good descriptive topic name for your patch.* Short, specific names are preferred. *snowcone-machine-protocol* is good, your name or your company name isn't.
- *Don't put unrelated things into one large patch.* A few smaller patches are usually easier to apply (but also don't put every changed line into a separate patch.

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.

NOTE

Please remember

Wireshark is a volunteer effort. You aren't paying to have your code reviewed and integrated.

Code Requirements

The core maintainers have done a lot of work fixing bugs and making code compile on the various platforms Wireshark supports.

To ensure Wireshark's source code quality, and to reduce the workload of the core maintainers, there are some things you should think about *before* submitting a patch.

WARNING

Pay attention to the coding guidelines

Ignoring the code requirements will make it very likely that your patch will be rejected.

- *Follow the Wireshark source code style guide.* Just because something compiles on your platform, that doesn't mean it'll compile on all of the other platforms for which Wireshark is built. Wireshark runs on many platforms, and can be compiled with a number of different compilers. See Coding Stylefor details.

- *Submit dissectors as built-in whenever possible.* Developing a new dissector as a plugin is a good idea because compiling and testing is quicker, but it’s best to convert dissectors to the built-in style before submitting for check in. This reduces the number of files that must be installed with Wireshark and ensures your dissector will be available on all platforms.

  This is no hard-and-fast rule though. Many dissectors are straightforward so they can easily be put into "the big pile", while some are ASN.1 based which takes a different approach, and some multiple source file dissectors are more suitable to be placed separately as plugins.

- *Ensure 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 suffice 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 will not automatically add the required Change-ID to your commit. In case you are not updating an existing patch you may generate a Change-ID by running git review -i (or git commit --amend if don’t use git review).

- *Fuzz test your changes!* Fuzz testing is a very effective way to automatically find a lot of dissector related bugs. You’ll take a capture file containing packets affecting your dissector and the fuzz test will randomly change bytes in this file, so that unusual code paths in your dissector are checked. There are tools available to automatically do this on any number of input files, see: https://wiki.wireshark.org/FuzzTesting for details.

Uploading your changes

When you’re satisfied with your changes (and obtained any necessary approval from your organization) you can upload them for review at https://code.wireshark.org/review. This requires a Gerrit Code Review account as described at The Wireshark Git repository.

Changes should be pushed to a magical "refs/for" branch in Gerrit. For example, to upload your new Snowcone Machine Protocol dissector you could push to refs/for/master with the topic "snowcone-machine":

```
$ git push ssh://my.username@code.wireshark.org:29418/wireshark
HEAD:refs/for/master/snowcone-machine
```

The username `my.username` is the one which was given during registration with the review system.

If you have `git-review` installed you can upload the change with a lot less typing:

```
# Note: The "-f" flag deletes your current branch.
$ git review -f
```

You can push using any Git client. Many clients have support for Gerrit, either built in or via an additional module.

You might get one of the following responses to your patch request:

- Your patch is checked into the repository. Congratulations!
- You are asked to provide additional information, capture files, or other material. If you haven't fuzzed your code, you may be asked to do so.
- Your patch is rejected. You should get a response with the reason for rejection. Common reasons include not following the style guide, buggy or insecure code, and code that won't compile on other platforms. In each case you'll have to fix each problem and upload another patch.
- You don't get any response to your patch. Possible reason: All the core developers are busy (e.g., with their day jobs or family or other commitments) and haven't had time to look at your patch. Don't worry, if your patch is in the review system it won't get lost.

If you're concerned, feel free to add a comment to the patch or send an email to the developer's list asking for status. But please be patient: most if not all of us do this in our spare time.

Backporting a change

When a bug is fixed in the master branch it might be desirable or necessary to backport the fix to a stable 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-1.10. Using "pure Git" commands you would do the following:

```
# Create a new topic branch for the backport.
$ git checkout -b backport-g1ab2c3d4 origin/master-1.10

# 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.
$ make
$ ...

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

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

# Upload the change to Gerrit
$ git push ssh://my.username@code.wireshark.org:29418/wireshark HEAD:refs/for/master-
1.10/backport-g1ab2c3d4
```

If you want to cherry-pick a Gerrit change ID (e.g. I5e6f7890) you can use `git review -X I5e6f7890` instead of `git cherry-pick` and `git review` instead of `git push` as described in the previous chapter.

Apply a patch from someone else

Sometimes you need to apply a patch to your private source tree. Maybe because you want to try a patch from someone on the developer mailing list, or you want to check your own patch before submitting.

WARNING

Beware line endings

If you have problems applying a patch, make sure the line endings (CR/LF) of the patch and your source files match.

Using patch

Given the file *new.diff* containing a unified diff, the right way to call the patch tool depends on what the pathnames in *new.diff* look like. If they're relative to the top-level source directory (for example, if a patch to *prefs.c* just has *prefs.c* as the file name) you'd run it as:

```
$ patch -p0 < new.diff
```

If they're relative to a higher-level directory, you'd replace 0 with the number of higher-level directories in the path, e.g. if the names are *wireshark.orig/prefs.c* and *wireshark.mine/prefs.c*, you'd run it with:

```
$ patch -p1 < new.diff
```

If they're relative to a *subdirectory* of the top-level directory, you'd run `patch` in *that* directory and run it with `-p0`.

If you run it without `-pat` all, the patch tool flattens path names, so that if you have a patch file with patches to *CMakeLists.txt* and *wiretap/CMakeLists.txt*, it'll try to apply the first patch to the top-level *CMakeLists.txt* and then apply the *wiretap/CMakeLists.txt* patch to the top-level *CMakeLists.txt* as well.

At which position in the filesystem should the patch tool be called?

If the pathnames are relative to the top-level source directory, or to a directory above that directory, you'd run it in the top-level source directory.

If they're relative to a **subdirectory** — for example, if somebody did a patch to *packet-ip.c* and ran `diff` or `git diff` in the *epan/dissectors* directory — you'd run it in that subdirectory. It is preferred that people **not** submit patches like that, especially if they're only patching files that exist in multiple directories such as *CMakeLists.txt*.

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.

Debian: .deb packages

The Debian Package is built using dpkg-buildpackage, based on information found in the source tree under *debian*. See http://www.debian-administration.org/articles/336 for a more in-depth discussion of the build process.

In the wireshark directory, type:

```
$ dpkg-buildpackage -rfakeroot -us -uc
```

to build the Debian Package.

Red Hat: .rpm packages

You can build an RPM package using the `rpm-package` target. The package version is derived from the current git HEAD, so you must build from a git checkout.

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 rpm-package
# ...or, if you're using GNU make...
$ make rpm-package
```

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`, `bison`, `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/macosx*. It must be built using CMake. In your build directory, type:

```
$ make dmg_package
```

to build the macOS Package.

Win32: 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/nsis/wireshark.nsi*.

When building with CMake you must first build the *nsis_package_prep* target, followed by the *nsis_package* target, e.g.

```
> msbuild /m /p:Configuration=RelWithDebInfo nsis_package_prep.vcxproj
> msbuild /m /p:Configuration=RelWithDebInfo nsis_package.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-2.9.0.exe* in the *packaging/nsis* directory in your build directory.

Win32: 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:

- NSIS Portable (Unicode)
- PortableApps.com Installer
- PortableApps.com Launcher
- PortableApps.com AppCompactor

When building with CMake you must first build the *nsis_package_prep* target (which takes care of general packaging dependencies), followed by the *portableapps_package* target, e.g.

```
> msbuild /m /p:Configuration=RelWithDebInfo nsis_package_prep.vcxproj
> msbuild /m /p:Configuration=RelWithDebInfo portableapps_package.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: *WiresharkPortable*2.9.0.paf.exe_ in the *packaging/portableapps* directory.

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: The tools should be commonly available on the supported UNIX platforms and for Windows platforms by using an emulation layer such as Cygwin.
- 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 Quick Setup.

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:

```
PS:\>iex ((new-object
net.webclient).DownloadString(_https://chocolatey.org/install.ps1_))
```

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:\Python27* or in %PROGRAMFILES%, e.g. *C:\Program Files\Python36*, Chocolatey tends to install it under *C:\ProgramData\chocolatey* or *C:\Tools*. If you want to avoid this behavior you'll probabaly want to install Python using the packages from python.org.

Windows: Cygwin

Cygwin provides a lot of UNIX based tools on the Windows platform. It uses a UNIX emulation layer which might be a bit slower compared to the native Windows tools, but at an acceptable level. The installation and update is pretty easy and done through a single utility, *setup-x86.exe* for 32-bit Windows and *setup-x86_64.exe* for 64-bit Windows. However, it can also be problematic. Cygwin utilities have a non-standard view of the filesystem, and sometimes things don't work as expected. For example, many files in */usr/bin* are symlinks which can't be run directly from Windows.

NOTE

Cygwin is no longer required

In the past the Wireshark development toolchain depended on Cygwin, but it it no longer required. Although you can often use the Cygwin version of a particular tool for Wireshark development that's not always the case.

CMake

Wireshark's build environment can be configured using CMake on Windows, Linux, macOS, and UNIX. 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. Our Buildbot runs CMake steps on Ubuntu, Win32, Win64, and macOS. In particular, the macOS and Windows packages are built using CMake.

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_CAP=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.

-DPYTHON_EXECUTABLE=c:/Python36/python

Force the Python path. Useful on Windows since Cygwin's */usr/bin/python* is a symlink.

-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`.

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/

Official documentation: https://cmake.org/documentation/

About CMake in general and why KDE4 uses it: http://lwn.net/Articles/188693/

Introductory tutorial/presentation: http://ait.web.psi.ch/services/linux/hpc/hpc_user_cookbook/tools/cmake/docs/Cmake_VM_2007.pdf

Introductory article in the Linux Journal: http://www.linuxjournal.com/node/6700/print

Useful variables: http://www.cmake.org/Wiki/CMake_Useful_Variables

Frequently Asked Questions: http://www.cmake.org/Wiki/CMake_FAQ

GNU compiler toolchain (UNIX only)

gcc (GNU compiler collection)

The GCC C compiler is available for most of the UNIX-like platforms.

If GCC isn't already installed or available as a package for your platform, you can get it at: http://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 there are some GUI frontends for it available, most notably GNU DDD.

If gdb isn't already installed or available as a package for your platform, you can get it at: http://www.gnu.org/software/gdb/gdb.html.

After correct installation:

```
$ gdb --version
```

should result in something like:

```
GNU gdb (Ubuntu 7.8-1ubuntu4) 7.8.0.20141001-cvs
Copyright (C) 2014 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.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
```

Your version string may vary, of course.

ddd (GNU Data Display Debugger)

The GNU Data Display Debugger is a good GUI frontend for GDB (and a lot of other command line debuggers), so you have to install GDB first. It is available for many UNIX-like platforms.

If GNU DDD isn’t already installed or available as a package for your platform, you can get it at: http://www.gnu.org/software/ddd/.

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: http://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.

Microsoft compiler toolchain (Windows native)

To compile Wireshark on Windows using the Microsoft C/C++ compiler, 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*, *msvcrt.lib*)
6. Windows platform headers and libraries (e.g. *windows.h*, *WSock32.lib*)
7. HTML help headers and libraries (*htmlhelp.h*, *htmlhelp.lib*)

Official Toolchain Packages And Alternatives

The official Wireshark 2.4.x releases are compiled using Microsoft Visual C++ 2015. The Wireshark 2.2.x and 2.0.x releases are compiled using Microsoft Visual C++ 2013. The Wireshark 1.12.x and 1.10.x releases were compiled using Microsoft Visual C++ 2010 SP1. The 1.8 releases were compiled using Microsoft Visual C++ 2010 SP1 as well. The 1.6, 1.4, and 1.2 releases were compiled using Microsoft Visual C++ 2008 SP1. Other past releases, including the 1.0 branch, were compiled using Microsoft Visual C++ 6.0.

Using the release compilers is recommended for Wireshark development work.

The older "Express Edition" compilers such as Visual C++ 2010 Express Edition SP1 can be used but any PortableApps packages you create with them will require the installation of a separate Visual C++ Redistributable package on any machine on which the PortableApps package is to be used. See 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:

Visual C++ 2013 Community Edition

IDE + Debugger?

Yes

Purchase required?

Free Download

SDK required for 64-bit builds?

No

CMake Generator: `Visual Studio 12`

Visual C++ 2010 Express Edition

IDE + Debugger?

Yes

Purchase required?

Free Download

SDK required for 64-bit builds?

Yes.

CMake Generator: `Visual Studio 10`

Remarks

Installers created using express editions require a C++ redistributable *vcredist_x86.exe* (3MB free download) is required to build Wireshark-win32-2.9.0.exe, and *vcredist_x64.exe* is required to build Wireshark-win64-2.9.0.exe. The version of *vcredist_x86.exe* or *vcredist_x64.exe must* match the version for your compiler including any service packs installed for the compiler.]

Visual Studio 2010

IDE + Debugger?

Yes

Purchase required?

Yes

SDK required for 64-bit builds?

No

CMake Generator: `Visual Studio 10`

Remarks

Building a 64-bit installer requires a a C++ redistributable (*vcredist_x86.exe*).footnoteref[vcredist]

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

```
PS:\> choco install VisualStudioCommunity2013
```

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.

Compiler Package	cl.exe	_MSC_VER	CRT DLL
Visual Studio 2015	14.0	1900	msvcr140.dll
Visual Studio 2013	12.0	1800	msvcr120.dll
Visual Studio 2012	11.0	1700	msvcr110.dll
Visual Studio 2010	10.0	1600	msvcr100.dll

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/{cpp} Optimizing Compiler Version 18.00.31101 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

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

However, the version string may vary.

Documentation on the compiler can be found at Microsoft MSDN

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 12.00.31101.0
Copyright (C) Microsoft Corporation.  All rights reserved.

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

However, the version string may vary.

Documentation on the linker can be found at Microsoft MSDN

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.

Depending on the Microsoft compiler version you use, some binary files coming from Microsoft might be required to be installed on Windows machine to run Wireshark. On a developer machine, the compiler setup installs these files so they are available - but they might not be available on a user machine!

This is especially true for the C runtime DLL (msvcr*.dll), which contains the implementation of ANSI and alike functions, e.g.: fopen(), malloc(). The DLL is named like: *msvcr**version**.dll*, an abbreviation for "Microsoft Visual C Runtime". For Wireshark to work, this DLL must be available on the users machine.

Starting with MSVC7, it is necessary to ship the C runtime DLL (*msvcr**version**.dll*) together with the application installer somehow, as that DLL is possibly not available on the target system.

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 MSDN link is recommended for the interested reader:

- Redistributing Visual C++ Files

In all cases where *vcredist_x86.exe* or *vcredist_x64.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.

msvcr120.dll / vcredist_x86.exe / vcredist_x64.exe - Version 12.0 (2013)

There are three redistribution methods that MSDN mentions for MSVC 2013 (see: "Choosing a Deployment Method"):

1. *Using Visual C++ Redistributable Package.* The Microsoft libraries are installed by copying *vcredist_x64.exe* or *vcredist_x86.exe* to the target machine and executing it on that machine (MSDN recommends this for applications built with Visual Studio 2013)
2. *Using Visual C++ Redistributable Merge Modules.* (Loadable modules for building msi installers. Not suitable for Wireshark's NSIS based installer)
3. *Install a particular Visual C++ assembly as a private assembly for the application.* The Microsoft libraries are installed by copying the folder content of *Microsoft.VC120.CRT* to the target directory (e.g. *C:\Program Files\Wireshark*)

To save installer size, and to make a portable version of Wireshark (which must be completely self-contained, on a medium such as a flash drive, and not require that an installer be run to install anything on the target machine) possible, when building 32-bit Wireshark with MSVC2013, method 3 (copying the content of *Microsoft.VC120.CRT*) is used (this produces the smallest package).

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 SDK's 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. The free Express (as in beer) downloadable C compiler versions (VC++ 2012 Express, VC++ 2012 Express, etc.) do not contain an SDK — you'll need to download a PSDK in order to have the required C header files and libraries.

Older versions of the SDK should also work. However, the command to set the environment settings will be different, try search for SetEnv.* in the SDK directory.

Documentation Toolchain

Wireshark's documentation is split across two directories. The `doc` directory contains man pages written in Perl's POD (Plain Old Documentation) format. The `docbook` directory contains the User's Guide, Developer's Guide, and the release notes, which are written in Asciidoctor markup.

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

Guide HTML

Asciidoctor → *DocBook XML* → xsltproc + DocBook XSL

Guide PDF

Asciidoctor

Guide HTML Help

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

Release notes HTML

Asciidoctor

Release notes text

Asciidoctor → *HTML* → html2text.py

Asciidoctor

Asciidoctor[https://asciidoctor.org/] comes in several flavors: a Ruby gem (Asciidoctor), a Java bundle (AsciidoctorJ), and transpiled JavaScript (Asciidoctor.js). Only the Asciidoctor and AsciidoctorJ flavors are supported for building the Wireshark documentation and AsciidoctorJ is recommended.

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. `compat-mode` [https://asciidoctor.org/docs/migration/] is currently enabled for the guides, but we are steadily migrating to Asciidoctor's modern (>= 1.5.0) syntax.

PDF output requires Asciidoctor PDF. It is included with AsciidoctorJ but *not* with Asciidoctor.

Xsltproc And DocBook

The single HTML, chunked HTML, and HTML Help guides are generated using DocBook XSL stylesheets. They in turn require an XSLT processor. We use `xsltproc`.

HTML Help

HTML Help is used to create the User’s and Developer’s Guide in .chm format. The User’s Guide .chm file is included with the NSIS and WiX installers and is used as Wireshark’s built-in help on Windows.

This compiler is used to generate a .chm file from a bunch of HTML files — in our case to generate the User’s and Developer’s Guide in .chm format.

The compiler is only available as the free (as in beer) "HTML Help Workshop" download. If you want to compile the guides yourself, you need to download and install this. If you don’t install it into the default directory, you may also have a look at the HHC_DIR setting in the file docbook/Makefile.

The files `htmlhelp.c` and `htmlhelp.lib` are required to be able to open .chm files from Wireshark and show the online help. Both files are part of the SDK (standalone (P)SDK or MSVC since 2002).

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\wsbuild32>set PATH="%PATH%;C:\Development\wsbuild32\run\RelwithDebInfo"
C:\Development\wsbuild32>wireshark.sln
```

for PowerShell use

```
PS C:\Development\wsbuild32>$env:PATH += ";$(Convert-Path run\RelWithDebInfo)"
PS C:\Development\wsbuild32>wireshark.sln
```

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\wsbuild32\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\wsbuild32\run\Debug\.

bash

The bash shell is needed to run several shell scripts.

UNIX: GNU Bash

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 http://www.gnu.org/software/bash/bash.html.

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

```
$ 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, documenation, and other tasks. Python 2.5 or later (including Python 3) should work fine and Python 3 is recommended. It may be required in the future.

Python is either included or available as a package on most UNIX-like platforms. Windows packages and source are available at http://python.org/download/. The Cygwin Python package is **not** recommended since */usr/bin/python* is a symbolic link, which causes confusion outside Cygwin.

You can also use Chocolatey to install Python:

```
PS:\> choco install Python3
```

or

```
PS:\> choco install Python2
```

Chocolatey installs Python into *C:\tools\python3* or *C:\tools\python2* by default. You can verify your Python version by running

```
$ python --version
```

on UNIX and Linux 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.

Perl

Perl is an interpreted programming language. The homepage of the Perl project is http://www.perl.com. Perl is used to convert various text files into usable source code. Perl version 5.6 and above should work fine.

UNIX: Perl

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

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 Native: Perl

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

You may also use Chocolatey to install either package:

```
PS:\> choco install ActivePerl
```

or

```
PS:\> choco install StrawberryPerl
```

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.

Bison

Bison is a parser generator used for some of Wireshark's file format support.

UNIX: Bison

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

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

After correct installation running the following

```
$ bison --version
```

should result in something like:

```
bison (GNU Bison) 2.3
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2006 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.

Windows Native: Win flex-bison and bison

A native Windows version of bison is available in the *winflexbison* Chocolatey package. Note that the executable is named *win_bison*.

Native packages are available from other sources such as GnuWin and Cygwin. They aren't officially supported but *should* work.

Flex

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

UNIX: flex

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 http://www.gnu.org/software/flex/.

After correct installation running the following

```
$ flex --version
```

should result in something like:

```
flex version 2.5.4
```

Your version string may vary.

Windows Native: Win flex-bison and flex

A native Windows version of flex is available in the *winflexbison* Chocolatey package. Note that the executable is named *win_flex*.

```
PS:\>choco install winflexbison
```

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 Obtain the Wireshark sources.

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

UNIX: git

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: http://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 Native: git

The Git command line tools for Windows can be found at http://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 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. Poshgit 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 http://git-scm.com/downloads/guis for details.

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 and Gerrit should handle patches for you.

You will find more instructions in Apply a patch from someone elseon how to use the patch tool.

UNIX: patch

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 http://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 native: patch

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

Windows: NSIS (optional)

The NSIS (Nullsoft Scriptable Install System) is used to generate *Wireshark-win32-2.9.0.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 http://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 Win32: NSIS .exe installer.

Windows: PortableApps (optional)

The PortableApps.com Installer is used to generate *WiresharkPortable-2.9.0.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 http://portableapps.com/.
- Install the following applications in the PortableApps.com environment:
  - PortableApps.com Installer
  - PortableApps.com Launcher
  - NSIS Portable (Unicode)
  - PortableApps.com AppCompactor

You can find more instructions on using the PortableApps.com Installer in Win32: PortableApps .paf.exe package.

Library Reference

Introduction

Several libraries are needed to build and run Wireshark. Most of them are split into three packages:

1. *Runtime*. System and third party libraries such as *MSVCR110.dll* and *libglib-2.0-0.dll*.
2. *Developer*. Documentation, header files, import libraries, and other files needed for compilation.
3. *Source*. Library sources, which are usually not required to build Wireshark.

TIP

Our libraries are freely available

All libraries required to build Wireshark on Windows are available for download at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/ and https://anonsvn.wireshark.org/wireshark-win64-libs/trunk/packages/. See Win32: Automated Library Download for an easier way to install them.

Binary library formats

Binary libraries are available in different formats, depending on the C compiler used to build it and of course the platform they were built for.

Unix

If you have installed unix binary libraries on your system, they will match the C compiler. If not already installed, the libraries should be available as a package from the platform installer, or you can download and compile the source and then install the binaries.

Win32: MSVC

Most of the Win32 binary libraries you will find on the web are in this format. You will recognize MSVC libraries by the .lib/.dll file extension.

Win32: 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-win32-libs and WIRESHARK_BASE_DIR\wireshark-win64-libs for 32 and 64 bit builds respectively.

You may also directly set the library directory with the environment variable WIRESHARK_LIB_DIR, but if you switch between 32 bit and 64 bit builds, the value of this must be set appropriately.

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 5.2 or later.

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 *qttools5-dev*, *qttools5-dev-tools*, *libqt5svg5-dev*, *qtmultimedia5-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.

Win32 MSVC

Qt5 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. Note that separate installations (into different directories) of Qt are required for 32 bit and 64 bit builds. The environment variable QT5_BASE_DIR should be set as appropriate for your environment and should point to the Qt directory that contains the bin directory, e.g. *C:\Qt\5.9.5\msvc2017_64*.

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 http://www.gtk.org/download.html.

Win32 MSVC

You can get the latest version at http://www.gtk.org/download.html.

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 http://www.ibr.cs.tu-bs.de/projects/libsmi/.

Win32 MSVC

Wireshark uses the source libSMI distribution at http://www.ibr.cs.tu-bs.de/projects/libsmi/. LibSMI is cross-compiled using MinGW32. It's stored in the libsmi zip archive at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/.

c-ares (optional)

C-Ares is used for asynchronous DNS resolution. This is the primary name resolution library in Wireshark.

Unix

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

Win32 MSVC

C-Ares is cross-compiled using MinGW32 and is available at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/.

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, http://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 http://www.zlib.net/.

Win32 MSVC

The zlib sources are downloaded from https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/ and compiled locally.

libpcap/WinPcap (optional)

Libpcap and WinPcap provide that 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 http://www.tcpdump.org/.

Win32 MSVC: WinPcap

You can get the "Windows packet capture library" at: https://www.winpcap.org/install/

GnuTLS (optional)

The GNU Transport Layer Security Library is used to dissect SSL and TLS protocols (aka: HTTPS).

Unix

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

Win32 MSVC

We provide a package cross-compiled using MinGW32 at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/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.

Win32 MSVC

Part of our GnuTLS package.

Kerberos (optional)

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

Unix

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

Win32 MSVC

We provide a package at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/.

LUA (optional)

The LUA library is used to add scripting support to Wireshark.

Unix

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

Win32 MSVC

We provide a copy of the official package at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/.

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 a package for Windows at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/.

WinSparkle (optional)

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

Win32 MSVC

We provide a copy of the WinSparkle package at https://anonsvn.wireshark.org/wireshark-win32-libs/trunk/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.

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:

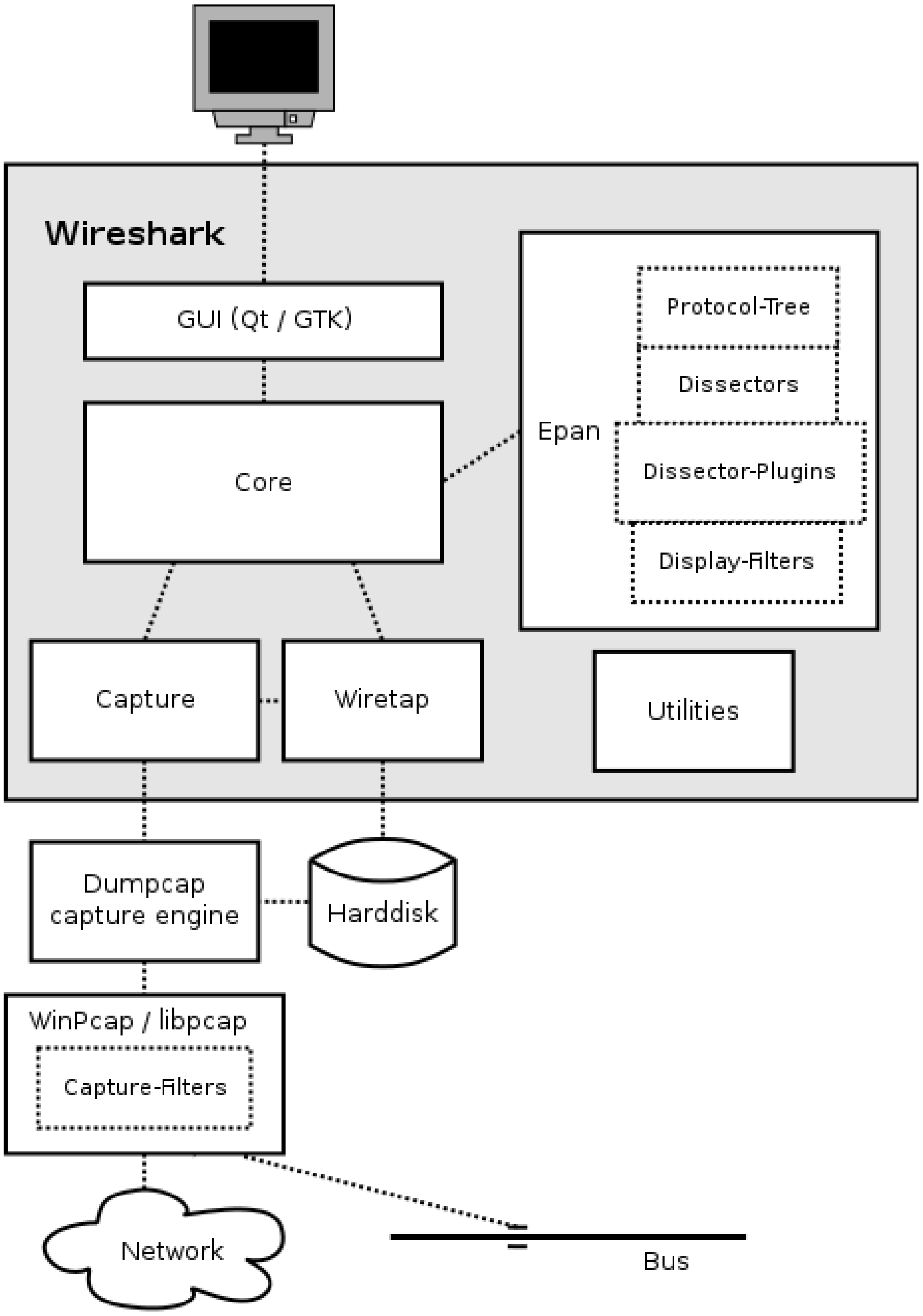

Figure 1. Wireshark function blocks

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 with the capture engine. Source code is in the root directory.

Dumpcap

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

WinPcap and libpcap

These are separate libraries that provide packet capture and filtering support on different platforms. The filtering WinPcap and libpcap works at a much lower level than Wireshark's display filters and uses a significantly different mechanism. That's why we have 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 into the `dumpcap` program. It's only this program that needs these privileges, allowing the main part 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 WinPcap (see libpcap/WinPcap (optional)) 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. In addition to this, as one of its strengths, Wireshark can read and write files in many different file formats of other network capturing tools. The wiretap library, developed together with Wireshark, provides a general purpose interface to read and write all the file formats. If you need to add support for another capture file format this is the place to start.

Dissect packets

While Wireshark is loading packets from a file each packet is dissected. Wireshark tries to detect the packet type and gets as much information from the packet as possible. In this run though, only the information shown in the packet list pane is needed.

As the user selects a specific packet in the packet list pane this packet will be dissected again. This time, Wireshark tries to get every single piece of information and put it into the packet details pane.

Introduction

Source overview

Wireshark consists of the following major parts:

- Packet dissection - in the */epan/dissector* and */plugin/*|* directories
- File I/O - using Wireshark's own wiretap library
- Capture - using the libpcap/winpcap library, in */wiretap*
- 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 "Code style" 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.

GLib contains lots of useful things for platform independent development. See https://developer.gnome.org/glib/ for details about GLib.

Packet capturing

This chapter needs to be reviewed and extended.

How to add a new capture type to libpcap

The following is an updated excerpt from a developer mailing list mail about adding ISO 9141 and 14230 (simple serial line card diagnostics) to Wireshark:

For libpcap, the first thing you'd need to do would be to get `DLT_*` values for all the link-layer protocols you'd need. If ISO 9141 and 14230 use the same link-layer protocol, they might be able to share a `DLT_*` value, unless the only way to know what protocols are running above the link layer is to know which link-layer protocol is being used, in which case you might want separate `DLT_*` values.

For the rest of the libpcap discussion, I'll assume you're working with libpcap 1.0 or later and that this is on a UN*X platform. 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.

Then you'd probably add to the `pcap_open_live()` routine, for whatever platform or platforms this code should work, something such as a check for device names that look like serial port names and, if the check succeeds, a call to a routine to open the serial port.

See, for example, the `#ifdef HAVE_DAG_API` code in *pcap-linux.c* and *pcap-bpf.c*.

The serial port open routine would open the serial port device, set the baud rate and do anything else needed to open the device. It'd allocate a `pcap_t`, set its `fd` member to the file descriptor for the serial device, set the `snapshot` member to the argument passed to the open routine, set the `linktype` member to one of the `DLT_*` values, and set the `selectable_fd` member to the same value as the `fd` member. It should also set the `dlt_count` member to the number of `DLT_*` values to support, and allocate an array of `dlt_count` `u_int`s, assign it to the `dlt_list` member, and fill in that list with all the `DLT_*` values.

You'd then set the various `_*_op` fields to routines to handle the operations in question. `read_op` is the routine that'd read packets from the device. `inject_op` would be for sending packets; if you don't care about that, you'd set it to a routine that returns an error indication. `setfilter_op` can probably just be set to `install_bpf_program`. `set_datalink` would just set the `linktype` member to the specified value if it's one of the values for OBD, otherwise it should return an error. `getnonblock_op` can probably be set to `pcap_getnonblock_fd`. `setnonblock_op` can probably be set to `pcap_setnonblock_fd`. `stats_op` would be set to a routine that reports statistics. `close_op` can probably be set to `pcap_close_common`.

If there's more than one `DLT_*` value, you definitely want a `set_datalink` routine so that the user can select the appropriate link-layer type.

For Wireshark, you'd add support for those `DLT_*` values to *wiretap/libpcap.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()`.

Packet dissection

How it 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 packet 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 will be decoded and displayed.

Dissection can be implemented in two possible ways. One is to have a dissector module compiled into the main program, which means it’s always available. Another way is to make a plugin (a shared library or DLL) that registers itself to handle dissection.

There is little difference in having your dissector as either a plugin or built-in. On the Windows platform you have limited function access through the ABI exposed by functions declared as WS_DLL_PUBLIC.

The big plus is that your rebuild cycle for a plugin is much shorter than for a built-in one. So starting with a plugin makes initial development simpler, 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 implementing 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, included in the main program, or a plugin.

Plugins are the easiest to write initially, so let’s start with that. With a little care, the plugin can be made to run as a built-in easily too so we haven’t lost anything.

Example 1. Dissector Initialisation.

```
#include "config.h"

#include <epan/packet.h>

#define FOO_PORT 1234

static int proto_foo = -1;


void
proto_register_foo(void)
{
    proto_foo = proto_register_protocol (
        "FOO Protocol", /* name       */
        "FOO",      /* short name */
        "foo"       /* abbrev     */
        );
}
```

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.

Next we have an int that is initialised to `-1` that records our protocol. This will get updated when we register this dissector with the main program. It’s good practice to make all variables and functions that aren’t exported static to keep name space pollution down. Normally this isn’t a problem unless your dissector gets so big it has to span multiple files.

Then a `#define` for the UDP port that carries *foo* traffic.

Now that we have the basics in place to interact with the main program, we’ll start with two protocol dissector setup functions.

First we’ll call `proto_register_protocol()` which registers the protocol. We can give it three names that will be used for display in various places. The full and short name are used in e.g. the "Preferences" and "Enabled protocols" dialogs as well as the generated field name list in the documentation. The abbreviation is used as the display filter name.

Next we need a handoff routine.

Example 2. Dissector Handoff.

```
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);
}
```

What's happening here? We are initialising the dissector. First we create a dissector handle; It is associated with the foo protocol and with a routine to be called to do the actual dissecting. Then we associate the handle with a UDP port number so that the main program will know to call us when it gets UDP traffic on that port.

The standard Wireshark dissector convention is to put `proto_register_foo()` and `proto_reg_handoff_foo()` as the last two functions in the dissector source.

Now at last we get to write some dissecting code. For the moment we'll leave it as a basic placeholder.

Example 3. Dissection.

```
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 out stuff in the info column */
    col_clear(pinfo->cinfo,COL_INFO);

    return tvb_captured_length(tvb);
}
```

This function is called to dissect the packets presented to it. The packet data is held in a special buffer referenced here as tvb. We shall become fairly familiar with this as we get deeper into the details of the protocol. 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.

For now we'll do the minimum we can get away with. In the first line we set the text of this to our protocol, 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 should have a basic dissector ready to compile and install. It doesn't do much at present, other than identify the protocol and label it.

In order to compile this dissector and create a plugin a couple of support files are required, besides the dissector source in *packet-foo.c*:

- *CMakeLists.txt* - Contains the CMake file and version info for this plugin.
- *packet-foo.c* - Your dissector source.
- *plugin.rc.in* - Contains the DLL resource template for Windows. (optional)

You can find a good example for these files in the gryphon plugin directory. *CMakeLists.txt* has to be modified with the correct plugin name and version info, along with 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 generated Wireshark or copy the plugin binary into the plugin directory of your Wireshark installation and run that.

Dissecting the details of the protocol

Now that we have our basic dissector up and running, let’s do something with it. The simplest thing to do to start with is to just label the payload. This will allow us to set up some of the parts we will need.

The first thing we will do is to build a subtree to decode our results into. This helps to keep things looking nice in the detailed display.

Example 4. 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);
}
```

What we’re doing here is adding a subtree to the dissection. This subtree will hold all the details of this protocol and so not clutter up the display when not required.

We are also marking the area of data that is being consumed by this protocol. In our case it’s all that has been passed to us, as we’re assuming this protocol does not encapsulate another. Therefore, we add the new tree node with `proto_tree_add_item()`, adding it to the passed in tree, label it with the protocol, use the passed in tvb buffer as the data, and consume from 0 to the end (-1) of this data.

ENC_NA ("not applicable") is specified as the "encoding" parameter.

After this change, there should be a label in the detailed display for the protocol, and selecting this will highlight the remaining contents of the packet.

Now let’s go to the next step and add some protocol dissection. For this step we’ll need to construct a couple of tables that help with dissection. This needs some additions to the `proto_register_foo()` function shown previously.

Two statically allocated arrays are added at the beginning of `proto_register_foo()`. The arrays are then registered after the call to `proto_register_protocol()`.

Example 5. Registering data structures.

```
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 gint *ett[] = {
        &ett_foo
    };

    proto_foo = proto_register_protocol (
        "FOO Protocol", /* name       */
        "FOO",      /* short name */
        "foo"       /* abbrev     */
        );

    proto_register_field_array(proto_foo, hf, array_length(hf));
    proto_register_subtree_array(ett, array_length(ett));
}
```

The variables `hf_foo_pdu_type` and `ett_foo` also need to be declared somewhere near the top of the file.

Example 6. Dissector data structure globals.

```
static int hf_foo_pdu_type = -1;

static gint ett_foo = -1;
```

Now we can enhance the protocol display with some detail.

Example 7. Dissector starting to dissect the packets.

```
    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, 0, 1, ENC_BIG_ENDIAN);
```

Now the dissection is starting to look more interesting. We have picked apart our first bit of the protocol. One byte of data at the start of the packet that defines the packet type for foo protocol.

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.

- *hf_foo_pdu_type* - The index for this node.
- *FOO PDU Type* - The label for this item.
- *foo.type* - This is the filter string. It enables us to type constructs such as `foo.type=1` into the filter box.
- *FT_UINT8* - This specifies this item is 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.

Example 8. Wrapping up the packet dissection.

```
...
static int hf_foo_flags = -1;
static int hf_foo_sequenceno = -1;
static int hf_foo_initialip = -1;
...

static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    gint 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,
            { "FOO PDU Initial IP", "foo.initialip",
            FT_IPv4, BASE_NONE,
            NULL, 0x0,
            NULL, HFILL }
        },
        ...
    ...
}
...
```

This dissects all the bits of this simple hypothetical protocol. We've introduced a new variable offsetinto 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.

Example 9. Naming the packet types.

```
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.

Example 10. Adding Names to the protocol.

```
   { &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.

Example 11. Adding Flags to the protocol.

```
#define FOO_START_FLAG 0x01
#define FOO_END_FLAG        0x02
#define FOO_PRIORITY_FLAG   0x04

static int hf_foo_startflag = -1;
static int hf_foo_endflag = -1;
static int hf_foo_priorityflag = -1;

static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    ...
        ...
        proto_tree_add_item(foo_tree, hf_foo_flags, tvb, offset, 1,
ENC_BIG_ENDIAN);
        proto_tree_add_item(foo_tree, hf_foo_startflag, tvb, offset, 1,
ENC_BIG_ENDIAN);
        proto_tree_add_item(foo_tree, hf_foo_endflag, tvb, offset, 1,
ENC_BIG_ENDIAN);
        proto_tree_add_item(foo_tree, hf_foo_priorityflag, tvb, offset, 1,
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. Note we keep the same offset for each of the flags.

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.

Example 12. Enhancing the display.

```
static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    gint offset = 0;
    guint8 packet_type = tvb_get_guint8(tvb, 0);

    col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear out stuff in the info column */
    col_clear(pinfo->cinfo,COL_INFO);
    col_add_fstr(pinfo->cinfo, COL_INFO, "Type %s",
             val_to_str(packet_type, packettypenames, "Unknown (0x%02x)"));

    proto_item *ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, ENC_NA);
    proto_item_append_text(ti, ", Type %s",
        val_to_str(packet_type, packettypenames, "Unknown (0x%02x)"));
    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;

    return tvb_captured_length(tvb);
}
```

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 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.

Example 13. Decompressing data packets for dissection.

```
    guint8 flags = tvb_get_guint8(tvb, offset);
    offset ++;
    if (flags & FLAG_COMPRESSED) { /* the remainder of the packet is compressed */
        guint16 orig_size = tvb_get_ntohs(tvb, offset);
        guchar *decompressed_buffer = (guchar*)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 `tvb_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.

```
msg_pkt ::= SEQUENCE {
    .....
    flags ::= SEQUENCE {
        fragment    BOOLEAN,
        last_fragment   BOOLEAN,
    .....
    }
    msg_id  INTEGER(0..65535),
    frag_id INTEGER(0..65535),
    .....
}
```

Example 14. Reassembling fragments - Part 1

```
#include <epan/reassemble.h>
   ...
save_fragmented = pinfo->fragmented;
flags = tvb_get_guint8(tvb, offset); offset++;
if (flags & FL_FRAGMENT) { /* fragmented */
    tvbuff_t* new_tvb = NULL;
    fragment_data *frag_msg = NULL;
    guint16 msg_seqid = tvb_get_ntohs(tvb, offset); offset += 2;
    guint16 msg_num = tvb_get_ntohs(tvb, offset); offset += 2;

    pinfo->fragmented = TRUE;
    frag_msg = fragment_add_seq_check(msg_reassembly_table,
        tvb, offset, pinfo,
        msg_seqid, NULL, /* ID for fragments belonging together */
        msg_num, /* fragment sequence number */
        tvb_captured_length_remaining(tvb, offset), /* fragment length - to the
end */
        flags & FL_FRAG_LAST); /* More fragments? */
```

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.

Example 15. Reassembling fragments part 2

```
    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()`.

Example 16. Reassembling fragments - Initialisation

```
static reassembly_table reassembly_table;

static void
proto_register_msg(void)
{
    reassembly_table_register(&msg_reassemble_table,
        &addresses_ports_reassembly_table_functions);
}
```

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.

Example 17. Reassembling fragments - Data

```
...
static int hf_msg_fragments = -1;
static int hf_msg_fragment = -1;
static int hf_msg_fragment_overlap = -1;
static int hf_msg_fragment_overlap_conflicts = -1;
static int hf_msg_fragment_multiple_tails = -1;
static int hf_msg_fragment_too_long_fragment = -1;
static int hf_msg_fragment_error = -1;
static int hf_msg_fragment_count = -1;
static int hf_msg_reassembled_in = -1;
static int hf_msg_reassembled_length = -1;
...
static gint ett_msg_fragment = -1;
static gint ett_msg_fragments = -1;
...
static const fragment_items msg_frag_items = {
    /* Fragment subtrees */
    &ett_msg_fragment,
    &ett_msg_fragments,
    /* Fragment fields */
    &hf_msg_fragments,
    &hf_msg_fragment,
    &hf_msg_fragment_overlap,
    &hf_msg_fragment_overlap_conflicts,
    &hf_msg_fragment_multiple_tails,
    &hf_msg_fragment_too_long_fragment,
    &hf_msg_fragment_error,
    &hf_msg_fragment_count,
    /* Reassembled in field */
    &hf_msg_reassembled_in,
    /* Reassembled length field */
    &hf_msg_reassembled_length,
    /* Tag */
    "Message fragments"
};
...
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 } },
...
static gint *ett[] =
{
...
&ett_msg_fragment,
&ett_msg_fragments
...
```

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*.

Example 18. Reassembling TCP fragments

```
#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 guint
get_foo_message_len(packet_info *pinfo _U_, tvbuff_t *tvb, int offset, void *data
_U_)
{
    /* TODO: change this to your needs */
    return (guint)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 you 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.

Example 19. Initialising a tap

```
#include <epan/packet.h>
#include <epan/tap.h>

static int foo_tap = -1;

struct FooTap {
    gint packet_type;
    gint 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.

Example 20. Calling a protocol tap

```
static int
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
       ...
    fooinfo = wmem_alloc(wmem_packet_scope(), 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.

Example 21. Initialising a stats interface

```
/* register all http trees */
static void register_foo_stat_trees(void) {
    stats_tree_register_plugin("foo", "foo", "Foo/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.

Example 22. Initialising a stats session

```
static const guint8* st_str_packets = "Total Packets";
static const guint8* 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, 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.

Example 23. Generating the stats

```
static int foo_stats_tree_packet(stats_tree* st, packet_info* pinfo,
epan_dissect_t* edt, const void* p)
{
    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, msgtypevalues, "Unknown packet type
(%d)"));
    return 1;
}
```

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 http://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
          ${CMAKE_CURRENT_SOURCE_DIR}/packet-3com-njack.c
          ...
  ```

  becomes

  ```
  DISSECTOR_SRC = \
          ${CMAKE_CURRENT_SOURCE_DIR}/packet-test-idl.c       \
          ${CMAKE_CURRENT_SOURCE_DIR}/packet-2dparityfec.c    \
          ${CMAKE_CURRENT_SOURCE_DIR}/packet-3com-njack.c     \
          ...
  ```

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. On my Linux box, it is PYTHONPATH=/usr/lib/python2.4/

Lua Support in Wireshark

Introduction

Wireshark has an embedded Lua interpreter. Lua is a powerful light-weight programming language designed for extending applications. Lua is designed and implemented by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Lua was born and raised at Tecgraf, the Computer Graphics Technology Group of PUC-Rio, and is now housed at Lua.org. Both Tecgraf and Lua.org are laboratories of the Department of Computer Science.

In Wireshark Lua can be used to write dissectors, taps, and capture file readers and writers.

Wireshark's Lua interpreter starts by loading `init.lua` that is located in the global configuration directory of Wireshark. Lua is enabled by default. To disable Lua the line variable *disable_lua* should be set to *true* in `init.lua`.

After loading *init.lua* from the data directory if Lua is enabled Wireshark will try to load a file named `init.lua` in the user's directory.

Wireshark will also load all files with `.lua` suffix from both the global and the personal plugins directory.

The command line option *-X lua_script:*`file.lua` can be used to load Lua scripts as well.

The Lua code will be executed once after all the protocol dissectors have being initialized and before reading any file.

Example of Dissector written in Lua

```
local p_multi = Proto("multi", "MultiProto");

local vs_protos = {
        [2] = "mtp2",
        [3] = "mtp3",
        [4] = "alcap",
        [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)
```

Example of Listener written in Lua

```
-- This program will register a menu that will open a window with a count of
```

```
occurrences
-- of every address in the capture

local 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

    -- 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.

Saving capture files

The classes/functions defined in this module are for using a `Dumper` object to make Wireshark save a capture file to disk. `Dumper` represents Wireshark’s built-in file format writers (see the `wtap_filetypes` table in `init.lua`).

To have a Lua script create its own file format writer, see the chapter titled "Custom file format reading/writing".

Dumper

Dumper.new(filename, [filetype], [encap])

Creates a file to write packets. `Dumper:new_for_current()` will probably be a better choice.

Arguments

filename

The name of the capture file to be created.

filetype (optional)

The type of the file to be created - a number entry from the `wtap_filetypes` table in `init.lua`.

encap (optional)

The encapsulation to be used in the file to be created - a number entry from the `wtap_encaps` table in `init.lua`.

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.

Returns

The MTP2 pseudoheader

Obtaining dissection data

Field

A Field extractor to 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.

Since: 1.11.3

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.

Since: 1.99.8

field.display

Mode: Retrieve only.

The full display name of this field, or nil.

Since: 1.99.8

field.type

Mode: Retrieve only.

The `ftype` of this field, or nil.

Since: 1.99.8

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:get_range()` to get the `TvbRange`, and then use `Tvb: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 rhs 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 in `init.lua`.

Since: 1.99.8

fieldinfo.source

Mode: Retrieve only.

The source `Tvb` object the `FieldInfo` is derived from, or nil if there is none.

Since: 1.99.8

fieldinfo.range

Mode: Retrieve only.

The `TvbRange` covering the bytes of this field in a Tvb.

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).

Since: 1.99.8

fieldinfo.is_url

Mode: Retrieve only.

Whether this field was marked as being a URL (boolean).

Since: 1.99.8

fieldinfo.little_endian

Mode: Retrieve only.

Whether this field is little-endian encoded (boolean).

Since: 1.99.8

fieldinfo.big_endian

Mode: Retrieve only.

Whether this field is big-endian encoded (boolean).

Since: 1.99.8

fieldinfo.name

Mode: Retrieve only.

The filter name of this field.

Since: 1.99.8

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

GUI support

ProgDlg

Manages a progress bar dialog.

ProgDlg.new([title], [task])

Creates a new `ProgDlg` progress dialog.

Arguments

title (optional)
Title of the new window, defaults to "Progress".

task (optional)
Current task, defaults to "".

Returns

The newly created `ProgDlg` object.

progdlg:update(progress, [task])

Appends text.

Arguments

progress
Part done (e.g. 0.75).

task (optional)
Current task, defaults to "".

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 stop button.

Returns

true if the user has asked to stop the progress.

progdlg:close()

Closes the progress dialog.

Returns

A string specifying whether the Progress Dialog has stopped or not.

Errors

- GUI not available

TextWindow

Manages a text window.

TextWindow.new([title])

Creates a new `TextWindow` text window.

Arguments

title (optional)
: Title of the new 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.

Arguments

text
: The text to be used.

Returns

The `TextWindow` object.

Errors

- GUI not available

textwindow:append(text)

Appends text

Arguments

text
: The text to be appended

Returns

The `TextWindow` object.

Errors

- GUI not available

textwindow:prepend(text)

Prepends text

Arguments

text
: The text to be appended

Returns

The `TextWindow` object.

Errors

- GUI not available

textwindow:clear()

Erases all 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)
  A boolean flag, defaults to true.

Returns

The `TextWindow` object.

Errors

- GUI not available

textwindow:add_button(label, function)

Adds a button to the text window.

Arguments

label
: The label of the button

function
: The Lua function to be called when clicked

Returns

The `TextWindow` object.

Errors

- GUI not available

Global Functions

gui_enabled()

Checks whether the GUI facility is enabled.

Returns

A boolean: true if it is enabled, false if it isn't.

register_menu(name, action, [group])

Register a menu item in one of the main menus.

Arguments

name
: The name of the menu item. The submenus are to be separated by '`/`'s. (string)

action
: The function to be called when the menu item is invoked. (function taking no arguments and returning nothing)

group (optional)
: The menu group into which the menu item is to be inserted. If omitted, defaults to MENU_STAT_GENERIC. One of:

  - MENU_STAT_UNSORTED (Statistics),
  - MENU_STAT_GENERIC (Statistics, first section),
  - MENU_STAT_CONVERSATION (Statistics/Conversation List),

    - MENU_STAT_ENDPOINT (Statistics/Endpoint List),
    - MENU_STAT_RESPONSE (Statistics/Service Response Time),
    - MENU_STAT_TELEPHONY (Telephony),
    - MENU_STAT_TELEPHONY_GSM (Telephony/GSM),
    - MENU_STAT_TELEPHONY_LTE (Telephony/LTE),
    - MENU_STAT_TELEPHONY_SCTP (Telephony/SCTP),
    - MENU_ANALYZE (Analyze),
    - MENU_ANALYZE_CONVERSATION (Analyze/Conversation Filter),
    - MENU_TOOLS_UNSORTED (Tools). (number)

new_dialog(title, action, …)

Pops up a new dialog

Arguments

title
  Title of the dialog's window.

action
  Action to be performed when OK'd.

…
  A series of strings to be used as labels of the dialog's fields.

Errors

- GUI not available
- At least one field required
- All fields must be strings

retap_packets()

Rescan all packets and just run taps - don't reconstruct the display.

copy_to_clipboard(text)

Copy a string into the clipboard.

Arguments

text
  The string to be copied into the clipboard.

open_capture_file(filename, filter)

Open and display a capture file.

Arguments

filename
: The name of the file to be opened.

filter
: A filter to be applied as the file gets opened.

get_filter()

Get the main filter text.

set_filter(text)

Set the main filter text.

Arguments

text
: The filter's text.

set_color_filter_slot(row, text)

Set packet-coloring rule for the current session.

Arguments

row
: The index of the desired color in the temporary coloring rules list.

text
: Display filter for selecting packets to be colorized.

apply_filter()

Apply the filter in the main filter box.

reload()

Reload the current capture file. Obsolete, use reload_packets()

reload_packets()

Reload the current capture file.

reload_lua_plugins()

Reload all Lua plugins.

browser_open_url(url)

Open an url in a browser.

Arguments

url
: The url.

browser_open_data_file(filename)

Open a file in a browser.

Arguments

filename
: The file name.

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` listener object.

Arguments

tap (optional)
: The name of this tap.

filter (optional)
: A filter that when matches the `tap.packet` function gets called (use nil to be called for every packet).

allfields (optional)
: Whether to generate all fields. (default=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.

Since: 1.11.3

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

```
function tap.packet(pinfo,tvb,tapinfo) ... end
```

NOTE `tapinfo` is a table of info based on the `Listener`'s type, or nil.

listener.draw

Mode: Assign only.

A function that will be called once every few seconds to redraw the GUI objects; in Tshark this funtion is called only at the very end of the capture file.

When later called by Wireshark, the `draw` function will not be given any arguments.

```
function tap.draw() ... end
```

listener.reset

Mode: Assign only.

A function that will be called at the end of the capture run.

When later called by Wireshark, the `reset` function will not be given any arguments.

```
function tap.reset() ... end
```

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:__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.

Since: 1.10.6

column:clear_fence()

Clear Column text fence.

Since: 1.11.3

Columns

The Columns of the packet list.

columns:__tostring()

Returns

The string "Columns", no real use, just for debugging purposes.

columns:__newindex(column, text)

Sets the text of a specific column.

Arguments

column
The name of the column to set.

text
The text for the column.

columns:__index()

Gets 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

Since: 2.4.0

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.

pinfo.len

Mode: Retrieve only.

The length of the frame.

pinfo.caplen

Mode: Retrieve only.

The captured length of the frame.

pinfo.abs_ts

Mode: Retrieve only.

When the packet was captured.

pinfo.rel_ts

Mode: Retrieve only.

Number of seconds passed since beginning of capture.

pinfo.delta_ts

Mode: Retrieve only.

Number of seconds passed since the last captured packet.

pinfo.delta_dis_ts

Mode: Retrieve only.

Number of seconds passed since the last displayed packet.

pinfo.curr_proto

Mode: Retrieve only.

Which Protocol are we dissecting.

pinfo.can_desegment

Mode: Retrieve or assign.

Set if this segment could be desegmented.

pinfo.desegment_len

Mode: Retrieve or assign.

Estimated number of additional bytes required for completing the PDU.

pinfo.desegment_offset

Mode: Retrieve or assign.

Offset in the tvbuff at which the dissector will continue processing when next called.

pinfo.fragmented

Mode: Retrieve only.

If the protocol is only a fragment.

pinfo.in_error_pkt

Mode: Retrieve only.

If we’re inside an error packet.

pinfo.match_uint

Mode: Retrieve only.

Matched uint for calling subdissector from table.

pinfo.match_string

Mode: Retrieve only.

Matched string for calling subdissector from table.

pinfo.port_type

Mode: Retrieve or assign.

Type of Port of .src_port and .dst_port.

pinfo.src_port

Mode: Retrieve or assign.

Source Port of this Packet.

pinfo.dst_port

Mode: Retrieve or assign.

Source Address 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.match

Mode: Retrieve only.

Port/Data we are matching.

pinfo.columns

Mode: Retrieve only.

Accesss to the packet list columns.

pinfo.cols

Mode: Retrieve only.

Accesss 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 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 `DissectorTables` 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 refererence 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.

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.

Since: 1.11.3

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 protocol short name.

Returns

A string of the protocol’s short name.

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])

Creates a new DissectorTable for your dissector’s use.

Arguments

tablename
: The short name of the table.

uiname (optional)
: The name of the table in the User Interface (defaults to the name given).

type (optional)
: Either `ftypes.UINT8`, `ftypes.UINT16`, `ftypes.UINT24`, `ftypes.UINT32`, or `ftypes.STRING` (defaults to `ftypes.UINT32`).

base (optional)
: Either `base.NONE`, `base.DEC`, `base.HEX`, `base.OCT`, `base.DEC_HEX` or `base.HEX_DEC` (defaults to `base.DEC`).

Returns

The newly created DissectorTable.

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.

Since: 1.11.3

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.

Since: 1.11.3

Returns

The array table of registered heuristic list names

DissectorTable.get(tablename)

Obtain a reference to an existing dissector table.

Arguments

tablename
: The short name of the table.

Returns

The DissectorTable.

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)

Remove existing dissectors from a table and add a new or a range of new dissectors.

Since: 1.11.3

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.

Since: 1.11.3

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 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.

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.

nil if not found.

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.

Since: 1.99.1

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 Protocol.

Pref.bool(label, default, descr)

Creates a boolean 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.

descr

A description of what this preference is.

Pref.uint(label, default, descr)

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.

descr

A description of what this preference is.

Pref.string(label, default, descr)

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.

descr

A description of what this preference is.

Pref.enum(label, default, descr, enum, radio)

Creates an enum 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.

descr

A description of what this preference is.

enum

An enum Lua table.

radio

Radio button (true) or Combobox (false).

Pref.range(label, default, descr, max)

Creates a range 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".

descr

A description of what this preference is.

max

The maximum value.

Pref.statictext(label, descr)

Creates a static text string to be added to a `Proto.prefs` Lua table.

Arguments

label

The static text.

descr

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.

pref

A valid but still unassigned Pref object.

Errors

- Unknown Pref type

prefs:__index(name)

Get the value of a preference setting.

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 more uses, the main one is to dissect a protocol. But they can also be just dummies used to register preferences for other purposes.

Proto.new(name, desc)

Arguments

name
: The name of the protocol.

desc
: A Long Text description of the protocol (usually lowercase).

Returns

The newly created protocol.

proto:__call(name, desc)

Creates a `Proto` object.

Arguments

name
: The name of the protocol.

desc
: 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.

Since: 1.11.3

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.

proto.init

Mode: Assign only.

The init routine of this dissector, a function you define.

The called init function 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`.

Since: 1.11.3

ProtoExpert

A Protocol expert info field, to be used when adding items to the dissection tree.

Since: 1.11.3

ProtoExpert.new(abbr, text, group, severity)

Creates a new `ProtoExpert` object to be used for a protocol's expert information notices.

Since: 1.11.3

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` or `expert.group.DECRYPTION`.

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.

Since: 1.11.3

ProtoField

A Protocol field (to be used when adding items to the dissection tree).

ProtoField.new(name, abbr, type, [valuestring], [base], [mask], [descr])

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.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 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` or `base.UNIT_STRING`.

mask (optional)

The bitmask to be used.

descr (optional)

The description of the field.

Returns

The newly created `ProtoField` object.

ProtoField.uint8(abbr, [name], [base], [valuestring], [mask], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

Creates a `ProtoField` of a signed 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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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` or `base.UNIT_STRING`.

valuestring (optional)

A table containing the text that corresponds to the values, or a table containing unit name for the values if base is `base.UNIT_STRING`.

mask (optional)

Integer mask of this field.

desc (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], [desc])

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 mask of this field, which must be 0 for framenum.

desc (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], [desc])

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 mask of this field.

desc (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], [desc])

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`.

desc (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], [desc])

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).

desc (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], [desc])

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.

desc (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], [desc])

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.

desc (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], [desc])

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`.

desc (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], [desc])

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`.

desc (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], [desc])

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`.

desc (optional)

Description of the field.

Returns

A `ProtoField` object to be added to a table set to the `Proto.fields` attribute.

ProtoField.ubytes(abbr, [name], [display], [desc])

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`.

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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], [desc])

Creates a `ProtoField` for a sub-protocol. Since 1.99.9.

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).

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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], [desc])

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).

desc (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).

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 shouild not be used for protocols whose PDU length cannot be determined from a fixed minimum portion, such as HTTP or Telnet.

Since: 1.99.2

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)

Adding information to the dissection tree

TreeItem

`TreeItem`s represent information in the packet-details pane of Wireshark, and the packet details view of Tshark. A `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: other children of the same `TreeItem` subtree.

During dissection, heuristic-dissection, and post-dissection, a root `TreeItem` is passed to dissectors as the third argument of the function callback (e.g., `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 `tree.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 the value of the extracted field (i.e., a number, `UInt64`, `Address`, etc.). If the value could not be extracted from the `TvbRange`, the child `TreeItem` is still returned, but the second returned value is `nil`.

Another new feature added to this function in Wireshark version 1.11.3 is the ability to extract native number `ProtoField`s from string encoding in the `TvbRange`, for ASCII-based and similar string encodings. For example, a `ProtoField` of as `ftypes.UINT32` type 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 (see init.lua). 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 Tvb named "tvb" contains the string "123":

```
     -- this is done earlier in the script
     local myfield = ProtoField.new("Transaction ID", "myproto.trans_id",
ftypes.UINT16)

     -- this is done inside a dissector, post-dissector, or heuristic function
     -- child will be the created child tree, and value will be the number 123 or nil
on failure
     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.

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` or `PI_DEBUG`.

severity (optional)

One of `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.

Since: 1.11.3

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.

Since: 1.11.3

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).

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=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 a Dissector. The Dissector form is usefull when you need to decide whether to call a sub-dissector.

Since: 2.4.0

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`.

Since: 1.99.8

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.

Since: 1.99.3

treeitem.visible

Mode: Retrieve only.

Get the `TreeItem`'s subtree visibility status (boolean).

Since: 1.99.8

treeitem.generated

Mode: Retrieve or assign.

Set/get the `TreeItem`'s generated state (boolean).

Since: 1.99.8

treeitem.hidden

Mode: Retrieve or assign.

Set/get `TreeItem`'s hidden state (boolean).

Since: 1.99.8

treeitem.len

Mode: Retrieve or assign.

Set/get `TreeItem`'s length inside tvb, after it has already been created.

Since: 1.99.8

Functions for handling packet data

ByteArray

ByteArray.new([hexbytes], [separator])

Creates a `ByteArray` object.

Starting in version 1.11.3, if the second argument is a boolean `true`, then the first argyument is treated as a raw Lua string of bytes to use, instead of a hexadecimal string.

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.

Since: 1.11.4

Arguments

first
: First array.

second
: Second array.

bytearray:prepend(prepended)

Prepend a `ByteArray` to this `ByteArray`.

Arguments

prepended
: `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: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`.

Since: 1.11.3

Returns

The created `ByteArray`.

bytearray:raw([offset], [length])

Obtain a Lua string of the binary bytes in a `ByteArray`.

Since: 1.11.3

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

Since: 1.11.3

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` (it gets added to the current frame too).

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:len()`.

WARNING	Tvbs are usable only by the current listener or dissector call and are destroyed as soon as the listener/dissector returns, so references to them are unusable once the function has returned.

tvb:__tostring()

Convert the bytes of a `Tvb` into a string, to be used for debugging purposes, as … will be appended if the string is too long.

Returns

The string.

tvb:reported_len()

Obtain the reported (not captured) length of a `Tvb`.

Returns

The reported length of the `Tvb`.

tvb:len()

Obtain the actual (captured) length of a `Tvb`.

Returns

The captured length of the `Tvb`.

tvb:reported_length_remaining()

Obtain the reported (not captured) length of packet data to end of a `Tvb` or -1 if the offset is beyond the end of the `Tvb`.

Returns

The captured length of the `Tvb`.

tvb:bytes([offset], [length])

Obtain a `ByteArray` from a `Tvb`.

Since: 1.99.8

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 until the end of the `Tvb`.

Returns

The TvbRange

tvb:raw([offset], [length])

Obtain a Lua string of the binary bytes in a `Tvb`.

Since: 1.11.3

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 `Tvb`.

tvb:__eq()

Checks whether the two `Tvb` contents are equal.

Since: 1.99.8

TvbRange

A `TvbRange` represents a usable range of a `Tvb` and is used to extract data from the `Tvb` that generated it.

`TvbRange`s are created by calling a `Tvb` (e.g. *tvb(offset,length)*). If the `TvbRange` span is outside the `Tvb`'s range the creation will cause a runtime error.

tvbrange:tvb()

Creates a (sub)`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.

tvbrange:ether()

Get an Ethernet Address from a `TvbRange`, as an `Address` object.

Returns

The Ethernet `Address` object.

Errors

- The range must be 6 bytes long

tvbrange:nstime([encoding])

Obtain a time_t structure from a `TvbRange`, 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

tvbrange:le_nstime()

Obtain a nstime from a `TvbRange`, as an `NSTime` object.

Returns

The `NSTime` object.

Errors

- The range must be 4 or 8 bytes long

tvbrange:string([encoding])

Obtain a string from a `TvbRange`.

Arguments

encoding (optional)
The encoding to use. Defaults to ENC_ASCII.

Returns

The string

tvbrange:ustring()

Obtain a Big Endian (network order) UTF-16 encoded string from a `TvbRange`.

Returns

The string.

tvbrange:le_ustring()

Obtain a Little Endian UTF-16 encoded string from a `TvbRange`.

Returns

The string.

tvbrange:stringz([encoding])

Obtain a zero terminated string from a `TvbRange`.

Arguments

encoding (optional)
The encoding to use. Defaults to ENC_ASCII.

Returns

The zero terminated string.

tvbrange:strsize([encoding])

Find the size of a zero terminated string from a `TvbRange`. The size of the string includes the terminating zero.

Since: 1.11.3

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 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`.

tvbrange:offset()

Obtain the offset in a `TvbRange`.

tvbrange:raw([offset], [length])

Obtain a Lua string of the binary bytes in a `TvbRange`.

Since: 1.11.3

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 `TvbRange`.

tvbrange:__eq()

Checks whether the two `TvbRange` contents are equal.

Since: 1.99.8

tvbrange:__tostring()

Converts the `TvbRange` into a string. Since the string gets truncated, you should use this only for debugging purposes or if what you want is to have a truncated string in the format 67:89:AB:…

Returns

A Lua hex string of the first 24 binary bytes in the `TvbRange`.

Custom file format reading/writing

The classes/functions defined in this section allow you to create your own custom Lua-based "capture" file reader, or writer, or both.

Since: 1.11.3

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.

Since: 1.11.3

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` in `init.lua` 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` in `init.lua` 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:

```
    { 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.

Since: 1.11.3

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` in init.lua 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
```

Since: 1.11.3

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` in init.lua 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.

Since: 1.11.3

FileHandler.new(name, shortname, description, type)

Creates a new FileHandler

Arguments

name
  The name of the file type, for display purposes only. E.g., "Wireshark - pcapng"

shortname
  The file type short name, used as a shortcut in various places. E.g., "pcapng". Note: The name cannot already be in use.

description
  Descriptive text about this file format, for 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`:

```
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
```

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.

Since: 1.11.3

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

file
: The File object userdata, provided by Wireshark previously in a reading-based callback.

length
: 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.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` in `init.lua` for values.

frameinfo.flags

Mode: Retrieve or assign.

The presence flags of the packet frame.

See `wtap_presence_flags` in `init.lua` 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` in `init.lua` for possible packet encapsulation types to use as the value for this field.

frameinfo.comment

Mode: Retrieve or assign.

A string comment for the packet, if the `wtap_presence_flags.COMMENTS` was set in the presence flags; nil if there is no comment.

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.

Since: 1.11.3

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.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` in `init.lua` for values.

frameinfoconst.flags

Mode: Retrieve only.

The presence flags of the packet frame - see `wtap_presence_flags` in `init.lua` 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` in `init.lua` for possible packet encapsulation types to use as the value for this field.

frameinfoconst.comment

Mode: Retrieve only.

A comment for the packet; nil if there is none.

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 and others.

IF the directory was created successfully, a boolean `true` is returned. If the directory cannot be made because it already exists, `false` is returned. If the directory cannot be made because an error occurred, `nil` is returned.

Since: 1.11.3

Arguments

name

The name of the directory, possibly including path.

Returns

Boolean `true` on success, `false` if already exists, `nil` on error.

Dir.exists(name)

Returns true if the given directory name exists.

If the directory exists, a boolean `true` is returned. If the path is a file instead, `false` is returned. If the path does not exist or an error occurred, `nil` is returned.

Since: 1.11.3

Arguments

name

The name of the directory, possibly including path.

Returns

Boolean `true` if the directory exists, `false` if it's a file, `nil` on error/not-exist.

Dir.remove(name)

Removes an 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.

This function only removes empty directories. To remove a directory regardless, use `Dir.remove_all()`.

Since: 1.11.3

Arguments

name

The name of the directory, possibly including path.

Returns

Boolean `true` on success, `false` if does not exist, `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.

Since: 1.11.3

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.

```
    for filename in Dir.open(path) do ... 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.

Since: 1.11.3

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.

Since: 1.11.3

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.

Since: 1.11.3

Returns

The pathname for the personal plugins directory.

Dir.global_plugins_path()

Gets the global plugins directory path.

Since: 1.11.3

Returns

The pathname for the global plugins directory.

dir:__call()

At every invocation will return one file (nil when done).

dir:close()

Closes the directory.

Utility Functions

Global Functions

get_version()

Gets a string of the Wireshark version.

Returns

version string

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:

```
        local my_info = {
            version = "1.0.1",
            author = "Jane Doe",
            repository = "https://github.com/octocat/Spoon-Knife"
        }

        set_plugin_info(my_info)
```

Since: 1.99.8

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 formated date

format_time(timestamp)

Formats a relative timestamp in a human readable form.

Arguments

timestamp
 A timestamp value to convert.

Returns

A string with the formated time

report_failure(text)

Reports a failure to the user.

Arguments

text
 Message text to report.

critical(...)

Will add a log entry with critical severity.

Arguments

...
 Objects to be printed

warn(...)

Will add a log entry with warn severity.

Arguments

...
 Objects to be printed

message(...)

Will add a log entry with message severity.

Arguments

...
 Objects to be printed

info(...)

Will add a log entry with info severity.

Arguments

...
: Objects to be printed

debug(...)

Will add a log entry with debug severity.

Arguments

...
: Objects to be printed

loadfile(filename)

Lua’s loadfile() has been modified so that if a file does not exist in the current directory it will look for it in wireshark’s user and system directories.

Arguments

filename
: Name of the file to be loaded.

dofile(filename)

Lua’s dofile() has been modified so that if a file does not exist in the current directory it will look for it in wireshark’s user and system directories.

Arguments

filename
: Name of the file to be run.

register_stat_cmd_arg(argument, [action])

Register a function to handle a `-z` option

Arguments

argument
: Argument

action (optional)
: Action

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 a 64 bit integer with full precision.

For details, see https://wiki.wireshark.org/LuaAPI/Int64.

Int64

`Int64` represents a 64 bit signed integer.

For details, see https://wiki.wireshark.org/LuaAPI/Int64.

Int64.decode(string, [endian])

Decodes an 8-byte Lua string, using given endianness, into a new `Int64` object.

Since: 1.11.3

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/nil, native host endian.

Returns

The `Int64` object created, or nil on failure.

Int64.new([value], [highvalue])

Creates a `Int64` Object.

Since: 1.11.3

Arguments

value (optional)
: A number, `UInt64`, `Int64`, or string of ASCII digits to assign the value of the new `Int64` (default=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 `Int64` object.

Int64.max()

Gets the max possible value.

Since: 1.11.3

Returns

The new `Int64` object of the max value.

Int64.min()

Gets the min possible value.

Since: 1.11.3

Returns

The new `Int64` object of the min value.

Int64.fromhex(hex)

Creates an `Int64` object from the given hex string.

Since: 1.11.3

Arguments

hex
: The hex-ascii Lua string.

Returns

The new `Int64` object.

int64:encode([endian])

Encodes the `Int64` number into an 8-byte Lua string, using given endianness.

Since: 1.11.3

Arguments

endian (optional)
: If set to true then little-endian is used, if false then big-endian; if missing/nil, native host endian.

Returns

The Lua string.

int64:__call()

Creates a `Int64` Object.

Since: 1.11.3

Returns

The new `Int64` object.

int64:tonumber()

Returns a Lua number of the `Int64` value - this may lose precision.

Since: 1.11.3

Returns

The Lua number.

int64:tohex([numbytes])

Returns a hex string of the `Int64` value.

Since: 1.11.3

Arguments

numbytes (optional)
The number of hex-chars/nibbles to generate, negative means uppercase (default=16).

Returns

The string hex.

int64:higher()

Returns a Lua number of the higher 32-bits of the `Int64` value. (negative `Int64` will return a negative Lua number).

Since: 1.11.3

Returns

The Lua number.

int64:lower()

Returns a Lua number of the lower 32-bits of the `Int64` value. (always positive).

Since: 1.11.3

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`, in a new `Int64`.

Since: 1.11.3

Returns

The new `Int64`.

int64:__add()

Adds two `Int64` together and returns a new one (this may wrap the value).

Since: 1.11.3

int64:__sub()

Subtracts two `Int64` and returns a new one (this may wrap the value).

Since: 1.11.3

int64:__mul()

Multiplies two `Int64` and returns a new one (this may truncate the value).

Since: 1.11.3

int64:__div()

Divides two `Int64` and returns a new one (integer divide, no remainder). Trying to divide by zero results in a Lua error.

Since: 1.11.3

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.

Since: 1.11.3

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).

Since: 1.11.3

Returns

The `Int64` object.

int64:__eq()

Returns true if both `Int64` are equal.

Since: 1.11.3

int64:__lt()

Returns true if first `Int64` < second.

Since: 1.11.3

int64:__le()

Returns true if first `Int64` ⇐ second.

Since: 1.11.3

int64:bnot()

Returns a `Int64` of the bitwise *not* operation.

Since: 1.11.3

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.

Since: 1.11.3

Returns

The `Int64` object.

int64:bor()

Returns a `Int64` of the bitwise *or* operation, with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

Since: 1.11.3

Returns

The `Int64` object.

int64:bxor()

Returns a `Int64` of the bitwise *xor* operation, with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

Since: 1.11.3

Returns

The `Int64` object.

int64:lshift(numbits)

Returns a `Int64` of the bitwise logical left-shift operation, by the given number of bits.

Since: 1.11.3

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.

Since: 1.11.3

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.

Since: 1.11.3

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).

Since: 1.11.3

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).

Since: 1.11.3

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.

Since: 1.11.3

Returns

The `Int64` object.

UInt64

`UInt64` represents a 64 bit unsigned integer, similar to `Int64`.

For details, see: https://wiki.wireshark.org/LuaAPI/Int64.

UInt64.decode(string, [endian])

Decodes an 8-byte Lua binary string, using given endianness, into a new `UInt64` object.

Since: 1.11.3

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/nil, native host endian.

Returns

The `UInt64` object created, or nil on failure.

UInt64.new([value], [highvalue])

Creates a `UInt64` Object.

Since: 1.11.3

Arguments

value (optional)
: A number, `UInt64`, `Int64`, or string of digits to assign the value of the new `UInt64` (default=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()

Gets the max possible value.

Since: 1.11.3

Returns

The max value.

UInt64.min()

Gets the min possible value (i.e., 0).

Since: 1.11.3

Returns

The min value.

UInt64.fromhex(hex)

Creates a `UInt64` object from the given hex string.

Since: 1.11.3

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.

Since: 1.11.3

Arguments

endian (optional)
If set to true then little-endian is used, if false then big-endian; if missing/nil, native host endian.

Returns

The Lua binary string.

uint64:__call()

Creates a `UInt64` Object.

Since: 1.11.3

Returns

The new `UInt64` object.

uint64:tonumber()

Returns a Lua number of the `UInt64` value - this may lose precision.

Since: 1.11.3

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.

Since: 1.11.3

Arguments

numbytes (optional)
: The number of hex-chars/nibbles to generate, negative means uppercase (default=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.

Since: 1.11.3

Returns

The `UInt64` object.

uint64:__add()

Adds two `UInt64` together and returns a new one (this may wrap the value).

Since: 1.11.3

uint64:__sub()

Subtracts two `UInt64` and returns a new one (this may wrap the value).

Since: 1.11.3

uint64:__mul()

Multiplies two `UInt64` and returns a new one (this may truncate the value).

Since: 1.11.3

uint64:__div()

Divides two `UInt64` and returns a new one (integer divide, no remainder). Trying to divide by zero results in a Lua error.

Since: 1.11.3

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.

Since: 1.11.3

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).

Since: 1.11.3

Returns

The `UInt64` object.

uint64:__eq()

Returns true if both `UInt64` are equal.

Since: 1.11.3

uint64:__lt()

Returns true if first `UInt64` < second.

Since: 1.11.3

uint64:__le()

Returns true if first `UInt64` ⇐ second.

Since: 1.11.3

uint64:bnot()

Returns a `UInt64` of the bitwise *not* operation.

Since: 1.11.3

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.

Since: 1.11.3

Returns

The `UInt64` object.

uint64:bor()

Returns a `UInt64` of the bitwise *or* operation, with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

Since: 1.11.3

Returns

The `UInt64` object.

uint64:bxor()

Returns a `UInt64` of the bitwise *xor* operation, with the given number/`Int64`/`UInt64`. Note that multiple arguments are allowed.

Since: 1.11.3

Returns

The `UInt64` object.

uint64:lshift(numbits)

Returns a `UInt64` of the bitwise logical left-shift operation, by the given number of bits.

Since: 1.11.3

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.

Since: 1.11.3

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.

Since: 1.11.3

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).

Since: 1.11.3

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).

Since: 1.11.3

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.

Since: 1.11.3

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 fist 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).
- ‘`i`*n*’ a signed integer with *n* bytes. An absent *n* means the native size of an int.
- ‘`I`*n*’ like ‘`i`*n*’ 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.
- ‘`c`*n*’ 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 ‘`c`*n*’, 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.
- ‘`x`*n*’ pad to *n* number of bytes, default 1.
- ‘`X`*n*’ 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.

Since: 1.11.3

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

GLib 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 GLib's Regex implementation, which itself is based on Perl Compatible Regular Expressions (PCRE). This engine is exposed into Wireshark's Lua engine through the well-known Lrexlib library, following the same syntax and semantics as the Lrexlib PCRE implementation, with a few differences as follows:

- There is no support for using custom locale/chartables
- *dfa_exec()* doesn't take *ovecsize* nor *wscount* arguments
- *dfa_exec()* returns boolean true for partial match, without subcapture info
- Named subgroups do not return name-keyed entries in the return table (i.e., in match/tfind/exec)
- The *flags()* function still works, returning all flags, but two new functions *compile_flags()* and *match_flags()* return just their respective flags, since GLib has a different and smaller set of such flags, for regex compile vs. match functions
- Using some assertions and POSIX character classes against strings with non-ASCII characters might match high-order characters, because glib always sets PCRE_UCP even if G_REGEX_RAW is set. For example, *[:alpha;]* matches certain non-ASCII bytes. The following assertions have this issue: *\b*, *\B*, *\s*, *\S*, *\w*, *\W*. The following character classes have this issue: [:alpha:], [:alnum:], [:lower:], [:upper:], [:space:], [:word:], and [:graph:].
- The compile flag G_REGEX_RAW is always set/used, even if you didn't specify it. This is because GLib runs PCRE in UTF-8 mode by default, whereas Lua strings are not UTF-aware.

Since: 1.11.3

This page is based on the full documentation for Lrexlib at http://rrthomas.github.io/lrexlib/manual.html

The GLib Regular expression syntax (which is essentially PCRE syntax) can be found at https://developer.gnome.org/glib/2.38/glib-regex-syntax.html

GRegex

GLib Regular Expressions based on PCRE.

Since: 1.11.3

Notes

All functions that take a regular expression pattern as an argument will generate an error if that pattern is found invalid by the regex library.

All functions that take a string-type regex argument accept a compiled regex too. In this case, the compile flags argument is ignored (should be either supplied as nils or omitted).

The capture flag argument *cf* may also be supplied as a string, whose characters stand for compilation flags. Combinations of the following characters (case sensitive) are supported:

- *i* = G_REGEX_CASELESS - Letters in the pattern match both upper- and lowercase letters. This option can be changed within a pattern by a “(?i)” option setting.
- *m* = G_REGEX_MULTILINE - By default, GRegex treats the strings as consisting of a single line of characters (even if it actually contains newlines). The “start of line” metacharacter (“^”) matches only at the start of the string, while the “end of line” metacharacter (“$”) matches only at the end of the string, or before a terminating newline (unless G_REGEX_DOLLAR_ENDONLY is set). When G_REGEX_MULTILINE is set, the “start of line” and “end of line” constructs match immediately following or immediately before any newline in the string, respectively, as well as at the very start and end. This can be changed within a pattern by a “(?m)” option setting.
- *s* = G_REGEX_DOTALL - A dot metacharater (“.”) in the pattern matches all characters, including newlines. Without it, newlines are excluded. This option can be changed within a pattern by a ("?s") option setting.
- *x* = G_REGEX_EXTENDED - Whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped “#” outside a character class and the next newline character, inclusive, are also ignored. This can be changed within a pattern by a “(?x)” option setting.
- *U* = G_REGEX_UNGREEDY - Inverts the “greediness” of the quantifiers so that they are not greedy by default, but become greedy if followed by “?”. It can also be set by a “(?U)” option setting within the pattern.

GRegex.new(pattern)

Compiles regular expression pattern into a regular expression object whose internal representation is corresponding to the library used. The returned result then can be used by the methods, e.g. match, exec, etc. Regular expression objects are automatically garbage collected.

Since: 1.11.3

Arguments

pattern
  A Perl-compatible regular expression pattern string

Returns

The compiled regular expression (a userdata object)

Errors

- A malformed pattern generates a Lua error

GRegex.flags([table])

Returns a table containing the numeric values of the constants defined by the regex library, with the keys being the (string) names of the constants. If the table argument is supplied then it is used as the output table, otherwise a new table is created. The constants contained in the returned table can then be used in most functions and methods where compilation flags or execution flags can be specified. They can also be used for comparing with return codes of some functions and methods for determining the reason of failure.

Since: 1.11.3

Arguments

table (optional)
  A table for placing results into

Returns

A table filled with the results.

GRegex.compile_flags([table])

Returns a table containing the numeric values of the constants defined by the regex library for compile flags, with the keys being the (string) names of the constants. If the table argument is supplied then it is used as the output table, otherwise a new table is created.

Since: 1.11.3

Arguments

table (optional)
  A table for placing results into

Returns

A table filled with the results.

GRegex.match_flags([table])

Returns a table containing the numeric values of the constants defined by the regex library for match flags, with the keys being the (string) names of the constants. If the table argument is supplied then it is used as the output table, otherwise a new table is created.

Since: 1.11.3

Arguments

table (optional)
  A table for placing results into

Returns

A table filled with the results.

GRegex.match(subject, pattern, [init], [cf], [ef])

Searches for the first match of the regexp pattern in the string subject, starting from offset init, subject to flags cf and ef. The pattern is compiled each time this is called, unlike the class method *match* function.

Since: 1.11.3

Arguments

subject
: Subject string to search

pattern
: A Perl-compatible regular expression pattern string or GRegex object

init (optional)
: start offset in the subject (can be negative)

cf (optional)
: compilation flags (bitwise OR)

ef (optional)
: match execution flags (bitwise OR)

Returns

On success, returns all substring matches ("captures"), in the order they appear in the pattern. false is returned for sub-patterns that did not participate in the match. If the pattern specified no captures then the whole matched substring is returned. On failure, returns nil.

GRegex.find(subject, pattern, [init], [cf], [ef])

Searches for the first match of the regexp pattern in the string subject, starting from offset init, subject to flags ef. The pattern is compiled each time this is called, unlike the class method *find* function.

Since: 1.11.3

Arguments

subject
: Subject string to search

pattern
: A Perl-compatible regular expression pattern string or GRegex object

init (optional)
: start offset in the subject (can be negative)

cf (optional)
: compilation flags (bitwise OR)

ef (optional)
: match execution flags (bitwise OR)

Returns

On success, returns the start point of the match (a number), the end point of the match (a number), and all substring matches ("captures"), in the order they appear in the pattern. false is returned for sub-patterns that did not participate in the match. On failure, returns nil.

GRegex.gmatch(subject, pattern, [init], [cf], [ef])

Returns an iterator for repeated matching of the pattern patt in the string subj, subject to flags cf and ef. The function is intended for use in the generic for Lua construct. The pattern can be a string or a GRegex object previously compiled with GRegex.new().

Since: 1.11.3

Arguments

subject
: Subject string to search

pattern
: A Perl-compatible regular expression pattern string or GRegex object

init (optional)
: start offset in the subject (can be negative)

cf (optional)
: compilation flags (bitwise OR)

ef (optional)
: match execution flags (bitwise OR)

Returns

The iterator function is called by Lua. On every iteration (that is, on every match), it returns all captures in the order they appear in the pattern (or the entire match if the pattern specified no captures). The iteration will continue till the subject fails to match.

GRegex.gsub(subject, pattern, [repl], [max], [cf], [ef])

Searches for all matches of the pattern in the string subject and replaces them according to the parameters repl and max. The pattern can be a string or a GRegex object previously compiled with GRegex.new().

Since: 1.11.3

For details see: http://rrthomas.github.io/lrexlib/manual.html#gsub

Arguments

subject
: Subject string to search

pattern
: A Perl-compatible regular expression pattern string or GRegex object

repl (optional)
: Substitution source string, function, table, false or nil

max (optional)
: Maximum number of matches to search for, or control function, or nil

cf (optional)
: Compilation flags (bitwise OR)

ef (optional)
: Match execution flags (bitwise OR)

Returns

On success, returns the subject string with the substitutions made, the number of matches found, and the number of substitutions made.

GRegex.split(subject, sep, [cf], [ef])

Splits a subject string subj into parts (sections). The sep parameter is a regular expression pattern representing separators between the sections. The function is intended for use in the generic for Lua construct. The function returns an iterator for repeated matching of the pattern sep in the string subj, subject to flags cf and ef. The sep pattern can be a string or a GRegex object previously compiled with GRegex.new(). Unlike gmatch, there will always be at least one iteration pass, even if there are no matches in the subject.

Since: 1.11.3

Arguments

subject
: Subject string to search

sep
: A Perl-compatible regular expression pattern string or GRegex object

cf (optional)
: compilation flags (bitwise OR)

ef (optional)
: match execution flags (bitwise OR)

Returns

The iterator function is called by Lua. On every iteration, it returns a subject section (can be an empty string), followed by all captures in the order they appear in the sep pattern (or the entire match if the sep pattern specified no captures). If there is no match (this can occur only in the last iteration), then nothing is returned after the subject section. The iteration will continue till the end of the subject.

GRegex.version()

Returns a returns a string containing the version of the used library.

Since: 1.11.3

Returns

The version string

gregex:match(subject, [init], [ef])

Searches for the first match of the regexp pattern in the string subject, starting from offset init, subject to flags ef.

Since: 1.11.3

Arguments

subject
: Subject string to search

init (optional)
: start offset in the subject (can be negative)

ef (optional)
: match execution flags (bitwise OR)

Returns

On success, returns all substring matches ("captures"), in the order they appear in the pattern. false is returned for sub-patterns that did not participate in the match. If the pattern specified no captures then the whole matched substring is returned. nil is returned if the pattern did not match.

gregex:find(subject, [init], [ef])

Searches for the first match of the regexp pattern in the string subject, starting from offset init, subject to flags ef.

Since: 1.11.3

Arguments

subject
    Subject string to search

init (optional)
    start offset in the subject (can be negative)

ef (optional)
    match execution flags (bitwise OR)

Returns

On success, returns the start point of the match (a number), the end point of the match (a number), and all substring matches ("captures"), in the order they appear in the pattern. false is returned for sub-patterns that did not participate in the match. On failure, returns nil.

gregex:exec(subject, [init], [ef])

Searches for the first match of the compiled GRegex object in the string subject, starting from offset init, subject to the execution match flags ef.

Since: 1.11.3

Arguments

subject
    Subject string to search

init (optional)
    start offset in the subject (can be negative)

ef (optional)
    match execution flags (bitwise OR)

Returns

On success, returns the start point of the first match (a number), the end point of the first match (a number), and the offsets of substring matches (“captures” in Lua terminology) are returned as a third result, in a table. This table contains false in the positions where the corresponding sub-pattern did not participate in the match. On failure, returns nil. Example: If the whole match is at offsets 10,20 and substring matches are at offsets 12,14 and 16,19 then the function returns the following: 10, 20, { 12,14,16,19 }.

gregex:dfa_exec(subject, [init], [ef])

Matches a compiled regular expression GRegex object against a given subject string subj, using a DFA matching algorithm.

Since: 1.11.3

Arguments

subject
: Subject string to search

init (optional)
: start offset in the subject (can be negative)

ef (optional)
: match execution flags (bitwise OR)

Returns

On success, returns the start point of the matches found (a number), a table containing the end points of the matches found, the longer matches first, and the number of matches found as the third return value. On failure, returns nil. Example: If there are 3 matches found starting at offset 10 and ending at offsets 15, 20 and 25 then the function returns the following: 10, { 25,20,15 }, 3

gregex:__tostring()

Returns a string containing debug information about the GRegex object.

Since: 1.11.3

Returns

The debug string

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

There are other Wireshark frontends which are not developed nor maintained by the Wireshark development team:

- Packetyzer. Native Windows interface, written in Delphi and released under the GPL. Not actively maintained. https://sourceforge.net/projects/packetyzer/
- hethereal Web interface. Not actively maintained and not finished.

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 Cretor, 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 *wireshark-qt.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 two modules, *main_window.cpp* and *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 InterCapConvention), GLib (which uses underscore_convention), and the Wireshark API (which also uses underscore_convention). As a general rule Wireshark’s Qt code uses InterCapConvention for class names, interCapConvention for methods, and underscore_convention for variables, with a trailing_underscore_ for member variables.

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/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 link:http://www. parashift.com/c++-faq/mixing-c-and-cpp.html:[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 *summary_dialog.[ch]* 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 *images/languages/XX.svg*. Update *image/languages/languages.qrc* accordingly.
- Run `lupdate ui/qt -ts ui/qt/wireshark_XX.ts` to generate/update your translation file.
- 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 Gerrit 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 (*$XDG_CONFIG_HOME/wireshark/languages/* or *$HOME/.wireshark/languages/* on UNIX).

For more information about Qt Linguist see its manual.

You can also manage translations online with Transifex.

Each week translations are automatically synchronized with the source code through the following steps:

- pull ts from Transifex
- lupdate ts file
- push and commit on Gerrit
- push ts on Transifex

Colors

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.

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.

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: http://developer.android.com/design/index.html. Wireshark doesn’t have a mobile frontend (not yet, at least) but there is still useful information here.
- GNOME Human Interface Guidelines: http://library.gnome.org/devel/hig-book/stable/
- The KDE Usability/HIG project: http://techbase.kde.org/Projects/Usability/HIG
- macOS Human Interface Guidelines: https://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/AppleHIGuidelines/Intro/Intro.html
- Design apps for the Windows desktop: http://msdn.microsoft.com/en-us/library/Aa511258.aspx
- 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.

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

Before running any tests you should build the "test-programs" target. It is required for the "utittests" suite.

The main testing script is `test.py`. It will attempt to test as much as possible by default, including packet capture. This means that you will probably either have to supply a capture interface (`--capture-interface <interface>`) or disable capture tests (`--disable-capture`).

To run all tests from CMake do the following: * Pass `-DTEST_EXTRA_ARGS=--disable-capture` or `-DTEST_EXTRA_ARGS=--capture-interface=<interface>` as needed for your system. * Build the "test" target or run ctest, e.g. `ctest --force-new-ctest-process -j 4 --verbose`.

On Windows, "ctest" requires a build configuration parameter, e.g. `ctest -C RelWithDebInfo --force -new-ctest-process -j 4 --verbose`.

To run all tests directly, run `test.py -p /path/to/wireshark-build/run-directory <capture args>`.

To see a list of all options, run `test.py -h` or `test.py --help`.

To see a list of all tests, run `test.py -l`.

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.

Suites, Cases, and Tests

The `test.py` script uses Python's "unittest" module. Our tests are patterned after it, and individual 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.case_tshark_capture_clopts.test_tshark_invalid_capfilter”.

Listing And Running Tests

Tests can be run via the `test.py` Python script. To run all tests, either run `test.py` in the directory that contains the Wireshark executables (`wireshark`, `tshark`, etc.), or pass the the executable path via the `-p` flag:

```
$ python test.py -p /path/to/wireshark-build/run
```

You can list tests by passing one or more complete or partial names to `tshark.py`. The `-l` flag lists tests. By default all tests are shown.

```
# List all tests
$ python test.py -l
$ python test.py -l all
$ python test.py --list
$ python test.py --list all

# List only tests containing "dumpcap"
$ python test.py -l dumpcap

# List all suites
$ python test.py --list-suites

# List all suites and cases
$ python test.py --list-cases
```

If one of the listing flags is not present, tests are run. If no names or `all` is supplied, all tests are run. Otherwise tests that match are run.

```
# Run all tests
$ python test.py
$ python test.py all

# Only run tests containing "dumpcap"
$ python test.py -l dumpcap

# Run the "clopts" suite
$ python test.py suite_clopts
```

Adding Or Modifying Tests

Tests must be in a Python module whose name matches “suite_*.py”. The module must contain one or more subclasses of “SubprocessTestCase” or “unittest.TestCase”. “SubprocessTestCase” is recommended since it contains several convenience methods for running processes, checking output, and displaying error information. Each test case method whose name starts with “test_” constitutes an individual test.

Success or failure conditions can be signalled using the “unittest.assertXXX()” or “subprocesstest.assertXXX()” methods.

The “config” module contains common configuration information which has been derived from the current environment or specified on the command line.

The “subprocesstest” class contains the following methods for running processes. Stdout and stderr is written to “<test id>.log”:

startProcess
: Start a process without waiting for it to finish.

runProcess
: Start a process and wait for it to finish.

assertRun
: Start a process, wait for it to finish, and check its exit code.

All of the current tests run one or more of Wireshark’s suite of executables and either checks their return code or their output. A simple example is “suite_clopts.case_basic_clopts.test_existing_file”, which reads a capture file using TShark and checks its exit code.

```
import config
import subprocesstest

class case_basic_clopts(subprocesstest.SubprocessTestCase):
    def test_existing_file(self):
        cap_file = os.path.join(self.capture_dir, 'dhcp.pcap')
        self.assertRun((config.cmd_tshark, '-r', cap_file))
```

Program output can be checked using “subprocesstest.grepOutput” or “subprocesstest.countOutput”:

```
import config
import subprocesstest

class case_decrypt_80211(subprocesstest.SubprocessTestCase):
    def test_80211_wpa_psk(self):
        capture_file = os.path.join(config.capture_dir, 'wpa-Induction.pcap.gz')
        self.runProcess((config.cmd_tshark,
                '-o', 'wlan.enable_decryption: TRUE',
                '-Tfields',
                '-e', 'http.request.uri',
                '-r', capture_file,
                '-Y', 'http',
            ),
            env=config.test_env)
        self.assertTrue(self.grepOutput('favicon.ico'))
```

Tests can be run in parallel. This means that any files you create must be unique for each test. “subprocesstest.filename_from_id” can be used to generate a filename based on the current test name. It also ensures that the file will be automatically removed after the test has run.

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.