Wireshark Developer's Guide

36390 for Wireshark 1.4

Ulf Lamping,

Wireshark Developer's Guide: 36390 for Wireshark 1.4

by Ulf Lamping

Copyright © 2004-2010 Ulf Lamping

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU General Public License, Version 2 or any later version published by the Free Software Foundation.

All logos and trademarks in this document are property of their respective owner.

Preface

1. 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.

2. 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.

3. 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.

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!

4. About this document

This book was developed by Ulf Lamping.

It is written in DocBook/XML.

You will find some specially marked parts in this book:

This is a warning!

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

This is a note!

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

This is a tip!

Tips will be helpful for your everyday work developing Wireshark.

5. Where to get the latest copy of this document?

The latest copy of this documentation can always be found at: http://www.wireshark.org/docs/ in PDF (A4 and US letter), HTML (single and chunked) and CHM format.

6. 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.

Part I. Wireshark Build Environment

Part I. 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.

Part II. Wireshark Development

The second part describes how the Wireshark sources are structured and how to change the sources (e.g. adding a new dissector).

Chapter 1. Introduction

1.1. Introduction

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

1.2. 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.

1.3. Platforms Wireshark runs on

Wireshark currently runs on most UNIX platforms and various Windows platforms. It requires GTK +, 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 GTK+ GUI library) which are available for a lot of 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 at least the following platforms:

1.3.1. Unix

- Apple Mac OS X
- BeOS
- FreeBSD
- HP-UX
- IBM AIX
- NetBSD
- OpenBSD
- SCO UnixWare/OpenUnix
- SGI Irix
- Sun Solaris/Intel
- Sun Solaris/Sparc
- Tru64 UNIX (formerly Digital UNIX)

1.3.2. 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

1.3.3. Microsoft Windows

Thanks to the Win32 API, development on all Windows platforms will be done in a very similar way. All Windows platforms referred to as Win32, Win or Windows may be used with the same meaning. Older Windows versions are no longer supported by Wireshark. As Windows CE differs a lot compared to the other Windows platforms mentioned, Wireshark will not run on Windows CE and there are no plans to support it.

Also the 64 bit Windows version are now supported by Wireshark. Although not all libraries are made 64 bit ready yet, basic operations are all available.

- Windows Server 2000
- Windows XP
- Windows Server 2003
- Windows Vista
- Windows 7
- Windows Server 2008

1.4. 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 oher 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 http://anonsvn.wireshark.org/wireshark/trunk/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 Section 1.7.5, “Mailing Lists” about the different Wireshark mailing lists available.

1.4.1. Programming language(s) used

Almost any part of Wireshark is implemented in plain ANSI 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.

1.4.2. 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: http://www.wireshark.org/download/.

1.5. Releases and distributions

The officially released files can be found at: http://www.wireshark.org/download/. 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.

1.5.1. 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 get a setup.exe file which will install all the required things for him.
- Win32 U3 (.u3 file). Special distribution for U3 capable USB memory sticks.
- 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 Red Hat Package Manager (RPM) based system obtains an .rpm file from which the package manager checks the dependencies and installs the software.
- MAC OS X (.dmg file). The typical MAC OS X end user is used to get a .dmg file which will install all the required things for him. The other requirement is to have the X11.app installed.
- 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 Section 3.12, “Binary packaging”.

1.5.2. 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 SVN sources. For details about the different ways to get the Wireshark source code see Section 3.3, “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.

1.6. 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.

1.6.1. 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: http://buildbot.wireshark.org/trunk/ 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: http://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.

1.6.2. What does the Buildbot do?

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

- checkout from the source repository
- build
- create binary package(s) / installer
- create source package (and check completeness)
- 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 XP x86 (Win32, little endian, VS 9)
- Windows XP x86-64 (Win64, little endian, VS 9)
- Ubuntu x86-64 (Linux, little endian, gcc)
- Solaris SPARC (Solaris, big endian, gcc)
- Mac OS-X PPC (BSD, big endian, gcc)
- Mac OS-X x86 (BSD, little endian, gcc)

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

1.7. 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).

1.7.1. Website

You will find lot's of useful information on the Wireshark homepage at http://www.wireshark.org.

1.7.2. Wiki

The Wireshark Wiki at http://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.

1.7.3. 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 the question(s) 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: http://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.

1.7.4. Other sources

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

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

Read the README!

The README.developer is packed full with all kinds of details relevant to the developer of Wireshark source code. It advices you around common pitfalls, shows you basic layout of dissector code, shows details of the API's available to the dissector developer, etc.

- the Wireshark source code
- tool documentation of the various tools used (e.g. manpages of sed, gcc, ...)
- the different mailing lists: see Section 1.7.5, “Mailing Lists”
- ...

1.7.5. 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 Section 1.7.6, “Bug database (Bugzilla)”.

wireshark-commits — This list is for Wireshark developers. Every time a change to the SVN repository is checked in, a mail to this mailing list is generated. If you want to be notified about all the changes to the SVN repository, join this list. Details about the SVN repository can be found in Section 3.2, “The Wireshark Subversion repository”.

You can subscribe to each of these lists from the Wireshark web site: http://www.wireshark.org. Simply select the **mailing lists** link on the left hand side of the site. The lists are archived at the Wireshark web site as well.

Tip!

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

1.7.6. 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.

1.7.7. Reporting Problems

Note!

Before reporting any problems, please make sure you have installed the latest version of Wireshark. Reports on older maintainance 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. GTK+, 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.

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.

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.

1.7.8. 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 Section 1.7.7, “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

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

Note

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.

Note

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

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

1.7.9. Reporting Crashes on Windows platforms

The Windows distributions don't contain the symbol files (.pdb), because they are very large. For this reason it's not possible to create a meaningful backtrace file from it. You should report your crash just like other problems, using the mechanism from Section 1.7.7, “Reporting Problems”.

Chapter 2. Quick Setup

2.1. 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 will typically use the installation package from your distribution (by your favourite package manager: aptitude, yum, synaptics, ...).

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.

2.2. Win32: Step-by-Step Guide

A quick setup guide for Win32 with recommended configuration.

Warning!

Unless you know exactly what you are doing, you should strictly follow the recommendations!

2.2.1. Install Microsoft C compiler and Platform SDK

You need to install:

1. C compiler: Download and install "Microsoft Visual C++ 2008 Express Edition SP1." (This is a very large download.)

Install MSVC the usual way. Don't forget to install `vcvars32.bat` or call it manually before building Wireshark. `vcvars32.bat` will set some required environment (e.g. the PATH) settings.

You can use other Microsoft C compiler variants!

It's possible to compile Wireshark with a wide range of Microsoft C compiler variants. For details see Section 4.4, “Microsoft compiler toolchain (Win32 native)”!

Don't use Cygwin's gcc!

Using Cygwin's gcc is not recommended and will certainly not work (at least without a lot of advanced tweaking). For further details on this topic, see Section 4.3, “GNU compiler toolchain (UNIX or Win32 Cygwin)”.

XXX - mention the compiler and PSDK web installers - which significantly reduce download size - and find out the required components

Wireshark development depends on several environment variables, particularly PATH. You can use a batch script to fill these in, for example

```
@echo off
echo Adding things to the path...
set PATH=%PATH%;.
set PATH=%PATH%;c:\cygwin\bin

echo Setting up Visual Studio environment...
call "c:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat"

title Command Prompt (VC++ 2008)
```

Why is this recommended? While this is a huge download, the 2008 Express Edition is the only free (as in beer) version that includes the Visual Studio integrated debugger. Visual C++ 2008 is also used to create official Wireshark builds, so it will likely have fewer development-related problems.

2.2.2. Install Cygwin

Download the Cygwin installer and start it.

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 click on the "Skip" item in the "New" column so it shows a version number for:

- Archive/unzip
- Devel/bison
- Devel/flex
- Interpreters/perl
- Utils/patch
- Web/wget

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

Why this is recommended: Cygwin's bash version is required, as no native Win32 version is available. As additional packages can easily be added, the perl and alike packages are also used.

2.2.3. Install Python

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

Why this is recommended: Cygwin's Python package doesn't work on some machines, so the Win32 native package is recommended.

2.2.4. Install Subversion Client

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

Why this is recommended: updating a personal source tree is significantly easier to do with Subversion than downloading a zip file and merging new sources into a personal source tree "by hand".

2.2.4.1. Subversion

If you want to work with the Wireshark Subversion source repositories (which is highly recommended, see Section 3.3, “Obtain the Wireshark sources”), it's recommended to install Subversion. This makes the first time setup easy and enables the Wireshark build process to determine your current source code revision. You can download the setup from http://subversion.tigris.org/ and simply install it.

2.2.4.2. TortoiseSVN

If you want to work with the Wireshark Subversion source repositories (which is highly recommended, see Section 3.3, “Obtain the Wireshark sources”), it's recommended to use TortoiseSVN for your everyday work. You can download the setup from http://tortoisesvn.tigris.org/ and simply install it.

2.2.5. Install and Prepare Sources

Tip

It's a good idea to successfully compile and run Wireshark at least once before you start hacking the Wireshark sources for your own project!

1. Download sources : Download Wireshark sources into: `C:\wireshark` using TortoiseSVN
   a. right click on the C:\ drive in Windows Explorer
   b. in the upcoming context menu select "SVN checkout..." and then set:
   c. i. URL of repository: " `http://anonsvn.wireshark.org/wireshark/trunk/`"
      ii. Checkout directory: `C:\wireshark`
   d. TortoiseSVN might ask you to create this directory - say yes
   e. TortoiseSVN starts downloading the sources
   f. if the download fails you may be behind a restrictive firewall, see Section 3.3, “Obtain the Wireshark sources” for alternative download methods
2. Edit config.nmake: edit the settings in `C:\wireshark\config.nmake`, especially:
   a. `VERSION_EXTRA` : Give Wireshark your "private" version info, e.g.: -myprotocol123 - to distinguish it from an official release!
   b. `PROGRAM_FILES` : Where your programs reside, usually just keep the default: `C:\Program Files` [2]
   c. `MSVC_VARIANT` : Make sure the variant for your compiler is uncommented, and that all others are commented out. For example, if you're using Visual C++ 2008 Express Edition, find the line

      ```
      #MSVC_VARIANT=MSVC2008EE
      ```

      and remove the comment character (#) from the beginning of the line. Then, find the line

      ```
      MSVC_VARIANT=MSVC2008
      ```

      and comment it out, by prefixing a hash (#). [1]

[1]Compiler dependent: This step depends on the compiler you are using. For compilers other than Visual C++ 2008, see the table at Section 4.4, “Microsoft compiler toolchain (Win32 native)”.

[2]International Windows might use different values here, e.g. a German version uses `C:\Programme` - take this also in account where `C:\Program Files` appears elsewhere.

2.2.6. Prepare cmd.exe

Prepare `cmd.exe` - set environment and current dir.

1. start **cmd.exe**
2. call **C:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat** to set environment variables of Visual C++ 2008 Express Edition. [1,2]
3. **cd C:\wireshark** to jump into the source directory

[1]Compiler dependent: This step depends on the compiler variant used, for other variants than the recommended Visual C++ 2008 Express Edition see the table at Section 4.4, “Microsoft compiler toolchain (Win32 native)”!

[2]International Windows might use different values here, e.g. a German version uses `C:\Programme` - take this also in account where `C:\Program Files` appears elsewhere. Note: You need to repeat steps 1 - 4 each time you open a new cmd.exe!

2.2.7. Verify installed tools

After you've installed the Wireshark sources (see Section 3.3, “Obtain the Wireshark sources”), you can check the correct installation of all tools by using the *`verify_tools`* target of the `Makefile.nmake` from the source package.

Warning!

You will need the Wireshark sources and some tools (nmake, bash) installed, before this verification is able to work.

Enter at the command line (cmd.exe, not Cygwin's bash!):

```
> nmake -f Makefile.nmake verify_tools
```

This will check for the various tools needed to build Wireshark:

```
Checking for required applications:
        cl: /cygdrive/c/Programme/Microsoft Visual Studio 8/VC/BIN/cl
        link: /cygdrive/c/Programme/Microsoft Visual Studio 8/VC/BIN/link
        nmake: /cygdrive/c/Programme/Microsoft Visual Studio 8/VC/BIN/nmake
        bash: /usr/bin/bash
        bison: /usr/bin/bison
        flex: /usr/bin/flex
        env: /usr/bin/env
        grep: /usr/bin/grep
        /usr/bin/find: /usr/bin/find
        perl: /usr/bin/perl
        env: /usr/bin/env
        C:/python27/python.exe: /cygdrive/c/python27/python.exe
        sed: /usr/bin/sed
        unzip: /usr/bin/unzip
        wget: /usr/bin/wget
```

If you have problems with all the first three items (cl, link, nmake), check if you called `vcvars32/SetEnv` as mentioned in Section 2.2.6, “Prepare cmd.exe” (which will "fix" your `PATH` settings). However, the exact text will be slightly different depending on the MSVC version used.

Unfortunately, the link command is defined both in Cygwin and in MSVC each with completely different functionality; you'll need the MSVC link. If your link command looks something like: **/usr/bin/link**, the link command of Cygwin takes precedence over the MSVC one. To fix this, you can change your `PATH` environment setting or simply rename the `link.exe` in Cygwin. If you rename it, make sure to remember that a Cygwin update may provide a new version of it.

2.2.8. Install Libraries

1. If you've closed **cmd.exe** in the meantime, prepare **cmd.exe** again.
2. **nmake -f Makefile.nmake setup** downloads libraries using **wget** and installs them - this may take a while ...
3. If the download fails you may be behind a restrictive firewall, see the proxy comment in Section 4.15, “Win32: GNU wget (optional)”.

2.2.9. Distclean Sources

The released Wireshark sources contain files that are prepared for a UNIX build (e.g. `config.h`).

You must distclean your sources before building the first time!

1. If you've closed **cmd.exe** in the meantime, prepare **cmd.exe** again
2. **nmake -f Makefile.nmake distclean** to cleanup the Wireshark sources

2.2.10. Build Wireshark

Now it's time to build Wireshark ...

1. If you've closed **cmd.exe** in the meantime, prepare **cmd.exe** again
2. **nmake -f Makefile.nmake all** to build Wireshark
3. wait for Wireshark to compile - this may take a while!
4. run **C:\wireshark\wireshark-gtk2\wireshark.exe** and check if it starts
5. check Help/About if it shows your "private" program version, e.g.: Version 1.4.x-myprotocol123 - you might run a release version previously installed!

Tip: If compilation fails for suspicious reasons after you changed some source files try to "distclean" the sources and make "all" again

2.2.11. Debug Environment Setup (XXX)

XXX - debug needs to be written, e.g. an idea is the create a simple MSVC workspace/project(s) to ease Visual Studio debugging

2.2.12. Optional: Create User's and Developer's Guide

Detailed information to build these guides can be found in the file `docbook/README.txt` in the Wireshark sources.

2.2.13. Optional: Create a Wireshark Installer

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

If you want to build your own `wireshark-win32-1.4.x-myprotocol123.exe`, you'll need NSIS.

1. NSIS: Download and install NSIS

   You may check the `MAKENSIS` setting in the file `config.nmake` of the Wireshark sources.

2. `vcredist_x86.exe` : Download the C-Runtime redistributable for Visual C++ 2008 Express Edition SP1 (`vcredist_x86.exe`) and copy it into `C:\wireshark-win32-libs` [1]
3. If you've closed **cmd.exe** in the meantime, prepare **cmd.exe** again
4. **nmake -f Makefile.nmake packaging** build Wireshark installer
5. run **C:\wireshark\packaging\nsis\wireshark-win32-1.4.x-myprotocol123.exe** and test it - it's a good idea to test also on a different machine than the developer machine.

[1]Compiler dependent: This step depends on the compiler variant used; for other variants than the recommended Visual C++ 2008 Express Edition SP1 see the table at Section 4.4, “Microsoft compiler toolchain (Win32 native)”!

Chapter 3. Work with the Wireshark sources

3.1. Introduction

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

- get the source
- compile the source
- submit changes
- ...

However, this chapter will not explain the source file contents in detail, such as where to find a specific functionality. This is done in Section 7.1, “Source overview”.

3.2. The Wireshark Subversion repository

Subversion is used to keep track of the changes made to the Wireshark source code. The Wireshark source code is stored inside Wireshark project's Subversion repository located at a server at the wireshark.org domain.

To quote the Subversion book about "What is Subversion?":

“Subversion is a free/open-source version control system. That is, Subversion manages files and directories over time. A tree of files is placed into a central repository. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your data, or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of "time machine". ”

Tip: Subversion and SVN is the same!

Subversion is often abbreviated as SVN, as the command-line tools are abbreviated that way. You will find both terms with the same meaning in this book, in mailing list discussions and elsewhere.

Using Wireshark's Subversion repository you can:

- keep your private sources up to date with very little effort
- get a mail notification if someone changes the latest sources
- 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 a lot more things related to the history of the Wireshark source code development

Subversion is divided into a client and a server part. Thanks to Gerald Combs (the maintainer of the Subversion server), no user has to deal with the maintenance of the Subversion server. You will only need a Subversion client, which is available as both a command-line and a GUI tool for many different platforms.

For further reference about Subversion, have a look at the homepage of the Subversion project: http://subversion.tigris.org/. There is a good and free book about it available at: http://svnbook.red-bean.com/.

Please note that Wireshark's public (anonymous) Subversion repository is separate from the main repository. It may take several minutes for committed changes to appear in the public repository - so please be patient for a few minutes if you desperately need a code change that was committed to the repository very recently.

3.2.1. The web interface to the Subversion repository

If you need a quick look at the Wireshark source code, you will only need a Web browser.

A `simple view` on the latest developer version can be found at:

http://anonsvn.wireshark.org/wireshark/trunk/.

A `comprehensive view` of all source versions (e.g. including the capability to show differences between versions) is available at:

http://anonsvn.wireshark.org/viewvc/viewvc.cgi/.

Of special interest might be the subdirectories:

- `trunk` - the very latest source files
- `releases` - the source files of all released versions

3.3. Obtain the Wireshark sources

There are several ways to obtain the sources from Wireshark's Subversion server.

Anonymous Subversion access is recommended!

It can make your life much easier, compared to updating your source tree by using any of the zip file methods mentioned below. Subversion handles merging of 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.

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.

3.3.1. Anonymous Subversion access

Recommended for development purposes.

Age: a few minutes.

You can use a Subversion client to download the source code from Wireshark's anonymous Subversion repository. The URL for the repository trunk is: http://anonsvn.wireshark.org/wireshark/trunk/.

See Section 4.11, “Subversion (SVN) client (optional)” on how to install a Subversion client.

For example, to check out using the command-line Subversion client, you would type:

```
$ svn checkout http://anonsvn.wireshark.org/wireshark/trunk wireshark
```

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

3.3.2. Anonymous Subversion web interface

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

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

The entire source tree of the Subversion repository is available via a web interface at: http://anonsvn.wireshark.org/viewvc/viewvc.cgi/. 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.

3.3.3. Buildbot Snapshots

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

Age: some number of minutes (a bit older than the anonymous Subversion 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: http://www.wireshark.org/download/automated/src/.

If anonymous Subversion 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 Subversion access if possible (believe it, it will save you a lot of time).

3.3.4. Released sources

Recommended for productive purposes.

Age: from days to weeks.

The officially released source files can be found at: http://www.wireshark.org/download/. You should use these sources if you want to build Wireshark on your platform for productive use.

The differences between the released sources and the sources stored at the Subversion repository will keep on growing until the next release is done (at the release time, the released and latest Subversion repository versions are then identical again :-).

3.4. 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 Subversion repository.

Take a look at the buildbot first!

As development evolves, the Wireshark sources are compilable most of the time - but not always. You may take a look at the Section 1.6, “Automated Builds (Buildbot)” first, to see if the sources are currently in a good shape.

3.4.1. ... with Anonymous Subversion access

After the first time checkout is done, updating your sources is simply done by typing (in the Wireshark source dir):

```
$ svn update
```

This will only take a few seconds, even on a slow internet connection. It will replace old file versions by new ones. If you and someone else have changed the same file since the last update, Subversion will try to merge the changes into your private file (this works remarkably well).

3.4.2. ... from zip files

Independent of the way you retrieve the zip file of the Wireshark sources (as described in Section 3.3, “Obtain the Wireshark sources”), the way to bring the changes from the official sources into your personal source tree is identical.

First of all, you will download the new zip 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.

Well, if you did change something in 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 zip 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.

3.5. Build Wireshark

The sources contain several documentation files, it's a good idea to look at these files first.

So after obtaining the sources, tools and libraries, the first place to look at is `doc/README.developer`, here you will get the latest infos for Wireshark development for all supported platforms.

Tip!

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

The following steps for the first time generation differ on the two major platforms.

3.5.1. Unix

Run the autogen.sh script at the top-level wireshark directory to configure your build directory.

```
./autogen.sh
./configure
make
```

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

```
./configure --help
```

to see what options you have.

3.5.2. Win32 native

The first thing to do will be to check the file `config.nmake` to determine if it reflects your configuration. The settings in this file are well documented, so please have a look at that file. However, if you've installed the libraries and tools as recommended there should be no need to edit things here.

Many of the file and directory names used in the build process go past the old 8.3 naming limitations. As a result, you should use the **cmd.exe** command interpreter instead of the old **command.com**.

Be sure that your command-line environment is set up to compile and link with MSVC++. When installing MSVC++, you can have your system's environment set up to always allow compiling from the command line, or you can invoke the vcvars32.bat script, which can usually be found in the `VC98\Bin` subdirectory of the directory in which Visual Studio was installed.

You should then cleanup any intermediate files, which are shipped for convenience of Unix users, by typing at the command line prompt (cmd.exe):

> **`nmake -f Makefile.nmake distclean`**

After doing this, typing at the command line prompt (cmd.exe):

> **`nmake -f Makefile.nmake all`**

will start the whole Wireshark build process.

After the build process has successfully finished, you should find a `wireshark.exe` and some other files in the root directory.

3.6. Run generated Wireshark

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.

3.6.1. 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 inpact 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 commandline 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.

3.6.2. Win32 native

During the build all relevant program files are collected in a subdirectory **wireshark-gtk2**. You can run the program from there by launching the wireshark.exe executable.

3.7. Debug your generated Wireshark

3.7.1. Unix/Linux

When you want to investigate a problem with Wireshark you want to load the program into your debugger. But loading wireshark into debugger fails because of the libtool build environment. You'll have to wrap loading wireshark into a libtool command: **libtool --mode=execute gdb wireshark**

If you prefer a graphic debugger you can use the Data Display Debugger (ddd) instead of GNU debugger (gdb).

Additional traps can be set on GLib by setting the `G_DEBUG` environment variable:**G_DEBUG=fatal_criticals libtool --mode=execute ddd wireshark**. See http://library.gnome.org/devel/glib/stable/glib-running.html

3.7.2. Win32 native

XXX - add more info here.

3.8. 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 your own new dissector
- change/extend an existing dissector
- fix a bug
- implement a new glorious feature :-)

The internal structure of the Wireshark sources will be described in Part II, “Wireshark Development (incomplete)”.

Tip!

`Ask the developer mailing list before you really start a new development task.` If you have an idea what you want to add/change, it's a good idea to contact the developer mailing list (see Section 1.7.5, “Mailing Lists”) and explain your idea. Someone else might already be working on the same topic, so double effort can be reduced, or someone can give you some tips that should be thought about (like side effects that are sometimes very hard to see).

3.9. 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 commit changes to the SVN 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 make a diff file (a patch) and upload it to the bug tracker.

3.9.1. What is a diff file (a patch)?

A diff file is a plain text file containing the differences between a pair of files (or a multiple of such file pairs).

Tip!

A diff file is often also called a patch, as it can be used to patch an existing source file or tree with changes from somewhere else.

The Wireshark community is using patches to transfer source code changes between the authors.

A patch is both readable by humans and (as it is specially formatted) by some dedicated tools.

Here is a small example of a patch for `file.h` that makes the second argument in `cf_continue_tail()` volatile. It was created using **svn diff**, described below:

```
Index: file.h
===================================================================
--- file.h      (revision 21134)
+++ file.h      (revision 22401)
@@ -142,7 +142,7 @@
  * @param err the error code, if an error had occurred
  * @return one of cf_read_status_t
  */
-cf_read_status_t cf_continue_tail(capture_file *cf, int to_read, int *err);
+cf_read_status_t cf_continue_tail(capture_file *cf, volatile int to_read, int *err);

 /**
  * Finish reading from "end" of a capture file.
```

The plus sign at the start of a line indicates an added line, a minus sign indicates a deleted line compared to the original sources.

We prefer to use so called "unified" diff files in Wireshark development, three unchanged lines before and after the actual changed parts are included. This makes it much easier for a merge/patch tool to find the right place(s) to change in the existing sources.

3.9.2. Generate a patch

There are several ways to generate patches. The preferred way is to generate them from an updated Subversion tree, since it avoids unnecessary integration work.

3.9.2.1. Using the svn command-line client

```
svn diff [changed_files] > svn.diff
```

Use the command line svn client to generate a patch in the required format from the changes you've made to your working copy. If you leave out the name of the changed file the svn client searches for all changes in the working copy and usually produces a patch containing more than just the change you want to send. Therefore you should always check the produced patch file.

If you've added a new file, e.g. `packet-myprotocol.c`, you can use **svn add** to add it to your local tree before generating the patch. Similarly, you can use **svn rm** for files that should be removed.

3.9.2.2. Using the diff feature of the GUI Subversion clients

Most (if not all) of the GUI Subversion clients (RapidSVN, TortoiseSVN, ...) have a built-in "diff" feature.

If you use TortoiseSVN:

TortoiseSVN (to be precise Subversion) keeps track of the files you have changed in the directories it controls, and will generate for you a unified diff file compiling the differences. To do so - after updating your sources from the SVN repository if needed - just right-click on the highest level directory and choose "TortoiseSVN" -> "Create patch...". You will be asked for a name and then the diff file will be created. The names of the files in the patch will be relative to the directory you have right-clicked on, so it will need to be applied on that level too.

When you create the diff file, it will include any difference TortoiseSVN finds in files in and under the directory you have right-clicked on, and nothing else. This means that changes you might have made for your specific configuration - like modifying `config.nmake` so that it uses your lib directory - will also be included, and you will need to remove these lines from the diff file. It also means that only changes will be recorded, i.e. if you have created new files -say, a new `packet-xxx.c` for a new protocol dissector- it will not be included in the diff, you need to add it separately. And, of course, if you have been working separately in two different patches, the .diff file will include both topics, which is probably not a good idea.

3.9.2.3. Using the diff tool

A diff file is generated, by comparing two files or directories between your own working copy and the "official" source tree. So to be able to do a diff, you should have two source trees on your computer, one with your working copy (containing your changes), and one with the "official" source tree (hopefully the latest SVN files) from http://www.wireshark.org.

If you have only changed a single file, you could type something like this:

```
diff -r -u --strip-trailing-cr svn-file.c work-file.c > foo.diff
```

To get a diff file for your complete directory (including subdirectories), you could type something like this:

```
diff -N -r -u --strip-trailing-cr ./svn-dir ./working-dir > foo.diff
```

It's a good idea to do a **`make distclean`** before the actual diff call, as this will remove a lot of temporary files which might be otherwise included in the diff. After doing the diff, you should edit the `foo.diff` file and remove unnecessary things, like your private changes to the `config.nmake` file.

Table 3.1. Some useful diff options

Option	Purpose
-N	Add new files when used in conjunction with -r.
-r	Recursively compare any subdirectories found.
-u	Output unified context.

Option	Purpose
--strip-trailing-cr	Strip trailing carriage return on input. This is useful for Win32
-x PAT	Exclude files that match PAT. This could be something like -x *.obj to exclude all win32 object files.

The diff tool has a lot options; they can be listed with:

```
diff --help
```

3.9.3. Some tips for a good patch

Some tips that will make the merging of your changes into the SVN tree much more likely (and you want exactly that, don't you :-):

- **Use the latest SVN sources, or alike.** It's a good idea to work with the same sources that are used by the other developer's, this makes it usually much easier to apply your patch. For information about the different ways to get the sources, see Section 3.3, “Obtain the Wireshark sources”.
- **Update your SVN sources just before making a patch.** For the same reasons as the previous point.
- **Do a "make clean" before generating the patch.** This removes a lot of unneeded intermediate files (like object files) which can confuse the diff tool generating a lot of unneeded stuff which you have to remove by hand from the patch again.
- **Find a good descriptive filename for your patch.** Think a moment to find a proper name for your patch file. Often a filename like `wireshark.diff` is used, which isn't really helpful if keeping several of these files and find the right one later. For example: If you want to commit changes to the datatypes of dissector foo, a good filename might be: `packet-foo-datatypes.diff`.
- **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 :-).
- **Remove any parts of the patch not related to the changes you want to submit.** You can use a text editor for this. A common example for win32 developers are the differences in your private `config.nmake` file.

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.

Please remember: you don't pay the person "on the other side of the mail" for his/her effort applying your patch!

3.9.4. 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.

Warn!

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 Section 7.2, "Coding styleguides" for details.

- **Submit dissectors as built-in whenever possible.** Developing a new dissector as a plugin is a good idea because compiling is quicker, but it's best to convert dissectors to the built-in style before submitting for checkin. 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-n-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 sourcefile dissectors are more suitable to be placed separate as plugin.

- **Verify that your dissector code does not use prohibited or deprecated APIs** This can be done as follows:

  ```
  perl <wireshark_root>/tools/checkAPIs.pl <source-filename(s)>
  ```

- **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: http://wiki.wireshark.org/FuzzTesting for details.

3.9.5. Sending your patch for inclusion

After generating a patch of your changes, you might want to have your changes included into the SVN repository.

To submit a patch, open a new ticket in the Wireshark bug database at https://bugs.wireshark.org/bugzilla/enter_bug.cgi?product=Wireshark. You must first create a bug, then attach your patch or patches.

- Set the Product, Priority, and Severity as needed.

- Add a Summary and Description, and create a bug using the Commit button. If your code has passed fuzz testing, please say so in the description.

- Once the bug has been created, select Create a New Attachment and upload your patch or patches. Set the **review_for_checkin** flag to **?**. If you skip this step, your patch won't show up in the patch request queue.

- If possible and applicable, attach a capture file that demonstrates your new feature or protocol.

- Don't set the bug's status to ASSIGNED and don't assign the bug to yourself--if you do the latter, the core developers won't see the updates made to the bug.

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

- Your patch is checked into the SVN 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: Don't worry, if your patch is in the bug tracker, it won't get lost. But it may be that all the core developers are busy (e.g., with their day jobs

or family or...) and haven't had time to look at your patch. 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.

3.10. 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!

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

3.10.1. 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 *`-p`* at all, the patch tool flattens path names, so that if you have a patch file with patches to `Makefile.am` and `wiretap/Makefile.am`, it'll try to apply the first patch to the top-level `Makefile.am` and then apply the `wiretap/Makefile.am` patch to the top-level `Makefile.am` 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 "svn 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 `Makefile.am`.

3.11. Add a new file to the Subversion repository

The "usual" way to commit new files is described in Section 3.9, “Contribute your changes”. However, the following might be of interest for the "normal" developer as well.

Note!

This action is only possible/allowed by the Wireshark core developers who have write access to the Subversion repository. It is put in here to have all information in one place.

If you (as a core developer) need to add a file to the SVN repository, then you need to perform the following steps:

1. Add the Wireshark boilerplate to the new file(s).
2. Add a line to each new file containing the following text (case is important, so don't write ID or id or iD):

   ```
   $Id$
   ```

3. Add the new file(s) to the repository:

   ```
   $ svn add new_file
   ```

4. Set the line ending property to "native" for the new file(s):

   ```
   $ svn propset svn:eol-style native new_file
   ```

5. Set version keyword to "Id" for the new file(s):

   ```
   $ svn propset svn:keywords Id new_file
   ```

6. Commit your changes, including the added file(s).

   ```
   $ svn commit new_file other_files_you_modified
   ```

Don't forget a brief description of the reason for the commit so other developers don't need to read the diff in order to know what has changed.

3.12. 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.

3.12.1. 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:

```
$ make debian-package
```

to build the Debian Package.

3.12.2. Red Hat: .rpm packages

The RPM is built using rpmbuild (http://www.rpm.org/), which comes as standard on many flavours of Linux, including Red Hat and Fedora. The process creates a clean build environment in `packaging/rpm/BUILD` every time the RPM is built. The settings controlling the build are in `packaging/rpm/SPECS/wireshark.spec.in`. After editing the settings in this file, `./configure` must be run again in the wireshark directory to generate the actual specification script.

Warn!

The SPEC file contains settings for the `configure` used to set the RPM build environment. These are completely independent of any settings passed to the usual Wireshark `./configure`.

In the wireshark directory, type:

```
$ make rpm-package
```

to build the RPM. Once it is done, there will be a message stating where the built RPM can be found.

Tip!

Because this does a clean build, as well as constructing the package, this can take quite a long time.

3.12.3. MAC OS X: .dmg packages

The MAC OS X Package is built using OS X packaging tools, based on information found in the source tree under `packaging/macosx`.

In the wireshark directory, type:

```
$ make osx-package
```

to build the MAC OS X Package.

3.12.4. Win32: NSIS .exe installer

The "Nullsoft Install System" is a free installer generator for Win32 based systems; instructions how to install it can be found in Section 4.17, “Win32: NSIS (optional)”. NSIS is script based, you will find the Wireshark installer generation script at: `packaging/nsis/wireshark.nsi`.

You will probably have to modify the MAKENSIS setting in the `config.nmake` file to specify where the NSIS binaries are installed.

In the wireshark directory, type:

```
> nmake -f makefile.nmake packaging
```

to build the installer.

Tip!

Please be patient while the compression is done, it will take some time (a few minutes!) even on fast machines.

If everything went well, you will now find something like: `wireshark-setup-1.4.exe` in the `packaging/nsis` directory.

Chapter 4. Tool Reference

4.1. 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 like platforms, but Win32 ports are also available. Therefore the tools are available in different "flavours":

- UNIX (or Win32 Cygwin): the tools should be commonly available on the supported UNIX platforms, and for Win32 platforms by using the Cygwin UNIX emulation
- Win32 native: some tools are available as native Win32 tools, no special emulation is required

Warning!

Unless you know exactly what you are doing, you should strictly follow the recommendations given in Chapter 2, *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.

Don't expect a lot of documentation regarding these tools in this document. If you need further documentation of a specific tool, you should find lot's of useful information on the web, as these tools are commonly used. You can also try to get help for the UNIX based tools with **`toolname --help`** or read the manpage **`man toolname`**.

You will find explanations of the tool usage for some of the specific development tasks in Chapter 3, *Work with the Wireshark sources*.

4.2. Win32: Cygwin

Cygwin provides a lot of UNIX based tools on the Win32 platform. It uses a UNIX emulation layer which might be a bit slower compared to the native Win32 tools, but at an acceptable level. The installation and update is pretty easy and done through a single (web based) setup.exe.

The native Win32 tools will typically be a bit faster, but more complicated to install, as you would have to download the tools from different webpages, and install them in different ways, tweaking the PATH and alike.

Note!

As there's no Win32 native bash version available, at least a basic installation of cygwin is required in any case.

Although Cygwin consists of several separate packages, the installation and update is done through a single setup.exe, which acts similar to other web based installers. All tools will be installed into one base folder, the default is `C:\cygwin`.

You will find this network based setup.exe at: http://www.cygwin.com/. Click on one of the "Install Cygwin now" appearances to download the `setup.exe`. After the download completed, start this `setup.exe` on your machine.

The setup will ask you for some settings, the defaults should usually work well for a first start. 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 click on the "Skip" item in the "New" column so it shows a version number for the required package

After clicking the Next button several times, the setup will then download and install the selected packages (this may take a while, depending on the package size).

Under: "Start#Programs#Cygwin#Cygwin Bash Shell" you should now be able to start a new Cygwin bash shell, which is similar to the command line (command.com/cmd.exe) in Win32, but much more powerful.

4.2.1. Add/Update/Remove Cygwin Packages

If you want to add additional, update installed or remove packages later, you have to start the setup.exe again. At the "Select Packages" page, the entry in the "New" column will control what is done (or not) with the package. If a new version of a package is available, the new version number will be displayed, so it will be automatically updated. You can change the current setting by simply clicking at it, it will change between:

- a specific version number - this different package version will be installed
- Skip - not installed, no changes
- Keep - already installed, no changes
- Uninstall - uninstall this package
- Reinstall - reinstall this package

4.3. GNU compiler toolchain (UNIX or Win32 Cygwin)

4.3.1. gcc (GNU compiler collection)

Win32: Warn!

Using Cygwin gcc to compile Wireshark is "EXPERT ONLY" and therefore NOT recommended. If you really want to try it anyway, see: http://wiki.wireshark.org/Development/CygwinGCC for some details!

The GCC C compiler is available for most of the UNIX-like platforms and as the Devel/gcc package from the Cygwin setup.

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 (GCC) 3.4.4 (cygwin special) (gdc 0.12, using dmd 0.125)
Copyright (C) 2004 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.
```

However, the version string may vary.

4.3.2. gdb (GNU project debugger)

GDB is the debugger for the GCC compiler. It is available for many (if not all) UNIX-like platforms and as the Devel/gdb package from the Cygwin setup

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 6.5.50.20060706-cvs (cygwin-special)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-cygwin".
```

However, the version string may vary.

4.3.3. 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 and as the ddd package from the Cygwin setup.

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/.

4.3.4. make (GNU Make)

Win32 Note!

Although some effort is made to use make from the Cygwin environment, the mainline is still using Microsoft Visual Studio's nmake.

GNU Make is available for most of the UNIX-like platforms and also as the Devel/make package from the Cygwin setup.

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 3.81
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.
```

However, the version string may vary.

4.4. Microsoft compiler toolchain (Win32 native)

To compile Wireshark on Windows using the Microsoft C/C++ compiler, you'll need:

1. C compiler (`cl.exe`)
2. Linker (`link.exe`)
3. Make (`nmake.exe`)
4. C runtime headers and libraries (e.g. `stdio.h, msvcrt.lib`)
5. Windows platform headers and libraries (e.g. `windows.h, WSock32.lib`)
6. HTML help headers and libraries (`htmlhelp.h, htmlhelp.lib`)

4.4.1. Toolchain Package Alternatives

The official Wireshark 1.4.x and 1.2.x releases are compiled using Microsoft Visual C++ 2008 SP1. 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. "Express Edition" compilers such as Visual C++ 2008 Express Edition SP1 can be used but any PortableApps or U3 packages you create will require the installation of a separate Visual C++ Redistributable package. See "C-Runtime "Redistributable" Files" below for more details.

However, you might already have a different Microsoft C++ compiler installed. With the considerations listed below, it should be possible to use it as well:

Compiler Package	IDE / Debugger?	Publicly available?	Platform SDK required?	config.nmake: MSVC_VARIANT	set compiler PATH and alike settings with:	Remarks
Visual Studio 6.0	Yes	Commercial [1]	Free Download (342MB)	MSVC6	Microsoft Visual Studio \VC98\Bin \vcvars32.bat	-
Visual Studio .NET (2002)	Yes	Commercial [1]	No [2]	MSVC2002	Microsoft Visual Studio .NET \Vc7\bin \vcvars32.bat	-
Visual Studio .NET 2003	Yes	Commercial [1]		MSVC2003	Microsoft Visual Studio .NET 2003\Vc7\bin \vcvars32.bat	-

Visual Studio 2005	Yes	Commercial		MSVC2005	Microsoft Visual Studio 8\VC\bin\vcvars32.bat	-
Visual C++ 2005 Express Edition	Yes	Free Download (474MB)	Free Download (420MB)	MSVC2005EE	Microsoft Visual Studio 8\VC\bin\vcvars32.bat	vcredist_x86.exe [3]
Visual Studio 2008	Yes	Commercial	No [2]	MSVC2008	Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat	-
Visual C++ 2008 Express Edition SP1 (recommended)	Yes	Free Download	No [2]	MSVC2008EE	Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat	vcredist_x86.exe [3]
Visual Studio 2010	Yes	Commercial	No [2]	MSVC2010	Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat	-
Visual C++ 2010 Express Edition	Yes	Free Download	No [2]	MSVC2010EE	Microsoft Visual Studio 10.0\VC\vcvarsall.bat"" x86	vcredist_x86.exe [3]
.NET Framework SDK version 1.0a	No	Free Download (104MB)	Free Download (420MB)	DOTNET10	Microsoft.NET\FrameworkSDK\Bin\corvars.bat	Can't build setup [4]
.NET Framework SDK Version 1.1 [5]	No	Free Download (106MB)		DOTNET11	Microsoft.NET\SDK\v1.1\Bin\sdkvars.bat	Can't build setup [4]
.NET Framework 2.0 SDK [5]	No	Free Download (363MB)		DOTNET20	Microsoft.NET\SDK\v2.0\Bin\sdkvars.bat	vcredist_x86.exe [3]
Windows SDK for Windows Vista and .NET Framework 3.0 Runtime Components	No	Free Download (1188MB)	No [2]	- (not yet implemented!)	Microsoft SDKs\Windows\v6.0\Bin\SetEnv.Cmd	vcredist_x86.exe [3]

[1] no longer officially available, might still be available through the MSDN subscriptions

[2]as the Platform SDK is already integrated in the package, you obviously don't need to install it and don't even need to call a separate environment setting batch file for the Platform SDK!

[3]`vcredist_x86.exe` (3MB free download) is required to build `Wireshark-win32-1.4.x.exe`. The version of `vcredist_x86.exe` MUST match the version for your compiler.

[4] `Wireshark-win32-1.4.x.exe` cannot be created with this package, as `msvcr*.dll` is not available or not redistributable!

[5]MSDN remarks that the corresponding .NET runtime is required. It's currently unclear if the runtime needs to be installed for the C compiler to work - or is this only needed to compile / run .NET programs?!?

Note!

The "Visual C++ Toolkit 2003" should NOT be used to compile Wireshark!

4.4.2. Legal issues with MSVC > V6?

Please note: The following is not legal advice - ask your preferred lawyer instead! It's the authors view, but this view might be wrong!

The myriad of Win32 support lib port projects all seem to believe there are legal issues involved in using newer versions of Visual Studio. This FUD essentially stems from two misconceptions:

1. Unfortunately, it is believed by many that the Microsoft Visual Studio 2003 EULA explicitly forbids linking with GPL'ed programs. This belief is probably due to an improper interpretation of the Visual Studio 2003 Toolkit EULA, which places redistribution restrictions only on SOURCE CODE SAMPLES which accompany the toolkit.

2. Other maintainers believe that the GPL itself forbids using Visual Studio 2003, since one of the required support libraries (MSVCR71.DLL) does not ship with the Windows operating system. This is also a wrongful interpretation, and the GPL FAQ explicitly addresses this issue.

Similar applies to Visual Studio 2005 and alike.

So in effect it should be perfectly legal to compile Wireshark and distribute / run it if it was compiled with any MSVC version > V6!

4.4.3. 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 6.0	6.0	1200	msvcrt.dll (Version 6)
Visual Studio .NET (2002)	7.0	1300	msvcr70.dll
.NET Framework SDK version 1.0a			
Visual Studio .NET 2003	7.10	1310	msvcr71.dll
.NET Framework SDK Version 1.1			
Visual Studio 2005	8.0	1400	msvcr80.dll
Visual C++ 2005 Express Edition			
.NET Framework 2.0 SDK			

Windows SDK for Windows Vista and .NET Framework 3.0 Runtime Components			
Visual Studio 2008	9.0	1500	msvcr90.dll
Visual Studio 2008 Express Edition			

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

> **cl**

should result in something like:

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

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

However, the version string may vary.

4.4.4. nmake.exe (Make)

nmake is part of the toolchain packages described above.

Instead of using the workspace (.dsw) and projects (.dsp) files, the traditional nmake makefiles are used. This has one main reason: it makes it much easier to maintain changes simultaneously with the GCC toolchain makefile.am files as both file formats are similar. However, as no Visual Studio workspace/project files are available, this makes it hard to use the Visual Studio IDE e.g. for using the integrated debugging feature.

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

> **nmake**

should result in something like:

```
Microsoft (R) Program Maintenance Utility   Version 6.00.9782.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

NMAKE : fatal error U1064: MAKEFILE not found and no target specified
Stop.
```

However, the version string may vary.

Documentation on nmake can be found at Microsoft MSDN

4.4.5. link.exe (Linker)

XXX - add info here

4.4.6. C-Runtime "Redistributable" Files

Please note: The following is not legal advice - ask your preferred lawyer instead! It's the authors view, but 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.

MSVC6 was using msvcrt.dll, which is already available on all recent windows systems - no need to redistribute anything. 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!

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 links are recommended for the interested reader:

- "Redistributing Visual C++ Files"
- "How to: Deploy using XCopy"
- "Redistribution of the shared C runtime component in Visual C++ 2005 and in Visual C++ .NET"

4.4.6.1. msvcrt.dll - Version 6.0

Redistributables weren't an issue with MSVC 6, as any realistic installer target system (>= Win95) already contains the corresponding msvcrt.dll.

4.4.6.2. msvcr70.dll - Version 7.0 (2002)

"Visual Studio .NET (2002)" - comes with this dll and it's mentioned in redist.txt.

".NET Framework SDK 1.0" doesn't even come with this dll. XXX - Is this file available with the .NET 1.0 runtime (dotnetfx.exe) - so it could be shipped instead?!? Do we want it that way?

4.4.6.3. msvcr71.dll - Version 7.1 (2003)

"Visual Studio .NET 2003" comes with this dll and it's mentioned in redist.txt.

".NET Framework SDK 1.1" comes with this dll, but it's NOT mentioned in redist.txt. XXX - Is this file available with the .NET 1.1 runtime (dotnetfx.exe) - so it could be shipped instead ?!? Do we want it that way?

4.4.6.4. msvcr80.dll / vcredist_x86.exe - Version 8.0 (2005)

There are three redistribution methods that MSDN mentions for MSVC 8 (see: " Choosing a Deployment Method"):

1. "Redistributable Merge Modules" (kind of loadable modules for building msi installers - not suitable for Wireshark's NSIS based installer)
2. copy the folder content of Microsoft.VC80.CRT to the target directory (e.g. "C:\program files \Wireshark")
3. vcredist_x86.exe (needs to be executed on the target machine - MSDN recommends this for the 2005 Express Editions)

To save installer size, MSVC2005 uses the content of Microsoft.VC80.CRT (method 2 - this is the smallest package). As MSVC2005EE and DOTNET20 doesn't provide the folder "Microsoft.VC80.CRT" they use method 3. You'll have to download a vcredist_x86.exe from

Microsoft that matches your compiler version. The best way to determine this version is to open one of the generated manifest files (e.g. wireshark.exe.manifest) and look for the version of the Microsoft.VC80.CRT entry.

- **8.0.50608.0**, from: "Microsoft Visual C++ 2005" (and probably the Express Edition as well): http://www.microsoft.com/downloads/details.aspx?FamilyId=32BC1BEE-A3F9-4C13-9C99-220B62A191EE
- **8.0.50727.762**, from: "Microsoft Visual C++ 2005 Express Edition - ENU Service Pack 1 (KB926748)": http://www.microsoft.com/downloads/details.aspx?familyid=200B2FD9-AE1A-4A14-984D-389C36F85647

Please report to the developer mailing list, if you find a different version number!

4.4.6.5. msvcr90.dll / vcredist_x86.exe - Version 9.0 (2008)

- **9.0.21022.8**, from: "Microsoft Visual C++ 2008 Redistributable Package (x86)": http://www.microsoft.com/downloads/details.aspx?FamilyID=9B2DA534-3E03-4391-8A4D-074B9F2BC1BF
- **9.0.30729.17**, from: "Microsoft Visual C++ 2008 SP1 Redistributable Package (x86)": http://www.microsoft.com/downloads/details.aspx?FamilyID=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2

Please report to the developer mailing list, if you find a different version number!

4.4.6.6. Version 10.0 (2010)?

As the corresponding C compiler is preliminary, it's too early to say!

4.4.7. Windows (Platform) SDK

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

When you purchase a commercial Visual Studio, it will include a PSDK. The free (as in beer) downloadable C compiler versions (VC++ 2005 Express, .NET Framework, ...) do not contain a PSDK - you'll need to download a PSDK in order to have the required C header files and libraries.

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

BTW: "Windows SDK" seems to be the new name of the Platform SDK for Vista. The current SDK name is misleading: "Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components" - translated this means: the Windows SDK for Windows Vista and Platforms (like WinXP) that have the .NET 3.0 runtime installed.

4.4.8. HTML Help

The HTML Help is used to create the User's and Developer's Guide in .chm format and to show the User's Guide as the Wireshark "Online Help".

Both features are currently optional, but might be mandatory in future versions.

4.4.8.1. HTML Help Compiler (hhc.exe)

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.

4.4.8.2. HTML Help Build Files (htmlhelp.c / htmlhelp.lib)

The files htmlhelp.c and htmlhelp.lib are required to be able to open .chm files from Wireshark - to show the "online help".

Both files are part of the Platform SDK (standalone PSDK or MSVC since 2002). If you still use MSVC 6, you can get them from the "HTML Help Workshop" mentioned above.

The related settings in config.nmake depend on the MSVC variant you use:

- MSVC 6: if the "HTML Help Workshop" is installed, set HHC_DIR to its directory
- > MSVC 6: set HHC_DIR to use it (the actual value doesn't matter in this case)

4.4.9. Debugger

Well, 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.

4.4.9.1. Visual Studio integrated debugger

You can use the integrated debugger of Visual Studio - only available in some of the toolchain packages.

However, setting up the environment is a bit tricky, as the Win32 build process is using makefiles instead of the .dsp/.dsw files usually used.

XXX - add instructions how to do it.

4.4.9.2. Debugging Tools for Windows

You could 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 at: http://www.microsoft.com/whdc/devtools/debugging/default.mspx (as links to Microsoft pages change from time to time, search for "Debugging Tools" at their page if this link should be outdated).

4.5. bash

The bash shell is needed to run several shell scripts.

4.5.1. UNIX or Win32 Cygwin: GNU bash

The bash shell is available for most of the UNIX-like platforms and as the bash package from the Cygwin setup.

If bash 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 3.1.17(6)-release (i686-pc-cygwin)
Copyright (C) 2005 Free Software Foundation, Inc.
```

However, the version string may vary.

4.5.2. Win32 native: -

The authors don't know of any working Win32 native bash implementation.

4.6. python

Python is an interpreter based programming language. The homepage of the python project is: http://python.org/. Python is used to generate some source files. Python 2.4 to 2.7 should work fine.

4.6.1. UNIX or Win32 Cygwin: python

Python is available for most of the UNIX-like platforms and as the python package from the Cygwin setup

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

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

```
$ python -V
```

should result in something like:

```
Python 2.4.3
```

However, the version string may vary.

4.6.2. Win32 native: python

Get Python 2.7, 2.6, 2.5, or 2.4 from http://python.org/download/. You can download an installation package there, which will install the Python system in the top level of your `C:` drive by default, e.g. `C:\Python27`.

You can check for a successful installation from a command prompt (cmd.exe):

```
C:\> cd python27
```

```
C:\Python27> python -V
```

The output should look something like:

```
Python 2.7
```

However, the version string may vary.

4.7. perl

Perl is an interpreter based 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 be working fine.

4.7.1. UNIX or Win32 Cygwin: perl

Perl is available for most of the UNIX-like platforms and as the perl package from the Cygwin setup.

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, v5.8.7 built for cygwin-thread-multi-64int
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2005, 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.com/, the Perl Home Page.
```

However, the version string may vary.

4.7.2. Win32 native: perl

A native Win32 perl package can be obtained from http://www.ActiveState.com. The installation should be straightforward.

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.

4.8. sed

Sed it the streaming editor. It makes it easy for example to replace specially marked texts inside a source code file. The Wireshark build process uses this to stamp version strings into various places.

4.8.1. UNIX or Win32 Cygwin: sed

Sed is available for most of the UNIX-like platforms and as the sed package from the Cygwin setup.

If sed isn't already installed or available as a package for your platform, you can get it at: http://directory.fsf.org/GNU/sed.html

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

```
$ sed --version
```

should result in something like:

```
GNU sed version 4.1.5
Copyright (C) 2003 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,
to the extent permitted by law.
```

However, the version string may vary.

4.8.2. Win32 native: sed

A native Win32 sed package can be obtained from http://gnuwin32.sourceforge.net/. The installation should be straightforward.

4.9. yacc (bison)

Bison is a free implementation of yacc.

4.9.1. UNIX or Win32 Cygwin: bison

Bison is available for most of the UNIX-like platforms and as the bison package from the Cygwin setup.

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, typing at the bash command line prompt:

```
$ 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.
```

However, the version string may vary.

4.9.2. Win32 native: bison

A native Win32 yacc/bison package can be obtained from http://gnuwin32.sourceforge.net/. The installation should be straightforward.

4.10. flex

Flex is a free implementation of lexx.

4.10.1. UNIX or Win32 Cygwin: flex

Flex is available for most of the UNIX-like platforms and as the flex package from the Cygwin setup.

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, typing at the bash command line prompt:

```
$ flex --version
```

should result in something like:

```
flex version 2.5.4
```

However, the version string may vary.

4.10.2. Win32 native: flex

A native Win32 lexx/flex package can be obtained from http://gnuwin32.sourceforge.net/. The installation should be straightforward.

4.11. Subversion (SVN) client (optional)

The Wireshark project uses its own Subversion (or short SVN) server to keep track of all the changes done to the source code. Details about the usage of Subversion in the Wireshark project can be found in Section 3.2, “The Wireshark Subversion 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 SVN client to get the latest source files. For detailed information about the different ways to obtain the Wireshark sources, see Section 3.3, “Obtain the Wireshark sources”.

You will find more instructions in Section 3.3.1, “Anonymous Subversion access” on how to use the Subversion client.

4.11.1. UNIX or Win32 Cygwin: svn

SVN is available for most of the UNIX-like platforms and as the Subversion package from the Cygwin setup

If Subversion isn't already installed or available as a package for your platform, you can get it at: http://subversion.tigris.org/ (together with the server software).

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

```
$ svn --version
```

should result in something like:

```
svn, version 1.0.5 (r9954)
   compiled Jun 20 2004, 23:28:30
```

```
Copyright (C) 2000-2004 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/).
...
```

However, the version string may vary.

4.11.2. Win32 native: svn

The Subversion command line tools for Win32 can be found at: http://subversion.tigris.org/. This will come with both client and server software - only the client software will be used.

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

```
> svn --version
```

should result in something like:

```
svn, Version 1.4.0 (r21228)

Copyright (C) 2000-2006 CollabNet.
...
```

However, the version string may vary.

4.12. Subversion (SVN) GUI client (optional)

Along with the traditional command-line client, several GUI clients are available for a number of platforms, see http://subversion.tigris.org/project_links.html.

Keep Subversion program versions in sync!

If you are working with both command line and GUI clients, keep the Subversion program versions in sync, at least the major/minor versions (e.g. 1.4).

4.12.1. UNIX or Win32 Cygwin: rapidSVN, subcommander

RapidSVN is a cross platform Subversion frontend based on wxWidgets. It can be found at: http://rapidsvn.tigris.org/. Subcommander is another cross platform Subversion frontend. It can be found at: http://subcommander.tigris.org/.

Cygwin doesn't provide any GUI client for Subversion.

4.12.2. Win32 native: TortoiseSVN

A good Subversion client for Win32 can be found at: http://tortoisesvn.tigris.org/. It will nicely integrate into the Windows Explorer window.

4.13. diff (optional)

Diff is used to get a file of all differences between two source files/trees (sometimes called a patch). The diff tool isn't needed for building Wireshark, but it's needed if you are going to commit your changes back to the Wireshark community.

Note!

The recommended way to build patches is using the Subversion client, see Section 4.11, "Subversion (SVN) client (optional)" for details.

You will find more instructions in Section 3.9.2.3, "Using the diff tool" on how to use the diff tool.

4.13.1. UNIX or Win32 Cygwin: GNU diff

Diff is available for most of the UNIX-like platforms and as the diffutils package from the Cygwin setup.

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

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

```
$ diff --version
```

should result in something like:

```
diff (GNU diffutils) 2.8.7
Written by Paul Eggert, Mike Haertel, David Hayes,
Richard Stallman, and Len Tower.

Copyright (C) 2004 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.
```

However, the version string may vary.

4.13.2. Win32 native: diff

A native Win32 diff package can be obtained from http://gnuwin32.sourceforge.net/. The installation should be straightforward.

The Subversion client TortoiseSVN has a built-in diff feature, see Section 4.12.2, "Win32 native: TortoiseSVN". It is currently unknown if this tool can be used to create diff files in the required format, so other persons can use them.

4.14. 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.

Tip!

Unless you are in the rare case needing to apply a patch to your private source tree, you won't need the patch tool installed.

You will find more instructions in Section 3.10, "Apply a patch from someone else" on how to use the patch tool.

4.14.1. UNIX or Win32 Cygwin: patch

Patch is available for most of the UNIX-like platforms and as the patch package from the Cygwin setup.

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.

4.14.2. Win32 native: patch

A native Win32 patch package can be obtained from http://gnuwin32.sourceforge.net/. The installation should be straightforward.

The Subversion client TortoiseSVN has a built-in patch feature, see Section 4.12.2, "Win32 native: TortoiseSVN". The last time tested (Version 1.1.0), this feature failed to apply patches known to be ok.

4.15. Win32: GNU wget (optional)

GNU wget is used to download files from the internet using the command line.

GNU wget is available for most of the UNIX-like platforms and as the wget package from the Cygwin setup.

You will only need wget, if you want to use the Win32 automated library download, see Section 5.3, "Win32: Automated library download" for details.

If GNU wget isn't already installed or available as a package for your platform (well, for Win32 it is available as a Cygwin package), you can get it at: http://www.gnu.org/software/wget/wget.html.

If wget is trying to download files but fails to do so, your Internet connection might use an HTTP proxy. Some Internet providers use such a proxy and it is common in many company networks today. Wireshark's setup script will try to discover your proxy settings automatically, but you may need to set the environment variable HTTP_PROXY by hand before using wget. For example, if you are behind proxy.com which is listening on port 8080, you have to set it to something like:

```
set HTTP_PROXY=http://proxy.com:8080/
```

If you are unsure about the settings, you might ask your system administrator.

4.16. Win32: GNU unzip (optional)

GNU unzip is used to, well, unzip the zip files downloaded using the wget tool.

GNU unzip is available for most of the UNIX-like platforms and as the unzip package from the Cygwin setup.

You will only need unzip, if you want to use the Win32 automated library download, see Section 5.3, "Win32: Automated library download" for details.

If GNU unzip isn't already installed or available as a package for your platform (well, for Win32 it is available as a Cygwin package), you can get it at: http://gnuwin32.sourceforge.net/packages/unzip.htm.

4.17. Win32: NSIS (optional)

The NSIS (Nullsoft Scriptable Install System) is used to generate `wireshark-win32-1.4.x.exe` from all the files needed to be installed, including all required DLL's and such.

To install it, simply download the latest released version (currently: 2.45) from http://nsis.sourceforge.net and start the downloaded installer. You will need NSIS version 2 final or higher.

You will find more instructions in Section 3.12.4, "Win32: NSIS .exe installer" on how to use the NSIS tool.

Chapter 5. Library Reference

5.1. Introduction

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

1. Runtime package: binaries (e.g. win32 DLL's) and alike
2. Developer package: documentation, header files and alike
3. Source package: library sources, usually not required to build Wireshark

Tip!

Win32: All libraries for the VS9 generation are available at: http://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/, but see Section 5.3, "Win32: Automated library download" for an easier way to install the libraries.

Tip!

Win64: All required libraries for the VS9 generation are available at: http://anonsvn.wireshark.org/wireshark-win64-libs/trunk/packages/, but see Section 5.3, "Win32: Automated library download" for an easier way to install the libraries. Not all libraries are available, yet.

5.2. 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.

5.2.1. 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.

5.2.2. 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.

5.2.3. Win32: cygwin gcc

Cygwin provides most of the required libraries (with file extension .a or .lib) for Wireshark suitable for cygwin's gcc compiler.

5.3. Win32: Automated library download

5.3.1. Initial download

You can download/install all required libraries by using the setup target of the `Makefile.nmake` from the source package.

Tip!

It's a really good idea to use the Win32 automated library download to install the required libraries as it makes this download very easy.

Note!

Before you start the download, you must have installed both the required tools (see Chapter 4, *Tool Reference*) and also the Wireshark sources (see Section 3.3, "Obtain the Wireshark sources").

By default the libraries will be downloaded and installed into `C:\wireshark-win32-libs`. You can change this to any other location by editing the file `config.nmake` and changing the line containing the WIRESHARK_LIBS setting to your favourite place (use an absolute path here).

Then enter at the command line:

```
> nmake -f Makefile.nmake setup
```

This will first check for all the various tools needed to build Wireshark, as described already in Section 2.2.7, "Verify installed tools".

Then it will download the zipped libraries (together around 30MB!) from the server location at: http://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/ into the directory specified by WIRESHARK_LIBS and install (unzip) all required library files there.

If you have problems downloading the library files, you might be connected to the internet through a proxy/firewall. In this case see the wget proxy comment in Section 4.15, "Win32: GNU wget (optional)".

5.3.2. Update of a previous download

As new versions of the libraries become available, maybe with bugfixes or some new functionality, your libraries get outdated.

You could simply remove everything in the WIRESHARK_LIBS dir and call the **`setup`** target again, but that would require a download of every file again, which isn't necessary.

The following will bring your libraries up to date:

- Update your Wireshark sources to the latest SVN files (see Section 3.3, "Obtain the Wireshark sources"), so the zip filenames in the setup target of Makefile.nmake are in sync with the library zip files on the server.

- Execute the library setup command as described above.

  ```
  > nmake -f Makefile.nmake setup
  ```

  Note that this command will automatically do a **`clean-setup`** which will remove all files previously unzipped from the downloaded files in your WIRESHARK_LIBS library path (all the subdirs, e.g. `c:\wireshark-win32-libs\gtk+`), except for the zip files located at the toplevel, which are the files downloaded the last time(s).

  Also note that as wget will download only the missing (updated) files, existing zip files in the WIRESHARK_LIBS dir won't be downloaded again. Remaining (outdated) zip files shouldn't do any harm.

5.4. GTK+ / GLib / GDK / Pango / ATK / GNU gettext / GNU libiconv

The Glib library is used as a basic platform abstraction library, it's not related to graphical user interface (GUI) things. For a detailed description about GLib, see Section 7.3, “The GLib library”.

The GTK and its dependent libraries are used to build Wireshark's GUI. For a detailed description of the GTK libraries, see Section 10.2, “The GTK library”.

All other libraries are dependent on the two libraries mentioned above, you will typically not come in touch with these while doing Wireshark development.

As the requirements for the GLib/GTK libraries have increased in the past, the required additional libraries depend on the GLib/GTK versions you have. The 2.x versions require all mentioned libs.

5.4.1. Unix

The GLib/GTK+ libraries are available for many unix-like platforms and cygwin.

If these libraries aren't already installed and also aren't available as a package for your platform, you can get them at: http://www.gtk.org/download.html.

5.4.2. Win32 MSVC

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

5.5. SMI (optional)

"Various tools relating to the SMI MIB Information"

5.5.1. 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/.

5.5.2. Win32 MSVC

Wireshark uses the source libSMI distribution at http://www.ibr.cs.tu-bs.de/projects/libsmi/. libSMI is compiled using MSVC++ 6.0. It's stored in the libsmi zip archive at http://anonsvn.wireshark.org/wireshark-win32-libs/trunk/packages/

5.6. c-ares (optional)

"Library for asynchronous name resolves."

This is the primary name resolving library for Wireshark. It replaces ADNS.

5.6.1. 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/.

5.6.2. Win32 MSVC

You can get the latest version at: http://c-ares.haxx.se/.

5.7. GNU adns (optional)

"Advanced, easy to use, asynchronous-capable DNS client library and utilities."

5.7.1. Unix

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

5.7.2. Win32 MSVC

You can get the latest version at: http://adns.jgaa.com/

5.8. PCRE (optional)

"Perl compatible regular expressions". PCRE provides the pattern matching functionality required for the "matches" display filter operator.

You probably don't need this. GRegex, part of GLib 2.14 and later is a wrapper around PCRE. Wireshark's configure script will use it instead of PCRE if it is present.

5.8.1. Unix

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

5.8.2. Win32 MSVC

You can get the latest version at: http://gnuwin32.sourceforge.net/packages/pcre.htm

5.9. 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."

5.9.1. Unix

If this library isn't already installed or available as a package for your platform, you can get it at: http://www.gzip.org/zlib/.

5.9.2. Win32 MSVC

You can get the latest version at: http://gnuwin32.sourceforge.net/packages/zlib.htm

(A version for the MSVC2003 compiler can be found at: http://www.winimage.com/zLibDll/)

5.10. libpcap/WinPcap (optional)

"packet capture library"

5.10.1. 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/.

5.10.2. Win32 MSVC: WinPcap

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

5.11. GnuTLS (optional)

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

5.11.1. Unix

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

5.11.2. Win32 MSVC

We roll our own version using: http://josefsson.org/gnutls4win/

5.12. Gcrypt (optional)

The "Gcrypt Library" is Low-level encryption library and provides support for many ciphers, such as DES, 3DES, AES, Blowfish, and others..

5.12.1. Unix

If this library isn't already installed or available as a package for your platform, you can get it at: http://directory.fsf.org/security/libgcrypt.html.

5.12.2. Win32 MSVC

Part of our homemade GnuTLS package.

5.13. Kerberos (optional)

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

5.13.1. 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/.

XXX - Is it supported on *NIX at all?

5.13.2. Win32 MSVC

You can get the latest version of KfW "Kerberos for Windows" at: http://web.mit.edu/Kerberos/dist/

5.14. LUA (optional)

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

5.14.1. 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.

5.14.2. Win32 MSVC

You can get the latest version at: http://luaforge.net/frs/?group_id=110

5.15. PortAudio (optional)

The PortAudio library enables audio output for RTP streams.

5.15.1. Unix

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

5.15.2. Win32 MSVC

You can get the latest version at: http://www.portaudio.com/download.html

5.16. GeoIP (optional)

MaxMind Inc. publishes a GeoIP database for use in open source software. It can be used to map IP addresses to geographical locations.

5.16.1. Unix

If this library isn't already installed or available as a package for your platform, you can get it at: http://www.maxmind.com/app/c.

5.16.2. Win32 MSVC

You can get the latest version at: http://www.maxmind.com/app/c

Part II. Wireshark Development (incomplete)

Part I. 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.

Part II. Wireshark Development

The second part describes how the Wireshark sources are structured and how to change the sources (e.g. adding a new dissector).

Chapter 6. How Wireshark Works

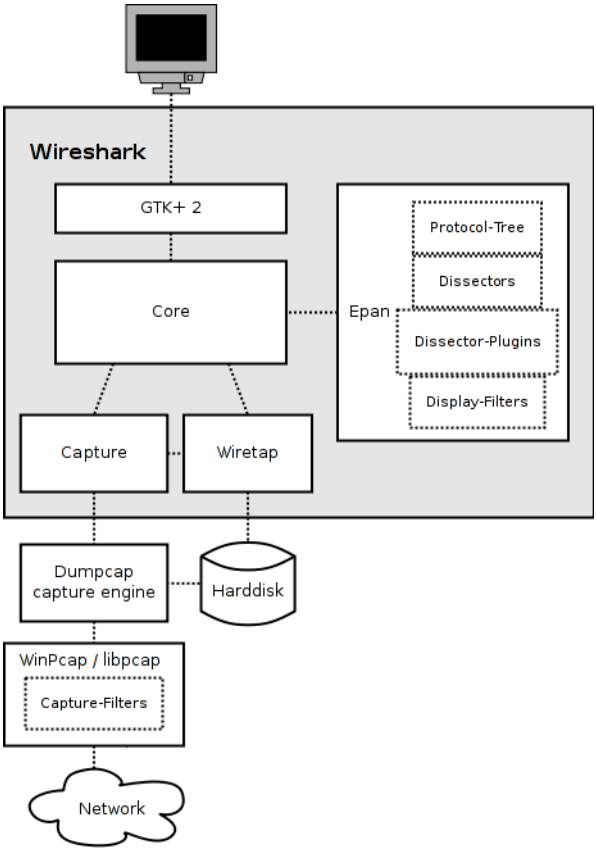
6.1. Introduction

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

6.2. Overview

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

Figure 6.1. Wireshark function blocks.



The function blocks in more detail:

GTK+ 2

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

Core

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

Epan

Ethereal Packet ANalyzer - the packet analyzing engine. Source code can be found in the `epan` directory.

- Protocol-Tree - Keep data of the capture file protocol information.
- Dissectors - The various protocol dissectors in `epan/dissectors`.

	• Dissector-Plugins - Some of the protocol dissectors are implemented as plugins. Source code can be found in `plugins`. • Display-Filters - the display filter engine at `epan/dfilter`.
Wiretap	The wiretap library is used to read/write capture files in libpcap and a lot of other file formats. Source code in the `wiretap` directory.
Capture	The interface with the capture engine. Source code in the root directory.
Dumpcap	The capture engine itself. This is the only part that is to execute with elevated privileges. Source code in the root directory.
WinPcap / libpcap (not part of the Wireshark package)	The platform dependent packet capture library, including the capture filter engine. That's the reason why we still have different display and capture filter syntax, as two different filtering engines are used.

6.3. Capturing packets

Capturing will take packets from a network adapter, and save them to a file on your harddisk.

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 as normal user program.

To hide all the lowlevel machine dependent details from Wireshark, the libpcap/WinPcap (see Section 5.10, “libpcap/WinPcap (optional)”) library is used. This library provides a general purpose interface to capture packets from a lot of different network interface types (Ethernet, Token Ring, ...).

6.4. Capture Files

Wireshark can read and write capture files in its natural file format, the libpcap format, which is used by many other network capturing tools, e.g. tcpdump. In addition to this, as one of its strengths, Wireshark can read/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/write all the file formats. If you need to add another capture file format, this is the place to start.

6.5. 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.

Chapter 7. Introduction

7.1. Source overview

Wireshark consists of the following major parts:

- Packet dissection - in the /epan/dissector and /plugin/* directory
- File I/O - using Wireshark's own wiretap library
- Capture - using the libpcap/winpcap library, in /wiretap
- User interface - using the GTK+ (and corresponding) libraries
- Help - using an external webbrowser and GTK text output

Beside this, some other minor parts and additional helpers exist.

Currently there's no clean separation of the modules in the code. However, as the development team switched from Concurrent Versions System (CVS) to Subversion (SVN) some time ago, directory cleanup is much easier now. So there's a chance that the directory structure will become clean in the future.

7.2. Coding styleguides

The coding styleguides for Wireshark can be found in the "Code style" section of the file `doc/README.developer`.

7.3. The GLib library

Glib is used as a basic platform abstraction library, it's not related to GUI things.

To quote the Glib documentation: “GLib is a general-purpose utility library, which provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on. It works on many UNIX-like platforms, Windows, OS/2 and BeOS. GLib is released under the GNU Library General Public License (GNU LGPL).”

GLib contains lot's of useful things for platform independent development. See http://developer.gnome.org/doc/API/2.0/glib/index.html for details about GLib.

Chapter 8. Packet capturing

XXX - this chapter has to be reviewed and extended!

8.1. How to add a new capture type to libpcap

The following is an 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 the current top-of-tree CVS version of libpcap, and that this is on a UN*X platform. You probably don't want to work with a version older than 0.8, 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()".

Chapter 9. Packet dissection

9.1. How it works

Each dissector decodes its part of the protocol, and then hands off decoding to subsequent dissectors for an encapsulated protocol.

So it might all start with a Frame dissector which dissects the packet details of the capture file itself (e.g. timestamps), passes the data on to an Ethernet frame dissector that decodes the Ethernet header, and then passes the payload 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/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 what's listed in `libwireshark.def`, but that is mostly complete.

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 deployment of the finished code may well be done as built-in dissector.

See also README.developer

The file `doc/README.developer` contains much detailed information about implementing a dissector (and may, in some cases, be more up-to-date than this document).

9.2. 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 IP address.

9.2.1. 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 9.1. Dissector Initialisation.

```
#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#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 we'll assume we are dissecting traffic for.

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 the `proto_register_protocol()` function 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 9.2. 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 stardard 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 9.3. Dissection.

```
static void
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear out stuff in the info column */
    col_clear(pinfo->cinfo,COL_INFO);
}
```

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`:

- Makefile.am - This is the UNIX/Linux makefile template
- Makefile.common - This contains the file names of this plugin
- Makefile.nmake - This contains the Wireshark plugin makefile for Windows
- moduleinfo.h - This contains plugin version info
- moduleinfo.nmake - This contains DLL version info for Windows
- packet-foo.c - This is your dissector source
- plugin.rc.in - This contains the DLL resource template for Windows

You can find a good example for these files in the interlink plugin directory. `Makefile.common` and `Makefile.am` have to be modified to reflect the relevant files and dissector name. `moduleinfo.h` and `moduleinfo.nmake` have to be filled in with the version information. Compile the dissector to a DLL or shared library and copy it into the plugin directory of the installation.

9.2.2. 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. Now the dissector is called in two different cases. In one case it is called to get a summary of the packet, in the other case it is called to look into details of the packet. These two cases can be distinguished by the tree pointer. If the tree pointer is NULL, then we are being asked for a summary. If it is non NULL, we can pick apart the protocol for display. So with that in mind, let's enhance our dissector.

Example 9.4. Plugin Packet Dissection.

```
static void
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{

    col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear out stuff in the info column */
    col_clear(pinfo->cinfo,COL_INFO);

    if (tree) { /* we are being asked for details */
        proto_item *ti = NULL;
        ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, FALSE);
    }
}
```

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. The FALSE we'll ignore for now.

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 9.5. 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 9.6. 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 9.7. Dissector starting to dissect the packets.

```
   if (tree) { /* we are being asked for details */
        proto_item *ti = NULL;
        proto_tree *foo_tree = NULL;

        ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, FALSE);
        foo_tree = proto_item_add_subtree(ti, ett_foo);
        proto_tree_add_item(foo_tree, hf_foo_pdu_type, tvb, 0, 1, FALSE);
    }
```

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, so that is why we use FALSE. Although for 1 byte there is no order issue it's best to keep this correct.

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_UNIT8 - 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 hexdecimal (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 `hf` array, and a couple more procedure calls.

Example 9.8. Wrapping up the packet dissection.

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

static void
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    gint offset = 0;

    ...

    if (tree) { /* we are being asked for details */
        proto_item *ti = NULL;
        proto_tree *foo_tree = NULL;

        ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, FALSE);
        foo_tree = proto_item_add_subtree(ti, ett_foo);
        proto_tree_add_item(foo_tree, hf_foo_pdu_type, tvb, offset, 1, FALSE); offset += 1;
        proto_tree_add_item(foo_tree, hf_foo_flags, tvb, offset, 1, FALSE); offset += 1;
        proto_tree_add_item(foo_tree, hf_foo_sequenceno, tvb, offset, 2, FALSE); offset += 2;
        proto_tree_add_item(foo_tree, hf_foo_initialip, tvb, offset, 4, FALSE); offset += 4;
    }
    ...
}

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 `offset` into the mix to help keep track of where we are in the packet dissection. With these extra bits in place, the whole protocol is now dissected.

9.2.3. 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 9.9. 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 9.10. 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 9.11. 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 void
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    ...
        ...
        proto_tree_add_item(foo_tree, hf_foo_flags, tvb, offset, 1, FALSE);
        proto_tree_add_item(foo_tree, hf_foo_startflag, tvb, offset, 1, FALSE);
        proto_tree_add_item(foo_tree, hf_foo_endflag, tvb, offset, 1, FALSE);
        proto_tree_add_item(foo_tree, hf_foo_priorityflag, tvb, offset, 1, FALSE); offset += 1;
        ...
    ...
}

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 9.12. Enhancing the display.

```
static void
dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    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)"));

    if (tree) { /* we are being asked for details */
        proto_item *ti = NULL;
        proto_tree *foo_tree = NULL;
        gint offset = 0;

        ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, FALSE);
        proto_item_append_text(ti, ", Type %s",
            val_to_str(packet_type, packettypenames, "Unknown (0x%02x)"));
        foo_tree = proto_item_add_subtree(ti, ett_foo);
        proto_tree_add_item(foo_tree, hf_foo_pdu_type, tvb, offset, 1, FALSE);
        offset += 1;
    }
}
```

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.

9.3. 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 9.13. 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*)g_malloc(orig_size);
    offset += 2;
    decompress_packet(tvb_get_ptr(tvb, offset, -1),
            tvb_length_remaining(tvb, offset),
            decompressed_buffer, orig_size);
    /* Now re-setup the tvb buffer to have the new data */
    next_tvb = tvb_new_real_data(decompressed_buffer, orig_size, orig_size);
    tvb_set_child_real_data_tvbuff(tvb, next_tvb);
    add_new_data_source(pinfo, next_tvb, "Decompressed Data");
} else {
    next_tvb = tvb_new_subset(tvb, offset, -1, -1);
}
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 g_malloc, 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_length_remaining()` function.

Next we build a new tvb buffer from this data, using the `tvb_new_real_data()` call. This data is a child of our original data, so we acknowledge that in the next call to `tvb_set_child_real_data_tvbuff`. Finally we add this data as a new data source, so that the detailed display can show the decompressed bytes as well as the original. One procedural step is to add a handler to free the data when it's no longer needed. In this case as `g_malloc()` was used to allocate the memory, `g_free()` is the appropriate function.

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 the `tvb_new_subset()` 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.

9.4. 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.

9.4.1. 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 9.14. 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(tvb, offset, pinfo,
        msg_seqid, /* ID for fragments belonging together */
        msg_fragment_table, /* list of message fragments */
        msg_reassembled_table, /* list of reassembled messages */
        msg_num, /* fragment sequence number */
        tvb_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 data.

- 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.
- The msg_fragment_table and the msg_reassembled_table are variables we need to declare. We'll consider these in detail later.
- 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 9.15. 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(tvb, offset, -1, -1);
    }
}
else { /* Not fragmented */
    next_tvb = tvb_new_subset(tvb, offset, -1, -1);
}

.....
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 9.16. Reassembling fragments - Initialisation

```
static GHashTable *msg_fragment_table = NULL;
static GHashTable *msg_reassembled_table = NULL;

static void
msg_init_protocol(void)
{
    fragment_table_init(&msg_fragment_table);
    reassembled_table_init(&msg_reassembled_table);
}
```

First a couple of hash tables are declared, and these are initialised in the protocol initialisation routine. 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 9.17. 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.

9.4.2. 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 9.18. Reassembling TCP fragments

```
#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include <epan/packet.h>
#include <epan/prefs.h>
#include "packet-tcp.h"

...

#define FRAME_HEADER_LEN 8

/* The main dissecting routine */
static void dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    tcp_dissect_pdus(tvb, pinfo, tree, TRUE, FRAME_HEADER_LEN,
                     get_foo_message_len, dissect_foo_message);
}

/* This method dissects fully reassembled messages */
static void dissect_foo_message(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
    /* TODO: implement your dissecting code */
}

/* determine PDU length of protocol foo */
static guint get_foo_message_len(packet_info *pinfo, tvbuff_t *tvb, int offset)
{
    /* TODO: change this to your needs */
    return (guint)tvb_get_ntohl(tvb, offset+4); /* e.g. length is at offset 4 */
}

...
```

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`* and *`tree`* 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.

9.5. 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 whats 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 9.19. 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 9.20. Calling a protocol tap

```
void dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree)
{
       ...
    fooinfo = ep_alloc(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);
}
```

This now enables those interested parties to listen in on the details of this protocol conversation.

9.6. 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 9.21. Initialising a stats interface

```
/* register all http trees */
static void register_foo_stat_trees(void) {
    stats_tree_register("foo", "foo", "Foo/Packet Types",
        foo_stats_tree_packet, foo_stats_tree_init, NULL);
}

G_MODULE_EXPORT const gchar version[] = "0.0";

G_MODULE_EXPORT void plugin_register_tap_listener(void)
{
    register_foo_stat_trees();
}

#endif
```

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()` function, which takes three strings, and three 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. The function that will called to generate the stats.
5. A function that can be called to initialise the stats data.
6. 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 9.22. 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 9.23. Generating the stats

```
static int foo_stats_tree_packet(stats_tree* st, packet_info* pinfo, epan_dissect_t* edt, const voi
{
    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.

9.7. How to use conversations

Some info about how to use conversations in a dissector can be found in the file `doc/README.developer` chapter 2.2.

Chapter 10. User Interface

10.1. Introduction

Wireshark can be "logically" separated into the backend (dissecting of protocols, file load/save, capturing, ...) and the frontend (the user interface). However, there's currently no clear separation between these two parts (no clear API definition), but this might change in the future.

The following frontends are currently maintained by the Wireshark development team:

- Wireshark, GTK 2.x based
- TShark, console based
- Wireshark, GTK 1.x based (was removed with the Wireshark 1.2.0 release)

There exist other Wireshark frontends, not developed nor maintained by the Wireshark development team:

- Packetyzer (Win32 native interface, written in Delphi and released under the GPL, see: http://www.paglo.com/opensource/packetyzer)
- hethereal (web based frontend, not actively maintained and not finished)

This chapter is focused on the Wireshark frontend, and especially on the GTK specific things.

10.2. The GTK library

Wireshark is based on the GTK toolkit, see: http://www.gtk.org for details. GTK is designed to hide the details of the underlying GUI in a platform independent way. As this is appreciated for a multiplatform tool, this has some drawbacks, as it will result in a somewhat "non native" look and feel.

GTK is available for a lot of different platforms including, but not limited to: Unix/Linux, Mac OS X and Win32. It's the foundation of the famous GNOME desktop, so the future development of GTK should be certain. GTK is implemented in plain C (as is Wireshark itself), and available under the LGPL (Lesser General Public License), being free to used by commercial and noncommercial applications.

There are other similar toolkits like Qt, wxwidgets, ..., which could also be used for Wireshark. There's no "one and only" reason for or against any of these toolkits. However, the decision towards GTK was made a long time ago :-)

At the time this document is written there are two major GTK versions available:

10.2.1. GTK Version 1.x

Please note: After the creation of the 1.0 branch further development removed support for GTK 1.x!

GTK 1.x was the first major release. Today there are 1.2.x and 1.3.x versions "in the wild", with only very limited differences in the API.

Advantages (compared to GTK 2.x):

- available on a lot of different platforms
- very stable as it's matured for quite a while now

Disadvantages:

- the look and feel is a bit old-fashioned
- not recommended for future developments (last GTK 1.x release in 2004)

GTK 1.x depends on the following libraries:

- GDK (GDK is the abstraction layer that allows GTK+ to support multiple windowing systems. GDK provides drawing and window system facilities on X11, Windows, and the Linux framebuffer device.)
- GLib (A general-purpose utility library, not specific to graphical user interfaces. GLib provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on.)

GTK 1.x is working on GLib 1.x (typical for Unix like systems) or 2.x (typical for Win32 like systems).

XXX: include Wireshark GTK1 screenshot

10.2.2. GTK Version 2.x

Advantages (compared to GTK 1.x):

- nice look and feel (compared to version 1.x)
- recommended for future developments
- stable (in productive code for years now)

Disadvantages:

- not available on all platforms (compared to version 1.x)
- more dependencies compared to 1.x, see below

GTK 2.x depends on the following libraries:

- GObject (Object library. Basis for GTK and others)
- GLib (A general-purpose utility library, not specific to graphical user interfaces. GLib provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on.)
- Pango (Pango is a library for internationalized text handling. It centers around the #PangoLayout object, representing a paragraph of text. Pango provides the engine for #GtkTextView, #GtkLabel, #GtkEntry, and other widgets that display text.)
- ATK (ATK is the Accessibility Toolkit. It provides a set of generic interfaces allowing accessibility technologies to interact with a graphical user interface. For example, a screen reader uses ATK to discover the text in an interface and read it to blind users. GTK+ widgets have built-in support for accessibility using the ATK framework.)
- GdkPixbuf (This is a small library which allows you to create #GdkPixbuf ("pixel buffer") objects from image data or image files. Use a #GdkPixbuf in combination with #GtkImage to display images.)
- GDK (GDK is the abstraction layer that allows GTK+ to support multiple windowing systems. GDK provides drawing and window system facilities on X11, Windows, and the Linux framebuffer device.)

XXX: include Wireshark GTK2 screenshot

10.2.3. Compatibility GTK versions

The GTK library itself defines some values which makes it easy to distinguish between the versions, e.g.: GTK_MAJOR_VERSION and GTK_MINOR_VERSION will be set to the GTK version at compile time inside the gtkversion.h header.

10.2.4. GTK resources on the web

You can find several resources about GTK.

First of all, have a look at: http://www.gtk.org as this will be the first place to look at. If you want to develop GTK related things for Wireshark, the most important place might be the GTK API documentation at: http://library.gnome.org/devel/gtk/stable/.

Several mailing lists are available about GTK development, see http://mail.gnome.org/mailman/listinfo, the gtk-app-devel-list may be your friend.

As it's often done wrong: You should post a mail to *help* the developers there instead of only complaining. Posting such a thing like "I don't like your dialog, it looks ugly" won't be of much help. You might think about what you dislike and describe why you dislike it and provide a suggestion for a better way.

10.3. GUI Reference documents

Although the GUI development of Wireshark is platform independent, the Wireshark development team tries to follow the GNOME Human Interface Guidelines (HIG) where appropriate. This is the case, because both GNOME and Wireshark are based on the GTK+ toolkit and the GNOME HIG is excellently written and easy to understand.

For further reference, see the following documents:

- GNOME Human Interface Guidelines at: http://library.gnome.org/devel/hig-book/stable/
- KDE user interface related documents at: http://developer.kde.org/documentation/standards/kde/style/basics/index.html
- Win32 styleguides available at: http://msdn.microsoft.com/en-us/library/aa511258.aspx

10.4. Adding/Extending Dialogs

This is usually the main area for contributing new user interface features.

XXX: add the various functions from gtk/dlg_utils.h

10.5. Widget naming

It seems to be common sense to name the widgets with some descriptive trailing characters, like:

- xy_lb = gtk_label_new();
- xy_cb = gtk_checkbox_new();
- XXX: add more examples

However, this schema isn't used at all places inside the code.

10.6. Common GTK programming pitfalls

There are some common pitfalls in GTK programming.

10.6.1. Usage of gtk_widget_show() / gtk_widget_show_all()

When a GTK widget is created it will be hidden by default. In order to show it, a call to gtk_widget_show() has to be done.

It isn't necessary to do this for each and every widget created. A call to gtk_widget_show_all() on the parent of all the widgets in question (e.g. a dialog window) can be done, so all of its child widgets will be shown too.

Appendix A. This Document's License (GPL)

As with the original license and documentation distributed with Wireshark, this document is covered by the GNU General Public License (GNU GPL).

If you haven't read the GPL before, please do so. It explains all the things that you are allowed to do with this code and documentation.

```
     GNU GENERAL PUBLIC LICENSE
        Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

       Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
```

modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not

excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,

INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.