TSDuck Developer's Guide

Thierry Lelégard

Version 3.38-3816, July 2024

Contents

Preface

TSDuck is a toolkit which manipulates MPEG Transport Streams (TS).

As a product, TSDuck provides many different command-line tools and plugins. Internally, the architecture of TSDuck is made of a large shareable library (`tsduck.dll` on Windows, `libtsduck.so` on Linux and BSD systems, `libtsduck.dylib` on macOS). Most of the valuable processing is performed by the various C++ classes from the TSDuck library. All command-line tools and plugins are usually small wrappers around this library.

The TSDuck library offers generic C++ classes as well as specialized classes for MPEG transport streams. This library can be used as a general-purpose C++ library for third-party applications, outside the TSDuck tools and plugins.

The following figure illustrates the TSDuck software architecture and how it interacts with third-party applications.

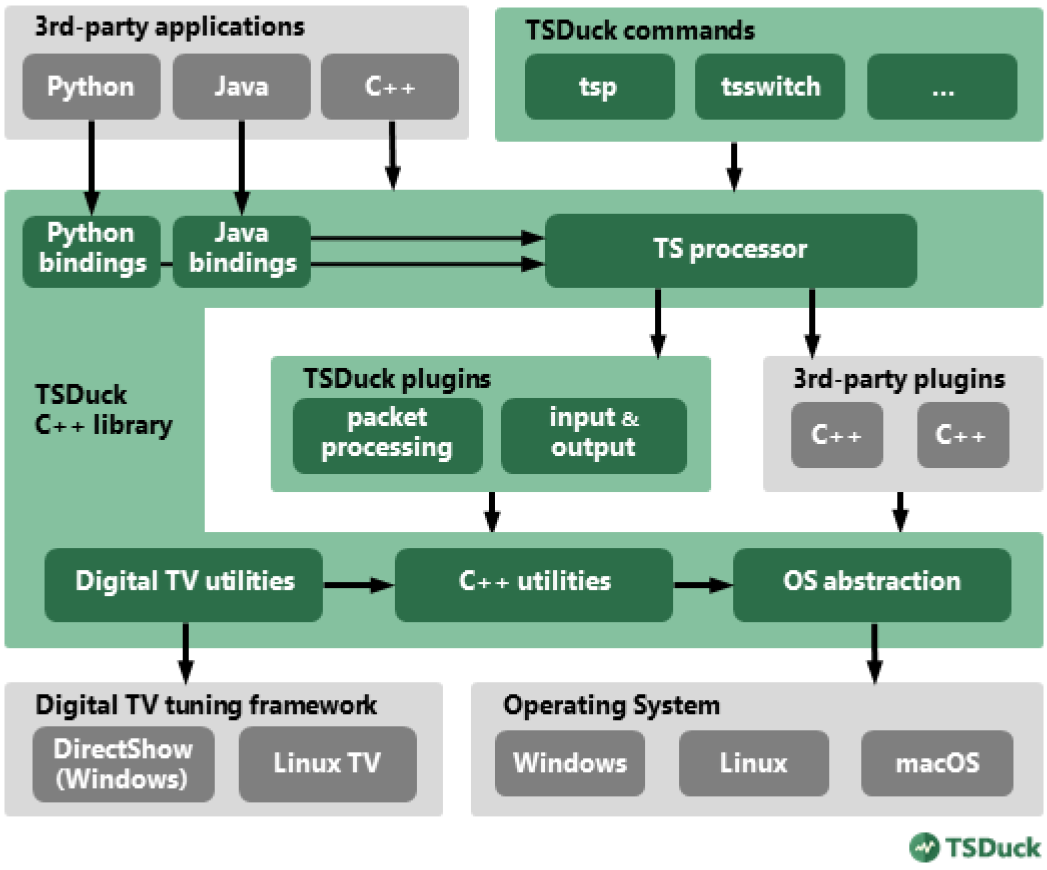

Figure 1. TSDuck software architecture

This document is the TSDuck Developer's Guide. It can be equally used by TSDuck maintainers and developers of third-party applications.

Structure of this guide:

- The chapter 1 describes how to build and install TSDuck. This section is especially useful on platforms for which there is not pre-built binary package. TSDuck users should read this chapter to build and install TSDuck on their system.
- The chapter 2 is an introduction for developers who use the TSDuck library in their application, outside TSDuck itself.
- The chapter 3 is a guide for advanced users who wish to contribute to TSDuck development, submit new features, improvements, or bug fixes.
- The chapter 4 is a reference section for TSDuck maintainers. It describes how the code is organized, the test and delivery processes, the automation of the project.

In addition to this developer's guide, a comprehensive reference programming documentation is available online. It documents all TSDuck public C++ classes, as well as the Python and Java API's. This is a reference documentation which is automatically generated from the source files using doxygen.

License

TSDuck is released under the terms of the license which is commonly referred to as "BSD 2-Clause License" or "Simplified BSD License" or "FreeBSD License". This is a liberal license which allows TSDuck to be used in a large number of environments. See the appendix B for more details.

Documentation format

The TSDuck developer's guide is built using asciidoctor, from a set of text files which are maintained alongside the source code, in the same `git` repository.

This guide is formatted for HTML. The file `tsduck-dev.html` is monolithic and self-sufficient, without reference to external images. Therefore, this HTML file can be downloaded, saved, and copied, as long as the license and content are not modified.

A PDF version tsduck-dev.pdf is also available. However, due to limitations in the PDF generator of asciidoctor, the rendering is sometimes not as good as the HTML document.

Documentation set

The TSDuck documentation set is made of:

1. TSDuck User's Guide (also from tsduck.io and in PDF format)
2. TSDuck Developer's Guide (also from tsduck.io and in PDF format)
3. TSDuck Programming Reference

Chapter 1. Building and Installing TSDuck

This chapter describes how to build and install TSDuck on various platforms. It is particularly useful if TSDuck has no pre-built binary package on your platform.

In the open source world, there are many different ways of building an application. Sometimes, building an open source application is straightforward. However, in other cases, it can be challenging, especially on non-Linux platforms.

TSDuck deliberately sides with the user, using simple and straightforward ways of building and installing.

On each platform, TSDuck always uses "native" mechanisms, tools, and procedures. The default compiler is always the "usual" one on the platform: GCC on Linux, Clang on macOS, Visual Studio on Windows. The packaging of installers, when available, is also "native": `.rpm` packages on Fedora and Red Hat clones, `.deb` packages on Ubuntu, Debian and the like, Homebrew distribution on macOS, NSIS-based executable installers on Windows.

1.1. Building TSDuck

TSDuck can be built on Windows, Linux, macOS and BSD systems.

Support for Dektec devices, DVB tuners and HiDes modulators is implemented only on Windows and Linux. MacOS and BSD systems can only support files and networking for TS input and output. VATek-based modulators, however, are supported on all platforms.

Some protocols such as SRT and RIST require external libraries which may not be available on all platforms or all versions of a specific distro.

1.1.1. Tested systems

TSDuck has been tested on the following operating systems on at least one CPU architecture.

Table 1. Tested operating systems

OS	Variants
macOS	Intel and Apple Silicon
Windows	Tested on Intel only
Linux	Ubuntu, Debian, Raspbian, Mint, Fedora, Red Hat, CentOS, Rocky, Alma, openSUSE, Arch, Alpine, Gentoo, Slackware
BSD	FreeBSD, OpenBSD, NetBSD, DragonFlyBSD

TSDuck has been tested on the following CPU architectures on at least one operating system.

Table 2. Tested CPU architectures

Architecture	Bits	Endianness
Intel x86	32 bits	Little endian
Intel x86-64	64 bits	Little endian
Armv7	32 bits	Little endian
Armv8	64 bits	Little endian
MIPS	32 and 64 bits	Little endian
RISC-V	64 bits	Little endian

Architecture	Bits	Endianness
PowerPC	64 bits	Big endian
IBM s390x	64 bits	Big endian

Some tests were done by contributors and were not verified. Some tests were performed using `qemu`, on an emulated platform, not a physical CPU. This is the case for RISC-V, PowerPC, IBM s390x.

TSDuck has been tested with the following compilers on at least one operating system.

Table 3. Tested compilers

Compiler	OS	Minimum version
GCC	Linux, BSD	11.0
Clang	Linux, macOS	5.0
MSVC	Windows	Visual Studio 2017 15.8

The required minimum version of each compiler depends on a correct implementation of C++17. Previous versions of each compiler have no or buggy C++17 support.

1.1.2. Building on UNIX systems (Linux, macOS, BSD)

On UNIX systems (Linux, macOS, BSD), building TSDuck simply means typing `make` in the top level directory of the project. More details and options are provided in the next sections.

TL;DR

If you don't like the details, just run this:

```
$ git clone https://github.com/tsduck/tsduck.git
$ cd tsduck
$ scripts/install-prerequisites.sh
$ make -j10 default docs-html
$ sudo make install
```

If you like thinking before doing, we recommend to read the following sections.

1.1.2.1. Pre-requisites

Operations in this section must be run once, before building TSDuck for the first time one a given system.

Execute the shell-script `scripts/install-prerequisites.sh`. It downloads and installs the requested packages which are necessary to build TSDuck. The list of packages and how to install them depend on the operating system distribution and version.

If you intend to use exclusion options in the `make` command line (for instance `NOSRT=1 NORIST=1`), specify them to `scripts/install-prerequisites.sh` too. This will prevent the installation of unused libraries.

In addition to the `make` exclusion options, `install-prerequisites.sh` supports `NOJAVA=1 NODOXYGEN=1`.

Currently, this script supports all UNIX operating systems which were listed in Table 1.

Since all packages are pulled from the standard repositories of each distro, there is generally no need to re-run this script later. The packages will be updated as part of the system system updates. Note, however, that a new version of TSDuck may require additional dependencies. In case of build error, it can be wise to run `scripts/install-prerequisites.sh` again and retry.

Although TSDuck has been built and tested on Slackware, the script `install-prerequisites.sh` does not support this distro yet. Slackware is not very friendly for automation. Some package shall be individually searched for a specific version and installed by hand. It has not been possible to find an automated way to setup the required environment to build TSDuck. Should this be possible, contributions from Slackware experts are welcome.

1.1.2.2. C++ compiler requirements

C++ language version

TSDuck now requires a C++17 compliant compiler. GCC is supposed to support C++17 from version 8 onwards. Clang needs version 5 at least.

However, building TSDuck with GCC versions 8 to 10 fails because of bugs in the compiler. C++17 support in GCC really works starting with version 11.

All recent Linux distros use GCC 11, 12 or 13. Some older distros which come with older GCC versions may propose alternative GCC packages with more recent versions.

If your distro is too old and doesn't provide any GCC 11 package, then you cannot build TSDuck version 3.36 and higher. On such systems, the highest TSDuck version which can be built is 3.35. This is the cost of obsolescence…

Using Clang as an alternative to GCC

If your distro is too old and doesn't provide any GCC 11 package, another alternative is to use LLVM/Clang. Most distros with old versions of GCC provide decently recent versions of Clang. To force a build with LLVM/Clang instead of GCC, defined the `make` variable `LLVM`:

```
$ make LLVM=1 ....
```

However, when the installed GCC is really old (typically before GCC 8), using Clang may not work either because Clang uses the GCC C/C++ standard libraries and their header files. If the GCC issue is a compilation issue on GCC 8 to 10, using Clang may work. With older versions of GCC, using Clang probably does not work because the corresponding standard library does not contain the C++17 features.

Red Hat 8.x example

Starting with Red Hat Entreprise Linux 9, all GCC versions correctly support C++17.

However, RedHat Entreprise Linux 8.8 comes with GCC 8.5.0. You can install and use GCC 11 using the following commands:

```
$ sudo dnf install gcc-toolset-11-gcc-c++ gcc-toolset-11-libatomic-devel
$ source /opt/rh/gcc-toolset-11/enable
$ make ...
```

The first command installs the GCC 11 packages. The second command defines the required environment variables in the current process. The last one builds TSDuck.

On RHEL, the GCC 11 packages are available in the AppStream repository. Make sure to have activated it first.

Other Linux distros

Older versions of other distros such as Ubuntu, Debian and others have equivalent alternative packages for GCC 11, with different names, when they come with an older version of GCC.

If there is no `enable` script (as in the example above) to setup the environment, you need to define the following variables, either as environment variables or on the make command line. The provided values are examples only and may be different in specific environments.

```
$ make CXX=g++-11 CC=gcc-11 GCC=gcc-11 CPP="gcc-11 -E" AR=gcc-ar-11 ...
```

Since `make` uses the environment for the initial values of its variables, it is also possible to define them as environment variables in some initialization script instead of using such a complex `make` command..

NetBSD example

As of this writing, the most recent version of NetBSD is 9.3, which comes with GCC 7.5.

More recent GCC packages are available. To install GCC 13:

```
$ sudo pkgin install gcc13
```

The compilation environment is installed in `/usr/pkg/gcc13`. Using GCC 13 is simply enabled by adding `/usr/pkg/gcc13/bin` at the beginning of the `PATH`:

```
$ export PATH="/usr/pkg/gcc13/bin:$PATH"
```

DragonFlyBSD example

As of this writing, the most recent version of DragonFlyBSD is 6.4.0, which comes with GCC 8.3. Even though DragonFlyBSD is supposed to be based on FreeBSD, its GCC version is way behind FreeBSD version 14.0 which comes with GCC 12.2.

More recent GCC packages are available for DragonFlyBSD. To install GCC 13:

```
$ sudo pkg install gcc13
```

However, because all *BSD systems are carefully incompatible between each other, using the alternative compiler is very different from NetBSD.

Building TSDuck:

```
$ gmake CXX=g++13 CC=gcc13 GCC=gcc13 CPP="gcc13 -E" AR=gcc-ar13 LDFLAGS_EXTRA="-Wl,-
rpath=/usr/local/lib/gcc13" ...
```

Since `make` uses the environment for the initial values of its variables, it is also possibe to define them as environment variables in some initialization script instead of using such a complex `make` command..

Note the command `gmake`, the GNU Make command. See section 1.1.2.4 for more details.

1.1.2.3. Hardware device libraries

Dektec DTAPI: The command `make` at the top level will automatically download the LinuxSDK from the Dektec site. There is no manual setup for DTAPI on Linux. Note that the Dektec DTAPI is available only for Linux distros on Intel CPU's with the GNU libc. Non-Intel systems (for instance Arm-based devices such as Raspberry Pi) cannot use Dektec devices. Similarly, Intel-based distros using a non-standard libc (for instance Alpine Linux which uses musl libc) cannot use Dektec devices either.

VATek API: On Linux, the command `make` at the top level will automatically download the Linux version of the VATek API from the GitHub. There is currectly no Linux package for the VATek API in the standard distros. On Windows and macOS, binary packages are available and are installed by the `install-prerequisites` scripts. Using VATek devices on BSD systems is currently not supported but should work if necessary (accessing VATek devices is performed through `libusb` and not a specific kernel driver).

1.1.2.4. Using make on BSD systems

On FreeBSD, OpenBSD, NetBSD, DragonFlyBSD, the standard BSD `make` command uses an old syntax. The makefiles in the TSDuck project use a GNU Make syntax and are not compatible with the BSD `make` command. As part of prerequisites for BSD systems, GNU Make is installed under the name `gmake`. In all build commands in this page, when `make` is mentioned, use `gmake` on all BSD systems.

1.1.2.5. Building the TSDuck binaries alone

Execute the command `make` at top level.

The TSDuck binaries, executables and shared objects (`.so` or `.dylib`), are built in directory `bin/release-<arch>-<hostname>` by default. Consequently, the same work area can be simultaneously used by several systems. Each system builds in its own area. You can also override the build directory using `make BINDIR=…`.

Note that TSDuck contains thousands of source files and building it can take time. However, since most machines have multiple CPU's, all makefiles are designed for parallel builds. On a quad-core machine with hyperthreading (8 logical cores), for instance, the command `make -j10` is recommended (10 parallel compilations), reducing the total build time to a few minutes.

As an example, on an Intel system from 2020, building TSDuck without parallelism takes several hours. On the same system, using `-j10`, it takes 20 minutes. On a recent iMac M3, using `-j10`, the build time is 2 minutes.

To cleanup the repository tree and return to a pristine source state, execute `make clean` at the top level.

1.1.2.6. Building without specialized dependencies

In specific configurations, you may want to disable some external libraries such as `libcurl` or `pcsc-lite`. Of course, the corresponding features in TSDuck will be disabled but the impact is limited. For instance, disabling `libcurl` will disable the input plugins `http` and `hls`.

The following `make` variables can be defined:

`NOTEST`	Do not build unitary tests.
`NODEKTEC`	No Dektec device support, remove dependency to `DTAPI`.
`NOHIDES`	No HiDes device support.
`NOVATEK`	No VATek device support (modulators based on VATek chips), remove dependency to `libvatek`.
`NOCURL`	No HTTP support, remove dependency to `libcurl`.
`NOPCSC`	No smartcard support, remove dependency to `pcsc-lite`.
`NOEDITLINE`	No interactive line editing, remove dependency to `libedit`.
`NOSRT`	No SRT support (Secure Reliable Transport), remove dependency to `libsrt`.
`NORIST`	No RIST support (Reliable Internet Stream Transport), remove dependency to `librist`.
`NOHWACCEL`	Disable hardware acceleration such as crypto instructions.
`ASSERTIONS`	Keep assertions in production mode (slower code).

The following command, for instance, builds TSDuck without dependency to `pcsc-lite`, `libcurl` and Dektec DTAPI:

```
$ make NOPCSC=1 NOCURL=1 NODEKTEC=1
```

Note that some dependencies such as `openssl` (cryptographic library) cannot be removed because they are deeply used inside TSDuck.

1.1.2.7. Building with specific debug capabilities

The following additional `make` variables can be defined to enable specific debug capabilities:

`DEBUG`	Compile with debug information and no optimization.
`GPROF`	Compile with code profiling using `gprof`.
`GCOV`	Compile with code coverage using `gcov`.
`ASAN`	Compile with code sanitizing using AddressSanitizer with default optimization.
`UBSAN`	Compile with code sanitizing using UndefinedBehaviorSanitizer with default optimization.

1.1.2.8. Displaying full build commands

Because of the number of include directories and warning options, the compilation commands are very long, typically more than 4000 characters, 30 to 50 lines on a terminal window. If the `make` commands displays all commands, the output is messy. It is difficult to identify the progression of the build. Error messages are not clearly identified.

Therefore, the `make` command only displays a synthetic line for each command such as:

```
[CXX] dtv/tables/dvb/tsAIT.cpp
[CXX] dtv/tables/atsc/tsATSCEIT.cpp
[CXX] dtv/tables/tsAbstractDescriptorsTable.cpp
```

In some cases, if can be useful to display the full compilation commands. To do this, define the variable `VERBOSE` as follow:

```
$ make VERBOSE=1
```

For convenience and compatibility with some tradition, `V` can be used instead of `VERBOSE`.

1.1.2.9. Building the TSDuck installation packages

Execute the command `make installer` at top level to build all packages.

Depending on the platform, the packages can be `.deb` or `.rpm` files. There is currently no support to build an installation package on other Linux distros and BSD systems.

There is no need to build the TSDuck binaries before building the installers. Building the binaries, when necessary, is part of the installer build.

All installation packages are dropped into the subdirectory `pkg/installers`. The packages are not deleted by the cleanup procedures. They are not pushed into the git repository either.

On macOS, there is no binary package for TSDuck on macOS. On this platform, TSDuck is installed using Homebrew, a package manager for open-source projects on macOS. See section 1.3.2 for more details.

1.1.2.10. For packagers of Linux distros

Packagers of Linux distros may want to create TSDuck packages. The build methods are not different. This section contains a few hints to help the packaging.

By default, TSDuck is built with capabilities to check the availability of new versions on GitHub. The `tsversion` command can also download and upgrade TSDuck from the binaries on GitHub. Packagers of Linux distros may want to disable this since they may prefer to avoid mixing their TSDuck packages with the generic TSDuck packages on GitHub. To disable this feature, build TSDuck with `make NOGITHUB=1`.

The way to build a package depends on the package management system. Usually, the build procedure includes an installation on a temporary fake system root. To build TSDuck and install it on `/temporary/fake/root`, use the following command:

```
$ make NOGITHUB=1 install SYSROOT=/temporary/fake/root
```

It is recommended to create two distinct packages: one for the TSDuck tools and plugins and one for the development environment. The development package shall require the pre-installation of the tools package.

If you need to separately build TSDuck for each package, use `make` targets `install-tools` and `install-devel` instead of `install` which installs everything.

```
$ make NOGITHUB=1 install-tools SYSROOT=/temporary/fake/root
$ make NOGITHUB=1 install-devel SYSROOT=/temporary/fake/root
```

1.1.2.11. Installing in non-standard locations

On systems where you have no administration privilege and consequently no right to use the standard installers, you may want to manually install TSDuck is some arbitrary directory.

You have to rebuild TSDuck from the source repository and install it using a command like this one:

```
$ make install SYSPREFIX=$HOME/usr/local
```

Unlike many open source applications on Linux, the TSDuck binaries are independent from the installation locations. There is no equivalent to `./configure --prefix …`. The same binaries can be installed in different locations, provided that the installation is consistent (typically using `make install …`).

The TSDuck commands are located in the `bin` subdirectory and can be executed from here without any additional setup. It is probably a good idea to add this `bin` directory in your `PATH` environment variable.

1.1.2.12. Using pkgconfig after installation

Applications may use the `pkgconfig` utility to reference the TSDuck library. A file named `tsduck.pc` is installed in the appropriate directory.

However, `pkgconfig` has its own limitations, specifically regarding the configured compilation options.

TSDuck is a C++ library which requires a minimum revision of the language. Currently, the minimum revision is C++17. All more recent revisions are supported. By default, most C++ compilers are based on older revisions. Therefore, compiling an application using TSDuck with the default options fails. At least, `-std=c++17` is required. To avoid compilation problems with most applications, `-std=c++17` is enforced in `tsduck.pc`.

However, some applications may need to explicitly specify an even more recent revision, such as `-std=c++20`, which conflicts with `-std=c++17` in `tsduck.pc`.

For that use case, you may install TSDuck without reference to the C++ revision using the following command:

```
$ make install NOPCSTD=1
```

The counterpart is that the applications *must* specify a `-std` option and the revision must be C++17 or more recent.

A generic solution would be that each library and the application all provide a *minimum* revision of the C++ language and pkgconfig would provide a synthetic `-std` option which fulfills all requirements. However, this feature does not exist in pkgconfig, hence this trick.

1.1.2.13. Running from the build location

It is sometimes useful to run a TSDuck binary, `tsp` or any other, directly from the build directory, right after compilation, without going through `make install`. This can be required for testing or debugging.

Because the binary directory name contains the host name, it is possible to build TSDuck using the same shared source tree from various systems or virtual machines. All builds will coexist using distinct names under the `bin` subdirectory.

For `bash` users who wish to include the binary directory in the `PATH`, simply "source" the script `scripts/setenv.sh`.

Example:

```
$ . scripts/setenv.sh
$ which tsp
/Users/devel/tsduck/bin/release-x86_64-mymac/tsp
```

This script can also be used with option `--display` to display the actual path of the binary directory. The output can be used in other scripts (including from any other shell than `bash`).

Example:

```
$ scripts/setenv.sh --display
/Users/devel/tsduck/bin/release-x86_64-mymac
```

Use `scripts/setenv.sh --help` for other options.

1.1.3. Building on Windows systems

On Windows systems, building a TSDuck installer simply means executing the PowerShell script `pkg\nsis\build-installer.ps1`. More details and options are provided in the next sections.

1.1.3.1. Pre-requisites

Operations in this section must be run once, before building TSDuck for the first time one a given Windows system. It should also be run to get up-to-date versions of the build tools and libraries which are used by TSDuck.

First, install Visual Studio Community Edition. This is the free version of Visual Studio. It can be downloaded here. If you already have Visual Studio Enterprise Edition (the commercial version), it is fine, no need to install the Community Edition.

Then, execute the PowerShell script `scripts\install-prerequisites.ps1`. It downloads and installs the requested packages which are necessary to build TSDuck on Windows.

If you prefer to collect the various installers yourself, follow the links to NSIS downloads, Git downloads, SRT downloads, RIST downloads, Dektec downloads, VATek downloads, Java downloads, Python downloads, Doxygen downloads, Graphviz downloads.

TSDuck now requires a C++17 compliant compiler. C++17 support started with Visual Studio 2017 15.8. We recommend to use Visual Studio 2022.

1.1.3.2. Building the binaries without installer

Execute the PowerShell script `scripts\build.ps1`. The TSDuck binaries, executables and DLL's, are built in directories named `bin\<target>-<platform>`, for instance `bin\Release-x64` or `bin\Debug-Win32`.

To cleanup the repository tree and return to a pristine source state, execute the PowerShell script `scripts\cleanup.ps1`.

1.1.3.3. Building the Windows installers

Execute the PowerShell script `pkg\nsis\build-installer.ps1`. By default, only the 64-bit installer is built. To build the two installers, for 32-bit and 64-bit systems, run the build script from a PowerShell window and add the option `-Win32`.

There is no need to build the TSDuck binaries before building the installers. Building the binaries, is part of the installer build.

All installation packages are dropped into the subdirectory `pkg/installers`. The packages are not deleted by the cleanup procedures. They are not pushed into the git repository either.

1.1.3.4. Installing in non-standard locations

On systems where you have no administration privilege and consequently no right to use the standard installers, you may want to manually install TSDuck is some arbitrary directory.

On Windows systems, a so-called *portable* package is built with the installers. This is a zip archive file which can be expanded anywhere. It is automatically built by `pkg\nsis\build-installer.ps1`, in addition to the executable installer.

1.1.3.5. Running from the build location

It is sometimes useful to run a TSDuck binary, `tsp` or any other, directly from the build directory, right after compilation. This can be required for testing or debugging.

The commands can be run using their complete path without additional setup. For instance, to run the released 64-bit version of `tsp`, use:

```
PS D:\tsduck> bin\Release-x64\tsp.exe --version
tsp: TSDuck - The MPEG Transport Stream Toolkit - version 3.12-730
```

For other combinations (release vs. debug and 32 vs. 64 bits), the paths from the repository root are:

```
bin\Release-x64\tsp.exe
bin\Release-Win32\tsp.exe
bin\Debug-x64\tsp.exe
bin\Debug-Win32\tsp.exe
```

1.1.4. Installer files summary

The following list summarizes the packages which are built and dropped into the `pkg/installers` directory, through a few examples, assuming that the current version of TSDuck is 3.37-3670.

`tsduck_3.37-3670.ubuntu23_amd64.deb`	Binary package for 64-bit Ubuntu 23.x
`tsduck_3.37-3670.ubuntu23_arm64.deb`	Binary package for Arm 64-bit Ubuntu 23.x
`tsduck_3.37-3670.debian12_amd64.deb`	Binary package for 64-bit Debian 12
`tsduck_3.37-3670.raspbian12_armhf.deb`	Binary package for 32-bit Raspbian 12 (Raspberry Pi)
`tsduck-3.37-3670.el9.x86_64.rpm`	Binary package for 64-bit Red Hat 9.x and clones
`tsduck-3.37-3670.el9.src.rpm`	Source package for Red Hat and clones
`tsduck-3.37-3670.fc39.x86_64.rpm`	Binary package for 64-bit Fedora 39
`tsduck-3.37-3670.fc39.src.rpm`	Source package for Fedora
`tsduck-dev_3.37-3670.ubuntu23_amd64.deb`	Development package for 64-bit Ubuntu 23.x
`tsduck-dev_3.37-3670.ubuntu23_arm64.deb`	Development package for Arm 64-bit Ubuntu 23.x
`tsduck-dev_3.37-3670.debian12_amd64.deb`	Development package for 64-bit Debian 12
`tsduck-dev_3.37-3670.raspbian12_armhf.deb`	Development package for 32-bit Raspbian (Raspberry Pi)
`tsduck-devel-3.37-3670.el9.x86_64.rpm`	Development package for 64-bit Red Hat 9.x and clones
`tsduck-devel-3.37-3670.fc39.x86_64.rpm`	Development package for 64-bit Fedora 39
`TSDuck-Win32-3.37-3670.exe`	Binary installer for 32-bit Windows
`TSDuck-Win64-3.37-3670.exe`	Binary installer for 64-bit Windows
`TSDuck-Win32-3.37-3670-Portable.zip`	Portable package for 32-bit Windows
`TSDuck-Win64-3.37-3670-Portable.zip`	Portable package for 64-bit Windows

On Linux systems, there are two different packages. The package `tsduck` contains the tools and plugins. This is the only required package if you just need to use TSDuck. The package named `tsduck-devel` (Red Hat family) or `tsduck-dev` (Debian family) contains the development environment. It is useful only to build third-party applications which use the TSDuck library.

On Windows systems, there is only one binary installer which contains the tools, plugins, documentation and development environment. The user can select which components shall be installed. The development environment is unselected by default.

On macOS systems, the Homebrew package `tsduck` installs all components.

1.2. Building the documentation

There are three sets of TSDuck documents:

1. TSDuck User's Guide (HTML and PDF)
2. TSDuck Developer's Guide (HTML and PDF)
3. TSDuck Programming Reference (HTML only)

The first two documents are written in Asciidoc format. Their HTML and PDF versions are built using Asciidoctor. The two HTML files are large standalone files, without reference to any other local file; they can be safely copied without breaking the navigation.

These two guides are installed with TSDuck on UNIX systems (Linux, macOS, BSD) and Windows (HTML version only).

The TSDuck Programming Reference contains the documentation of all public classes which can be used by applications in C++, Java, or Python. This reference is built using Doxygen.

Asciidoctor and Doxygen are automatically installed by the scripts `install-prerequisites.sh` on UNIX systems (Linux, macOS, BSD) and `install-prerequisites.ps1` on Windows.

On large libraries, Doxygen is extremely verbose. The TSDuck Programming Reference is made of a large number of HTML files, more than 14,000 files and directories. It also takes some time to generate. Therefore, the Programming Reference is neither built by default nor installed with the rest of TSDuck. Every night, a fresh copy is generated and published online at https://tsduck.io/doxy.

1.2.1. Building on UNIX systems (Linux, macOS, BSD)

The user's guide and the developer's guide are built using the target `docs`. The HTML and PDF files are built in subdirectory `bin/doc`.

```
$ make docs
```

Because the two guides are installed with the rest of TSDuck, they are automatically rebuilt as part of `make install`.

The following targets are also available to build a subset of the guides:

`userguide-html`	Build the user's guide HTML version
`userguide-pdf`	Build the user's guide PDF version
`userguide`	Build the user's guide HTML and PDF versions
`open-userguide-html`	Build the user's guide HTML version and opens it with the default HTML viewer
`open-userguide-pdf`	Build the user's guide PDF version and opens it with the default PDF viewer
`open-userguide`	Build the user's guide HTML and PDF versions and opens them with their default viewers
`devguide-html`	Build the developer's guide HTML version
`devguide-pdf`	Build the developer's guide PDF version
`devguide`	Build the developer's guide HTML and PDF versions
`open-devguide-html`	Build the developer's guide HTML version and opens it with the default HTML viewer
`open-devguide-pdf`	Build the developer's guide PDF version and opens it with the default PDF viewer
`open-devguide`	Build the developer's guide HTML and PDF versions and opens them with their default viewers
`docs`	Build the four document, user and developer, HTML and PDF
`docs-html`	Build the user and developer's guide in HTML format
`docs-pdf`	Build the user and developer's guide in PDF format

The programming reference is built using the target `doxygen`.

```
$ make doxygen
```

The set of files is built in subdirectory `bin/doxy/html`.

1.2.2. Building on Windows

The user's guide and the developer's guide are built using the PowerShell script `doc\build-doc.ps1`. The HTML and PDF files are built in subdirectory `bin\doc`. By default, they are automatically opened using the default HTML and PDF viewers of the system.

Because the two guides are installed with the rest of TSDuck, this script is automatically executed as part of the script `pkg\nsis\build-installer.ps1`.

The programming reference is built using the PowerShell script `doc\doxy\build-doxygen.ps1`. The set of files is built in subdirectory `bin\doxy\html`. By default, the start page is automatically opened using the default HTML viewer of the system.

When used in an automation system, the two scripts `doc\build-doc.ps1` and `pkg\nsis\build-installer.ps1` can be called with options `-NoOpen -NoPause` to skip the opening of documents using the default viewers and exit without waiting for a user to close the command window.

1.3. Installing TSDuck

TSDuck can be installed on Windows, Linux, macOS and BSD systems.

1.3.1. Installing on Windows

On Windows systems, TSDuck can be installed using a binary installer (traditional method) or using the `winget` package manager (modern method).

1.3.1.1. Using winget

TSDuck is installable on Windows systems using the winget package manager.

`winget` is now the preferred package manager for open source and third-party products on Windows systems. It is documented and supported by Microsoft. It should be pre-installed on all recent Windows 10 and Windows 11 systems.

The TSDuck installation command is simply:

```
PS C:\> winget install tsduck
```

1.3.1.2. Download an installer

Executable binary installers for the latest TSDuck version are available for 64-bit Windows on Intel systems.

All tools, plugins and development environments are in the same installer. Running the installer provides several options:

- Tools & Plugins
- Documentation
- Python Bindings (optional)
- Java Bindings (optional)
- C++ Development (optional)

Older versions of TSDuck remain available on GitHub.

Nightly builds and pre-releases can be found on the TSDuck Web site.

To automate the installation, the executable binary installer can be run from the command line or a script.

- The option `/S` means "silent". No window is displayed, no user interaction is possible.
- The option `/all=true` means install all options. By default, only the tools, plugins and documentation are installed. In case of upgrade over an existing installation, the default is to upgrade the same options as in the previous installation.

1.3.2. Installing on macOS

TSDuck is installable on macOS systems using Homebrew, the package manager for open-source projects on macOS.

If you have never used Homebrew on your system, you can install it using the following command (which can also be found on the Homebrew home page):

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once Homebrew is set up, you can install TSDuck using:

```
$ brew install tsduck
```

All tools, plugins and development environments are installed.

After installation, to upgrade to latest version:

```
$ brew update
$ brew upgrade tsduck
```

When Homebrew upgrades packages, the old versions are not removed. The new versions are just added. After a while, megabytes of outdated packages accumulate on disk. To remove outdated packages:

```
$ brew cleanup
```

To uninstall TSDuck:

```
$ brew uninstall tsduck
```

If you would like to install the lastest test version (HEAD version) use the following command. Be aware that it takes time since TSDuck is locally recompiled.

```
$ brew install --HEAD tsduck
```

1.3.3. Installing on Linux

Pre-build packages for the latest TSDuck version are available for the following configurations:

- Fedora (64-bit Intel)
- Ubuntu (64-bit Intel and Arm)
- RedHat, CentOS, Alma Linux (64-bit Intel)
- Debian (64-bit Intel)
- Raspbian (32-Bit Arm, Raspberry Pi)

The type of package, `.rpm` or `.deb`, depends on the configuration. The pre-built packages are provided for the latest version of each distro only.

For each distro, two packages exist: the `tsduck` package installs the TSDuck commands, plugins, Java and Python bindings, the `tsduck-devel` or `tsduck-dev` package installs the development environment for C++ programmers.

Older versions of TSDuck remain available on GitHub. Nightly builds and pre-releases for Ubuntu can be found on the TSDuck Web site.

To use older versions of the above distros, rebuilding the packages is easy:

```
$ make installer
```

To install TSDuck on other types of Linux systems for which no package is available:

```
$ make -j10 default docs-html
$ sudo make install
```

More details on how to build TSDuck are available in section 1.1.

1.3.4. Installing on BSD systems

There is currently no installer for FreeBSD, OpenBSD, NetBSD, DragonFlyBSD. You need to build and install as follow:

```
$ gmake -j10 default docs-html
$ sudo gmake install
```

Note that GNU Make (`gmake`) shall be used instead of the standard BSD `make`.

Chapter 2. Developing Applications with TSDuck

The TSDuck library is made of a large number of general-purpose C++ classes for Digital TV. They can be used by any application to manipulate MPEG transport streams, the signalization, and various data which are carried in MPEG-TS.

A subset of high-level features is also accessible from Java and Python applications.

This chapter describes how to use the TSDuck library in third-party applications.

It also describes how to develop TSDuck plugins and TSDuck extensions outside the TSDuck project. Although we encourage open source contributions to enrich the project, it can be useful to extend TSDuck in private applications to test proprietary features.

2.1. Building an application with the TSDuck library

2.1.1. Pre-requisites

To be able to build applications or `tsp` plugins with the TSDuck library, you must install the TSDuck development environment first.

- On Windows systems, you must select the optional "Development" component during the installation.
- On Fedora, Red Hat and clones, you must install the package `tsduck-devel`.
- On Ubuntu and the Debian family, you must install the package `tsduck-dev`.
- On macOS systems, the development environment is always installed with TSDuck using Homebrew.
- If you build TSDuck from sources, use `make install` (which is equivalent to `make install-tools install-devel`).

2.1.2. Building applications on UNIX systems (Linux, macOS, BSD)

The command `tsconfig` generates the appropriate build options for the current operating system. See the TSDuck user's guide for more details on `tsconfig`.

The following sample makefile illustrates the creation of a simple application named `myexec` using one single source file `myexec.cpp`.

```
CXXFLAGS += $(shell tsconfig --cflags)
LDLIBS += $(shell tsconfig --libs)

default: myexec
```

This is as simple as that.

Just run `make` to build the application.

```
$ make
```

C++ language level

By default, the command `tsconfig --cflags` forces C++17 as level of C++ language standard. If your application requires a more recent level, define the environment variable `TS_NOSTDCPP` to any non-empty value. This disables the C++ standard option in `tsconfig`. The application shall then define its own C++ standard in its command line. This user-specified C++ standard cannot be lower than C++17.

Alternatively, the command `tsconfig --nostdcpp --cflags` can be used to omit the C++ standard from the compilation options without defining the environment variable `TS_NOSTDCPP`.

2.1.3. Building applications on Windows

The "Development" option of the TSDuck installer provides the build environment for Visual Studio 2022, in debug and release mode, for 64-bit Intel platforms. It may be compatible with Visual Studio 2019 or earlier, but without guarantee.

The environment variable `TSDUCK` is defined to the root of the TSDuck installation tree. A Visual Studio property file named `tsduck.props` is installed here. It provides all definitions and options to use the TSDuck library.

Create the solution and projects for your application. Then, manually edit the project file, named for instance `app.vcxproj`, and insert the following line just before the final `</Project>` closing tag:

```
<Import Project="$(TSDUCK)\tsduck.props"/>
```

Then build your project normally.

C++ language level

By default, the property file `tsduck.props` forces C++17 as level of C++ language standard. If your application requires a more recent level, define the environment variable `TS_NOSTDCPP` to any non-empty value. This disables the C++ standard option in `tsduck.props`. The application shall then define its own C++ standard in its project files. This user-specified C++ standard cannot be lower than C++17.

2.2. Overview of the TSDuck library

The TSDuck library contains general-purpose C++ classes and utilities to handle MPEG transport streams.

For programming details, see the reference documentation online (doxygen-generated).

Roughly, the TSDuck library provides two categories of features:

- Operating system abstraction layer to make the application code fully portable between heterogeneous platforms. This is similar to frameworks such as Qt, but much more lightweight.
- Handling of MPEG transport streams and signalization, including DVB, ATSC and ISDB features.

In early versions of TSDuck, the OS abstraction layer contained many more classes. Starting with C++11 and C++17, the standard library of the language was enriched with many more system features. Some low-level TSDuck classes became obsolete. The code was migrated to use standard C++17 features and the corresponding low-level classes were removed from TSDuck.

All C++ declarations are located inside the namespace `ts`, either directly within `ts` or inside inner namespaces. All preprocessor’s macros are named with prefix `TS_`.

2.2.1. C++ features

2.2.1.1. Portability issues

The file `tsPlatform.h` contains some very low level definitions such as macros defining the environment (processor, compiler, operating system, endianness), byte and bit manipulation, etc.

2.2.1.2. C++ strings

C and C++ strings are made of 8-bit characters which are notoriously unable to represent international character sets. The usage of `std::string` with the TSDuck library is discouraged in favor of *Unicode strings*.

2.2.1.3. Unicode strings

The class `ts::UString` implements Java-like Unicode strings. Each character uses 16 bits of storage. Formally, `ts::UString` uses UTF-16 representation. This means that all characters from all modern languages can be represented as one single character. Characters from archaic languages may need two UTF-16 values, called a "surrogate pair".

Technically, `ts::UString` is a subclass of `std::u16string`. So any operation on standard C++ strings is also available to `ts::UString`. But many more operations have been added to manipulate Unicode strings.

For consistency, the type `ts::UChar` is an alias for `char16_t`. The header file `tsUChar.h` defines some utility functions on `ts::UChar`. It also defines constants for most Unicode characters like `ts::COLON` or more complex ones such as `ts::LATIN_CAPITAL_LETTER_A_WITH_ACUTE`, among hundredths of others.

Some interesting features in class `ts::UString` are:

- Explicit and implicit conversions between UTF-8 and UTF-16.
- Including automatic conversion to UTF-8 when writing to text streams.
- Conversions with DVB and ARIB character sets.
- Conversions with HTML encoding.
- Management of "display width", that is to say the amount of space which is used when the string is displayed. This can be different from the string length in the presence of combining diacritical characters or surrogate pairs.
- String padding, trimming, truncation, justification, case conversions.
- Substring, prefix or suffix detection, removal or substitution.
- Splitting and joining strings based on separators or line widths.
- Reading or writing text lines from or to a text file.
- Data formatting using `format()`, `Format()`, `Decimal()`, `Hexa()`, `Dump()`.
- Data scanning using `scan()`.

Unicode strings can be converted to and from DVB or ARIB (Japan) strings. Most DVB-defined character sets are implemented (see the classes `ts::Charset` and `ts::DVBCharset`) and recognized when a string is read from a descriptor. When a string is serialized into a binary descriptor, the most appropriate DVB character set is used. In practice, a few known DVB character sets are used and, when the string cannot be encoded in any of them, UTF-8 is used (UTF-8 is always a valid DVB character set).

2.2.1.4. Binary data

The class `ts::ByteBlock` represents a raw block of bytes. It is a subclass of `std::vector<uint8_t>` and consequently benefits from all standard vector operations. It also adds useful methods for data serialization or deserialization in any byte order.

For data serialization or deserialization over arbitrary memory areas, the header file `tsMemory.h` provides low-level functions to access integer values of 8, 16, 24, 32, 40, 48 and 64 bits in any byte order.

The class `ts::Buffer` provides a higher-level abstraction layer over a memory area to parse or generate bitstreams. It gives access to data of any bit-size at any bit position, any endianness, either as a continuous stream or seeking at random bit positions.

The principles of the C++ class `ts::Buffer` were freely inspired by the Java class `java.nio.ByteBuffer`. There are differences between the two but the main principles are similar.

Its subclass `ts::PSIBuffer` provides primitives to serialize and deserialize MPEG and DVB structures such as list of descriptors, DVB, ARIB and ATSC strings or "Modified Julian Dates".

2.2.1.5. Singletons and static data

The *singleton* design pattern is simple in theory, but not so simple to implement correctly in practice. The TSDuck library encapsulates the implementation difficulties using the two macros `TS_DECLARE_SINGLETON()` (in a header file) and `TS_DEFINE_SINGLETON()` (in the corresponding compilation unit).

Similarly, using static data can be a nightmare because it is impossible to manage the initialization order of modules in C++. Again, the TSDuck library encapsulates these implementation difficulties using the macro `TS_STATIC_INSTANCE()`. This is a variant of the singleton pattern, privately used inside a compilation unit.

2.2.1.6. Error reporting

All TSDuck classes use a consistent error reporting mechanism through the `ts::Report` abstract class.

This interface defines several levels of severity in the type `ts::Severity`, ranging from `ts::Severity::Debug` to `ts::Severity::Fatal`. Each instance of `ts::Report` defines which levels of message are reported to the user. This is usually triggered by command-line options such as `--verbose` or `--debug`.

Most classes or methods from the TSDuck library use a reference to an instance of `ts::Report` to report messages and errors. The actual reporting object is often built at application level and then propagated to all layers of code.

Some interesting subclasses of `ts::Report` are:

- `ts::CerrReport`, a singleton which reports errors to `std::cerr`. The macro `CERR` can be used as a shortcut to the instance of the singleton.
- `ts::NullReport`, a singleton which drops all messages. The macro `NULLREP` can be used as a shortcut to the instance of the singleton.
- `ts::ReportFile` which logs messages in a file. It can be made thread-safe using a `ts::ThreadSafety` value as template argument.
- `ts::ReportBuffer` which logs messages in a memory buffer. It can be made thread-safe using a `ts::ThreadSafety` value as template argument.
- `ts::Args` (see section 2.2.1.9) which defines the syntax and handling of command line arguments. This is the typical instance of `ts::Report` which is used at application-level.
- `ts::Plugin`, the superclass of all `tsp` plugins. A plugin reports its messages directly in its own instance. Each `tsp` plugin executes in a separate thread and asynchronously logs messages without slowing down the plugin’s thread.

2.2.1.7. Exceptions

As a general rule, TSDuck prefers the usage of error reporting interface and error status over exceptions. However, for a limited number of unrecoverable conditions which should never occur in practice, exceptions are used.

All TSDuck exceptions inherit from the superclass `ts::Exception`. An instance of this exception is able to embed an error message and an optional system error code.

Each specific exception should be a subclass of `ts::Exception`. Instead of rewriting the subclass code, applications should use the macro `TS_DECLARE_EXCEPTION()`.

2.2.1.8. Pseudo-enumeration data

An instance of the class `ts::Enumeration` associates a list of integer or `enum` values with strings. It can be used to display meaningful strings instead of integer values.

It is even more useful to decode command line arguments. When an option accepts a predefined list of values, the input string can be either an integer value or a name. When it is a name, it can even be abbreviated as long as it is not ambiguous in the corresponding `ts::Enumeration`. This is transparent for the application which receives the corresponding integer value.

2.2.1.9. Command-line arguments

The class `ts::Args` implements a generic handling of command line arguments.

Each application typically defines its own subclass of `ts::Args`. A plugin is always a subclass of `ts::Args`, through the intermediate class `ts::Plugin`.

A subclass of `ts::Args` defines the command line syntax and the corresponding help text. The superclass `ts::Args` automatically parses the command line, reports errors and handle common options such as `--help` or `--version`.

The value of command line options can be free strings, integer values or enumeration values. Integer values are recognized in decimal or hexadecimal form (prefix `0x`) and thousands separators (‘,’) which are present for clarity are ignored. Enumeration values are handled through `ts::Enumeration`.

2.2.1.10. XML data

The TSDuck library embeds an XML parser and several classes to handle a DOM structure.

See the class `ts::xml::Node`, the abstract base class of the DOM hierarchy.

2.2.1.11. JSON data

The TSDuck library embeds a JSON parser and several classes to handle JSON values.

See the class `ts::json::Value`, the abstract base class of the JSON hierarchy.

2.2.2. Cryptography

The TSDuck library contains a few cryptographic classes. The TSDuck library is **not** a cryptographic library and will never be. Cryptography is a serious matter which should be left to cryptographers.

Some transport stream processing operations require some cryptography, essentially block ciphers and hash functions. The TSDuck library proposes an homogeneous API over them. Standard cryptographic primitives are implemented using the standard system libraries, *OpenSSL* on UNIX systems (Linux, macOS, BSD), *BCrypt* on Windows. Less standard primitives are directly implemented in TSDuck.

The abstract class `ts::BlockCipher` is the root of a hierarchy of symmetric cryptography classes, including chaining modes. The main block cipher classes are `ts::AES128`, `ts::AES256`, `ts::TDES` and `ts::DES`.

DES is an obsolete and insecure algorithm. TDES (a.k.a. 3-DES or Triple DES) is also deprecated. However, the two are still used in some ATSC Digital TV systems.

Chaining modes are template classes which inherit from the abstract class `ts::CipherChaining`. The template parameter is a block cipher class. The main chaining modes are `ts::ECB`, `ts::CBC`, various flavors of `ts::CTSx` or more exotic modes from the DTV world such as `ts::DVS042`.

Additionally, `ts::CipherChaining` is also a subclass of `ts::BlockCipher` because it remains a symmetric cipher. So, ciphers like `ts::AES` or `ts::CBC<ts::AES>` can be used through the same `ts::BlockCipher` interface.

The class `ts::Scrambling` implements DVB-CSA-2, the Digital Video Broadcasting Common Scrambling Algorithm. This implementation is older than the open-source libdvbcsa library and is probably less efficient.

The abstract class `ts::Hash` is the root of a hierarchy of hash functions classes. The main hash functions are `ts::SHA1`, `ts::SHA256` or `ts::SHA512`.

The abstract class `ts::RandomGenerator` is the root of pseudo-random generators.

The subclass `ts::SystemRandomGenerator` is a portable interface to the system-provided PRNG. Usually, this is not the best PRNG on earth, but it is fine for most usages in TSDuck applications. For more critical usages (such as encryption key generation), use `ts::BetterSystemRandomGenerator`. This PRNG class uses `ts::SystemRandomGenerator` with an additional security layer.

The class `ts::Xoshiro256ss` implements the Xoshiro256** PRNG. It is a fast and deterministic PRNG, with a low level of security. The same seed will always produce the same pseudo-random sequence. It can be used in cases where many random numbers are required, without strong security criteria. It is typically used in fuzzing tools.

2.2.3. Operating system features

2.2.3.1. Miscelleaneous system utilities

The header files `tsSysUtils.h`, `tsFileUtils.h`, `tsEnvironment.h`, declare utility functions on top of the operating system.

With the introduction of C++17, many of these functions have been removed in favor of new standard functions. However, a number of additional features manipulate:

- File paths.
- File attributes.
- Creating or deleting files and directories.
- Environment variables.
- Process identifiers.
- System error codes.

2.2.3.2. Time

The class `ts::Time` is a portable implementation of time (both local and UTC time).

Many operations are provided, such as:

- Getting system time in various forms.
- Arithmetic operations on time.
- Analysing and building time values.
- Formatting time values as strings.

2.2.3.3. Multithreading

TSDuck is heavily multi-threaded. The abstract class `ts::Thread` manages a thread. To define an actual thread, derive this class and implement the virtual method `main()`.

The class `ts::ThreadAttributes` contains all mandatory or optional attributes of a thead. An application typically builds a `ts::ThreadAttributes` object and then creates threads using these attributes.

In earlier versions of TSDuck, synchronization primitives used to be implemented through specific classes (`ts::Mutex`, `ts::Condition`). They are now removed and new C++11 classes such as `std::mutex` and `std::condition_variable` are used instead.

Note that the C++11 class `std::thread` is not used. Its API is too limited to be useable in complex environments: it does not allow to customize the priority or the stack size *before* the creation of the thread. Therefore, TSDuck exclusively uses `ts::Thread` and `ts::ThreadAttributes` instead.

TSDuck relies on C++ mechanisms to track the usage of resources. Standard classes such as `std::lock_guard` or `std::unique_lock` are used to ensure that no dangling lock is lost through the *guard design pattern*.

2.2.3.4. Virtual memory

The class `ts::ResidentBuffer` implements a buffer which is locked in physical memory, preventing paging or swapping on this buffer. This is useful for large data buffers with high performance constraints.

This is a template class. The template parameter is the type of the elementary data in the buffer.

The core data of the `tsp` processor is a `ts::ResidentBuffer<ts::TSPacket>`. The incoming packets are directly written into this buffer by the input plugin. Each packet processing plugin directly reads and writes the packets here. And the output plugin reads the packet there, at the very same place they were written by the input plugin. Given that this global buffer is locked in physical memory, the best performances are guaranteed.

Note however that most operating systems require that the application has privileges to lock physical memory.

2.2.3.5. Processes

To track potential memory leaks and the impact of the application on the system, the class `ts::SystemMonitor` creates a background thread which reports the process metrics of the application at regular intervals.

The class `ts::ForkPipe` is a portable and convenient way to create a process running a specific command and creates an outgoing pipe from the calling application to the standard input of the created process. The pipe is open in binary mode (when it makes sense for the operating system) and can be used to pass an entire transport stream when necessary.

2.2.3.6. Networking

The classes `ts::IPv4Address` and `ts::IPv4SocketAddress` define an IPv4 address and a corresponding socket address (an IPv4 address and a port number). Host name resolution and multicast are supported.

Equivalent classes exist for IPv6 and MAC (Ethernet addresses).

The classes `ts::TCPSocket` and `ts::UDPSocket` implement TCP/IP and UDP/IP endpoints.

The class `ts::UDPSocket` can be used directly to send and receive datagrams. Multicast is supported.

The class `ts::TCPSocket` can be used only through two subclasses. The subclass `ts::TCPConnection` is a TCP/IP communication endpoint, either on client or server side. It is used to send or receive data streams. The subclass `ts::TCPServer` is used to implement a TCP server. It accepts incoming client connections and initiates a `ts::TCPConnection` for each new connection. On the client side, the class `ts::TCPConnection` is directly used to connect to the server.

Subclasses of `ts::TCPConnection` are used to implement specific protocols on top of TCP/IP. Currently, the available subclasses are `ts::TelnetConnection` and `ts::tlv::Connection`. The latter is used to handle communications using the "DVB SimulCrypt head-end protocols". See section 2.2.6 for more details.

The class `ts::WebRequest` performs simple Web requests using HTTP, HTTPS or FTP. Using a URL, the result can be downloaded in memory or in a file. Multiple redirections and SSL/TLS are automatically handled. This class is built on top of native system libraries, *libcurl* on UNIX systems (Linux, macOS, BSD), *WinInet* on Windows.

2.2.3.7. Shared libraries

The TSDuck library contains classes to load shared libraries (`.dll` on Windows, `.so` on Linux and BSD, `.dylib` on macOS) and lookup symbols inside them in a portable way. These classes are typically used to load `tsp` plugins but can be used in any application.

The class `ts::SharedLibrary` manipulates any type of shared library.

The subclass `ts::ApplicationSharedLibrary` searches a shared library using TSDuck rules: if the file is not found "as it is", an optional prefix and a list of directories are used. This is how, on Windows for instance, searching the shared library named `zap` will end up loading the file `tsplugin_zap.dll` in the same directory as the application executable file.

2.2.3.8. Smart-card interface

Applications which interact with smart-cards shall use the PC/SC interface. PC/SC is a standard interface which was originally developped for Windows but which is also available on Linux and macOS.

The TSDuck library does not embed or hide PC/SC but it provides a few utilities like transmitting an APDU and read the response in one single function or searching a smart-card with some characteristics in the ATR from all connected smart-cards.

All these utilities are grouped in the namespace `ts::pcsc`.

2.2.3.9. Windows specificities

The class `ts::COM` provides a portable and reliable way to make sure that the Common Object Model (COM) is properly initialized and terminated on Windows systems. This class is defined on all platforms but does nothing on non-Windows systems. It is consequently safe to use it everywhere without tedious conditional compilation directives.

Other classes manipulate Windows-specific objects and are not available on non-Windows systems.

The template class `ts::ComPtr` is the equivalent of a smart pointer for COM objects. The reference count of a COM object is properly incremented and decremented when the COM object is manipulated through a `ts::ComPtr`. The COM object is automatically released when no more reference exists.

There is little advantage to develop an intrinsicly non-portable COM object class. However, in order to access tuner devices, TSDuck needed a few custom internal COM classes to interact with the DirectShow framework. These internal classes needed some COM support functions which are available to applications (just in case…)

2.2.4. MPEG features

2.2.4.1. Transport streams

The class `ts::TSPacket` defines a transport stream packet. It is in fact a flat structure which occupies exactly 188 bytes in memory. It is safe to use arrays or vectors of `ts::TSPacket`. The packets are guaranteed to be contiguous in memory.

The class `ts::TSPacket` also adds many operations on the TS packet to read or modify properties like the PID (type `ts::PID`), the continuity counters or deeper structures like PCR, DTS or PTS.

The class `ts::TransportStreamId` contains the identification of an MPEG/DVB transport stream.

The class `ts::Service` contains all possible properties of a DVB service. Not all properties need to be set at the same time. Each property can be individually set, cleared or queried.

Transport stream files are implemented by classes `ts::TSFileInput` and `ts::TSFileOutput`. They respectively read and write transport stream files with specific features such as repeating the reading of a part of the file.

The subclass `ts::TSFileInputBuffered` provides additional, but limited, capabilities to seek forward and backward on non-seekable files such as pipes.

The subclass `ts::TSFileOutputResync` adds resynchronization capabilities on continuity counters and PID's.

The class `ts::TSAnalyzer` consumes all TS packets from a transport stream and analyzes virtually everything from the stream. This is the class which is used by the command `tsanalyze` and the plugin `analyze` to collect the vast amount of information it reports.

The class `ts::PCRAnalyzer` is a useful tool to evaluate the bitrate of a transport stream. It performs the analysis of the Program Clock Reference (PCR) which are present in the transport stream in order to evaluate the bitrate of the stream. If PCR are not found, the class can also use Decoding Time Stamps (DTS) to evaluate the bitrate. This is less precise than PCR but can be used as a backup.

2.2.4.2. Audio, video and PES packets

The TSDuck library provides classes to manipulate PES packets and a few audio and video attributes. These features are limited to the analysis of a transport stream. There is no video or audio decoding features. Specialized libraries exist for this and are out of scope for TSDuck.

The class `ts::PESPacket` implements a PES packet and can manipulate its attributes, header and payload.

The class `ts::PESDemux` extracts PES packets from a transport stream. It can also notify the application of the changes in audio or video attributes.

The abstract class `ts::AbstractAudioVideoAttributes` is the root of a hierarchy of classes which contains attributes for audio or video streams. Currently, specialized classes exist for MPEG-2 video, AVC/H.264, HEVC/H.265, VVC/H.266 video, MPEG-2 audio and AC-3 audio.

The class `ts::AVCParser` performs the parsing of an AVC, HEVC, or VVC bitstream.

2.2.5. Signalization

The MPEG signalization is built from sections, tables and descriptors. All these concepts are implemented in the TSDuck library.

2.2.5.1. Binary, specialized and XML formats

Signalization objects, sections, tables and descriptors, can be manipulated in several formats: binary objects, specialized classes and XML.

Tables in JSON format are also supported through automatic XML-to-JSON translation.

The classes `ts::Section`, `ts::BinaryTable` and `ts::Descriptor` implement binary forms of the signalization objects.

A binary table are made of a collection of sections. A binary table is valid when all binary sections are present. Each section contains its section number in the table and the total expected number of sections inside the table.

All sections and descriptors can be represented by the classes `ts::Section` and `ts::Descriptor`. They simply contain the complete binary content of the object and can manipulate the various components. An instance of `ts::Section` stores the *table_id* and manipulates the various components of the section header. For long sections, the final CRC32 can be checked for consistency or recomputed after modification of the section content.

Tables can be stored in binary files. The format of these files is quite simple. They just contain raw binary sections, without any encapsulation. Tables can also be stored in XML or JSON files. The class `ts::SectionFile` reads and writes tables or section from files, independently of the format, either a binary section file or an XML file.

Tables and descriptors can also be manipulated using specialized classes such as `ts::PAT` or `ts::PMT` for tables and `ts::ContentDescriptor` or `ts::ShortEventDescriptor` for descriptors.

All specialized classes inherit from a common abstract root named `ts::AbstractSignalization`. All descriptors inherit from the intemediate class `ts::AbstractDescriptor`. All tables inherit from the intemediate class `ts::AbstractTable`. Tables with long sections inherit from `ts::AbstractLongTable`.

Most tables and descriptors are implemented, from MPEG, DVB, ATSC, ISDB and a few private descriptors. Unimplemented descriptors shall be manipulated in binary form (or be implemented…)

Binary tables or descriptors are converted from or to specialized classes using `serialize()` and `deserialize()` methods. The validity of a binary or specialized object can be checked using the `isValid()` method.

Sample deserialization code:

```
void someFunction(ts::DuckContext& duck, const ts::BinaryTable& table)
{
    ts::PMT pmt;
    if (table.isValid() && table.tableId() == ts::TID_PMT) {
        pmt.deserialize(duck, table);
        if (pmt.isValid()) {
            processPMT(pmt);
        }
    }
}
```

The deserialization can also be done in the constructor. And the validity and *table_id* checking is done anyway in the deserialization. So, the previous code can be simplified as:

```
void someFunction(ts::DuckContext& duck, const ts::BinaryTable& table)
{
    ts::PMT pmt(duck, table);
    if (pmt.isValid()) {
        processPMT(pmt);
    }
}
```

Sample serialization:

```
ts::DuckContext duck;

ts::PMT pmt;
pmt.version = 12;
pmt.service_id = 0x1234;
// Declare one component, PID 0x345, carrying H.264/AVC video.
pmt.streams[0x345].stream_type = ts::ST_AVC_AUDIO;

ts::BinaryTable table;
pmt.serialize(duck, table);
```

Note that an instance of the class `ts::DuckContext` can store various information about the way to interpret incorrect signalization or preferences. Its default value is appropriate for a standard PSI/SI processing.

Each time the instance of `ts::DuckContext` is used, it accumulates information. For instance, if it is used to deserialize an ATSC MGT table, the information that the TS is an ATSC one is retained. Later, if the same instance of `ts::DuckContext` is used to deserialize a descriptor for which there is an ambiguity (the tag is used in two standards for instance), the ATSC version of the descriptor will be used.

It is also possible to automatically define and load command line options to preset the state of the instance of `ts::DuckContext`. See section 2.2.5.3 for more details.

Finally, specialized classes for tables and descriptors can be converted to and from XML using the methods `toXML()` and `fromXML()`.

These methods are typically used by the class `ts::SectionFile` which represents a file containing sections and tables in binary or XML format. The class can be used to load a set of tables in XML format or to store table objects in XML format.

The class `ts::SectionFile` is the core of the `tstabcomp` utility, the tables compiler (or decompiler).

2.2.5.2. Demux and packetization

Signalization objects can be extracted from transport streams using the class `ts::SectionDemux` and inserted back into transport streams using the class `ts::Packetizer`. These two classes also have specialized subclasses.

An instance of `ts::SectionDemux` can extract sections or complete tables in binary form.

Tables with long sections are usually cycled. A given table with a given version number and a given *table id extension* is reported only once, after collecting all its sections. The same table will be reported again only when its version number changes.

On the contrary, short tables are all reported since they do not implement versioning.

It is also possible to use a `ts::SectionDemux` to be notified of all individual sections.

2.2.5.3. Application preferences contexts

The class `ts::DuckContext` carries various preferences about the standards or localizations. Typically, each application has a given context. Using `tsp`, each plugin has it own context.

The preferences which are carried by a context include the default standard (DVB, ATSC, ISDB), the default character sets in PSI/SI, the default private data specifier (for DVB private descriptors), the HF region (for terrestrial or satellite frequency mapping)

The `ts::DuckContext` class can automatically define command-line arguments to explicitly specify preferences (options `--atsc` or `--default-charset` for instance). Thus, the preferences are setup from the beginning.

But preferences are also accumulated all along the execution. For instance, as soon as an ATSC table is demuxed, the fact that the transport stream contains ATSC data is stored in the context. Later, when an MPEG table (a PMT for instance) contains an ambiguous descriptor tag which is used by DVB and ATSC, then the ATSC alternative will be used.

2.2.6. DVB SimulCrypt protocols

The communications inside a DVB SimulCrypt head-end is defined by the standard ETSI TS 103 197, "Head-end implementation of DVB SimulCrypt".

Most of these protocols use the same principles. They use binary TLV (Tag/Length/Value) messages, asynchronous communications, concepts of *channels*, *streams*, status and error messages.

The generic handling of these messages is implemented by classes in the namespace `ts::tlv`. All TLV messages inherit from `ts::tlv::Message`. Channel-level messages inherit from `ts::tlv::ChannelMessage` and stream-level messages inherit from `ts::tlv::StreamMessage`.

The syntax of a given protocol is defined by subclassing `ts::tlv::Protocol`.

Currently, the TSDuck library implements the following protocols:

- ECMG□SCS in namespace `ts::ecmgscs`.
- EMMG/PDG□MUX in namespace `ts::emmgmux`.

2.2.7. Conditional access systems

The class `ts::CASMapper` analyzes the signalization of a transport stream, locates ECM and EMM stream and associates each of them with a *CA_System_Id*.

An instance of `ts::CASMapper` can then be queried for ECM, EMM streams or CAS vendors.

2.2.8. Other forms of demux

We have already mentioned the classes `ts::SectionDemux` and `ts::PESDemux`. Other specialized forms of demux can be implemented.

The class `ts::T2MIDemux` demuxes T2-MI (DVB-T2 Modulator Interface) packets and extracts encapsulated transport streams. Similarly, the class `ts::TeletextDemux` extracts Teletext subtitles from TS packets.

Since all forms of demux share a number of properties, they all inherit from a root abstract class named `ts::AbstractDemux`.

2.2.9. Digital TV tuners

The class `ts::Tuner` interfaces DVB/ATSC/ISDB tuner devices in a portable way. This is quite a challenge since Linux and Windows use very different tuner frameworks. Some very-specific features are available either only on Linux or Windows.

The abstract class `ts::TunerParameters` is the root of a hierarchy of classes containing tuning parameters. Subclasses exist for DVB-S, DVB-T, DVB-C and ATSC. ISDB-S and ISDB-T are currently unsupported.

The class `ts::TSScanner` reads a TS from a `ts::Tuner` until all scanning information is found, typically until the PAT, NIT and SDT are received. This is the basis for scanning a DTV network.

Note that tuner devices are supported on Linux and Windows only. On macOS, the above classes are defined but return "unimplemented" errors when used.

2.2.10. Interface to Dektec devices

TSDuck can manipulate ASI and (de)modulator devices from Dektec. The TSDuck library includes the DTAPI library, a proprietary C++ interface which is provided by Dektec. The DTAPI is not available in source form and not part of the TSDuck source repository. However, when TSDuck is built, the DTAPI is downloaded in binary from Dektec and included in the TSDuck library.

Such a packaging is authorized by the DTAPI license (see the file `OTHERS.txt` in the TSDuck source repository or installation tree).

An application should not directly call the DTAPI. In practice, this works on Linux but not on Windows. So if you want portability, do not do this. The reason is that the structure of Windows DLL's is such that exported code from a DLL must be compiled using specific attributes. But the DTAPI, as provided by Dektec, was not compiled with these attributes. So, when the DTAPI is included in `tsduck.dll`, the DTAPI can be called from inside `tsduck.dll` but is not accessible from the application.

This is why accessing the DTAPI from the application must be done through some TSDuck proxy class. The classes `ts::DektecControl`, `ts::DektecInputPlugin` and `ts::DektecOutputPlugin` provide the features which are required by the utility `tsdektec` and the plugin `dektec`. They can be used by third-party applications.

Note that Dektec devices are supported on Linux and Windows only. On macOS, the above classes are defined but return "unimplemented" errors when used.

2.3. Java and Python bindings

2.3.1. Overview

Starting with version 3.25, TSDuck includes Java and Python bindings to some high-level features.

Although subject to enhancements, these bindings will never aim at supporting the full TSDuck feature set since this would be too large. Only a small subset of TSDuck high-level features are targeted.

The Java classes are documented in the Java bindings reference section.

The Python classes are documented in the Python bindings reference section.

Sample Java and Python applications are available in the TSDuck source tree.

Currently, the TSDuck Java and Python bindings provide access to the features in the following table. Equivalences are provided between C++, Java, Python and command line tools.

The first three classes implement high-level features which have direct counterparts as command line tools. The others are support classes which are only required to use the high-level classes.

Table 4. Equivalence between commands, C++, Java, Python classes

Command	C++ class	Java class	Python class
`tsp`	`ts::TSProcessor`	`io.tsduck.TSProcessor`	`tsduck.TSProcessor`
`tsswitch`	`ts::InputSwitcher`	`io.tsduck.InputSwitcher`	`tsduck.InputSwitcher`
`tstabcomp`	`ts::SectionFile`	`io.tsduck.SectionFile`	`tsduck.SectionFile`
n/a	`ts::DuckContext`	`io.tsduck.DuckContext`	`tsduck.DuckContext`
n/a	`ts::Report`	`io.tsduck.AbstractSyncReport`	`tsduck.AbstractSyncReport`
n/a	`ts::AsyncReport`	`io.tsduck.AbstractAsyncReport`	`tsduck.AbstractAsyncReport`
n/a	`ts::SystemMonitor`	`io.tsduck.SystemMonitor`	`tsduck.SystemMonitor`
n/a	`ts::PluginEventHandlerInterface`	`io.tsduck.AbstractPluginEventHandler`	`tsduck.AbstractPluginEventHandler`
n/a	`ts::PluginEventContext`	`io.tsduck.PluginEventContext`	`tsduck.PluginEventContext`

2.3.2. Support classes

2.3.2.1. TSDuck execution context

The `DuckContext` class is used to define and accumulate regional or operator preferences. In the TSDuck C++ programming guide, it is referred to as *TSDuck execution context*. Most of the time, using the default state of a new instance is sufficient.

The application *sample Japanese tables*, available in Java and Python, demonstrates how it can be necessary to override the defaults in specific cases.

2.3.2.2. Reporting classes

The reporting classes (`ts::Report` C++ class hierarchy) are used to report logs, errors and debug. They are consistently used all over TSDuck and are required to use the high level features. There is a large hierarchy of classes in the three languages which can be classified according to two sets of criteria:

- Synchronous vs. asynchronous:

- Synchronous report classes log messages in the same thread as the caller. They are usually not thread-safe.
- Asynchronous report classes, on the other hand, can be used in a multi-threaded environment and the actual message logging (such as writing in a log file) is performed in a separate thread. As a consequence, an asynchronous report instance must be explicitly *terminated*. An asynchronous report class is required when using heavily multi-threaded classes such as `TSProcessor` or `InputSwitcher`.

- Native vs. abstract:
  - Native classes are the C++ classes which are used in all the TSDuck command line tools. They are typically used to report to standard output, standard error, files or dropping the logs. They can be used from Java and Python directly but cannot be derived or customized. They are typically used when predefined error logging is sufficient.
  - Abstract classes are pure Java or Python base classes which are designed to be derived in applications. Such application-defined classes shall override the method `logMessageHandler` (Java) or `log` (Python) to intercept and process the message lines.

The asynchronous abstract classes can be useful to collect events, tables and sections in XML, JSON or binary / hexadecimal form in Java or Python applications when using `TSProcessor` or `InputSwitcher`. Some of the sample Java and Python applications illustrate this mechanism.

Table 5. Categories of report classes in C++, Java, Python

Category	C++ class	Java class	Python class
Synchronous, native	`ts::CerrReport`	`io.tsduck.ErrReport`	`tsduck.StdErrReport`
	`ts::NullReport`	`io.tsduck.NullReport`	`tsduck.NullReport`
Asynchronous, native	`ts::AsyncReport`	`io.tsduck.AsyncReport`	`tsduck.AsyncReport`
Synchronous, abstract	`ts::Report`	`io.tsduck.AbstractSyncReport`	`tsduck.AbstractSyncReport`
Asynchronous, abstract	`ts::AsyncReport`	`io.tsduck.AbstractAsyncReport`	`tsduck.AbstractAsyncReport`

2.3.2.3. Resource monitoring

The `SystemMonitor` class is available in all languages, C++, Java and Python. It can be used at the top-level of an application to implement the `--monitor` option as found in `tsp` and `tsswitch`. An instance of a thread-safe `Report` class is used to report monitoring messages.

The `SystemMonitor` class is very simple to use. Examples are available in Java and Python.

2.3.2.4. Plugin events

For developers, TSDuck plugins can *signal events* which can be handled by the application. Each event is signalled with a user-defined 32-bit *event code*. An application can register *event handlers* in the `ts::TSProcessor` instance (see the class `ts::PluginEventHandlerRegistry`, knowing that `ts::TSProcessor` is a subclass of `ts::PluginEventHandlerRegistry`). The event handler registration can include various *selection criteria* such as event code value or originating plugin (see the inner class `ts::PluginEventHandlerRegistry::Criteria`).

C++ developers who create their own plugins can signal any kind of event that they later handle in their application. This is illustrated in a C++ sample custom application. In this sample code, everything is customized in the application: the plugin, the event it signals, the associated event data, the application handling of the event.

Since developing a TSDuck plugin is only possible in C++, Java and Python developers have more limited options. Some standard TSDuck plugins such as `tables`, `psi` or `mpe` provide the option `--event-code`. Using this option, the plugins signal event using the specified event code for each data they handle (sections or MPE datagrams depending on the plugin).

Java and Python applications can derive from class `AbstractPluginEventHandler` to define and register their own event handlers. Thus, binary sections or MPE datagrams can be handled directly from the plugin to the Java or Python application.

Some plugins are even dedicated to application developers and are useless on `tsp` command lines. This is the case of the plugin `memory` (both an input and an output plugin). This plugin, when used in a `TSProcessor` instance, performs direct transport stream input and output from and to the application using memory buffers. The memory buffers are signalled using plugin events. The `memory` input plugin is an example of an application-defined event handler returning data to the plugin. See this sample code in the TSDuck source code tree.

2.3.3. Application/plugin communication in Java or Python

At high level, Java and Python applications can only run `TSProcessor` or `InputSwitcher` sessions, just like a shell-script would do with commands `tsp` and `tsswitch`.

The communication from the Java and Python applications to the plugins is performed using plugin options. These options may contain file names or UDP ports which can be created by the application.

More effectively, most file contents can be provided directly on the command line, avoiding the burden of creating temporary files. For instance, wherever an input XML file name is expected, it is possible to use the XML content instead. Any "XML file name" which starts with `<?xml` is considered as inline XML content. Similarly, if an input "JSON file name" starts with `{` or `[`, it is considered as inline JSON content.

On reverse side, there is some limited form of communication from the plugins to the Java or Python application. There are basically two ways to handle plugin information in the application: the logging system and plugin events.

Using the logging system:

Some plugins support options such as `--log-xml-line`, `--log-json-line` or `--log-hexa-line`. With these options, the extracted data (table, section, MPE datagram) are "displayed" as one single line in the designated format on the logging system. Using user-defined Java or Python asynchronous abstract reporting classes, the application receives all logged lines and can filter and manipulate the data which were extracted and logged by the plugins.

Using plugin events:

Some plugins support the option `--event-code`. With this option, the extracted data are *signalled* by the plugin as an event. Using and registering user-defined Java or Python plugin event handlers, the application is directly notified of the data.

Which mechanism, logging system or plugin events, should be used depends on the application.

- Logging system:
  - Pros:
    - The log lines are asynchronously processed in the context of the low-priority logging thread. Any lengthy processing in the Java or Python application does not hurt the dynamics of the plugins.
  - Cons:
    - If the application needs to process binary data, the additional serialization process in the log line adds some useless overhead.
    - Because the logging system is non-intrusive by design, log messages may be lost if there are more messages than the logging thread can process without making plugin threads wait. This can be mitigated using the *synchronous log* option in the `AbstractAsyncReport` consttructor.
- Plugin events:
  - Pros:
    - The binary data are directly passed from the plugin to the application without any serialization,

logging or multi-threading overhead.

- Cons:
    - The application-defined event handlers execute in the context of the plugin thread. Any lengthy processing at this stage slows down the plugin.

The following sample applications can be used as a starting point:

Table 6. Sample Java and Python communication applications

Communication type	Java	Python
Logging (XML)	SampleAnalyzeSDT	sample-analyze-sdt.py
Logging (JSON)	SampleAnalyzeTS	sample-analyze-ts.py
Logging (bin/hexa)	SampleFilterTablesLog	sample-filter-tables-log.py
Plugin events (sections)	SampleFilterTablesEvent	sample-filter-tables-event.py
Plugin events (MPE datagrams)	SampleMPE	sample-mpe.py
Plugin events (input/output)	SampleMemoryPlugins	sample-memory-plugins.py

2.3.4. Using TSDuck Java bindings

All TSDuck Java classes are defined in a package named `io.tsduck`.

A few examples are provided in the directory `sample/sample-java` in the TSDuck source code package.

2.3.4.1. Linux

The TSDuck Java bindings are installed with TSDuck in `/usr/share/tsduck/java`. All classes are in a JAR file named `tsduck.jar`. Simply add this JAR in the environment variable `CLASSPATH` to use TSDuck from any Java application:

```
$ export CLASSPATH="/usr/share/tsduck/java/tsduck.jar:$CLASSPATH"
```

2.3.4.2. macOS

This is similar to Linux, except that instead of `/usr/share`, use `/usr/local/share` (Intel Macs) or `/opt/homebrew/share` (Apple Silicon Macs).

```
$ export CLASSPATH="/usr/local/share/tsduck/java/tsduck.jar:$CLASSPATH"
$ export CLASSPATH="/opt/homebrew/share/tsduck/java/tsduck.jar:$CLASSPATH"
```

2.3.4.3. Windows

On Windows, Java bindings are optional components of the TSDuck installer. When they are selected for installation, they are installed in the TSDuck area and the environment variable `CLASSPATH` is modified at system level to include the JAR file of the TSDuck Java bindings.

Thus, any Java program can use TSDuck directly.

2.3.5. Using TSDuck Python bindings

All TSDuck bindings are defined in a module named `tsduck`. All Python programs using TSDuck shall consequently start with:

```
import tsduck
```

A few examples are provided in the directory `sample/sample-python` in the TSDuck source code package.

2.3.5.1. Linux

The Python bindings are installed with TSDuck in `/usr/share/tsduck/python`. Simply add this directory in the environment variable `PYTHONPATH` to use TSDuck from any Python application:

```
$ export PYTHONPATH="/usr/share/tsduck/python:$PYTHONPATH"
```

2.3.5.2. macOS

This is similar to Linux, except that instead of `/usr/share`, use `/usr/local/share` (Intel Macs) or `/opt/homebrew/share` (Apple Silicon Macs).

```
$ export PYTHONPATH="/usr/local/share/tsduck/python:$PYTHONPATH"
$ export PYTHONPATH="/opt/homebrew/share/tsduck/python:$PYTHONPATH"
```

2.3.5.3. Windows

On Windows, Python bindings are optional components of the TSDuck installer. When they are selected for installation, they are installed in the TSDuck area and the environment variable `PYTHONPATH` is modified at system level to include the root directory of the TSDuck Python bindings.

Thus, any Python program can use TSDuck directly.

2.3.5.4. Python prerequisites

The code was initially tested with Python 3.7 and higher. Python 2.x is not supported. Intermediate versions may work but without guarantee.

2.3.5.5. Implementation notes

There are usually two ways to call C/C++ from Python:

- Using the predefined `ctypes` Python module to call C functions,
- Implementating a full native Python module in C/C++.

The second option is usually more flexible and more generic. However, the generated binary depends on the version of Python. If such an option is used, the binary installation of TSDuck would require a specific version of Python (or a specific set of versions of it). But each system has it own requirements on Python and it is difficult for a product like TSDuck to impose a specific version of Python.

Consequently, the less flexible `ctypes` approach was chosen. The TSDuck binary library contains C++ wrapper functions to some features of TSDuck and these carefully crafted functions are directly called from Python code using `ctypes`, regardless of the version of Python. Note, however, that these C++ functions are hidden inside the Python bindings and are invisible to the C++ application developer.

2.4. Developing a TSDuck plugin

2.4.1. Plugin development workflow

When some new kind of transport stream processing is needed, several solutions are possible:

- First, check if an existing plugin or a combination of existing plugins can do the job.
- Check if an existing plugin can be extended (by adding new options for instance).
- As a last resort, develop a new plugin, which is relatively easy.

New plugins can be developed either as part of the TSDuck project or as independent third-party projects.

2.4.1.1. Developing independent third-party plugins

If you create your own third-party plugins (ie. if you are not a TSDuck maintainer), it is recommended to develop your plugins outside the TSDuck project.

Do not modify your own copy of the TSDuck project with your private plugins. This could create useless difficulties to upgrade with new versions of the project.

Consider developing your plugins in their own projects, outside TSDuck. You do not even need to get the full source code of TSDuck. It is sufficient to install the TSDuck development environment (see section 2.1).

An example of a third party plugin project is provided in the directory sample/sample-plugin.

2.4.1.2. Developing plugins for the TSDuck project

To develop a new plugin named `foo`, follow these steps:

- Create a source file named `tsplugin_foo.cpp` in the `tsplugins` subdirectory.
- On UNIX systems (Linux, macOS, BSD), this new source file will be automatically recognized by the Makefile and the new plugin will be built.
- On Windows systems, the plugin needs a "project file" for Visual Studio and MSBuild. This project file shall be referenced in the TSDuck "solution file".

The last step is automated using the Python script `scripts/build-project-files.py`. This script explores the source files for all commands and plugins. It automatically generates missing project files and references them in the solution file.

This script can be run on UNIX systems (Linux, macOS, BSD) or Windows systems. On Windows, it can be easier to launch the PowerShell script `scripts/build-project-files.ps1`, which simply calls the Python script.

2.4.2. Development guidelines

Don't write a plugin from scratch. Use an existing plugin as code base (beware however of the pitfalls of careless copy / paste). The simplest code bases can be found in the plugins `null` (input), `drop` (output) , `skip` (basic packet processing), `nitscan` (reading content of PSI/SI), `svrename` (modifying PSI/SI on the fly).

Always create plugins which perform simple and elementary processing. If your requirements can be divided into two independent processing, create two distinct plugins. The strength of TSDuck is the flexibility, that is to say the ability to combine elementary processing independently and in any order.

2.4.2.1. Class hierarchy

In the source file of the plugin, create a C++ class, derived from either `ts::InputPlugin`, `ts::OutputPlugin` or `ts::ProcessorPlugin`. If your plugin implements two capabilities (both input and output for instance), implement the corresponding two classes in the same source file.

See the class diagram of `ts::Plugin` in the reference programming documentation for a global view of the plugin classes.

Specialized plugins which manipulate exiting tables derive from `ts::AbstractTablePlugin`. Examples of such plugins are `pmt`, `pat`, `nit`, etc. The actual plugin subclasses focus on the modification of the target table while the superclass automatically handles demuxing, remuxing and creation of non-existing tables.

Specialized descrambling plugins derive from `ts::AbstractDescrambler`. This abstract class performs the generic functions of a descrambler: service location, ECM collection, descrambling of elementary streams. The concrete classes which derive from `ts::AbstractDescrambler` must perform CAS-specific operations: ECM streams filtering, ECM deciphering, control words extraction. Most of the time, these concrete classes must interact with a smartcard reader containing a smartcard for the specific CAS.

2.4.2.2. Invoking tsp from a plugin, the ts::TSP callbacks

In its constructor, each plugin receives an associated `ts::TSP` object to communicate with the `tsp` main executable. This instance of `ts::TSP` is a protected field named `tsp` which can be freely accessed by the code of the plugin.

A plugin shared library must exclusively use that `tsp` object for text display and must never use `std::cout`, `printf` or the like. The class `ts::TSP` is a subclass of `ts::Report` and supports all reporting methods such as `info()`, `verbose()`, `error()`, `debug()`, etc.

When called in a multi-threaded context, the supplied `tsp` object is thread-safe and asynchronous (the methods return to the caller without waiting for the message to be printed).

Note that the plugin instance is also a subclass of `ts::Report` and automatically redirects all messages to its `tsp` field. Therefore, the code of the plugin can transparently use its own methods `info()`, `error()`, etc. This is equivalent to calling its `tsp`.

2.4.2.3. Joint termination support

A plugin can decide to terminate `tsp` on its own (returning end of input, output error or `ts::ProcessorPlugin::TSP_END`). The termination is unconditional, regardless of the state of the other plugins. Thus, if several plugins have termination conditions, `tsp` stops when the first plugin decides to terminate. In other words, there is an "or" operator between the various termination conditions.

The idea behind joint termination is to terminate `tsp` when several plugins have jointly terminated their processing. If several plugins have a "joint termination" condition, `tsp` stops when the last plugin triggers the joint termination condition. In other words, there is an "and" operator between the various joint termination conditions.

First, a plugin must decide to use joint termination. This is usually done in method `ts::Plugin::start()`, using `ts::TSP::useJointTermination(bool)` when the option `--joint-termination` is specified on the command line.

Then, when the plugin has completed its work, it reports this using `ts::TSP::jointTerminate()`. After invoking this method, any packet which is processed by the plugin may be ignored by `tsp`.

2.5. Developing a TSDuck extension

Applications or `tsp` plugins can be developed on their own. But it is also possible to develop fully integrated *extensions* to TSDuck.

An extension not only adds new plugins and commands, it can also augment the features of standard TSDuck commands and plugins. An extension can also be packaged as a binary installer which can be deployed on top of an existing installation of TSDuck.

The possible features of a TSDuck extension are:

- Handling third-party tables and descriptors. The new tables and descriptors can be manipulated in XML or

JSON, analyzed and displayed with the standard TSDuck tools.

- Handling third-party Conditional Access Systems, based on a range of *CA_system_id* values. The ECM's, EMM's and private parts of the *CA_descriptor* are correctly analyzed and displayed with the standard TSDuck tools.
- Adding filtering capabilities based on specific or private conditions on sections for command `tstables` and plugin `tables`.
- Additional plugins for `tsp`.
- Additional command-line utilities.

A complete example of a TSDuck extension is provided in the TSDuck source tree. This example also provides scripts to build standard installers (`.exe` on Windows, `.rpm` and `.deb` on Linux). The generated packages install the extension on top of a matching version of TSDuck.

2.5.1. Files in an extension

A TSDuck extension typically contains the following types of files:

- Additional utilities. They are executable files without predefined naming. They are installed in the same directory as the TSDuck commands.
- Additional `tsp` plugins. They are dynamic libraries named `tsplugin_XXX.so`, `.dylib` or `.dll`. The plugins are loaded by `tsp` when invoked by their names `XXX`.
- Extension shared libraries named `tslibext_XXX.so`, `.dylib` or `.dll`. All shareable libraries named `tslibext_XXX` in the same directory as the TSDuck binaries or in the path `TSPLUGINS_PATH` are automatically loaded when any TSDuck command is invoked (in fact any time the TSDuck library `tsduck.dll` or `libtsduck.so` or `.dylib` is used). Such libraries typically install hooks into the core of TSDuck to handle third-party signalization.
- XML files describing the XML models for third-party signalization (tables and descriptors). There is no mandatory naming template for those files but `tslibext_XXX.xml` is recommended. These XML files must be registered by the extension dynamic library (details below).
- Name files describing third party identifiers (table ids, descriptor tags, CA system id, stream types, etc.) These files are used by TSDuck to better identify the various entities. There is no mandatory naming template for those files but `tslibext_XXX.names` is recommended. These files must be registered by the extension dynamic library (details below).

2.5.2. The extension dynamic library

All shareable libraries named `tslibext_XXX.so`, `.dylib` or `.dll` are automatically loaded by any TSDuck command or plugin. The initialization of the library is responsible for registering various hooks which implement the additional features.

2.5.2.1. Identification of the extension

This is an optional but recommended step. One C++ module inside the `tslibext_XXX` library shall invoke the macro `TS_REGISTER_EXTENSION` as illustrated below:

```
TS_REGISTER_EXTENSION(u"foo",                       // extension name
                      u"Sample foo extension",      // short description
                      {u"foot", u"foobar"},         // list of provided plugins for tsp
                      {u"footool", u"foocmd"});     // list of provided command-line tools
```

Using this declaration, the extension is identified and listed using the command `tsversion --extensions`.

Without the declaration, the extension is loaded and functional but it is not identified.

2.5.2.2. Providing an XML model file for additional tables and descriptors

To analyze input XML files containing tables, TSDuck uses an XML model to validate the syntax of the input XML file. There is a predefined large XML file which describes all supported tables and descriptors.

An extension may provide additional smaller XML files which describe the new tables or descriptors. See the sample extension for more details. The XML files shall be installed in the same directory as the rest of the extension (and TSDuck in general).

For each additional XML file, there must be one C++ module inside the `tslibext_XXX` library which invokes the macro `TS_REGISTER_XML_FILE` as illustrated below:

```
TS_REGISTER_XML_FILE(u"tslibext_foo.xml");
```

2.5.2.3. Providing a names files for additional identifiers

The usage rules and conventions are identical to the XML file above. The declaration macro for each names file is `TS_REGISTER_NAMES_FILE` as illustrated below:

```
TS_REGISTER_NAMES_FILE(u"tslibext_foo.names");
```

Here is an example, from the sample "foo" extension, which defines additional names for a table, a descriptor and a range of *CA_system_id*.

```
[TableId]
0xF0 = FOOT

[DescriptorId]
0xE8 = Foo

[CASystemId]
0xF001-0xF008 = FooCAS
```

2.5.2.4. Providing support for additional tables

If your environment defines a third-party table which is unsupported or unknown in TSDuck, you can implement it in your extension library.

First, define the C++ class implementing the table:

```
class FooTable : public ts::AbstractLongTable { ... };
```

In the implementation of the table, register hooks for the various features you support. In this example, we register a C++ class for `FooTable`:

```
TS_REGISTER_TABLE(FooTable,                 // C++ class name
                  {0xF0},                   // table id 0xF0
                  ts::Standards::NONE,      // not defined in any standard
                  u"FOOT",                  // XML name is <FOOT>
                  FooTable::DisplaySection);
```

The last argument to `TS_REGISTER_TABLE` is a static method of the class which displays the content of a section of this table type.

The XML model for the table is included in the XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tsduck>
  <_tables>
    <FOOT version="uint5, default=0" current="bool, default=true" foo_id="uint16, required"
name="string, optional">
      <_any in="_descriptors"/>
    </FOOT>
  </_tables>
</tsduck>
```

for the following binary layout, using the same conventions as MPEG/DVB standards:

```
table_id                      8 bits   = 0xF0
section_syntax_indicator      1 bit    = '1'
reserved                      3 bits
section_length               12 bits
foo_id                       16 bits
reserved                      2 bits
version_number                5 bits
current_next_indicator        1 bit
section_number                8 bits
last_section_number           8 bits
name_length                   8 bits
for(i=0;i<N;i++){
    name_char                 8 bits
}
reserved_future_use           4 bits
descriptors_length           12 bits
for (i=0;i<N;i++){
    descriptor()
}
CRC_32
```

2.5.2.5. Providing support for additional descriptors

Similarly, it is possible to implement a third-party descriptor as follow:

```cpp
class FooDescriptor : public ts::AbstractDescriptor { ... };
```

In the implementation of the descriptor, we register hooks for the various features. Since this is a non-DVB descriptor with descriptor tag `0xE8`, greater than `0x80`, we must set the private data specifier to zero in the ts::EDID ("extended descriptor id").

```cpp
TS_REGISTER_DESCRIPTOR(FooDescriptor,                 // C++ class name
                       ts::EDID::Private(0xE8, 0),    // "extended" descriptor id
                       u"foo_descriptor",             // XML name is <foo_descriptor>
                       FooDescriptor::DisplayDescriptor);
```

The last argument to `TS_REGISTER_DESCRIPTOR` is a static method of the class which displays the content of a descriptor.

The XML model for the descriptor is included in the XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<tsduck>
  <_descriptors>
    <foo_descriptor name="string, required"/>
  </_descriptors>
</tsduck>
```

for the following binary layout:

```
descriptor_tag          8 bits = 0xE8
descriptor_length       8 bits
for(i=0;i<N;i++) {
    name_char           8 bits
}
```

2.5.2.6. Implementing advanced section filtering capabilities

The command `tstables` (and its plugin counterpart `tables`) can process vast amounts of tables. To extract specific tables or sections, the command provides filtering options such as `--pid`, `--tid` or `--tid-ext`.

For specific sections, it is possible to define additional filtering options to the `tstables` command.

The extension library shall provide a C++ class implementing `ts::TablesLoggerFilterInterface`. The sample `foo` extension provide an option `--foo-id` which selects instances of `FooTable` containing specific values for some `foo_id` field.

```
class FooFilter: public ts::TablesLoggerFilterInterface { ... };
```

See the documentation of `ts::TablesLoggerFilterInterface` for more details.

In the implementation of the class, we register it as a section filter for `tstables`:

```
TS_REGISTER_SECTION_FILTER(FooFilter);
```

2.5.2.7. Providing support for additional Conditional Access Systems

If you work with a specific Conditional Access System, you probably manipulate confidential information that cannot be published in an open-source tool such as TSDuck. The solution is to develop a private closed-source extension.

In the extension library, you may register functions to display the structure of the ECM's, EMM's or private part of the *CA_descriptor*. The registration is based on a range of *CA_system_id* (here the constants `CASID_FOO_MIN` and `CASID_FOO_MAX`).

```
// Display a FooCAS ECM on the output stream.
// Compatible with ts::DisplaySectionFunction profile.

void DisplayFooCASECM(ts::TablesDisplay& display, const ts::Section& section, int indent);

// Display a FooCAS EMM on the output stream.
// Compatible with ts::DisplaySectionFunction profile.

void DisplayFooCASEMM(ts::TablesDisplay& display, const ts::Section& section, int indent);

// Display the payload of a FooCAS ECM on the output stream as a one-line "log" message.
// Compatible with ts::LogSectionFunction profile.

ts::UString LogFooCASECM(const ts::Section& section, size_t max_bytes);

// Display the payload of a FooCAS EMM on the output stream as a one-line "log" message.
// Compatible with ts::LogSectionFunction profile.

ts::UString LogFooCASEMM(const ts::Section& section, size_t max_bytes);

// Display the private part of a FooCAS CA_descriptor on the output stream.
// Compatible with ts::DisplayCADescriptorFunction profile.

void DisplayFooCASCADescriptor(ts::TablesDisplay& display, const uint8_t* data, size_t size, int
indent, ts::TID tid);
```

See the documentation for `ts::DisplaySectionFunction`, `ts::LogSectionFunction` and `ts::DisplayCADescriptorFunction`.

To register the display handlers in TSDuck:

```
TS_REGISTER_SECTION({ts::TID_ECM_80, ts::TID_ECM_81},
                    ts::Standards::NONE,  // not defined in any standard
                    DisplayFooCASECM,     // display function
                    LogFooCASECM,         // one-line log function
                    {},                   // no predefined PID
                    CASID_FOO_MIN,        // range of CA_system_id
                    CASID_FOO_MAX);

TS_REGISTER_SECTION(ts::Range<ts::TID>(ts::TID_EMM_FIRST, ts::TID_EMM_LAST),
                    ts::Standards::NONE,  // not defined in any standard
                    DisplayFooCASEMM,     // display function
                    LogFooCASEMM,         // one-line log function
                    {},                   // no predefined PID
                    CASID_FOO_MIN,        // range of CA_system_id
                    CASID_FOO_MAX);

TS_REGISTER_CA_DESCRIPTOR(DisplayFooCASCADescriptor, CASID_FOO_MIN, CASID_FOO_MAX);
```

2.5.3. Building cross-platform binary installers for an extension

See the sample `foo` extension in the TSDuck source tree.

Scripts are provided to build `.exe` installers on Windows, `.rpm` and `.deb` packages on Linux.

To avoid unexpected issues, an extension is only compatible with the version of TSDuck it was compiled with. When you install a new version of TSDuck, make sure to rebuild your extension with the development environment of this specific version of TSDuck. Then, install the new version of the extension on top of the same new version of TSDuck.

Chapter 3. Contributing to TSDuck Development

TSDuck development is managed using Git. The reference repository is on GitHub.

Code contributions from external developers are welcome and will be reviewed (without guaranteed response time however). Contributions shall be submitted using pull requests on GitHub exclusively.

This chapter summarizes the main actions to help developers and integrators to work with pull requests. This is the minimal set of actions.

More details can be found on GitHub documentation. Several articles also describe the GitHub standard fork & pull request workflow. We specifically recommend this one.

3.1. Transparency of contributions

All commits in a Git pull request shall have a clear and transparent identification of the author. The author name shall be the true first and last names of the contributor. No pseudo or other forms or anonymity is allowed. Preferably (although not required), the author's e-mail should be a real address where the contributor can be contacted.

This requirement for transparency is not arbitrary. There is a reason for it. The world of Digital TV is roughly divided in industries, service providers, TV operators, and pirates. A technical toolbox such as TSDuck is useful to everyone, equally. But it must be clear to everyone that TSDuck is made by engineers for engineers. TSDuck shall remain a fully transparent project: open source, identified web sites, identified authors and contributors.

In the world of Pay-TV, anonymity equals piracy. This may seem unfair but this is the way it is perceived by the industry. So, to maintain the trust in TSDuck, let's keep anonymity away from it. We hope that all contributors understand this position.

3.2. Contributor workflow

3.2.1. Initial setup

The first requirement is to create a GitHub account, if you do not already have one.

Initially, create your own fork of the TSDuck repository. Go to the TSDuck reference repository and click on the "Fork" button.

Clone your GitHub forked repository on your development system. Use one of the two following commands.

```
$ git clone https://USERNAME@github.com/USERNAME/tsduck.git
$ git clone git@github.com:USERNAME/tsduck.git
```

In the first case, you will need to provide a GitHub personal access token each time you push to GitHub. In the second case, you need to first upload your SSH public key to GitHub and then simply push without password.

You may want to track more precisely the master branch of the reference repository. See more details in the above mentioned article.

3.2.2. Contributing code

To facilitate merging, each contribution should be provided in a specific branch. Let's call it `newfeature` here:

```
$ git branch newfeature
$ git checkout newfeature
```

Then, do your coding work.

If the contribution brings new features, be sure to document them in the TSDuck user's guide. The user's guide is made of text files in Asciidoc format in the directory `doc/user`.

New features and bug fixes should also be documented in the file `CHANGELOG.txt`.

When all modifications are ready, commit and push the work to GitHub:

```
$ git push origin newfeature
```

Finally, create the pull request. Go to your forked repository on GitHub, something like `https://github.com/username/tsduck`, and select the branch `newfeature`. Select the "Pull requests" tab and click on the green button "New pull request". Select the branch for the pull request and click on "Create pull request".

The pull request is transmitted to the project main repository `tsduck/tsduck`. The Continuous Integration (CI) process is automatically started on your code. During this CI process, the code is compiled and tested on Linux, macOS and Windows using various compilers and variants of the C++ standard. Then, all TSDuck tests are run on your code.

In case of failure of the CI job, you will be notified by mail and the pull request is retained on hold. You may then review the failures, update your code, commit and push on your branch again. The new commits are added to the pull request and the CI job is run again.

Most users work in one environment, one operating system, one compiler. Even if the code works in this environment, it may fail to build or run in another environment, operating system or compiler. This is why the CI process is useful because you can review the impact of your modifications in other environments.

3.2.3. Testing your code

Before pushing your code and creating the pull request, you will test your code. Additionally, when your add new features or support for new signalization, tables or descriptors, it is recommended to update the TSDuck test suite.

See the section 4.1 for more details.

To update the TSDuck test suite according to your new code, fork the tsduck-test repository, update it, and create pull requests on this repo using the same method as the main tsduck repository.

Pay attention to the interactions between the `tsduck` and `tsduck-test` repositories.

The `tsduck-test` repository contains tests and reference outputs for those tests. When you update the TSDuck code, the test reference output may need to be updated accordingly. You do that in your fork of the `tsduck-test` repository. When you create a pull request on the main `tsduck` repository, the CI job checks the origin of the pull request. In your case, this is your `username/tsduck` forked repository. The CI job checks if you also have a `username/tsduck-test` forked repository. If it exists, it is used to run the test suite. If you do not have a fork of the test repository, the reference `tsduck/tsduck-test` repository is used.

Consequently, the recommended workflow depends on the type of code contribution you provide.

- If you provide a simple code update which has no impact on the test suite, then you should fork the `tsduck/tsduck` repository only. Your code will be tested against the `tsduck/tsduck-test` repository to make sure it does not break the project.
- If your contribution is more substantial and needs an update of the test suite, then you need to fork the `tsduck/tsduck` and `tsduck/tsduck-test` repositories. Once your code and tests are complete, create the commits and push the two repositories. At the end, create the pull requests on the two repositories. The CI

job for the `tsduck` repository will then use your `username/tsduck-test` repository for the test suite. If all tests pass on all operating systems, your contributions in `tsduck` and `tsduck-test` will be merged.

One last point: If you maintain your fork of `USERNAME/tsduck-test`, be sure to keep it synchronized with the reference `tsduck/tsduck-test` repository because your `USERNAME/tsduck-test` will always be used in your CI jobs. If one day, you submit a small code update which did not need any update in the test suite and your `USERNAME/tsduck-test` is not up-to-date, your CI job may fail.

3.3. Integrator workflow

On your local development system, configure your TSDuck development git repository to track all pull requests. In the file `.git/config`, add the following line in section `[remote "origin"]`:

```
[remote "origin"]
    ... existing lines ...
    fetch = +refs/pull/*/head:refs/pull/origin/*
```

To integrate a pull request number *NNN*, fetch it in a local branch named `NNN`:

```
$ git fetch origin
$ git checkout -b NNN pull/origin/NNN
```

To merge the pull request into the `master` branch:

```
$ git checkout master
$ git merge NNN
```

Additional review and fix may be necessary before pushing the contribution.

Chapter 4. Maintaining TSDuck Code

4.1. Testing TSDuck

4.1.1. Testing overview

TSDuck is highly flexible, allowing an unlimited number of configurations. Testing it is consequently challenging. Moreover, some use cases are difficult to automate or require specific hardware. Testing live TS reception using all possible DVB tuners or Dektec devices requires a lot of material, human and time resources which are far beyond the capabilities of a free open-source project.

From a strict industrial standpoint, TSDuck is consequently not fully tested.

However, on a pragmatic standpoint, test suites have been setup to extract the best of what could be done with limited resources.

Thus, although it is impossible to guarantee that a given release of TSDuck is bug-free, we may be relatively confident that no major regression has been introduced.

4.1.2. Organization of the tests

The code of TSDuck is divided in two parts, a large C++ library (`tsduck.dll`, `libtsduck.so`, or `.dylib`) and a collection of small command line tools and plugins.

Similarly, the tests for TSDuck are divided in two parts.

- The TSDuck library has its own unitary test suite based on a custom framework named "TSUnit". This test suite is part of the main tsduck repository for TSDuck in directory `src/utest`.
- The tools and plugins are less easy to test. They work on large transport stream files which would clutter the TSDuck repository. The repository tsduck-test contains those tests and the relevant scripts and data files.

The two test suites are fully automated.

4.1.3. The TSDuck library test suite

In the main TSDuck repository, the directory `src/utest` contains the source file for one single program named `utest`. This program is divided in many source files (or test suites). Each source file contains many unitary tests. The test infrastructure is based on a custom framework named "TSUnit".

The structure of TSUnit is freely inspired by equivalent frameworks such as CppUnit and JUnit. It has been adapted to TSDuck specificities and focuse on fast and easy development of test suites.

The `utest` executable is built twice, once using the TSDuck shared library and once using the static library. On UNIX systems (Linux, macOS, BSD), both versions of the test suite are built and run using `make test` as illustrated below

```
$ make -j10 test
  .....
  [CXX] utestXML.cpp
  [LD] /home/tsduck/bin/release-x86_64-vmubuntu/utest
  [LD] /home/tsduck/bin/release-x86_64-vmubuntu/utest_static
TSPLUGINS_PATH=/home/tsduck/bin/release-x86_64-vmubuntu LD_LIBRARY_PATH=/home/tsduck/bin/release-
x86_64-vmubuntu /home/tsduck/bin/release-x86_64-vmubuntu/utest
```

```
OK (619 tests, 28730 assertions)

TSPLUGINS_PATH=/home/tsduck/bin/release-x86_64-vmubuntu LD_LIBRARY_PATH=/home/tsduck/bin/release-
x86_64-vmubuntu /home/tsduck/bin/release-x86_64-vmubuntu/utest_static

OK (616 tests, 28707 assertions)

$
```

The number of tests and assertions changes with new versions of TSDuck. When new features are added, the corresponding new tests are added.

Note that the statically linked version contains slightly less tests. The missing tests are related to plugin activations and they can run only in a shared library environment.

On Windows, the Visual Studio project builds two executables named `utests-tsduckdll.exe` and `utests-tsducklib.exe`. They can be run manually or from Visual Studio:

```
D:\tsduck> bin\Release-x64\utests-tsduckdll.exe

OK (619 tests, 28747 assertions)

D:\tsduck> bin\Release-x64\utests-tsducklib.exe

OK (616 tests, 28724 assertions)

D:\tsduck>
```

By default, the test program runs all tests and reports failures only. But `utest` also accepts a few command options which make it appropriate to debug individual features.

The available command line options are:

`-d`	Debug messages are output on standard error
`-l`	List all tests but do not execute them.
`-t name`	Run only one test or test suite.

First, the option `-l` is used to list all available tests:

```
$ utest -l
AlgorithmTest
    AlgorithmTest::testEnumerateCombinations
ArgsTest
    ArgsTest::testAccessors
    ArgsTest::testAmbiguousOption
........
    WebRequestTest::testNoRedirection
    WebRequestTest::testReadMeFile
XMLTest
    XMLTest::testChannels
    XMLTest::testCreation
    XMLTest::testDocument
    XMLTest::testEscape
    XMLTest::testFileBOM
    XMLTest::testInvalid
    XMLTest::testKeepOpen
    XMLTest::testTweaks
    XMLTest::testValidation
$
```

The left-most names are test suite names. They represent one source file in `src/utest`. It is possible to run all tests in a test suite or one specific test. For instance:

```
$ utest -t ArgsTest
$ utest -t XMLTest::testValidation
```

The option `-d` is used to produce debug message (see examples in the test source files).

Thus, `utest` alone can be used as an automated fairly complete non-regression test suite for the TSDuck library or as a debug environment for a given feature under development (in all good "test-driven development" approaches, the code is written at the same time as its unitary test).

4.1.4. The TSDuck tools and plugins test suite

The Git repository tsduck-test contains high-level tests for the TSDuck tools and plugins.

This test suite is fully automated and works on test files only. No specific hardware (DVB tuner or Dektec device), no live stream is tested. This is a known limitation.

The principle is simple. Reference input files are stored in the repository, mainly transport stream files which were once captured from real live sources. Test scripts contain deterministic commands which should always produce the same outputs (data, logs, etc). These reference outputs are also stored in the repository.

The test suite runs using a given version of TSDuck and its outputs are compared with the reference outputs. If differences are detected, there is a potential problem.

Since different versions of TSDuck may produce slightly different outputs, a given version of the test suite formally applies to one version of TSDuck only. Git tags are aligned in both repositories (or should be…) to indicate the target version.

4.1.4.1. Structure of the test suite

In short, execute the script `run-all-tests.sh` to run the complete test suite.

The repository contains the following subdirectories:

`tests`	Contains one script per test or set of tests. The name for test *NNN* is `test-NNN.sh`. Each test script can be executed individually. All tests are executed using the script `run-all-tests.sh`.
`common`	Contains utilities and common script.
`input`	Contains input data files for the tests.
`reference`	Contains reference output files for the various tests. There is one subdirectory `test-NNN` per test which contains all output files for that test.
`tmp`	Contains output files which are created by the execution of the tests. These files are typically compared against reference output files in `reference`. These files are temporary by definition. The subdirectory `tmp` is present on test machines only and is excluded from the Git repository.

4.1.4.2. Adding new tests

To add a new test:

- Allocate a new test number and document the purpose of the new test in the file `README.md`.

- Add input files in subdirectory `input`. For test *NNN*, all input files should be named `test-NNN.*`. There is generally zero or one input file per test, sometimes more.
- Create the script `test-NNN.sh` in subdirectory `tests`. Use other existing test scripts as templates.
- Run the command `tests/test-NNN.sh --init`. If the test is properly written, this creates the reference output files in the subdirectory `reference/test-NNN`. Manually check the created files, verify that they are correct. Be careful with this step since these files will be used as references.
- Run the same command without the `--init` option. This time, the output files are created in `tmp` and are compared with files in `reference`. Verify that all tests pass. Errors may appear if the test script is not properly written or if the output files contain unique, non-deterministic, time-dependent, system-dependent or file-system-dependent information. Make sure the output files are totally reproduceable in all environments. At worst, add code in the test script to remove any information from the output files which is known to be non-reproduceable.

Sometimes, TSDuck is modified in such in a way that an output file is modified on purpose. Usually, this starts with a failed test. When analysing the test failure, it appears that the modification of the output is intentional. In that case, re-run the command `tests/test-NNN.sh --init` to update the reference output files. Do not forget to manually validate them since they will act as the new reference.

The reference output files are stored in the Git repository. Therefore, the best way to have a quick overiew of what changed in the output reference files is simply `git diff`.

4.1.4.3. Testing a development version

By default, the test suite uses the TSDuck command from the system path. Typically, it will use the installed version.

To test a development version, the two Git repositories `tsduck` and `tsduck-test` shall be checked out at the same level, side by side in the same parent directory. First, TSDuck shall be rebuilt in its repository.

Then, when the option `--dev` is specified to a test script or to `run-all-tests.sh`, the test suite automatically uses the TSDuck executables from the development repository.

4.1.4.4. Large files

The `tsduck-test` repository contains large files, typically transport stream files.

Initially, these files were not stored inside the regular GitHub repository. Instead, they used the Git Large File Storage (LFS) feature of GitHub. However, using LFS on GitHub happended to be a pain, as experienced by others and explained in this article.

As a consequence, the transport stream files were re-integrated into the Git repository as regular files. But we now limit their size to 20 MB.

4.2. Automation

TSDuck is a free open-source project which is maintained on spare time by very few developers (only one at the time of this writing). As a consequence, the maintenance of the project is driven by the lack of time and resource. To face these constraints, automation is essential.

This is briefly explained in this presentation.

For future maintainers or contributors, this page lists a few automation procedures which are used by the project.

4.2.1. Continuous integration

Each time a commit or a pull request is pushed to GitHub, the workflow `.github/workflows/continuous-integration.yml` is automatically run to verify that the project can be built on most supported platforms and contains no regression.

- The workflow is run on Linux Ubuntu, macOS, and Windows (64 and 32 bits).
- The compilation is done using two revisions of the C++ language: C++17 and C++20.
- On Linux, the builds are repeated using two compilers, `gcc` and `clang`.
- In each configuration of platform, compiler and language level:
  - TSDuck is fully built.
  - The unitary tests are run.
  - The full test-suite is downloaded and run.
  - The sample programs are built using a typical TSDuck installation of the binaries which were just built (Linux and macOS only).
- The documentation is rebuilt: user's guide, developer's guide, programming reference using doxygen. Missing documentation for new features can be found in the log. This is done on Ubuntu only.

4.2.2. Nightly builds

Every night, if there were any modification in the TSDuck repo during the last day, "nightly builds" are automatically produced and published on the TSDuck web site. No manual action is required, everything is automated.

The workflow `.github/workflows/nightly-build.yml` is automatically run every day at 00:40 UTC.

- On Linux Ubuntu and Windows (64 bits only), the TSDuck binary packages are built for the current state of the master branch in the repository.
- The produced binaries are installed on the CI/CD system.
- The full test-suite is downloaded and run.
- The complete set documentation (user's guide, developer's guide, programming reference) is built and packaged in an archive. If the installed version of doxygen is known to contain bugs in the generation of the TSDuck documentation (many were found), the source code for a known good version of doxygen is downloaded and rebuilt first. This is done to ensure that the documentation is always correct.
  - The Linux Ubuntu binaries, the Windows binaries and the documentation package are published as "artefacts" of the workflow. They are publicly downloadable from GitHub.

At the end of the `nightly-build.yml` workflow, when all build jobs are completed, the update job triggers a signal on the TSDuck web site. A PHP script is automatically run on the web site to retrieve, download and publish the latest nighly binaries and documentation.

4.2.3. Release creation

Creating an "official" TSDuck release is relatively easy. Although the build time for all binaries can be long, everything is automated and run in the background without requiring the maintainer's attention.

Since TSDuck is maintained with little resource and time, there is no product management cycle. Thanks to the continuous integration process, any state of the repository can be used as a release candidate, as long as there is no issue in the CI process and no major development or fix is in progress.

As a rule of thumb, a new release is produced every 3 to 6 months, when enough new features or fixes are available, based on the CHANGELOG file.

4.2.3.1. Building the various binaries

Binaries must be built for Windows (64 bits only) and a few major Linux distros (Fedora, RedHat, Ubuntu, Debian, and 32-bit Raspian). Additional binaries are now built for some Arm64 Linux distros.

The idea is to have a central system (Linux, macOS or Windows) from which the builds are orchestrated. The script `pkg/build-remote.sh` can be used to build TSDuck binaries on various remote systems and collect the resulting installers.

The remote systems can be physical systems or virtual machines running on the central system. When a virtual machines is not active, it is automatically booted, the binaries are built and collected, and then the virtual machines is shut down. The currently supported hypervisors to managed virtual machines are VirtualBox, VMware and Parallels Desktop (macOS). All systems shall be preconfigured so that the central user is authorized to connect on each of them using `ssh` without password (use public key authentication, not passwordless accounts!)

The maintainer shall typically have a personal script, outside the repository, which calls `build-remote.sh` on all systems.

Typical example of such a script, with one physical remote system (a Raspberry Pi) and 5 virtual machines on the local host:

```
$HOME/tsduck/pkg/build-remote.sh --host raspberry
$HOME/tsduck/pkg/build-remote.sh --host vmfedora --parallels Fedora
$HOME/tsduck/pkg/build-remote.sh --host vmredhat --parallels RedHat
$HOME/tsduck/pkg/build-remote.sh --host vmubuntu --parallels Ubuntu
$HOME/tsduck/pkg/build-remote.sh --host vmdebian --parallels Debian
$HOME/tsduck/pkg/build-remote.sh --host vmwindows --parallels Windows --windows --directory
Documents/tsduck --timeout 20
```

So, building a release is as simple as running that script. After "some time", all binaries and build log files are available in the `pkg/installers` subdirectory.

Be sure to check that all binaries are present. Check the log files if some of them are missing.

4.2.3.2. Creating the GitHub release

Once the binary installers are collected, simply run this command to create and publish the release on GitHub:

```
$ pkg/github/release.py --create
```

The current state of the repo on GitHub is used as base for the release. A tag is created with the version number. The release is created with a explanatory description. All binaries are published as assets of that release.

4.2.3.3. Creating the HomeBrew release

The GitHub release contains binaries for Linux and Windows. On macOS, TSDuck is distributed through HomeBrew. The binaries for all macOS versions, Intel and Arm platforms, are built inside the HomeBrew CI/CD pipeline when an updated formula is pushed in a pull request.

After the creation of the release on GitHub, create the pull request for the new TSDuck version in HomeBrew using the following command:

```
$ pkg/homebrew/brew-bump-formula-pr.sh -f
```

The option `-f` forces the creation of the pull request. Without it, it works in "dry run" mode. It can be safe to start with a dry run, as a test.

When the HomeBrew CI/CD pipeline has finished to process the pull request, the new version of TSDuck is available for all flavors of macOS.

Recently, TSDuck has been placed by the Homebrew maintainers in the "autobump" list of packages. The original projects for the 2600+ packages in this list are regularly monitored by a Homebrew workflow. Whenever one of these projects publishes a new version, the update of the formula is automatically done. Next time a new version of TSDuck is published, it should be worth waiting a couple of days before requesting an update using a pull request, to check if a similar pull request is automatically submitted by the "autobump" workflow.

4.2.3.4. Updating the version number

Once a release is published, the minor version number of TSDuck `TS_VERSION_MINOR` must be updated in the source file `src/libtsduck/tsVersion.h`.

This is currently not automated and shall be manually updated before the first commit following the publication of a new release.

4.2.4. Cleanup of long-standing issues

The issues area on GitHub is used to report problems, ask questions, and support any discussion about TSDuck. When an issue is obviously completed, because a complete answer was provided or a fixed is pushed, the issue is closed. Sometimes, a plausible response or fix is provided but some feedback is expected from the user to confirm this. When a positive feedback is provided, the issue is closed.

However, some users never provide a feedback after their problem is solved. In that case, the issue remains open forever.

To solve this, there is a label named "close pending". When a plausible response, solution or fix is provided, the maintainer of the project sets the "close pending" label on the issue. It remains open. However, if the issue is not updated in the next 150 days, it will be automatically closed.

This is achieved by the workflow `.github/workflows/cleanup-issues.yml`. This workflow is scheduled every week on Sunday at 02:00 UTC. It runs the Python script `pkg/github/close-pending.py` which automatically closes all issues with label "close pending" and no update within the last 150 days.

4.3. TSP design

This section is a brief description of the design and internals of `tsp`. It contains some reference information for `tsp` maintainers.

`tsp` is designed to clearly separate the technical aspects of the buffer management and dynamics of a chain of plugins from the specialized plugin processing (TS input, TS output, packet processing).

4.3.1. Plugin Executors

Each plugin executes in a separate thread. The base class for all plugin threads is `ts::tsp::PluginExecutor`. Derived classes are used for input, output and packet processing plugins.

4.3.2. Transport packets buffer

There is a global buffer for TS packets. Its structure is optimized for best performance.

The input thread writes incoming packets in the buffer. All packet processors update the packets and the output thread picks them at the same place. No packet is copied or moved in memory.

The buffer is an array of `ts::TSPacket`. It is a memory-resident buffer, locked in physical memory to avoid virtual memory paging (see class `ts::ResidentBuffer`).

The buffer is managed in a circular way. It is divided into logical areas, one per plugin thread (including input and output). These logical areas are sliding windows which move when packets are processed.

Inside a `ts::tsp::PluginExecutor` object, the sliding window which is currently assigned to the plugin thread is defined by the index of its first packet (`_pkt_first`) and its size in packets (`_pkt_cnt`).

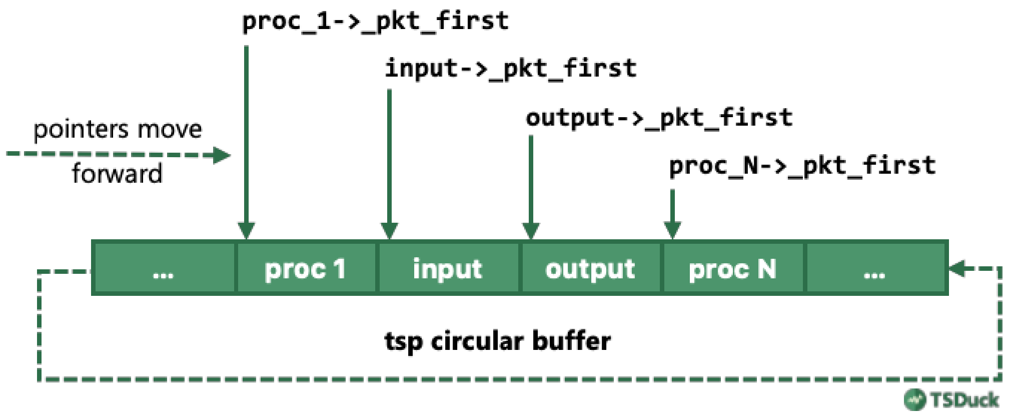

Figure 2. Flat (non-circular) view of the buffer:

When a thread terminates the processing of a bunch of packets, it moves up its first index and, consequently, decreases the size of its own area and accordingly increases the size of the area of the next plugin.

The modification of the starting index and size of any area must be performed under the protection of a mutex. There is one global mutex for simplicity. The resulting bottleneck is not so important since updating a few pointers is fast.

When the sliding window of a plugin is empty, the plugin thread sleeps on its `_to_do` condition variable. Consequently, when a thread passes packets to the next plugin (ie. increases the size of the sliding window of the next plugin), it must notify the `_to_do` condition variable of the next thread.

When a packet processor decides to drop a packet, the synchronization byte (first byte of the packet, normally 0x47) is reset to zero. When a packet processor or the output executor encounters a packet starting with a zero byte, it ignores it. Note that this is transparent to the plugin code in the shared library. The check is performed by the `ts::tsp::ProcessorExecutor` and `ts::tsp::OutputExecutor` objects. When a packet is marked as dropped, the plugin is not invoked.

All `ts::tsp::PluginExecutor` are chained in a ring. The first one is input and the last one is output. The output points back to the input so that the output executor can easily pass free packets to be reused by the input executor.

The `_input_end` flag indicates that there is no more packet to process after those in the plugin's area. This condition is signaled by the previous plugin in the chain. All plugins, except the output plugin, may signal this condition to their successor.

The `_aborted flag` indicates that the current plugin has encountered an error and has ceased to accept packets. This condition is checked by the previous plugin in the chain (which, in turn, will declare itself as aborted). All plugins, except the input plugin may signal this condition. In case of error, all plugins should also declare an `_input_end` to their successor.

4.4. Adding PSI/SI tables or descriptors

Adding support for new PSI/SI tables or descriptors is a welcome contribution to TSDuck. Users from various continents, using different standards, or participating in standardization processes, are in the best position to implement new tables or descriptors.

We recommend to release these contributions in open source, as part of the TSDuck project.

This section summarizes the main steps when implementing new tables or descriptors.

See chapter 3 for more details on the contribution process.

4.4.1. Code base selection

The main recommendation to start with is: do not develop a new table or descriptor from scratch. Use an existing and proven one as code base and adapt to your new structure.

To identify the right existing structure as code base, use some of these criteria:

- Short single-section table vs. long multi-section table.
- Flat vs. structured section, with descriptors or not, including substructures with descriptors (e.g. PMT).
- Use of strings, DVB strings, ATSC strings, ISDB strings.
- Public descriptor, DVB private descriptor, table-specific descriptor.

Because the number of possible descriptor *tags* (a.k.a. descriptor ids) is limited to 256 values, there is no room for all possible descriptors. For this reason, the various standard organizations use tricks such as *extended descriptors*, *private descriptors*, or *table-specific descriptors*.

In a MPEG/DVB context, the allocation of descriptor ids is the following:

`0x00-0x3E`	MPEG-defined descriptors
`0x3F`	MPEG extension descriptor
`0x40-0x7E`	DVB-defined descriptors
`0x7F`	DVB extension descriptor
`0x80-0xFE`	Private descriptors
`0xFF`	Reserved, not allocated, typically used as "null" value

4.4.1.1. Extended descriptors

MPEG and DVB separately define the concept of *extended descriptors*. Because of the shortage of descriptor ids, each of the two standards have defined an *extension_descriptor*, typically using their last allocated descriptor id. This descriptor is a generic envelope for specialized descriptors.

The first byte of the descriptor payload is an *extended_descriptor_id* which identifies the actual descriptor type. This allows the definition of up to 256 additional descriptors.

TSDuck does not use any specific type for the generic *extended_descriptor*. Instead, there is a distinct type for each form of *extended_descriptor*. Each of them has its own XML element and C++, just like any other descriptor. The extended descriptor mechanism is only considered as a binary serialization detail, not a different type of descriptor.

If you have to implement a MPEG-defined extended descriptor, you may use the *HEVC_timing_and_HRD_descriptor* as code base.

If you have to implement a DVB-defined extended descriptor, you may use the *supplementary_audio_descriptor* as code base.

4.4.1.2. Private descriptors (DVB)

In the DVB standard, descriptor ids 0x80 to 0xFE are "private". They are reserved for use by private entities, typically TV operators, broadcast equipment vendors, Conditional Access Systems (CAS) vendors.

To determine which semantics should be associated with a given descriptor id in that range, DVB defines a *private_data_specifier_descriptor* which contains a 32-bit *private_data_specifier* (PDS). DVB allocates a unique PDS to any private organization which requests it. See ETSI-101-162 and the DVB services for more details on allocated PDS values.

In a descriptor list, all private descriptors which come after a *private_data_specifier_descriptor* are defined by the private organization which is identified in the *private_data_specifier_descriptor*. If several private descriptors from distinct defining entities must be placed in the same descriptor list, several *private_data_specifier_descriptor* are allowed to switch from one entity to another.

A descriptor with a tag in the range 0x80-0xFE and no preceding *private_data_specifier_descriptor* is illegal.

That being said, in practice, the experience has exhibited two families of bugs:

- Rogue signalization: Some organizations ignore this rule and define their own private descriptors, in the range 0x80-0xFE, without officially allocated PDS value.
- Signalization bugs: Some broadcasters "forget" to insert the right *private_data_specifier_descriptor* before a well-defined private descriptor.
- Implementation bugs: Some implementers of receivers (set-top box, TV set) ignore the PDS rule or forget to check the previous PDS. They blindly interpret some private descriptor based on some expected descriptor id in the range 0x80-0xFE. If the same private descriptor id is used in the context of another PDS, the receiver incorrectly interprets the binary descriptor.

All these bugs are real and were regularly found during the development and usage of TSDuck. If you implement a private descriptor, be sure to follow the rules.

You may use the EACEM-defined *eacem_stream_identifier_descriptor* as code base.

Most of the commonly used private descriptors are some forms of *logical_channel_number_descriptor*. The Logical Channel Number (LCN) is the usual concept of TV channel number, the oldest and most traditional way of identifying a TV channel. Surprisingly, neither MPEG nor DVB defined it. Therefore, operators or equipment vendors have to define their own way of identifying LCN’s. For this reason, there is a wide range of variants of private *logical_channel_number_descriptor* which all contain the same kind of information. If you implement such a descriptor, with a similar implementation, your class should be a subclass of `AbstractLogicalChannelDescriptor`. Use *eacem_logical_channel_number_descriptor* as code base.

4.4.1.3. Table-specific descriptors

So-called *table-specific descriptors* are specific descriptors which exist only in the context of a couple of specific tables. They usually re-use the tag of a standard descriptor, typically in the MPEG-defined range. Of course, it is assumed that the standard descriptor, the tag of which has been hijacked, will never be used in those specific tables to avoid ambiguities.

Let’s take an example, the *target_IP_address_descriptor*. This is a DVB-defined descriptor which can be used only inside an INT or a UNT, two DVB-defined tables. The *target_IP_address_descriptor* uses tag 0x09, which is normally used by a MPEG-defined *CA_descriptor*. When TSDuck analyzes a descriptor list and encounters a tag 0x09, it usually starts to analyze a *CA_descriptor*, except when the table is an INT or a UNT, in which case it analyzes a *target_IP_address_descriptor*.

This situation is supported by TSDuck. If you have to implement such a table-specific descriptor, use *target_IP_address_descriptor* as code base.

4.4.2. Affiliation to a standard

Each table or descriptor is defined either by a standard body or an organization, committee or private company. Check if other PSI/SI from this organization is already implemented in TSDuck. This is important because source files for PSI/SI are organized by standard.

Tables are implemented in the directory `src/libtsduck/dtv/tables`, using one subdirectory per standard. The current subdirectories are `atsc`, `dvb`, `isdb`, `mpeg`, `scte`. Currently, only renown standard bodies define tables.

Descriptors are implemented in the directory `src/libtsduck/dtv/descriptors`, using one subdirectory per standard. The current subdirectories are the same as tables, plus various organizations such as `avs` or `uwa`, plus private companies which define private DVB descriptors such as `eacem`, `dtg`, `sky`.

Try to find the right subdirectory for your new structure. Create another directory if required.

In that descriptor, you will have to create three or four files (the last is optional). For instance, the MPEG-defined *ISO_639_language_descriptor* is implemented as:

```
src/libtsduck/dtv/descriptors/mpeg:

    tsISO639LanguageDescriptor.xml
    tsISO639LanguageDescriptor.h
    tsISO639LanguageDescriptor.cpp
    tsISO639LanguageDescriptor.names
```

More details follow in the next sections.

4.4.3. Declaring identifiers

Your table or descriptor must have a 8-bit identifier. You need to add it in the TSDuck source code.

Table ids and descriptor ids are defined in file `src/libtsduck/dtv/signalization/tsPSI.h`, in enum lists `TID` and `DID`, respectively. The ids are grouped by standard, be sure to add it at the right place.

In the case of a table, if that table is expected on some predefined PID, also add this PID in file `src/libtsduck/dtv/transport/tsTS.h`, in the enum list `PID`.

In the case of a private DVB descriptor, your descriptor is valid only after a *private_data_specifier_descriptor* which contains the *private_data_specifier* (PDS) of the organization which defines the descriptor. Check if that PDS value is present in file `src/libtsduck/dtv/signalization/tsPSI.h`, in enum list `PDS`. Add it if not present.

For TSDuck to display meaningful identifiers, the source tree contains *names files*, with a `.names` extensions. These files associate a unique value with a name. There are several sections (for PID, TID, DID, for instance). In each section, a value can be present only once and values must be declared in ascending order.

Add the table or descriptor name in the file `src/libtsduck/dtv/signalization/tsPSI.names`, in sections `TableId` or `DescriptorId`, respectively. Carefully read the comments at the beginning of each section. It explains the encoding of each unique value.

For table ids, the value includes the standard and the optional *CAS_id* (useful for ECM and EMM only).

For descriptor ids, the value includes the PDS for private descriptors or the *table_id* for table-specific descriptors. Note that, for historical reasons, ATSC and ISDB descriptors are encoded with a "fake" dedicated PDS.

If you have added a new PDS value, add its name in the `PrivateDataSpecifier` section of `tsPSI.names`.

If you implement a MPEG-defined or DVB-defined extended descriptor, add the corresponding *extended_descriptor_id* in `src/libtsduck/dtv/signalization/tsPSI.h`, in enum lists with `MPEG_EDID_` and `EDID_` symbols.

Also add the corresponding name in `src/libtsduck/dtv/signalization/tsPSI.names`, in sections

`MPEGExtendedDescriptorId` or `DVBExtendedDescriptorId`, respectively.

4.4.4. XML definition

You must define an XML representation for your table or descriptor in a `.xml` file. Use the selected code base as reference.

This XML file is an *XML model file*, as defined in the TSDuck User’s Guide.

A table shall be defined as one XML element inside the following envelope:

```
<?xml version="1.0" encoding="UTF-8"?>
<tsduck>
  <_tables>
    <my_table_name ...>
      <_any in="_metadata"/>
      ...
    </my_table_name>
  </_tables>
</tsduck>
```

Note the mandatory `<_any in="_metadata"/>`.

A descriptor shall be defined as one XML element inside the following envelope:

```
<?xml version="1.0" encoding="UTF-8"?>
<tsduck>
  <_descriptors>
    <my_descriptor ...>
      ...
    </my_descriptor>
  </_descriptors>
</tsduck>
```

For attributes and element names, preferably use the exact same names as defined in the standard for your table or descriptor.

Do not blindly copy the binary structure in the XML description. Define an XML equivalent representation.

For instance, a common pattern for optional fields in binary structures is to define a one-bit *foo_flag* and a subsequent optional *foo* field. The *foo* field is typically present only when *foo_flag* is 1. Do not define *foo_flag* in the XML structure. Just define a *foo* attribute and document it as optional.

The template value of XML attributes is a short informal type declaration. For integer values, always start the description string with `uintN` or `intN`, when *N* is the size in bits of the binary field. This `uintN` or `intN` is used by the automatic XML-to-JSON translation to generate a JSON number instead of a JSON string.

Your `.xml` file will be automatically grabbed by the TSDuck build system and integrated into the final configuration files.

4.4.5. C++ class

The C++ header (`.h`) and body (`.cpp`) files for the table or descriptor class are mandatory. Start with the selected code base and carefully replace the structure names.

In the `.cpp` file, there is a fundamental macro: `TS_REGISTER_TABLE()` for tables and `TS_REGISTER_DESCRIPTOR()` for descriptors. This is a C++ trick which automatically registers your structure in the PSI/SI repository during the initialization of the module. If you omit this macro, your table will not be recognized.

The registration macro may take various forms depending on the type of structure (standard descriptor, table-specific descriptor, extended descriptor, etc.) Be careful to select a code base with the same characteristics in order to copy the same type of registration.

4.4.6. Names file

If necessary, you may provide a `.names` file. This is useful when a field of your structure can get distinct values with distinct meanings. When displaying a structure, it is more convenient for the user to get a meaningful name rather than a value.

A `.names` file is organized in several sections. By convention, use section names which start with the XML name of your structure, followed by a dot.

Example of the file `tsISO639LanguageDescriptor.names`, for the *ISO_639_language_descriptor*:

```
[ISO_639_language_descriptor.audio_type]
Bits = 8
0x00 = undefined
0x01 = clean effects
0x02 = hearing impaired
0x03 = visual impaired commentary
```

In the C++ source file, use the inherited static method `DataName()` to retrieve a meaningful name, with optional formatting of the value before or after the name.

Example of the file `tsISO639LanguageDescriptor.cpp`:

```
void ts::ISO639LanguageDescriptor::DisplayDescriptor(TablesDisplay& disp, PSIBuffer& buf, const
UString& margin, DID did, TID tid, PDS pds)
{
    ...
    disp << ", Type: " << DataName(MY_XML_NAME, u"audio_type", buf.getUInt8(), NamesFlags::FIRST) <<
std::endl;
```

Your `.names` file will be automatically grabbed by the TSDuck build system and integrated into the final configuration files.

4.4.7. Documentation

Your new table or descriptor shall be documented in two ways:

1. The XML structure is documented in the TSDuck User's Guide (asciidoc format).
2. The C++ class is documented in the TSDuck Programming Reference (doxygen format).

4.4.7.1. User's guide

The user's guide must be manually updated.

The asciidoc files (`.adoc`) for the PSI/SI XML structures are in the directory tree `doc/user/si-xml`. Just like the source files, there is one subdirectory per standard. In each subdirectory, there is one single `.adoc` file for all tables and one single `.adoc` file for all descriptors.

Edit or create the corresponding file. If you create a new file, add an include directive in the file `doc/user/20D-app-si-xml.adoc`.

Some guidelines:

- In each file, keep tables and descriptors organized in alphabetical order.
- Copy the surrounding asciidoc syntax from other existing tables or descriptors.
- Remove the enclosing `<tsduck>`, `<_tables>`, `<_descriptors>` structures, just keep your structure.
- In tables, remove the `<_any in="_metadata"/>`. It is meaningless for the user.
- Add any comment or formatting which makes the result more informative to the user.

At the beginning of the section, add a reference to the defining standard, for instance:

```
Defined by MPEG in <<ISO-13818-1>>.
```

The reference between `<<` and `>>` must be a valid one from the bibliography in file `doc/user/20F-app-references.adoc`, for instance:

```
* [[[ISO-13818-1]]] ISO/IEC 13818-1:2018 | ITU-T Recommendation H.222 (2017):
  "Generic coding of moving pictures and associated audio information: Systems" (also known as "MPEG-2
System Layer").
```

If the reference does not exist yet in the bibliography, add it. Keep the references sorted in alphabetical order.

4.4.7.2. Programming reference

All public structures and fields in the C++ header file must be documented using Doxygen tags. See examples in existing structure. This is the way your structure will become documented in the programming reference.

No public element shall be left undocumented. To verify this, generate the documentation and check any error. Undocumented elements are reported.

- On UNIX systems (Linux, macOS, BSD), run `make doxygen`.
- On Windows systems, run the PowerShell script `doc\doxy\build-doxygen.ps1`.

In the initial descriptor of your C++ class, make sure it is properly identified with the right group and standard. For instance:

```
//!
//! Representation of a Program Association Table (PAT).
//! @see ISO/IEC 13818-1, ITU-T Rec. H.222.0, 2.4.4.3
//! @ingroup table
//!
```

or:

```
//!
//! Representation of an ISO_639_language_descriptor
//! @see ISO/IEC 13818-1, ITU-T Rec. H.222.0, 2.6.18.
//! @ingroup descriptor
//!
```

The directive `@ingroup` is used by Doxygen to assign the class in the right group.

The directive `@see` is important in three ways.

1. It is included in the Doxygen documentation.
2. It helps the future maintainers of the code to find the right documentation and directly the section number

where to look.

3. It is also used in the automatic generation of the appendix A of this document.

4.4.8. Tests

There are lots of traps and pitfalls in the coding of a table or descriptor. It is crucial to test it thoroughly.

First, become familiar with the TSDuck test suite as described in section 4.1.4.

Once you have cloned your forked versions of the two repositories, `tsduck` and `tsduck-test`, side by side in the same parent directory, you can implement a test for your table or descriptor.

This kind of test is standardized. The idea is to start from an XML file containing several samples of your table or descriptor. Then, invoke the common script `standard-si-test.sh`.

This standard test compiles the XML file in binary, decompiles it to generate XML and JSON, recompiles the output, inject the tables in a transport stream, extract them in text form, etc. All intermediate results are kept as reference.

This kind of test is interesting in two ways. First, during the initial test, after development, it is a good tool to debug the serialization, deserialization, binary and XML. Second, the reference outputs will track any future regression.

For instance, the test 027 is the reference test for SCTE 35 tables and descriptors. All tested structures are in the file `tsduck-test/input/test-027.xml`. The test script `tsduck-test/tests/test-027.sh` is very simple:

```
#!/usr/bin/env bash
source $(dirname $0)/../common/testrc.sh
test_cleanup "$SCRIPT.*"
source "$COMMONDIR"/standard-si-test.sh $SCRIPT.xml
```

In practice, *all* test scripts for that kind of PSI/SI test are identical. Only the input `.xml` file changes.

If your table or descriptor belongs to a set of structures which are already tested in an existing test, you may simply add your tested XML definitions in the existing test and update its reference output.

Otherwise, especially if you plan to implement several structures, you may create a new test. Just use existing tests with `standard-si-test.sh` as a starting point.

Pay attention to the XML structures you want to test. Keep in mind that you test one given structure in all possible ways, regardless of real applications. Your tested structures do not need to carry meaningful values. You test the *syntax* of your table or descriptor, not its *semantics*. You just want to test code, nothing else.

Here are some guidelines:

- If you test a descriptor, your don't care about which table it is in. Use a `<CAT>` for instance, a table which only contains descriptors and nothing else.
- If you test a table which contains descriptors, use any kind of simple descriptors, *ISO_639_language_descriptor* for instance. You do not care if such a descriptor does not make sense in your table.
- If you test a table which contains descriptors, test each descriptor list with zero, one, two descriptors.
- Test optional fields in structures where they are present and in other structures where they are omitted.
- More generally, when your code takes different steps or branches in the presence of different forms of input, test all possible forms of input.
- Test adjacent fields with different values. If two flags are in consecutive bits in the binary structure, test once

with a `true`/`false` combination and once with a `false`/ `true` combination.

- Use integer values which use the full width of a binary field to detect incorrect truncation or size errors. For instance, in a `uint32` field, use value 0xDEADBEEF, for instance, not 0 or 1.

When the result is satisfactory, submit a pull request for each repository, `tsduck` and `tsduck-test`. See section 3.2 for more details on that.

Chapter 5. Coding guidelines

5.1. Rationale for coding guidelines

In the computing community, there are many ways to develop software, many programming languages, many paradigms (functional, object oriented, etc.) and many conceptions of the art of programming. Each world has its own set of coding guidelines. But each world does have coding guidelines.

In software development, industrial or open-source, the keys to success are the management of the global cost of the software lifetime, the reliability and the security of the software.

- **Costs:** In the lifetime of a software product, the initial development often costs less than the maintenance. Consequently, an important requirement of software development is reducing the maintenance cost. And sometimes this is at the expense of the initial development cost. Writing good and maintainable code may cost a little bit more initially but the final cost will be beneficial. Writing good and maintainable code requires a set of coding guidelines which are followed by the whole development team.
- **Reliability:** The reliability of the software is planned from the design and is achieved through the quality of the code. The quality of the code also requires a common set of coding guidelines.
- **Security:** The security of the code has been quite challenged by hackers in the last twenty years. Most attacks take advantage of flaws in the code, either plain bugs or complex vulnerabilities in code which is otherwise functionally correct. The experience on those vulnerabilities results in a set of secure coding guidelines.

This chapter defines the coding guidelines for the TSDuck project.

Year after year, the code base of TSDuck has proven to become more robust. The maintenance process improved the code quality through regular refactoring operations, preserving or improving the modular structure of the project. Modifications, improvements or implementation of new features were always developed in a very short time and limited to a local set of source files. Applying the present set of guidelines has been of great help in this continuously improving robustness.

A previous document named *TSDuck Coding Guidelines* contained more generic coding rules. This document was derived from a more general one which was written by the author for professional software development. Because it was too generic, it is no longer relevant in the TSDuck documentation set. For reference, a copy is still available online.

5.2. Classification of coding guidelines

There is no unique term for coding guidelines. In the industrial, academic or open-source software, coding guidelines are sometimes named *coding standard* or *coding rules*. But they all refer to the same type of document and serve the same purpose.

There are several types of coding guidelines. Some guidelines are generic; they describe general software development practices. Some guidelines are more specific; they apply to details in the writing of source code for specific programming languages.

Some rules are strictly necessary to write good code and are not negotiable in any environment.

Some others are simple coding conventions, for instance the naming conventions for identifiers. Even within one single programming language, many different coding conventions exist and are strongly advocated by their respective supporters. These types of conventions can be initially discussed but, once they are selected, they must be adopted by all developers in the organization or project.

Several sets of conventions can be individually satisfactory but using several sets of conventions in the same software project is not. Good and maintainable software must be easily understandable and consistent coding conventions are a key part of the understanding of the code.

This document adopts the following classification:

- **Rule:** A mandatory regulation which must be applied without exception in all contexts. It cannot be negotiated or modified. A rule is preceded by ***[Rule]*** in the text.
- **Recommendation:** A regulation which must be applied everywhere it is possible. But there are circumstances where it is difficult to apply. Situations where a recommendation must be applied or, on the contrary, may be omitted shall be documented. A recommendation is preceded by ***[Recommendation]*** in the text.
- **Convention:** A mandatory regulation which must be applied without exception. Alternative conventions could have been possible in the first place but a single one had to be chosen. A convention is preceded by ***[Convention]*** in the text.

We use the word *guideline* as a generic term for rule, recommendation and convention.

In this document, we group conventions in separate sections. This separation is made on purpose to highlight the fact that rules and recommendations are not negotiable while conventions may differ between projects.

5.3. Generic coding guidelines

This section describes generic coding practices, independently of any specific programming language or framework.

Some of these guidelines may appear very generic in fact and evaluating the correctness of a code regarding these guidelines may be sometimes subjective or biased. However, this document prefers to be helpful rather than normative. And while generic advices can hardly be considered as normative, they may be quite helpful to the developer.

5.3.1. Generic coding rules and recommendations

5.3.1.1. Software architecture

[Rule] The software architecture shall be driven by the following keywords: Simplicity, Clarity, Modularity, Independence, Reusability, Maintainability.

- **Simplicity:** The simpler a system is, the more reliable it is. Apply the well-known KISS principle: Keep It Simple and Stupid.
- **Clarity:** The software architecture shall be immediately understandable. And so must be the source code. Anyone shall be able to understand the source code without effort, especially new contributors.
- **Modularity:** Break the design in smaller modules which are easier to maintain and understand.
- **Independence:** The modules should be potentially used independently of others. Modularity is not the synonym of independence. Some modular systems have so many dependencies between modules than the modularity is only an artificial division of a very big spaghetti-like module. Typical pitfalls which break the independence of modules are the usage of global variables, excessive usage of implicit state, module interdependencies, etc. Drawing a dependency graph of the modules can give a first impression; if there are loops in the graph, the independence principle is broken.
- **Reusability:** Reusability is the key to software cost reduction and speed of maintenance. A module shall be written in a generic way. It should be reusable in environments which are different from the original one. Each time you write a module, try to understand what is specific to your situation and what could be used outside of it. Write the module for the general case with a customized behavior from parameters and express your situations by applying parameters to your generic code. The DRY principle ("Don't Repeat Yourself") is an application of reusability.
- **Maintainability:** This is a corollary to all the preceding points. The maintainability is the key to success in software development, industrial or open-source.

5.3.1.2. Source code structure

This section describes the common rules on the structure of source files. Additional rules may be defined for various programming languages.

[Recommendation] A source file should be limited to a maximum of 500 lines. For complex modules which can be hardly split into smaller pieces, 1000 lines is the maximum.

These limits are raw file size, including blank and comment lines, not only source code (but excluding the standard legal headers).

As explained before, simplicity is a key to maintainability. If your file is larger than these limits, it is probably poor quality and you should consider redesigning it.

This is a recommendation rather than a rule because there are pathological cases where it makes sense to list many cases, or rich generic classes with a lot of features and doxygen comments. We must admit that there are some of them in the TSDuck source code.

[Rule] All source files shall start with a legal header which references the copyright and license information for the project.

All lines in the header must start with the appropriate syntax for comment lines in the language of the source file.

Application to TSDuck: the text of the legal header is the following:

```
TSDuck - The MPEG Transport Stream Toolkit
Copyright (c) 2005-2024, author's first name and last name
BSD-2-Clause license, see LICENSE.txt file or https://tsduck.io/license
```

[Rule] A source file shall contain at least 30% comments. More precisely, after removing the initial standard legal header and all blank lines, at least 30% of the remaining lines in the file shall contain non-empty comments.

As explained before, clarity is a key to maintainability. A new maintainer shall not need to analyze the code to understand it, reading it should be sufficient. The code, its structure and its environment shall be described in comments. Comments must be everywhere, not only in the file header. Write comments both for you and for future maintainers. Explain why you invoke a function; do not assume that the maintainer knows what it is. If you use a specific code structure for a good reason, explain it in comments to avoid a future maintainer to "optimize" it later, breaking your intent.

Comments shall explain the code, not paraphrase it. Do not explain *what* the code does (no need to explain that `i++` increments the variable `i`). Explain *why* it does it that way, at that point of the code.

This 30% comment requirement does not take into account the standard legal headers. The 30% proportion of comments shall apply to the description of the specific code and its environment, after removing all standard and non-significant headers. The standard comments which are processed by automated documentation tools such as Doxygen or Javadoc are included in the 30% comment requirement since they describe the actual code.

In the TSDuck source code, C++ source files (`.cpp` files) contain slightly over 30% comments and blank lines. In the C++ header files (`.h` files), this proportion is raised to 60%, mostly thanks to Doxygen comments.

[Rule] Comments should be written at the same time as the code and not after the code is completed.

When you write the code, you know precisely what you are doing at this specific time. This is the right time to explain what you are doing in comments. When the code is completed, some important thoughts you had when writing the code are already gone.

[Rule] Use source code self-documentation tools such as Doxygen or Javadoc.

These tools use specially formatted comments to produce a set of HTML pages or PDF documents with the documentation of all the code. With these tools, it is no longer necessary to write separate detailed design documents; the documentation is automatically extracted from the code.

This is both a cost reduction and a guarantee that the code is consistent with the documentation.

Document everything correctly. A common pitfall is to build the minimum comment structure so that the documentation tool does not complain. The result is a short and obscure description of a function and a list of parameters without description. This is useless. Generate the documentation and read it or have it read by someone else. If the result is not satisfactory, complete the self-documentation comments.

[Rule] Maintain the code self-documentation.

Another pitfall is to modify the code later without applying the corresponding modifications (if necessary) to the self-documentation comments.

5.3.1.3. Revision control system

[Rule] All source files shall be managed in a revision control system.

The TSDuck project is managed using Git. The reference repository is on GitHub within the organization named `tsduck` [TSDuck-Source]. Additional Git repositories are available in the same GitHub organization for unitary tests, third-party device drivers, etc.

[Rule] Each developer shall be accurately identified in the revision control logs. The identification includes the real name of the developer and his e-mail address.

This is necessary for accurate history tracking.

Example: Git tries to guess the real name and e-mail address of the author of each commit. However, the results are not always brilliant. A safe way to set the proper user identification for git is through the following shell commands:

```
$ git config --global user.name "Firstname Lastname"
$ git config --global user.email "username@domain.name"
```

The modifications are permanent. They are saved in the file `$HOME/.gitconfig`.

[Rule] Keep the source code repository clean and well organized. Specifically, ensure that dirty or non-original files are never pushed into the repository: temporary files, binary files, compilation products and more generally everything that can be regenerated from source files in the repository.

Most revision control systems have automatic ways of excluding dirty files, typically specifying file naming templates.

With Git, excluding dirty files is achieved through `.gitignore` files. See the man page `gitignore(5)` for more details.

Executables and objects are compilation products, not sources. They should be ignored in most cases. But UNIX executable files have no defined suffix. It is not possible to set up a global rule to ignore them (like ignoring all `*.exe` in Windows). Thus, in each directory where one or more executables are produced, there must be a local `.gitignore` file which contains the exact names of the executables to ignore.

Binary objects or libraries (`*.o`, `*.a`, etc.) are ignored by the global rule, at the root directory of the project. When a directory contains a third-party binary for which no source is available, this binary must not be ignored. There must be a local `.gitignore` file which contains a "not" (`!`) directive to avoid ignoring this file.

Example: Content of a local `.gitignore` file illustrating these rules:

```
# Executable files to ignore
my_app
other_app

# Third-party binary library to include in the repository (not to be ignored)
```

```
!libthird.a
```

5.3.1.4. Internationalization

[Rule] All elements in a source file shall be written in English. Comments are in English. Identifiers use English words.

Whether you like it or not, English has replaced Latin as the standard international communication language for decades. As of today, Klingon is not yet eligible as a valid international communication language.

[Rule] Text messages and output shall be written in English.

Generally speaking, if there is a need for internationalization, external localization files shall be used for non-English messages. There are standard methods for this in the various environments and programming languages. This rule is particularly important for GUI applications which are used by the general public.

Since TSDuck is a very technical project, it is used by engineers who have at least some understanding of English. Consequently, there is no internationalization of the messages in TSDuck.

5.3.1.5. Modularity and compatibility

[Rule] Modularity is a contract definition. Define the contract and respect the contract.

A module shall be clearly defined by a public interface. This interface defines a contract between the module and its users. This contract shall be clearly described in the module interface, typically in the self-documentation comments. Whenever the module is modified, respect the original contract.

[Recommendation] Always preserve ascending compatibility.

A contract is a contract and you should not be not allowed to break it. If you make substantial modifications in an existing module, you may add features but you should not remove features or change the contract of existing features.

As a general rule, you must ensure that all existing code that could have used your module still compile and work correctly without modification.

For projects with a long history, this recommendation is not always easy to apply. When adding many features or restructuring poorly designed modules, preserving ascending compatibility may create chaos in the code, making it more difficult to maintain in the future.

So, the right balance shall be found. It also depends on the nature of the project. A standard library which is used by thousands of developers and maintained by a large team shall prioritize ascending compatibility. Libraries with very few maintainers and not too many users may choose to prioritize the maintainability of the code, at the expense of ascending compatibility.

TSDuck belongs the latter category. TSDuck went through several code cleanup cycles, taking advantage of new standard features in C++11 or C++17. In these operations, old features were removed to simplify the code base.

[Recommendation] Use Object Oriented Design (OOD) wherever it is possible.

TSDuck is developed in C++ and takes advantage of all OOD features of the language.

The Java and Python bindings follow the same principles.

[Rule] Do not expose implementation details.

The interface of a module or class shall expose only public declarations which are part of the contract. Implementation details shall be hidden. Not only undocumented, but logically hidden. Even if you do not document them, someone will discover them in some definition or header file and may use them. Application code trying to use implementation details should not compile.

In C++, implementation details shall be implemented as *private* fields and methods in classes.

[Rule] Expose opaque data types only.

This is an application of the preceding rule. If you need to expose the existence of a data type, you should hide its internal structure. Some fields may be strictly private, for internal use of your module. Some fields may be read-only for the user. Some fields may be modifiable by the user but with some control from your implementation.

And in the future, you may need to redesign the structure of the data type but you will also need to preserve the contract, preserve the logical access to these fields.

Thus, do not expose public fields. Use *accessors*, also named *getter* and *setter* methods.

If you think that this is a loss in performance, you are wrong. Most compilers have the ability to inline the code of a function. If you define an inline accessor method, the generated code will be only a data access. You get the same performance as a public exposure of the field but with a better contract and better maintainability of the code.

If you think that coding and documenting accessors is boring, well, you are right. But many IDE's can do most of the work automatically for you.

[Rule] Global variables are evil. Do not use them.

Global variables break the contract of a module. They also break the independence principle. And they also break the reusability principle.

[Recommendation] Static variables are evil. Avoid using them.

Static variables (in C parlance) are slightly different from global variables since they are not shared between modules. But in the OOD approach, a static variable is shared by all instances of its module. Most of the time, this breaks the reusability principle.

If some kind of persistent unique data is necessary, prefer the singleton design pattern over the static variable.

5.3.1.6. Naming conventions

This section lists a few generic naming conventions which apply to all programming languages. For rules which specifically apply to C++, see section 5.4.2.3.

Usually, naming is addressed by interchangeable *conventions*. However, these conventions shall be managed by higher level common sense rules.

[Rule] Use meaningful names which are immediately understandable by the reader or maintainer.

Example:

```
int logicalFileIndex;    // OK, self-explanatory
int logical_file_index;  // OK, just another naming convention
int lfi;                 // Questionable. Is LFI a really widely used acronym for Logical File Index?
                         // Or is it just your own convention?
int i;                   // OK if this is just a loop index in a small function.
                         // WRONG if this is a Logical File Index
```

[Rule] Do not use names which differ only in letter case, differ by one character only or too similar.

This is confusing for the reader and error-prone for the maintainer.

Counter-example: The following sets of identifiers can be easily confused.

```
int broadcastIndex;  // lower case 'c'
int broadCastIndex;  // upper case 'C'

int counter;
int counters[10];    // trailing 's'
int index1:          // digit 1
int indexl;          // lower case 'L'
int indexI;          // upper case 'i'
```

5.3.1.7. Coding principles

[Rule] RTFM. Read The F... Manual. Read the documentation of all functions, classes or libraries you use.

This is such an obvious rule that it should not be worth mentioning. However, experience proves that it is not always followed.

Each time, *all the time, really*, read the corresponding documentation before invoking a function. Even if this is a function you personally developed some time ago. Understand the error cases, the returned error codes, the side effects, the exact usage of the parameters, etc.

Get prepared for reading documentation. Have a shell window ready for a `man` command, a browser with prepared bookmarks pointing to the online help, Doxygen or Javadoc pages. When coding on a secure network which is not connected to the Internet, make sure that a copy of all online documentations is available on a server within the secure perimeter.

[Rule] If it ain't broken, don't fix it!

Sometimes, a programmer reads some code and finds it clunky. Some programmers cannot resist the temptation to "fix" the ugliness of the code. This is a dangerous practice. In some cases, this modification may introduce a bug and turns a clunky but perfectly functional code into a buggy one. In front of an existing and operational code, in the absence of other intent such as refactoring or generalization, the question shall never be "is it elegant?" but "does it work?" And if it works, do not modify it.

As a consequence of this rule, do not rewrite existing code for the sole purpose of applying coding guidelines. When the coding guidelines evolve, it would be both counter-productive and dangerous to review and modify the existing code base to retrofit the new coding guidelines. The coding guidelines apply only to new code or code which is modified for valid maintenance reasons.

[Rule] Do not uselessly over-optimise without profiling.

Do not make hypotheses on the performance of the code. Do not write more complicated code because you think it will be faster. You do not know what the compiler will generate; only the compiler knows. Let the compiler optimize the code. Recent compilers are very good at optimization.

Even if a local construct is slightly more efficient, you do not know the actual impact on the global execution time. It is dangerous to write more complicated code to gain 0.001% of execution time. On the contrary, because the code you write is more complex, it is more error-prone and more complex to maintain.

Write the code using a clean structure. When the final application code is ready, do some profiling in the real context of the application. This will give you the actual bottlenecks in the code. Then, you will know which parts of the code are worth optimizing.

[Rule] Write endian-neutral code. Make sure your code works identically on big-endian and little-endian processors.

Never assume that a piece of code will work forever on the same type of processor. Even embedded code will eventually be ported on other platforms.

Some CPU architectures are little-endian (Intel), some are big-endian (SPARC, IBM s390) and some can work in either mode (Arm, PowerPC, MIPS). To make provision for future ports, make sure that your code is not dependent on a specific endianness.

The endianness applies only to integer, floating-point data, UTF-16 and UTF-32 characters strings. ANSI and UTF-8 characters strings, cryptographic keys and other types of raw data do not have any endianness; they are just suites of bytes.

The endianness only matters for external data interchange or storage. Write or use *serialization* and *deserialization* libraries to switch back and forth between the native and external representations of data. Each programming language has specific ways of handling this gracefully.

Do not make any assumption about performance. For instance, if you handle external data in little-endian representation and your code targets a little-endian CPU, do not avoid serialization routines simply because you think that this would add a useless overhead. Well-designed serialization libraries do not add any overhead when the native and target representations are identical. In this case, the serialization and deserialization routines are inlined empty functions. But using them in the source code is beneficial. When you port the code to another platform with the opposite endianness, the program will work correctly without modification.

It is difficult, and sometimes impossible, to test that a program is truly endian-neutral, especially when the native and external data representations are identical. When possible, test your code on a little-endian and on a big-endian platform from the beginning.

In the case of MPEG transport stream, most data are serialized in big-endian order. TSDuck provides low-level serialization functions for individual values and high-level classes for serialized data buffer management. Make sure to always use them.

Nowadays, most commonly used platforms are little-endian. The only well supported big-endian platform is IBM s390x (this is one of the three best available Linux distros, with Intel and Arm). It is difficult and expensive to obtain IBM s390x hardware. However, TSDuck is periodically tested on emulated IBM s390x using `qemu`.

[Recommendation] Use design patterns.

A number of design patterns are available in the literature. These patterns are proven constructs. Do not invent hazardous new designs, use proven design patterns. See [GAMMA] for instance.

5.3.1.8. Secure coding

[Rule] Do not assume anything.

This is the common root of many security flaws.

- Do not assume that the caller has checked a piece of data, check it.
- Do not assume that this object is in such state because you think that this is logical in this context, check it.
- Do not assume that a connection is established, check it.
- Do not assume that a buffer has such minimal size, check it.
- Do not assume that this pointer is non-zero, check it.

Etc. etc. etc.

[Rule] Do not trust any other code.

This is a corollary of the preceding rule: do not assume that a function does what its documentation says it should do. Specifically, if there is a simple way to improve the robustness and security of your code by checking what the other code did or returned, then check.

[Rule] Avoid temporary solutions or quick & dirty fixes.

If you are tempted to write a temporary fix to a problem, do not do that. Take your time and implement the good and final solution right away. Otherwise, because of the daily workload, you will delay the final solution, then you will forget about it and finally your temporary (and most certainly insecure) fix will become permanent.

Counter-example: A well-known product had a random number generation routine which was used to initialize cryptographic operations. The body of this routine was merely something like `return 3;`. It is clear that the developer intended to implement a proper solution later. But it never came.

[Rule] Inputs are evil. Never trust inputs. Check all inputs.

When you write a module, the inputs you receive come from an untrusted world by definition. You must always check all inputs. Addresses shall be validated (check null pointers, alignments and range constraints). Numerical values shall be checked for allowed ranges. The size of memory areas and buffers shall be checked. C-strings shall be checked for null termination. The format of special strings shall be checked (date and time, request, etc.)

All functions shall check the validity of all inputs, always, without exception.

[Rule] Check all intra-application inputs also, at all levels.

Do not assume that the inputs of a function are valid simply because the caller is part of the same application. The caller code may have bugs. The caller code may be modified later and introduce new bugs. Your function can be reused in another context (remember that we encourage reusability) and the new caller may have bugs.

[Rule] Buffer overflow is your enemy. Check buffer sizes all the time.

If there is any theoretical, practical or syntactical possibility that some data could be larger than the memory area where you are going to copy it, check the data size, the index or whatever.

Use the capabilities of your programming language to define safe buffer objects with well-designed primitives instead of using raw memory buffers.

[Rule] Pay attention to the stack usage in a multi-threaded environment.

In a multi-threaded environment, all threads are not equal. The size of the stack of the main thread is often "unlimited", or at least extremely large, in the multi-mega-byte range. On the contrary, the size of the stack of all other threads is constrained. The default size depends on the platform but it is in the multi-kilo-byte range, much smaller than the main thread.

When you write a function, always remember that it will be potentially used in various environments, single-threaded or multi-threaded, with small stacks or large stacks.

Allocating very large objects on the stack may lead to unpredictable results if they overflow the current stack. If you are lucky enough, the implementation allocates non-accessible *guard* pages of memory before and after the stack of all threads. Thus, if you overflow a stack by a few bytes, the program will crash with an access violation error. However, if you allocate a very-very large object on the stack, the start of the object may step over the guard pages and lie into the stack of another thread. Thus, accessing the highest parts of your local object, you will corrupt local objects in other functions executing in another thread. Finding such a bug is a nightmare.

[Rule] Limit of size of local variables and all data which are allocated on the stack.

This is a corollary of the previous rule. Allocating large objects on stack may overflow the stack of a given thread. Large local objects shall be dynamically allocated on the heap and released before going out of scope.

[Rule] Always explicitly specify the size of the stack of all threads you create.

This is another corollary of the same rule. If you do not explicitly specify the size of the stack of a thread, it will get the default size which is defined by the implementation. And this size is typically different from one platform to another. This means that your code is not deterministic. It may run on one platform and crash on another.

Be very conservative when computing the required size of the stack of a thread. Do not try to be too clever. It is useful to carefully compute the accumulated sizes of all local objects in all nested functions. But what should you do with the result? Clearly, do not use it as the stack size. The stack is cluttered with many stack frames or temporary working data which are allocated by the compiler. This extra size depends on the platform, the ABI, the compiler, the compilation options, etc. In practice, you cannot accurately compute the required stack size. So, use various sources of information. Compute your local data sizes, run tests and measure the maximum stack size and, at the end, double it just in case.

When you define an API, there are cases where the user passes the address of a *handler* or *callback*, some code which is developed by the user but which is invoked by your library. If your library creates internal threads and some user handler or callback is invoked in the context of these threads, how do you compute the stack size of the threads you create? This cannot be transparent to the user. In the API of your library, you must give a way to specify the required stack usage of the user handler. The user shall be able to pass the handler code address *and* the corresponding required stack usage of his handler. Internally, to compute the stack size of your threads, you must add your own usage and the user's usage.

[Rule] Always check error codes. Take appropriate action and error processing.

Most functions return an error code or some indication that the processing failed somehow. Read the documentation of the function and always handle error cases.

This is specifically important for memory allocation routines.

But there are less trivial cases.

Example: Consider file management.

- We probably always check the error code which is returned by a file opening or creation operation because we anticipate the risk that the file does not exist or we have no permission to create it.
- When reading from a file, we generally always check the returned error code because we can reach the end of file at any time.
- However, how many of us check the error code which is returned by a *write* operation? Nonetheless, writing to successfully opened files can fail at any time, because the volume is full, because the device has been removed, etc. In some cases, failing to write critical data such as transaction information or cryptographic keys without noticing can have disastrous effects.

[Rule] In case of error, fail safely.

When writing complex functions or module, you may invoke hundreds of external functions. Any of them can return an error situation. In the meantime, you may have opened, allocated, locked or somehow initialized many resources. If each error processing is individual, the code is bloated and potentially insecure.

Write a common error processing path which checks which resources should be cleaned and which safely cleans them.

Object-oriented languages can greatly help you in this process. Carefully designed destructors should always automatically clean up resources.

[Rule] Don't write spaghetti-like code, write state machines.

Avoid deeply nested `if … then … else` structures. Avoid multiple processing of the same cases in various places of the code. When applicable (and it is more often that you may think), redesign the code as a state machine.

[Rule] Physically clean up secure information from memory.

Passwords, encryption keys and less critical information are stored in memory while being processed. Once the local processing is completed, overwrite the corresponding memory with binary zeroes or random data. If some memory was dynamically allocated on the heap or on the stack, clean it up before freeing it from the heap or returning from the current function.

[Rule] Make sure that secure information is never accidentally written to disk in clear form.

Most high-level operating systems such as UNIX or Windows use virtual memory. When the physical memory of the system is about to fill, the system swaps or paginates, which means that some memory pages belonging to running processes are temporarily written to disk. If those memory pages contain sensitive data such as passwords or cryptographic keys, this becomes a security breach.

You must ensure that this never happens.

With virtual memory systems, make sure that all data buffers which contain sensitive data are locked in physical memory. Thus, the corresponding memory pages will never be written to disk.

Example: On Linux systems, use `mlock()` to lock a memory area in physical memory.

[Rule] Sensitive information shall be exchanged or written to disk in encrypted form only.

Persistent sensitive information usually needs to be stored or exchanged. Storage shall be performed in encrypted form only. Exchanging sensitive information shall be performed using an approved cryptographic protection layer, either individually or using a secure tunnel such as TLS 1.3.

All versions of SSL and versions 1.0 and 1.1 of TLS are considered insecure. TLS 1.2 shall be considered as deprecated.

Encryption keys shall be adequately protected. Passwords and other similar data which need to be compared but not retrieved in plain form shall not be stored. Instead a salted hash is stored.

Use approved encryption algorithms only. Also use approved security protocols and approved key management methods only. Never use your own encryption algorithm or security protocol. In case of doubt, ask an expert.

[Rule] Usage of encryption shall be validated by a cryptologist or security expert.

Cryptography is a smoking gun. Encryption and security protocols are hard to secure. Naïve implementations or careless straightforward usage of encryption primitives are known to be vulnerable to various types of attacks.

All the following mechanisms shall be validated by a cryptologist or a security expert before being used:

- Encryption, decryption.
- Signature generation and verification.
- Encryption keys and other random data generation.
- Key management and protection.
- Security protocols and secure data exchange.
- Usage of hash, nonce and challenge.
- Implementation of cryptographic primitives, choice of cryptographic libraries.

[Rule] Take care of reentrancy in reusable components.

Basic reusable components increase the productivity and the maintainability of the code. However, using the same component from concurrent threads may lead to race conditions and crashes.

Clearly document whether a component is thread-safe or not.

You may internally implement the required locking feature from the beginning to make the component thread-safe. However, this may have several drawbacks. There may be useless performance degradation in applications which do not share the component in multiple threads. Additionally, the concepts of threads and their associated locking features may be quite different between systems and it is not always easy to write portable thread-safe code.

One option is to write a simple and non-thread-safe component and then write a thread-safe variant of the component. Do not re-implement the component in the thread-safe variant. Encapsulate the calls to services of the non-thread-safe component in the services of the thread-safe variant.

Object-oriented languages with template or genericity features such as C++ and Java may help to write a unique portable thread-safe implementation. The component focuses on its core features only and accepts synchronization primitives or objects as template or generic parameters.

[Rule] Use the principle of "least privilege".

This is a fundamental rule of secure software. It means that a code shall not run with any privilege it does not absolutely require for its nominal execution. In case of security breach, when an attacker takes control of your code, this attacker shall not be able to take advantage of extra privilege.

Example: If an application manipulates files on behalf of a specific user, the process shall have no more privilege than this user. More generally, the application shall run with the identity and privileges of the user it works for.

[Rule] Do not run any code using the root account or any kind of privileged user account.

This is a corollary of the previous rule. Only the operating system or low-level system services need system-wide privileges. There is usually no good reason to use the *root* or *administrator* account or whatever privilege to run an application. If you really think that you need to be *root*, you are wrong. If you really insist, see next rule.

[Rule] Drop privileges as soon as possible.

This is another corollary of the same rule. There are rare cases where your application needs to have elevated privileges for a few very specific operations. In that case, use the following scenario:

- Start the application with the strictly minimum set of privileges which are required to perform the privileged operations.
- Perform the privileged operations right at the beginning of the application.
- Drop all extra privileges immediately after.

Example: An HTTP server shall listen on TCP port 80 by default. This port number lies in the system range of ports. On UNIX systems, you need to be root to open this port. If your application implements an HTTP server on this port, it shall be started as root. At the very beginning of the application, opens the port 80. Do not do anything else. Do not work on files or system resources which can lead to security problems if the application is misused. When the TCP port 80 is successfully open, immediately switch to some non-privileged user account like `httpd`. The open file descriptor of the socket on port 80 is still valid and can be used from the unprivileged user account.

[Recommendation] Lock your application in an isolated enclave without access to the rest of the system.

This is yet another corollary of the above rule. When your application is successfully attacked and control is taken by an attacker, the "pwned" application shall not be able to access the rest of the operation system.

Define the required environment for your application; define the corresponding boundaries and lock the environment within these boundaries inside a subsystem without access to the rest of the system. The available confinement mechanisms depend on the operating system.

Example:

- `chroot()` on Linux and some other UNIX systems.
- Linux containers (LXC, Docker).
- Virtual machines under control of a hypervisor (VMware, VirtualBox, XEN, HyperV, etc.)

5.3.1.9. Software evolution

[Rule] Avoid copy/paste source code. Use refactoring instead.

Most of the value in software engineering comes from the software reusability and its corollary, development and maintenance cost reduction. Software reusability means generic and reusable code. But writing clean generic code right from the beginning is not natural. The process of producing generic code is typically extending and generalizing existing code into generic software.

This is named *refactoring*, one of the most important concepts in modern software engineering.

Creating new code from existing code using copy / paste is duplicating, not refactoring. Duplicating code means duplicating bugs, duplicating maintenance costs and more generally increasing the entropy of the system.

Remember: Copy/paste propagates bugs but does unfortunately not propagate fixes.

[Recommendation] Avoid creating too many branches in the source code repository.

The presence of multiple persistent branches in the source code repository is a wide-scale variant of the copy / paste syndrome. It multiplies the maintenance costs and ruins the source code consistency. Fixing old bugs becomes a nightmare because of the multiple and divergent branches where the bug is present. Creating a branch may save some time on the very short term but the extra cost on the mid and long term is overwhelming.

Exception: Creating short-lived branches for exploration, integration or tests may be accepted under the firm condition that the branch will be rapidly merged into the main trunk and deleted.

5.3.1.10. Compilation errors and warnings

[Rule] Use the most paranoid "warning mode" of the compiler and fix all warnings without exception.

Compilation warnings are never gratuitous. A compiler warning always draws attention to a potential bug. Even if the code works as expected on the tested platform (i.e. the set of compiler & target execution system) the incriminated section of code may fail on another platform. Ignoring tons of uncorrected compilation warnings is a very bad practice. When porting to a new platform, you may have to fix hundredths of bugs that would have been avoided if the warnings were considered in time.

Even warnings which appear to be really harmless after analysis are dangerous because they create a culture of ignoring warnings. A listing of hundredths of "harmless" warnings hides new warnings which may indicate actual bugs.

The most paranoid warning mode depends on the compiler. Different compilers have different options. Some compilers find warnings that some other compilers don’t. As a general rule, fix all warnings from all compilers. Your code must not generate any warning on any platform.

[Rule] Turn compilation warnings into errors.

Many compilers have an option which turns warnings into errors. Use it in all code without exception. Thus, the presence of one single warning prevents the code generation.

[Rule] Understand an error or a warning before fixing it.

This seems obvious but this is too often not applied in practice.

Fixing a compilation error or warning is not simply making it disappear. It is tempting to simply find a modification in the source code which gets the compilation message away. But is this the right fix? Does this remove the potential bug that was behind? Did you even understand the potential bug?

Example: You may get an error about a pointer type mismatch or a warning about a comparison between signed and unsigned integers. Of course, casting one object to the appropriate type removes the message.

And sometimes, this is the appropriate fix. But sometimes, this is not.

You need to deeply understand the situation to decide whether you should use a type cast or modify the code structure. Do you understand why `if (a < b)` produces a warning when `a` is a signed integer and `b` an unsigned one (or the opposite)? Do you understand that the underlying risks include an index mismatch leading to a buffer overflow, one of the major sources of security breaches?

If you don’t, do not fix the code yourself, get help first.

5.3.1.11. Makefiles

This section describes the rules to use GNU Make to build native software on UNIX systems (Linux, macOS, BSD) only.

[Recommendation] The command `make` should be usable at any point in the source tree. A makefile shall be present in all directories of the source tree. The name of the makefile shall be `Makefile`.

For most projects, the source tree is complex. Sub-projects, sub-systems, test suites are implemented as sub-directory trees. Some developers work on specific sub-trees while others work the entire project tree (integrators for instance).

Consequently, running `make` should be possible anywhere in the source tree.

This recommendation applies to "logical entities" only. A library may contain too many files to use one single directory. The source files are spread all over a tree of subdirectories. However, if the compilation of the library is homogeneous, it is acceptable to have a makefile at the root of the library source tree only.

[Recommendation] As a general rule, when run in a specific directory, the command `make` shall recursively operate on all subdirectories.

In the source tree, any directory and its entire tree of subdirectories shall be considered as a consistent subsystem to build. See the preceding rule.

Consequently, running `make` somewhere in the source tree shall build the corresponding subsystem, meaning the current directory and all subdirectories, recursively.

The setup from a common file shall provide the required tools to simplify this process (see next rule).

When a directory is simply an intermediate level where there is nothing to build locally, simply include the common file and nothing more. The first target in the common file shall recurse into the subdirectories.

[Recommendation] All makefiles shall include a common file. This file contains the common set of rules, variables, options and targets which are used by all projects.

The actual content of this common makefile is outside the scope of this section. Use a relative path to include the common file since the source code repository can be checked out at different locations on different systems.

Example:

```
include ../../Makefile.inc
```

In the TSDuck source code, there is a file named `Makefile.inc` at the top level of the source tree. It is included by all makefiles, at all levels.

[Rule] Do not try to explicitly enumerate the header file dependencies of C++ files. Setup an automatic dependency resolution system.

It is difficult to manually maintain such dependencies. Include directives are added and removed from nested header files and there is no reliable way to manually maintain this on the long run. This can lead to subtle bugs when a module is not recompiled simply because its dependency on a recently updated header file was missing.

The idea is to automatically generate a text file `module.dep` which contains a makefile dependency rule for each `module.cpp`. The `module.dep` file lists all header files which are directly or indirectly included by `module.cpp`. Generating such a file is a feature of the C preprocessor (at least with GCC and Clang).

All `.dep` are automatically included in the makefile. The makefile automatically regenerates obsolete `.dep` before including them. This is possible using the GCC or Clang compilers and GNU Make.

In TSDuck, this system is implemented in the top-level `Makefile.inc`.

Other building systems such as `cmake`, `qmake`, MSBuild (Microsoft Visual Studio) implement their own automatic dependency resolution system. Use the one that suits you or use the above method with plain makefiles, but do not try to maintain the dependencies manually.

[Rule] Do not try to explicitly enumerate all source files. Setup an automatic file discovery system.

This rule is similar to the previous one. In a large system, with many source files (there as 2300 source files in TSDuck), it is easy to forget to add new or moved files.

Some source files are not explicitly referenced in the application. In TSDuck, this is the case of all descriptors, tables, filters or plugins. There are hundredths of them and they register themselves in a central repository upon initialization. This means that omitting to build such a source file will not prevent the application from being built. However, subtle bugs will emerge from the absence of a module.

This is why this rule is not a matter of lazyness, avoiding to list source files. This is a functional and security requirement.

Tools such as GNU make have features to explore the source files. There are heavily used in the TSDuck makefiles. Simply adding a source file somewhere in the directory tree under `src/libtsduck` automatically includes it in the build process of the TSDuck library. Similarly, adding a source file in `src/tsplugins` or `src/tstools` automatically builds a new plugin or tool.

Automatic discovery of source files is sometimes a disputed topic. Some developers advocate the explicit listing of all source files in `Makefile` or `CMakeLists.txt`. This is usually the result of a lack of experience in very large and dynamic projects.

5.3.1.12. Unit testing

[Rule] Use unitary and non-regression tests automation.

Use a unitary testing framework (Cunit, CppUnit, Junit, etc.) For each module, build the corresponding unit tests and integrate them in the framework. Cover most common legal, illegal and limit cases in unit tests.

With TSDuck, the test suite is split in two parts. The first part addresses low-level unitary tests, typically for C++ classes, using a dedicated testing framework named `TSUnit`. The second part includes non-regression tests for TSDuck commands and plugins. See section 4.1 for more details.

Use a continuous integration framework (Jenkins, for instance) to run all unitary tests nightly and report failures.

With TSDuck, continuous integration is managed using GitHub Actions. See section 4.2 for more details.

[Rule] Use a Test-Driven Development (TDD) approach.

Using TDD, each module is written in parallel with its unitary tests. Modern methods such as Agile or eXtreme Programming (XP) advocate that the module and its tests shall be developed in parallel by distinct developers.

The typical scenario is the following:

- Create the module in the main development area. The module is initially empty. Make sure it is compiled and inserted in the main product (library, executable, whatever).
- Create the corresponding test in the unitary tests area. Make sure that the test suite is correctly built and executed, although the test suite is initially empty.
- Develop one feature of the module.
- Develop the corresponding unitary tests.
- Run the tests and debug all potential problems.
- Write another feature and its tests.
- And so on.

This approach has several advantages:

- The developer is forced to define a clear API of the module from the beginning. Otherwise, the unitary tests are difficult to define.
- In case of failure, the unitary tests provide a simple environment to debug.
- The module is immediately inserted in the automated non-regression tests. During further developments of the module, if you later break something that was previously tested, the new bug will be automatically caught.
- The integration time is reduced since each module is individually tested at development time.

[Rule] Add new tests for each fixed bug.

Each time a bug is found beyond unit testing, in integration phase or even production, there is a bug in the code. But there is also a bug in the unit test suite because it failed to catch the bug in the code.

The proper fix scenario is:

- Fix the unit test suite first: Add a unit test which exhibits the bug.
- Run the unit test suite and verify that it fails (the bug is caught).
- Fix the application code.
- Re-run the unit test suite and verify that it passes.

This method is also well suited to fight the "resurrecting bug syndrome". We have all encountered situations where a bug was fixed in the past but reappears later. This is usually due to some human error with the configuration management system. By enriching the unit test suite with test cases for all known bugs, we are able to detect the resurrection of outdated buggy versions.

5.3.1.13. Integration of open source software

[Rule] Always check the license of Free and Open Source Software (FOSS) before using it. In this context, "using" means integrating and linking with FOSS modules as well as copy / paste of FOSS source code.

TSDuck is open-source software. So, it may seem weird to worry about integrating other open source software. However, all open source software are not equal in terms of usage and integration because they use distinct, and sometimes incompatible, licenses.

Today, a vast amount of generic or reusable software is available for free (*"free as a beer"*) in the form of Free (*"free as in freedom"*) and Open Source Software.

Several topics must be studied when using FOSS:

- The license of *your* software (FOSS or not).
- The license of the FOSS you plan to use.
- Your usage of that FOSS.
- The planned deployment of your software.

Many different types of licenses exist for FOSS. The Wikipedia article *"comparison of free and open source software licenses"* lists more than 40 different licenses. Some of them are obscure or written by legal illiterate programmers and are difficult to understand. Actual legal cases which are based on these licenses are disputed or even contradictory. Therefore, do not underestimate the difficulty.

Generally speaking, there are two main categories of FOSS licenses: *permissive* licenses (BSD License for instance, as used by TSDuck) and *invasive* or *copyleft* licenses (GPL for instance). While the former may be safely used in proprietary software with some care, the latter cannot.

The way the target FOSS is used is also important. A license may apply differently when the FOSS is copied / pasted in source form, statically linked or used as a side component. The LGPL variant of the GPL is typically designed for dynamically linked libraries. But some libraries are LGPL while others are GPL. And some cases are borderline. Dynamic reference (`dlopen()`) of GPL’ed shared libraries on Linux is a disputed subject for instance.

Because the TSDuck source code is released under the BSD-2-Clause license, it is usually permitted to copy and paste a few lines of code from another project which is released under another permissive license. This license must be compatible with the BSD-2-Clause license. Don’t forget to check and take any requested action, such as mentioning the original author, software, copyright or license. However, it is not permitted to copy source code which is released under a restrictive license such as GPL or LGPL. These licenses are not compatible with permissive licences such as the BSD licences.

Finally, the type of deployment of your software matters. For proprietary software which is distributed outside the company where it is developed (commercially or even for free), the license of the FOSS applies. But for internal proprietary tools which are never distributed, you may usually do what you want. This rule does not apply to community projects such as TSDuck but it is worth remembering.

As a rule of thumb, FOSS with a permissive license such as TSDuck may freely use other FOSS with another permissive license. However, using FOSS with a restrictive license must be done with care:

- LGPL libraries can be accessed through dynamic linking only. Static linking or dynamic loading (`dlopen()`) are prohibited.
- GPL applications can be started through `exec()` system calls or equivalent.
- GPL libraries cannot be used.

As an example, `libreadline` is one of the few libraries with a GPL license, not LGPL. As a consequence, a BSD-licensed software such as TSDuck cannot use it. Fortunately, there is an equivalent library named `libedit` with almost the same features as `libreadline` and a BSD license. Therefore, TSDuck uses `libedit` instead of `libreadline`.

In practice, `libedit` has been developed only because of the unfriendly license of `libreadline`. Similarly, many new software now avoid the GPL license.

In all cases, this subject is too complicated for us, the developers. In case of doubt, get a legal advice first.

5.3.2. Generic coding conventions

This section is present to fulfil the required separation of immutable *rules* and *recommendations* from potentially replaceable *conventions*, as explained in section 5.2.

5.3.2.1. Control characters

[Convention] Use exclusively line-feeds (LF, '\n', 0x0A) as line delimiters.

This is the usual UNIX vs. Windows or LF vs. CR-LF line format. All compilers on Windows understand files with LF as line delimiters. On the contrary, some compilers or interpreters in the UNIX world have problems with CR-LF line delimiters. The `bash` shell, for instance, considers the CR as part of the line.

Most editors and IDE’s have an option to force the usage of LF as line delimiters, even on Windows systems.

Automatic clean-up scripts should be used on a regular basis on the code base to convert CR-LF sequences into LF.

Exception: When working with Git, it is possible to automatically switch back and forth between LF and CR-LF during check-out and commit on all or selected types of text files. This feature is managed though the `autocrlf` configuration option and `.gitattributes` files in the repository tree. If you use that feature, make sure that text files are committed with LF lines. If you want some Windows-specific text files (such as `.sln` or `.vcxproj` files) to be committed with CR-LF, make sure to properly reference them in a global `.gitattributes` file, as used in TSDuck.

[Convention] Do not use the tabulation character, use spaces.

In source code, the indentation is essential. It is tempting to use tab characters to indent. However, the size of a tab character may vary between tools. Most system tools use 8 characters per tab. But many IDE’s use 4 characters (the usual single indentation width). When editing a source file, depending on the typing and automatic indentation features, the result is often a mixture of tabs and spaces in the indentations. When such a file appears correctly in an IDE with 4 spaces per tab, for instance, the same file appears totally messed up in an editor with 8 spaces per tab.

Most editors and IDE’s have an option to force the usage of multiple spaces instead of a tab.

Automatic clean-up scripts should be used on a regular basis on the code base to convert tab characters into spaces (but a uniform tab width must be applied and may not match the original environment).

Exception: The presence of an actual tabulation character is required in some specific file formats such as makefiles. But most editors and IDE’s can handle this exception.

5.3.2.2. Character encoding

There is always some misunderstanding between the characters in a text and their binary representation in a file. In the most general case, a character is universally represented by a 32-bit UNICODE value. But files are rarely represented in UTF-32 format.

In practice, UTF-32 is untransformed UNICODE. Some environments, such as Windows, erroneously name "UNICODE" a 16-bit representation which is actually UTF-16 with surrogate characters. UTF-16 means Unicode Transformation Format - 16 bits.

When a file is transferred between systems or accessed from a shared disk by multiple heterogeneous systems, the physical content of the file remains the same but the various operating systems or production tools often interpret this physical content in different ways, leading to unpredictable results.

The following rules attempt to mitigate this problem.

[Convention] Restrict the character set of source files to the 7-bit ASCII set.

There are few reasons to use characters outside the 7-bit ASCII set in software source files. The syntax of all modern programming languages uses ASCII only. All identifiers and comments shall be written in English which is ASCII only. Any people name in comment (author, developer, etc.) shall be used without accent or local "decoration".

Avoid "national pride" of non-English characters. Contributors shall accept to drop them from their name, if any. As an example, the main author of TSDuck is named "Lelégard", with an accent on the second "e", but the name is simply stored as "Lelegard" (no accent) in all source files. All contributors are kindly requested to accept this rule.

[Convention] Use UTF-8 encoding when internationalization is required.

It is sometimes required to use international texts. Always uses specific internationalization files for non-English languages. The actual format of these files depends on the toolset. Unless specified otherwise by a toolset, always use the UTF-8 encoding for these files. When present, the option "UTF-8 without BOM" should be used.

BOM: Byte Order Mark, a feature of UTF-16 and UTF-32 which is normally useless in UTF-8. A specific leading binary sequence has been defined as "UTF-8 BOM". It can be used to assert that the file format is UTF-8. However, several tools are unable to process this sequence correctly. Therefore, it is recommended to avoid it.

When the syntax of a file format allows the explicit specification of the character encoding, use it to specify UTF-8.

Example: All XML files shall be saved in UTF-8 format and start with:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Exception: Some tools may impose or generate files using their specific encoding. Always use the character encoding which is the safest one for such tools.

5.4. C++ coding guidelines

This section defines the coding guidelines which are specific to the C++ language.

5.4.1. C++ coding rules and recommendations

5.4.1.1. Language selection

[Recommendation] Unless it is impossible for a very good reason, do not use C, use C++.

C is an old and unsafe language. C++ is much safer than C. It is possible to improve the safety of C using all advanced features of ANSI C99 and the most paranoid warning mode of the C compiler, but C will never be as safe as C++.

Good reasons to use C instead of C++ include:

- Maintenance of a legacy application written in C.
- Embedded system development on a platform without C++ compiler.

All other reasons are usually bad reasons.

If you need at least two essential reasons to use C++ instead of C, one is named *constructors* and the other one is named *destructors*. These features are the essential bases of safe coding and they are not available in C.

For the record, the very first version of the ancestor of TSDuck, back in 2005, so-called "TSDuck V1", was developed in C. When the code base grew, this quickly appeared to be a fatal mistake. All the code was scrapped and rewritten in C++, starting "TSDuck V2".

[Rule] Performance is never a good reason for not using C++.

There is a common misconception that C++ code is slower than C code. This is wrong. The confusion comes from the fact that C++ has much more features than C and seems more complicated. But more features do not mean more generated code.

Most C++ features are source-level structure and safety. If you are compiler aware, you realize that the generated code overhead is insignificant. It is quite possible to write good object-oriented C++ code with performance in mind.

As an example, a good object design involves lots of very small methods. But if you declare them as inline, you get the performance of C with the features of C++.

5.4.1.2. Modularity

[Rule] For each file `module.cpp` which is not a main program, there must be exactly one corresponding `module.h` which contains the declaration of the public interface of the module. No internal or private definitions shall be present in the header. No part of the public interface of the module shall be declared in another header file.

This is the natural C++ implementation of a module. A module must have exactly one public interface and one implementation. There is one file for each.

Exception: In some cases, there is no need for an implementation and the `.cpp` file is not present, everything is present in the header file. This can be the case where the module contains declarations only, or template methods or classes, or when all functions and methods are short enough to be declared inline.

[Rule] Each module shall contain only one C++ class. If the class has inner classes, the inner classes are declared and defined in the same module at the top-level class.

This simplifies maintenance.

It is not possible to declare an inner class in a different header file from its enclosing class. To keep the one-to-one association between `.h` and `.cpp` files, the definitions of the inner class methods are implemented into the same `.cpp` file as the enclosing class.

Inner classes (also known as nested classes) shall not be confused with subclasses.

[Rule] If a data type is used only inside a module and is consequently private to the module, it shall

be declared within the `module.cpp` file and not in any header file.

If the type is private, it should not be useable by any other module. If the type were declared in a header file, even a specific one, different from the `module.h` public interface, it could be included by another module and used outside its normal scope.

In the case of C++ "private" declarations, the syntax of the language requires to declare them in the class declaration, meaning in the header file of the class. However, being private, these declarations cannot be used outside their outer class definition, which preserve the data type hiding rule.

In the early times of the C language, in the "Kernighan & Ritchie" era, it was common to declare all data types in one header file, outside of the source files. This practice is clearly outdated and should be banned. This separation of data types and code is purely based on the syntax, not on the semantics and is a violation of the principle of modularity.

[Rule] A module implementation file `module.cpp` shall start with an `#include` directive of its own interface file `module.h`.

Thus, the successful compilation of `module.cpp` validates two important points:

- The header file is self-sufficient.
- The header file is consistent with its implementation.

[Rule] A header file must allow multiple inclusions without generating errors.

You cannot manage how your header will be used. The compilation of a user application which includes your header file shall never generate an error simply because of your header.

Use case: A user code explicitly includes your header `module.h` as well as some other header `other.h`. If this `other.h` happens to also include your `module.h`, there must be no error.

In C++, the `#pragma once` is defined for this purpose. The old C tradition of conditional compilation using `#ifdef MODULE_H` is outdated and should be avoided.

Example: C++ header file `module.h`:

```
#pragma once
// ... file content here ...
```

[Rule] The structure of a header file shall be self-sufficient. The users of the header file must not have to specify other header files in order to use it. There must be no ordering requirement of the `#include` directives for the user.

This means that a header file must include all other header files which are needed by the declarations it contains.

[Rule] The header file for a C module shall be safely compiled when included from a C++ module. A C module shall be safely linkable with C++ modules.

In a C header file, do not use C++ reserved names.

In a C header file, all language constructs which are specific to C or otherwise incompatible with C++ shall be conditionally compiled using the predefined macro `__cplusplus` (with two leading underscores). This macro is defined by the compiler when the compilation occurs in a C++ environment.

In a C header file, all C declarations which generate a reference to a linker symbol (function or data) shall be enclosed in `extern "C" {…}` when compiled in C++.

Example: The following structures are inserted at the beginning and end of the declarations in the C header file:

```
#if defined (__cplusplus)
```

```
extern "C" {
#endif

/* all C declarations here ... */

#if defined (__cplusplus)
}  /* extern "C" */
#endif
```

[Rule] Use the anonymous namespace to define elements (data, classes, functions, etc.) which are internal to a module.

This avoids name clashes and namespace pollution.

This is the C++ equivalent of `static` declarations in C. Note that `static` has a different meaning in C++ and should not be used for that purpose.

Example:

```
namespace {
    void SomeInternalMethod()
    {
        ...
    }
}
```

[Rule] Never use static variables in inline functions.

If you do that, there will be one static variable per module where the function is used, breaking the semantic of the static attribute.

5.4.1.3. Naming and syntax formatting

Most of these topics are covered by coding conventions in section 5.4.2. However, a few topics are clearly *rules* rather than *conventions* because they impact the reliability and good understanding of the code.

[Rule] Always use a namespace when declaring entities. Never define anything in the default namespace.

This avoids the name clashes during the link. In TSDuck code, everything is defined in the namespace named `ts` or in one of its inner namespaces.

[Rule] The directive `using namespace` is strictly forbidden. There is no exception.

This avoids ambiguities. This also avoids some portability issues, especially with Visual C++ on Windows platforms.

All C++ standard entities must be referenced with the `std::` prefix.

Remember that there is no need to explicitly specify the namespace to reference an entity which is declared in the current namespace or an outer one. Therefore, inside the source code of TSDuck, it is not necessary to repeat the prefix `ts::` everywhere. Only third-party applications need to use it.

[Rule] Explicitly use the `::` prefix for predefined entities which are defined in the default namespace.

This indicates an explicit reference to the default namespace and avoids ambiguities with entities which are declared in the current namespace or an outer one.

Example:

```
open("file");    // WRONG: may conflict with an open() method of this class
```

```
::open("file");  // OK, there is no ambiguity
```

[Rule] Do not place code after a comment on the same line.

This is confusing. The trailing code can easily remain unnoticed.

Example:

```
// comment line is OK
a = 1; // comment after code is OK
b = x /* WRONG: code after comment is confusing */ + 3;
```

See also next rule.

[Rule] The comments always start with a `//` and extend up to the end of line. The usage of the C comment syntax `/*…*/` is discouraged.

A common problem with the C syntax `/*…*/` is the potential absence of closing `*/`. In some cases, the subsequent lines of codes are silently "swallowed" by the comment, up to the end of the next comment. Such bugs are very difficult to track.

By using the `//` syntax, all comments automatically terminate at the end of line.

Counter-example:

```
/* innocent comment */
a = 1;
/* WRONG: unterminated comment
b = 2;
/* the preceding comment actually ends here -->*/
c = 3;
```

In this example, the instruction `b = 2;` is excluded from the compiled code. Good luck to find the bug!

[Rule] Using the `{ }` braces in conditional and loop statements is mandatory, even if there is only one or no instruction in the block.

Omitting the braces is dangerous for the maintainability of the code. If the indentation is incorrect or if the unique statement in the block is complicated or spans multiple lines, omitting the braces makes the code hard to understand.

Good code:

```
if (x > a) {
    x = 0;
}

for (i = 0; i < max; i++) {
    print(a[i]);
}

while (readFile()) {
}
```

Bad code:

```
if (x > a)
    x = 0;


for (i = 0; i < max; i++)
    print(a[i]);


while (readFile());
```

Even worse:

```
if (x > a) x = 0;
```

Notes and anecdotes:

- Yes, Python is bad. Know your history. Using indentation for structuring code was abandoned in the 70's after the Fortran era.

- Failing to apply this rule was the root cause of the famous security vulnerability nick-named "gotofail" on macOS and iOS.
- The Linux kernel coding rules specifically prohibit the usage of braces when there is only one instruction. Do these guys ever heard of "gotofail"?

5.4.1.4. Coding style

[Rule] Avoid numerical constants, use `static constexpr` declarations for these values.

This is the C++ way. Preprocessing macros shall not be used for constants in C++. Use preprocessing macros only for conditional compilation or for numerical values which are evaluated in the context of conditional compilation.

Exception: Usual constants such as 0 and 1 which are used in obvious contexts (reset, increment) shall be used without names since they are self-explanatory.

[Rule] The value of all preprocessing macros shall be enclosed into parentheses (in the absence of other syntactic constraints).

This avoids compilation ambiguities.

Example:

```
#define TS_GOOD (TS_FOO + 3)
#define TS_BAD   TS_FOO + 3    // WRONG: what about "a = 5 * TS_BAD" ?
```

[Rule] In all preprocessing macros, the parameters shall be enclosed into parentheses when referenced in the definition.

This avoids compilation ambiguities.

Example:

```
#define TS_GOOD(x) (2 * (x))
#define TS_BAD(x)  (2 * x)    // WRONG: what about "TS_BAD(a + b)" ?
```

[Rule] In all preprocessing macros, each parameter shall be referenced exactly once in the definition.

Macro preprocessing is only text substitution. When the actual parameter of a macro has side effects, it is evaluated as many times as referenced in the macro definition while the caller expects exactly one evaluation.

Example:

```
#define TS_GOOD(x) (f((x)))       // OK:    TS_GOOD(a++) => a incremented
#define TS_BAD1(x) (g((x), (x)))  // WRONG: TS_BAD1(a++) => a incremented twice
#define TS_BAD2(x) (h())          // WRONG: TS_BAD2(a++) => a not modified
```

Exception: There are macros which are reserved to specific usages where the actual parameters are not general-purpose expressions but must be lvalues or special syntactic structures.

Alternative: When it is required to reference a macro parameter several times, you cannot use a macro. You should implement a real function and declare it `inline`. See next rule.

[Rule] Preprocessing macros should be avoided for code structures. Use inline function definitions.

This is the C++ way. This avoids all macro substitution pitfalls which are described in the preceding rules.

There is no difference in terms of code size or performance. The generated code for inline functions is expanded inline, just like a preprocessing macro. If the inline function is not used, no code is generated.

[Rule] The preprocessing directives `#ifdef` and `#ifndef` should not be used. Use `#if` followed by an expression instead.

This is more consistent with directives containing multiple conditions or `#elif` directives. For complex conditions, the code is more compact and more readable.

Good code:

```
#if defined(A)
    #define X 1
#elif defined(B) && defined(C)
    #define X 2
#endif
```

Bad code:

```
#ifdef A
    #define X 1
#else // not A
    #ifdef B
        #ifdef C
            #define X 2
        #endif // C
    #endif // B
#endif // A
```

[Rule] Use the preprocessing directive `#error` wherever there is no valid default alternative.

The `#error` will draw the attention of the developer when the code is compiled in a new environment which was not previously addressed. The developer who does the porting can precisely locate where there is some specific work to do.

Example 1: A simple list of mutually exclusive alternatives, ensuring that one branch is taken.

```
#if defined(TS_WINDOWS)
    typedef ... SystemError;
#elif defined(TS_UNIX)
    typedef ... SystemError;
#else
    #error "unknow O/S, please update SystemError in this header file"
#endif
```

Example 2: A more complex list of alternatives. But exactly one symbol must be declared at the end.

```
#if (defined(__i386) || defined(__x86_64)) && !defined(TS_LITTLE_ENDIAN)
    #define TS_LITTLE_ENDIAN 1
#elif (defined(__sparc) || defined(__powerpc)) && !defined(TS_BIG_ENDIAN)
    #define TS_BIG_ENDIAN 1
#endif

// ... more definitions

#if !defined(TS_LITTLE_ENDIAN) && !defined(TS_BIG_ENDIAN)
    #error "unknow endian, please update this header file"
#endif

#if defined(TS_LITTLE_ENDIAN) && defined(TS_BIG_ENDIAN)
    #error "conflicting endianness, please review this header file"
#endif
```

[Rule] When declaring multiple variables with the same base type, create multiple individual declarations, one per line.

The following two code excerpts are perfectly valid according to the C and C++ standards and have identical effects. However, in the bad code example, it is not clear that the line declares objects which are not `int` (the last two variables are a pointer and an array). It is also not clear that `j` is pre-initialized to zero while `i` is not.

Good code:

```
int i;
int j = 0;
int* p;
int a[10];
```

Bad code:

```
int i, j = 0, *p, a[10];
```

[Rule] The order of evaluation of the operators in C and C++ is complex. Do not hesitate to add extra parentheses in order to make the code more readable.

Do you know what `++p->a` means? Does it increment the pointer first and then dereference it? Or does it dereference the pointer first and then increment the pointed field? Actually, the second alternative is right. But how many maintainers will know that?

By the way, did you even know that the self-increment operator `++` has distinct precedencies when it is used as a prefix or a suffix?

Example: Adding theoretically useless but helpful parentheses:

```
++p->a;     // WRONG: nobody really knows what this mean
++(p->a);   // GOOD: same as "++p->a" but more readable
(++p)->a;   // not the same as "++p->a" so parentheses are required anyway
```

[Rule] A `switch` statement must not contain any implicit "fall through" path, i.e. all cases must end with a break.

The `switch`/`case` implicit fall-through path is a rare and confusing construct. Most of the time, no `break` at the end of a `case` is a forgotten one, i.e. a bug. By allowing implicit fall-through paths, we cannot clearly identify if a missing `break` is a bug or a feature. So, to be safe, we forbid it.

Exception: It is allowed to have no break when there is no instruction at all after the `case`. This situation is not a real fall-through, this is a `case` with multiple equivalent values.

Example:

```
switch (x) {
    case 1:
        print("something ");
        // WRONG: implicit fall-through path is confusing, bug or feature ?
    case 2:
        print("else");
        break;
    case 3:  // OK, not a real fall-through but a multiple case entry
    case 4:
        print("ok");
        break;
    default:
        print("error");
        break;
}
```

In TSDuck, when allowed as a compiler option, the implicit fallthrough paths are detected and rejected using the appropriate warning configuration.

[Rule] In a `switch` statement, if a "fall through" path is really necessary, it must be explicitly declared using a `[[fallthrough]]` attribute.

These constructs shall remain rare and reserved to cases where any other structure would be even more confusing.

Example:

```
switch (x) {
    case 1:
        print("something ");
        [[fallthrough]];
    case 2:
        print("else");
        break;
    default:
        print("error");
        break;
}
```

[Rule] A `switch` statement must end with a `default` entry.

This ensures that unexpected values are always processed, typically with an error processing.

This is especially useful when the `switch` is applied on all values of an `enum` type. You may think that the `default` alternative is useless since all values are handled. However, what will happen when you update the `enum` type and add a value? If several `switch` statements are used in various modules, you will probably forget to update the list of cases in one of them. Without `default` alternative, the new value will be silently ignored and the bug will be either hard to find or (worse) remain undetected. In the presence of a `default` alternative which triggers error code, the run-time error will immediately detect the bug.

In TSDuck, when allowed as a compiler option, the absence of `default` alternative is detected and rejected using the appropriate warning configuration.

[Rule] If a local variable is required in a `case` or `default` entry in a `switch`, declare a local block using `{ }` for the entry.

Do not declare a local variable in a `switch` `case` without enclosing `{ }` block. The scope of the local variable extends to the subsequent entries in the `switch`, which is usually not what you indented.

Note that the C and C++ languages have distinct interpretations here.

Good code: **Bad code:**

```
switch (x) {
    case 1: {  // <== note the "{"
        int i; // local to this entry
        ...
        break;
    }          // <== note the "}"
    case 2:
        ...
        break;
    default:
        ...
        break;
}
```

```
switch (x) {
    case 1:
        int i; // C: compilation error
        ...
        break;
    case 2:
        // C++: i is still visible here
        ...
        break;
    default:
        ...
        break;
}
```

[Rule] In conditional expressions, explicitly compare against zero for non-boolean values such as integers or pointers.

The implicit interpretation of an integer or pointer value as a boolean is confusing. Sometimes, the "common interpretation" is even the opposite of the reality.

Example: The common sense is that a function which returns a boolean value would return true on success and false on error because the word "true" reveals a positive feeling while "false" reveals a negative one. Many UNIX system calls do not return a boolean value. They return an integer which is zero on success and a non-zero error code on error.

See how this can be confusing.

Confusing code:

```
if (close(fd)) {
    // You think it's a success ?
    // In fact, it's an error
}
```

Cleaner code:

```
if (close(fd) != 0) {
    // error processing
}
```

[Rule] Never use boolean operators (`!`, `&&`, `||`) on non-boolean values such as integers or pointers.

Same reasons as the preceding rule.

[Rule] Do not use assignments in conditional expressions, even for boolean variables.

This is confusing. The lazy reader will misinterpret `if (a = b)` as `if (a == b)`. So, assign first, then use the resulting variable in the conditional expression:

Confusing code:

```
if (big = isLargeFile(f)) {

}
```

Cleaner code:

```
big = isLargeFile(f);
if (big) {

}
```

In TSDuck, when allowed as a compiler option, an assignment in a boolean expression is detected and rejected using the appropriate warning configuration.

Anecdote: A long time ago, someone attempted to introduce a backdoor in the Linux kernel, pushing code containing something like `if (proc->uid = 0) {…}`. The careless reader could thing that the kernel code tested if the caller was root. If practice, the code forced to the caller to become root, which was a severe security breach. If you need only one reason for this coding rule, use this anecdote.

[Rule] Reduce the scope of variables to the smallest possible block.

This is more readable and easier to maintain.

Example: Loop indexes should be declared inside the `for` statement when possible:

Good code:

```
for (int i = 0; i < max; i++) {
    const int j = 3 * i;
    ...
}
```

Bad code:

```
int i, j;
...
for (i = 0; i < max; i++) {
    j = 3 * i;
    ...
}
```

[Rule] Delay the declaration of variables until they are used for the first time.

This avoids lengthy initial declarations which are not immediately useful. This also gives you a chance to declare the variable as `const` for instance.

Good code:

```
int a;
...
a = ...;
...
const int max = 2 * a + 1;
for (int i = 0; i < max; i++) {
    ...
}
```

Bad code:

```
int a, i, max;
...
a = ...;
...
max = 2 * a + 1;
for (i = 0; i < max; i++) {
    ...
}
```

[Rule] To set the initial value of an object, always prefer the initialization syntax over an assignment.

This is faster to execute. In the case of the initialization by assignment, the default constructor is first invoked, then the assignment operator is invoked.

Example: Objects `b`, `c` and `d` have an identical initial value. But the initialization of `b` is faster.

```
ts::Foo a;
ts::Foo b(a);    // Invoke the copy constructor
ts::Foo c = a;   // Invoke the default constructor, then the assignment operator
ts::Foo d;       // Invoke the default constructor
d = a;           // Invoke the assignment operator
```

In the case of objects `c` and `d`, the default constructor does something which is immediately undone by the assignment operator.

Sometimes, the compiler may optimize this but do not count on it since a deep code analysis of the constructor and assignment is required to allow this optimization. Without this deep source code analysis, the compiler does not know if the semantics and side effects of the default constructor plus assignment are equivalent to the copy constructor.

[Rule] The use of `goto` is prohibited.

The `goto` statement is the most discussed and infamous construct in computer software history.

Possible exception: The only acceptable exception is a common error path within one function, at the end of the function, after the return of the normal (non-error) path.

However, this should be reserved to special cases, when working on low-level system calls, in the presence of multiple potential error cases, when using another more complex structure would be even more confusing. In modern C++ code, using structured objects with a proper destructor should prevent this.

Example: Valid and (possibly) acceptable usage:

```
class Foo
{
public:
    Foo(); // default constructor
private:
    FILE* f1 = nullptr;
    FILE* f2 = nullptr;
};

Foo::Foo()
{
    // Open each resource
    f1 = fopen("file1", "r");
    if (f1 == nullptr) {
        goto error;
    }
    f2 = fopen("file2", "r");
    if (f2 == nullptr) {
        goto error;
    }

    // Do other initial processing
    // ...
    return;

 error:
    if (f1 != nullptr) {
        fclose(f1);
        f1 = nullptr;
    }
    if (f2 != nullptr) {
        fclose(f2);
        f2 = nullptr;
    }
}
```

[Rule] A single function shall not exceed 50 lines of actual code or approximately 100 raw lines including comments.

Very long functions are hard to understand and maintain. They should be redesigned to extract local treatments in other well documented local functions, even if these local functions are called only once.

Exception: A very large `switch` construct with a lot of short entries may be acceptable. In fact, breaking it apart into several functions may be less maintainable.

[Rule] Do not pass parameters of non-elementary type by value.

Passing a parameter by value is legal but it requires copying the parameter value, typically on stack. While this is harmless for elementary types (integers, floats, enums and pointers), it can be costly for structured types.

In C++, this can even be devastating with polymorphic types (classes with at least one virtual function) when the type of the formal parameter is a superclass of the actual parameter. In this case, we get the *slicing problem*: the pointer to the *vtable* and the superclass parts are copied while the derived parts are not. The result is an inconsistent object with virtual methods possibly using fields that do not exist. In C++, the common way is to pass such parameters *by reference* (a C++ specific mechanism, not to be confused with passing *by address* or *by pointer* in C).

Example:

```
class Foo {...};

void f(int x);          // OK: elementary type by value ("in" parameter)
void f(int* x);         // OK: elementary type by address ("out" parameter)
void f(Foo x);          // WRONG: structured type by value, don't do that
void f(const Foo* x);   // OK: structured type by address ("in" parameter)
void f(Foo* x);         // OK: structured type by address ("out" parameter)
void f(const Foo& x);   // OK: structured type by reference (C++)
```

[Recommendation] In `for` loops, prefer the modern C++11 syntax instead of explicit iterators.

The code is more readable, mode compact, and less error-prone.

Example:

```
std::list<Foo> flist;

// Using iterators
for (std::list<Foo>::iterator iter = flist.begin(); iter != flist.end(); ++iter) {
    process(*iter);
}

// Modern C++11 syntax
for (auto Foo& f : flist) {
    process(f);
}
```

Exception: There are cases where the iterator is required for intermediate operations of specific processing at the end of each iteration. This is the case when it is necessary to insert or remove elements in the list in an iteration.

[Rule] In modern C++11 `for` loops, use a reference in the iteration parameter.

This saves the useless copy of a temporary object.

Example:

```
std::list<Foo> flist;

// INCORRECT code
for (auto Foo f : flist) {
    // In each iteration, f is a new temporary object which is constructed
    // from the value of the element in the list
    process(f);
}

// correct code
for (auto Foo& f : flist) {
    // f is a reference to the element in the list
    process(f);
}
```

5.4.1.5. Strict typing

To detect as many potential errors as possible directly at compilation time, use strict types and attributes on all declarations and definitions.

[Rule] Use the `const` attribute wherever an entity is used as read-only.

This applies to parameter declarations in function profiles and in object declarations.

Example 1: A value is computed once and reused later without modification.

```
const size_t max = count * sizeof(something) + offsetFoo;
size_t i;
for (i = 0; i < max; i++) {
```

Example 2: If a reference or pointer value is used to access a read-only area (at least through this pointer or reference), use the `const` attribute. This is especially important for function declarations since this establishes a more precise contract with the caller.

```
void LogMessage(const char* msg, const Time& time);
```

[Rule] Use the `volatile` attribute wherever an entity is potentially asynchronously updated by another thread or some hardware mechanism.

This guarantees that the generated code will not optimize the access to the variable by caching it in a register for instance.

[Rule] Do not use `volatile` on a non-elementary types (i.e. not integer and not boolean).

The compiler can hardly guarantee an atomic access on such types and there is a risk to get randomly corrupted values.

Redesign the code so that accesses to non-elementary types are explicitly synchronized.

[Rule] Be careful when using the `const` attribute on pointers. Do you want a variable pointer to constant data, a constant pointer to variable data, or a constant pointer to constant data? A good practice is to use type names for complex pointer types.

The order of the `const` attribute, the type name and the `*` sign makes the difference. This is often the source of obscure compilation error messages.

Example:

```
const char* a;              // a variable pointer to "const char"
char* const b = ...;        // a constant pointer to "char"
const char* const c = ...;  // a constant pointer to "const char"

a = "abcd";   // OK
*a = 'x';     // ERROR
b = "abcd";   // ERROR
*b = 'x';     // OK
c = "abcd";   // ERROR
*c = 'x';     // ERROR
```

It is more readable to use type declarations:

```
using CharPointer = char*;
using ConstCharPointer = const char*;

ConstCharPointer a = nullptr;
const CharPointer b = ...;
const ConstCharPointer c = ...;
```

[Rule] Never use built-in integer types such as `int` or `long`, unless explicitly required by the context.

The C and C++ standards do not define the implementation of the `int`, `short`, `long` types and their variants (`unsigned`, `long long`, etc.).

Specifically, the types `int` and `long` are known to have different sizes on different platforms. Using them reduces the portability of the code by introducing subtle side effects on large values.

Of course, in the presence of a legacy library which uses `int` or some of its variants in an API, you are obliged to use the same type for the data which are used with this API.

Use the following standard types when the size of integer values matter. They are defined in the standard headers `stdint.h` and `inttypes.h`.

`int8_t`	`uint8_t`
`int16_t`	`uint16_t`
`int32_t`	`uint32_t`
`int64_t`	`uint64_t`

[Rule] Use the predefined type `size_t` for values which hold the size in bytes of a C/C++ object.

This is the standard definition in the C/C++ language. The C and C++ standards define `size_t` as the return type for the `sizeof` operator. The modern C/C++ standard libraries use this type in the profile of their functions for size values.

The actual implementation of `size_t` differs from one platform to another. It has typically the same size as a pointer. But there is no unique correspondence with the built-in integer types.

For instance, some 64-bit platforms define `int` as 32-bit and `long` as 64-bit while other 64-bit platforms define both `int` and `long` as 64-bit. But in all cases, `size_t` is 64-bit on these platforms. This is especially true when porting between Windows / Visual Studio and Linux / GCC.

So, there are clearly at least three distinct sets of integer semantics:

- Built-in types (`int` et al.): not portable, no specific semantic, unpredictable, do not use.
- Size of objects (`size_t`).
- Values represented by a given number of bits (`uint8_t` et al.)

Variant: The standard type `ssize_t` declares a signed integer type of the same size of `size_t`.

[Rule] Never use the built-in type `char` for any other purpose that 7-bit ASCII characters.

Do not use `char` to represent array of bytes, use either `int8_t` or `uint8_t`. Do not use `char` to represent characters outside the 7-bit ASCII range. In the context of internationalization, the binary representations vary. This can be a dedicated 8-bit character set (Latin-1, Latin-9, etc.) or some binary representation of UNICODE (UTF-8, UTF-16, UTF-32, etc.)

In all cases, the management of internationalized character strings shall be encapsulated into some dedicated library.

In TSDuck source code, the types `ts::UChar` and `ts::UString` implement Java-like characters and strings (UTF-16 with surrogate pairs).

[Rule] Never use the built-in type `unsigned char`, nowhere, never.

The type `unsigned char` represents nothing. It has no semantics at all.

Either a data is an ASCII character and it is a `char` or it is a byte and it is an `int8_t` or `uint8_t`.

[Rule] Always use literal constants which are type-compatible with the context.

Example:

```
int   i = 17;
long  l = 17L;      // 'L' suffix means a literal of type long
char  c = '\0';     // do not use integer literal such as: char c = 0;
void* p = nullptr;  // C++ notation for a null pointer, not always binary zero
```

It is important to understand why using the `L` suffix is essential in integer literals which are used in the context of an expression the final type of which is `long`. Consider the following example.

```
const int i = 1;
const long l1 = i + 0xFFFFFFFFL;
const long l2 = i + 0xFFFFFFFF;   // same literal without 'L' suffix
```

You may think that `l1` and `l2` have the same value. This may be the case. But they don't on some platforms where `l1` gets `4294967296` (the expected value) while `l2` gets zero.

Why?

On some platforms, `int` is a 32-bit value while `long` is a 64-bit value. In the case of `l2`, the intermediate context into which the literal is evaluated is `int` since `i` is an `int`. So, the literal `0xFFFFFFFF` is *first* evaluated into the context of an `int` and its value is `-1`. The addition is performed on `int` values, giving zero. *Finally*, the result is promoted to long, staying zero.

On the contrary, in the case of `l1`, the literal is explicitly of type `long` because of the trailing `L` suffix. Thus, the intermediate context into which the addition is performed is `long`. This time, `i` is *first* promoted to `long` and *then* the addition is performed on `long` values, giving the expected result.

Portability issue: The `L` suffix is the standard way to represent `long` literals. But we discourage the usage of predefined integer types (see the corresponding rule above) for portability reasons. For integer types which are explicitly smaller than 64 bits (`int8_t`, `int16_t`, `int32_t`, etc.), using `int` literals is usually safe. For explicit 64-bit integer types (`int64_t`, `uint64_t`), it is recommended to use the suffix `LL` in literals. The corresponding value is of type `long long`. The size of this type is not guaranteed either but it is at least 64 bits on all known implementations.

[Rule] Use the keyword `nullptr` for null pointers, not the literal `0`, not the macro `NULL`.

This is the only non-ambiguous way of specifying a null pointer.

The old macro `NULL` was specified for the C language, not C++. Early specifications of the C++ language commanded to use the literal `0` for null pointers. This has led to many ambiguous or incorrect code since this literal has two distinct semantics: integer zero of type `int` and null pointer to any type.

In TSDuck, when allowed as a compiler option, using zero as null pointer literal is detected and rejected using the appropriate warning configuration (e.g. option `-Wzero-as-null-pointer-constant` with GCC and Clang).

[Rule] Never change the order of `enum` values in a type declaration.

The `enum` types are strictly defined in the C and C++ standards. Specifically, in the absence of explicit numerical value, `enum` values are defined as implicitly consecutive numerical values. It is safe for C/C++ code to compare `enum` values using `<` or `>` operators.

If you change the order of the declarations in an `enum` type for simple aesthetic reasons (sort in alphabetical order for instance), you change the semantics of the type and you potentially break the user's code. This is why re-ordering the values of an `enum` type should be avoided.

If you add a new value in an `enum` type, think about why you add it:

- If this is simply a new value, add it at the end of the type, regardless of aesthetic reasons.
- You may insert it somewhere else only if there is a very good reason to insert it between two existing values, for instance because this is new logic state between two previously consecutive states in a strictly ordered state machine.

[Rule] Always use a type definition (`using`) for each meaningful type. Do not use basic types which somehow reflect the implementation of the type.

Example:

```
using FooIndex = uint16_t;
```

This facilitates the maintenance if the implementation changes someday.

In the above example, if a 16-bit integer becomes too short some day for a `Foo` index, just change the type definition. Otherwise, you would have to change all references to `uint16_t` as `uint32_t` in all variables which have the semantics of a `Foo` index and only those `uint16_t`. Needless to say that the probability to introduce a bug at this stage is very high.

[Rule] In type definitions, use the C++ syntax `using name = definition`. Do not use the old C `typedef` syntax.

The `using` syntax is more clear than `typedef`. The type name is clearly isolated before the `=` sign.

With `typedef`, in some complex cases such as function pointers, the type name is buried into the type definition and not easy to spot.

5.4.1.6. Assertions

Assertions are statements which are inserted in the middle of executable code. They assert a condition and abort the execution if this condition is false. Assertions are included in the generated code under some specific compilation options and removed by the compiler for the final production code.

Assertions are very useful to check the consistency of the code.

[Rule] Assertion must be used to check the internal consistency of code. Assertions must never be used to check input values or other external conditions.

Assertions are debug tools exclusively. Removing them shall not change the semantics of the code.

An assertion shall be used only to check the internal consistency of the code. This means that an assertion shall be used to assert something that you know is always true, regardless of the external environment, if your code is right.

In other words, an assertion contains a condition which is always true if your code is bug-free, even in the presence of incorrect inputs or incorrect environment.

For instance, if your code is such that an index `i` shall always be less than some maximum value at the end of a loop, you may write `assert(i < max)`.

Similarly, if you are about to write a data structure `ds` into a memory area `ma` which is defined as an array of `uint8_t`, you may write `assert(sizeof(ds) <= sizeof(ma))` and then `memcpy(&ma, &ds, sizeof(ds))`.

But, under no circumstances, you should use some external condition such as an input parameter of the current function in an assertion. The semantic of an assertion is that it shall have no effect on correct code. The compiler may include or remove the assertion from the generated code and the execution of correct code should be exactly the same in both cases.

Checking an external or input parameter in an assertion is illegal since an invalid external condition is not a symptom of incorrect code (at least your code). If you use an input parameter in an assertion in correct code, this code will either run or fail in the presence of invalid input, depending on the compilation option.

The correct way of checking inputs is to use plain checking code, not assertions. In the presence of invalid inputs, write the appropriate error management path for the context (return an error code, log an error, perform a default action or whatever is required by the context).

Example:

```
int8_t buf1[(3 * TS_SIZE1) + 4];
int8_t buf2[TS_SIZE1 + TS_SIZE2];

// correct, sizes are compile-time constants
assert(sizeof(buf1) >= sizeof(buf2));
memcpy(&buf1, &buf2, sizeof(buf2));

int8_t* buf3 = (int8_t*)(malloc(someComputedSize));

// incorrect, depends on system run-time resources
assert(buf3 != nullptr);

// correct, valid run-time check regardless of compilation options
if (buf3 == nullptr) {
    // do some error processing
}

// incorrect, depends on a run-time value
assert(sizeof(buf1) >= someComputedSize);

// correct, valid run-time check regardless of compilation options
if (sizeof(buf1) < someComputedSize) {
    // do some error processing
}
memcpy(&buf1, buf3, someComputedSize);
```

[Rule] Assert everything that could be asserted.

Use assertions as often as necessary. When writing code, always try to locate all the implicit assumptions you make (data size, `enum` values ordering, etc.). Locate all the conditions that must be true and which would corrupt something if incorrect (final index values, counters, etc.) Assert all of this.

When writing code, you easily assume conditions. You know that they are true by design. But you may be wrong (nobody is perfect). And some future maintenance may break the design. So, even if this seems futile, write assertions on critical conditions.

And remember that assertions are automatically removed by the compiler in production code. So, do not hesitate to use assertions, they are performance-free on production code.

[Rule] The debug code which is conditionally compiled (typically using some preprocessor symbol such as `DEBUG`) shall not modify the functional behavior of the code.

The reasons are the same as for the previous rule.

Such debug can typically only log debug information.

Be aware, however, that the presence of debug code may have a significant impact on timing.

5.4.1.7. Secure coding

[Rule] Always initialize data with predictable values. Use zero for integers and `nullptr` for pointers if there is no other meaningful values in the context.

Uninitialized variables are a common source of bugs which may be difficult to track.

Getting different behaviors on different platforms or versions depending on different unmanaged initial values is even worse.

Example:

```
uint32_t x = a + b;
uint32_t y = 0;
uint32_t* p = nullptr;
```

[Rule] Do not use `memset()` to initialize structures. Use appropriate initial value for each type.

All binary zero is often not appropriate as initial value for a data structure.

Good code:

```
struct Foo {
    uint8_t* p;
    size_t   size;
};

Foo obj;
obj.p = nullptr; // not always binary zero
obj.size = 0;
```

Bad code:

```
struct Foo {
    uint8_t* p;
    size_t   size;
};

Foo obj;
memset(&obj, 0x00, sizeof(obj));
```

[Rule] Make no assumption about the size or memory layout of a data structure.

Each platform has its constraints on data alignment. The compiler takes them into account when representing a data structure. The same data structure may have different sizes or memory layout on different platforms.

Example:

```
struct Foo {
    uint8_t  a;
    uint32_t b;
} Foo;
```

The size of this structure may be 5 or 8 bytes, depending on the platform. The field `b` may be at offset 1 or 4 from the beginning of the structure.

If your code depends on the fact that `b` is assumed to be at a specific offset, you should assert this. The following example asserts that `b` is at offset 1 after `a`:

```
assert(((char*)&((Foo*)nullptr)->b - (char*)&((Foo*)nullptr)->a) == 1);
```

This is quite a strange expression since the null pointer is explicitly dereferenced. But it is used only to evaluate an address and not a content. Thus, the pointer is never really dereferenced.

For the sake of clarity, the above code can be rewritten as:

```
assert(offsetof(Foo, b) == 1);
```

[Rule] Make no assumptions about the evaluation order of operands or function parameters.

The C and C++ languages do not specify the order of evaluation in expressions and function calls. If any operand or parameter has side effects, the final result is unpredictable and may vary from one platform to another.

Counter-example: Bad code which rely on the order of evaluation:

```
int a = 2;
int b = a * a++;        // WRONG: result can be either 4 or 6
f(print(a), print(b));  // WRONG: a and b may be printed in any order
```

[Rule] Do not assume that a null pointer (`nullptr` in C++) is represented by binary zeroes.

The C and C++ standards state that a zero value in a pointer context is interpreted as a *null pointer*. But, on some platforms where zero is a valid address, the compiler may use a non-zero bit pattern to represent the concept of a null pointer.

This rarity makes the problem even more dangerous. If your code relies on the fact that a null pointer is represented by a zero bit pattern, it will work on most platforms. Later, when recompiling the code on a new platform where null pointers are represented differently, the code will no longer work. And it will take ages to find the problem.

[Rule] In a function, never return a pointer or a reference to a local variable or a parameter of this function.

The local variables go out of scope upon return. The returned pointer or reference points to a portion of the stack which is no longer used. Worse, the referenced memory area will be reused by the local variables of another function call.

Consequently, returning pointer or a reference to a local variable or a parameter may lead to subtle random memory corruptions in the later functions. This kind of problem is a nightmare to debug.

[Rule] As a generalization of the preceding rule, always consider the lifetime of an object before passing it using a pointer or a reference to some other entity.

The preceding rule addresses this problem for statically allocated objects in the context of a mono-thread application. But similar problems exist with dynamically allocated objects and multi-threaded applications.

Object lifetime: First, what is the lifetime of an object? This depends on the nature of the object. If the object is directly declared in a function or code block, its birth is its declaration point and its death occurs when the execution flow exits from the function or block. If the object is dynamically allocated (`malloc()` or equivalent in C, `new` in C++), the allocation is its birth and the corresponding deallocation (`free()` or equivalent in C, `delete` in C++) is its death.

Dynamic allocation: When a function dynamically allocates an object and returns this pointer or passes it to multiple modules, the function loses all tracks of the usage of the pointer by other entities. When shall this object be deallocated is a complex question which must be addressed at the application design level.

C++ may mitigate this problem using implementations of the smart pointer design pattern, typically `std::shared_ptr` in moderm C++.

Multi-threaded applications: If a function passes the address of one of its local variables to another thread, there is a risk that the function returns while the other thread still uses the referenced data. To protect from this, the function may synchronize its return point with the termination of the other thread (*join* operation). Otherwise, this problem must be addressed at the application design level.

[Rule] When you declare a function accepting some form of array or raw data buffer, always add a parameter to specify the size of the data (input) or maximum size of the buffer (output).

Never assume that the format of the data will reliably define the end of the data.

Example:

```
// WRONG: how much data should I send?
void SendData1(const void* data);

// OK
void SendData2(const void* data, size_t dataSize);
```

Of course, in the second case, the correct declaration does not mean that the implementation is correct. But it is possible to write a correct implementation for this function.

On the other hand, the first function can never be safe and reliable.

[Rule] Never call any function with the following characteristics:
- A parameter is an address where the function is supposed to write to.
- The function may write more than one element at that address.
- No parameter gives the maximum number of elements to write to.

Many functions write data to a user-supplied buffer without letting the user specify the size of this buffer. These functions are simply badly defined and badly designed. They must be thrown away, simply.

[Rule] As a corollary of the preceding rule, never use any unsafe predefined string functions such as `sprintf()`, `strcpy()`, `strcat()`, ***all other*** `strXXX()` ***functions of the standard*** `libc` ***and many other standard functions.***

Many standard functions are inherited from the old days of the UNIX operating system in the 70's. They are obviously flawed in their definition. Just think twice about using standard functions, especially old ones.

Note that safe alternative versions exist for most of these functions: `snprintf()`, `strncpy()`, etc. Although safer than their ancestors, these functions still require some care. The pain with C-strings is the final null termination character. When a safe standard C-string function (one with an `n` in the middle of the name) copies a string into a buffer and the string is potentially larger than the buffer, the final string is truncated to the size of the buffer. In this case, truncation means no final null character. This also means that the result is *not a valid C-string*. If you use it as a C-string, the results are unpredictable (collecting memory garbage or crash).

Thus, when using the safe `n` C-string functions, you must always either check the return value to detect overflow (if supported by the function) or systematically overwrite the last byte of the buffer with a null character.

Example: The function, here `strncpy()`, does not indicate if a truncation occurred:

```
char s[10];
strncpy(s, longText, sizeof(s));
s[sizeof(s) - 1] = '\0';  // force valid C-string if truncated
```

Example: The function, here `snprintf()`, returns a value which can be used to detect the truncation:

```
if (snprintf(s, sizeof(s), "%s", longText) >= int(sizeof(s))) {
    // result was truncated,
    // either process error or force valid C-string as above
}
```

[Rule] Zero out pointers after `free` ***(C) or*** `delete` ***(C++).***

By zeroing the pointer, any accidental usage will result into an access violation (immediate crash) instead of a *reuse-after-free* or *double-deallocation* (subtle memory corruption which may run undetected for sometimes and hard to debug).

Example:

```
Foo* p = new Foo(...);
...
if (...) {
    delete p;
    p = nullptr;
}
*p = ...;      // immediate crash if preceding branch was taken
```

[Rule] Never cast a pointer into an integer.

First, converting between integers and pointers is bad design. Second, you do not know the size of a pointer and, on some platforms, casting a pointer to an integer results in a loss of information.

If you need to declare a field which stores a generic pointer, use `void*`.

If you need to perform low-level address arithmetic on bytes (instead of C elements), cast the pointers into `uint8_t*`.

[Rule] Be careful when mixing the `sizeof` operator with pointer or array index arithmetic. Remember that the pointer arithmetic uses the element size as a base while the `sizeof` operator returns a value in bytes.

Do not use `sizeof` to compute the last index in an array; this works only for arrays of `char` and 8-bit integers.

Counter-example: Incorrectly setting the last element of an array, leading to a crash (at best) or some random memory corruption (at worst).

```
uint32_t a[100];

// WRONG: not the last element but far, far away in memory
a[sizeof(a) - 1] = 0;
```

[Rule] To determine the number of elements in an array, do not use the explicit size from the declaration of this array. Use `sizeof` in an appropriate formula.

Otherwise, the maintenance becomes difficult if the declaration of the array is modified.

Example:

```
Foo array[SOME_COUNT];

// WRONG: what if the declaration of array is modified ?
const size_t numberOfElements = SOME_COUNT;

// This method is independent from the declaration of the array
const size_t numberOfElements = sizeof(array) / sizeof(array[0]);
```

[Rule] Do not declare arrays as local variables.

This is a corollary of the rules on buffer overflow and stack overflow.

An array can be very large and may overflow the stack of the current thread.

Moreover, using an array is subject to buffer overflow, even if you make your best efforts to avoid that. On a security standpoint, a buffer overflow is much more dangerous on the stack than on the heap. The stack is full of "code addresses", typically return addresses or exception handlers. A buffer overflow on the stack is the typical vulnerability which can be exploited for malware code injection. Even if there are some "code addresses" in the heap, they are less common. This is why a buffer overflow is less dangerous on the heap than on the stack.

This problem is more frequent in C than C++. With C++, we do not use arrays; we use vectors or other container classes. In instance of such a class is typically implemented as a short data structure containing pointers. The actual data are contained in dynamically allocated areas which are transparently managed by the container class.

[Rule] Never mix signed and unsigned integers.

Mixing signed and unsigned integers, typically dealing with index or offset values, is extremely dangerous and should be banned. The weird effect appears when offset values are decremented towards zero. After zero, the signed value goes negative while the unsigned one wraps up to 2^n-1. The two values are subsequently desynchronized.

Some coding guidelines advocate the usage of unsigned integers to manipulate sizes "because they never become negative". This is not always a good idea. An unsigned value never becomes negative for sure, but it becomes "extremely large" instead, which is not better or even worse in some case. It is true that the C/C++ standards use the type `size_t` for sizes and `size_t` is unsigned. But standards may have bad ideas too.

Counter-example: The following code is a very common way to explore a buffer backward.

```
i = size;
while (--i >= 0) {
    // do something on buffer[i]
}
```

The code is valid when the iterator `i` is signed. However, if `i` is unsigned, of type `size_t` for instance, this code loops forever (or most certainly crashes if `i` is used to access memory).

Do you understand why Java, a language "designed with security in mind", has no unsigned type?

[Rule] Anticipate and prevent integer arithmetical overflow and underflow.

In C/C++, the integer types always implement modular arithmetic. All operations are performed modulo 2^n. Arithmetical operations never overflow, they wrap up or down. In some cases, this behaviour is fine. But in most cases it is not. In fact, integer arithmetic and modular arithmetic are two different domains with distinct usages but C/C++ implements only one of the two. Other programming languages have a different approach. The Ada language, for instance, defines two distinct families of integer types: integer types and modular types. All operations on integer types remain in the range of the type and throw an exception in case of underflow or overflow. Modular types, on the other hand, implement the same semantics as C/C++. Based on the application requirements, the developer chooses the appropriate type. But this is not possible in C/C++ or Java. The developer has to implement all checks manually.

The type of check depends on the operation (addition, subtraction, multiplication) and the signedness of the integer type. Note that an integer division never overflows or underflows but the divider shall be checked to prevent "divide by zero" errors. To check overflows on addition and subtraction, we need to know the maximum and minimum values of the integer type. To check overflows on multiplication with signed types, we need to know the proper function to compute an absolute value.

In C++, the standard header `<limits>` declares the template class `std::numeric_limits`. This class provides a generic mechanism to determine various properties of numeric types, either integer or floating point. The header `<cstdlib>` declares overloaded versions of `std::abs()` for all predefined signed integer types.

The characteristics of an application-defined integer type `Foo` are:

- Minimum value: `std::numeric_limits<Foo>::min()`

- Maximum value: `std::numeric_limits<Foo>::max()`
- True when the type is signed: `std::numeric_limits<Foo>::is_signed`
- Absolute value: `std::abs(value)`

For some reason, in the template class `std::numeric_limits`, `min()` and `max()` have the syntax of a function while `is_signed` has the syntax of a constant. But the functions are inlined and return a constant. So, there is no performance penalty.

Detecting underflow on subtraction with unsigned integers

The following code illustrates a very dangerous situation of underflow.

```
size_t start = ...;            // some starting index inside an array
size_t current = ...;          // current exploration index
size_t size = current - start; // size the explored area
```

When `current` is less that `start`, the operation `current - start` underflows since `size_t` is an unsigned type. The variable `size` receives a very large positive value, close to 2^{32} or 2^{64}, depending on the platform. When `size` is later used as an actual data size, the most likely result is a buffer overflow, the best known security vulnerability, leading to all sorts of malware infections.

So, the code should be rewritten as follow:

```
size = current < start ? 0 : current - start;
```

The test `current < start` anticipates and detects the possible underflow in the subsequent subtraction. In case of detected overflow, the code uses an appropriate alternative (here a zero value).

Detecting overflow on addition with unsigned integers

Detecting overflows is a bit more complicated than underflows because the wrapping value, where the overflow occurs, depends on the size of the integer types. The following code illustrates how to detect a potential overflow:

```
size_t start = ...;  // some starting index inside an array
size_t size = ...;   // size of an area inside the array
size_t next;         // will receive the index after the area

if (start > std::numeric_limits<size_t>::max() - size) {
    // process arithmetic overflow
}
else {
    next = start + size;
    if (next >= size_of_array) {
        // process buffer overflow
    }
    else {
        // now you can safely process the array at index "next".
    }
}
```

This is the minimum required code to safely move forward into a buffer using index values. Note the different cases for buffer overflow and arithmetic overflow on index computation.

Checking the potential overflow must be done using operations which never overflow or underflow. The operation `std::numeric_limits<size_t>::max() - size` is safe because the integer type is unsigned and `std::numeric_limits<size_t>::max()` is its maximum value.

On the other hand, the naive test `if (start + size > std::numeric_limits<size_t>::max())` is both wrong and useless. It is wrong because the addition may overflow and its value is not reliable. And it is useless because the test is always false since no `size_t` value can be greater than `std::numeric_limits<size_t>::max()`.

Detecting underflow or overflow on addition with signed integers

With signed integers, the addition generates an error in the following cases.

- Overflow: Both operands are positive and the result is negative.
- Underflow: Both operands are negative and the result is positive.

Summary of underflow or overflow detection by type and operation

The following table summarizes the right ways to detect error conditions on the various arithmetical operations on signed and unsigned types. The generic names `MIN`, `MAX` and `ABS()` are used to designate the minimum value, maximum value and function returning the absolute value for the given type. In C, you have to select the right constants and functions. In C++, use the generic mechanisms of the language. The operands are named `a` and `b`; the result is named `c`.

Addition (unsigned):

```
if (a > MAX - b) {
    // ERROR
}
else {
    c = a + b;
}
```

Addition (signed):

```
c = a + b;
if ((a > 0 && b > 0 && c <= 0) || (a < 0 && b < 0 && c >= 0))
{
    // ERROR
}
```

Subtraction (unsigned):

```
if (a < b) {
    // ERROR
}
else {
    c = a - b;
}
```

Subtraction (signed):

```
c = a - b;
if ((a > 0 && b < 0 && c <= 0) || (a < 0 && b > 0 && c >= 0))
{
    // ERROR
}
```

Multiplication (unsigned):

```
if (b != 0 && a > (MAX / b)) {
    // ERROR
}
else {
    c = a * b;
}
```

Multiplication (signed):

```
if (b != 0 && ABS(a) > ABS(MAX / b)) {
    // ERROR
}
else {
    c = a * b;
}
```

Division (signed or unsigned):

```
if (b == 0) {
    // ERROR
}
else {
    c = a / b;
}
```

Due to rounding effects, the check on multiplication is a bit too aggressive. It detects all error cases but some marginal valid cases, close to the limit, may be incorrectly detected as errors. However, implementing accurate overflow detection on multiplication in the general case is very costly in terms of performance. The proposed solution is a tradeoff.

In TSDuck, the header file `tsIntegerUtils.h` declares template functions such as `ts::bound_check()`, `ts::bounded_add()` or `ts::bounded_sub()`.

[Rule] Anticipate and mitigate floating point rounding

Floating point types are less prone to underflow and overflow than integers because they are always signed and their minimum and maximum values are very large. If you really need to check for overflow and underflow, use the same technique as for signed integers.

The main problem with floating point types is rounding and the accumulation of imprecisions after an arbitrary large sequence of arithmetical operations.

Never use the equality (`==`) or inequality (`!=`) operators on floating point values because these operators check the (in)equality of binary representations without taking into account rounding and imprecisions.

There are various complex ways to safely compare floating point values. But an accurate implementation should take into account the complete sequence of arithmetical operations which led to the values to compare. For a given sequence of operations, there is a given accumulated imprecision and this imprecision must be taken into account when comparing values. In practice, this is too complicated - and way too slow - to implement.

So, we must adopt a pragmatic approach when comparing floating point values.

Example 1: If we need a relative precision, we should reduce the comparison toward `1.0` and compare it with the *epsilon* value of the floating type. However, since epsilon is defined in the C++ standard as the *"the difference between 1 and the least value greater than 1 that is representable in the given floating point type"*, the accumulated impression may have exceeded epsilon. So, in practice, it is recommended to compare to some small multiple of epsilon.

The following C++ function implements a generic numerical equality operator. It takes an additional parameter named `impr` (for imprecision) which is the small multiple of epsilon. The default value is 4 and is adequate in most cases. However, if you compare values which were created by long sequences of arithmetical operations, it is probably better to use a higher value.

```
template <typename T>
bool probablyEqual(T a, T b, int impr = 4)
{
    if (std::numeric_limits<T>::is_integer) {
        // T is an integer type, equality is valid.
        return a == b;
    }
    else {
        // T is a floating point type, equality is fuzzy.
        const T c = std::max(std::abs(a), std::abs(b));
        return c == 0.0 || std::abs(a - b) / c <= impr * std::numeric_limits<T>::epsilon();
    }
}
```

This example is just for clarity. In practice, we would implement one version for integer types and one version for floating point types, using SFINAE.

Usage:

```
float a, b;
...
if (probablyEqual(a, b)) {
    // Consider that a "pragmatically" equals b.
}
```

In TSDuck, the header file `tsFloatUtils.h` declares the template function `ts::equal_float()`.

Example 2: In some contexts, a relative precision is useless and an absolute precision is sufficient in practice. If you handle financial values for instance, the smallest amount of currency is usually a cent.

This means that everything smaller than `0.01` is irrelevant. So, regardless of the values and how they were computed, it is much simpler to write comparisons this way:

```
float a, b;
...
if (std::abs(a - b) < 0.005) {
    // Consider that a "pragmatically" equals b for financial data.
}
```

5.4.1.8. C++ classes

[Rule] The `struct` data types shall be avoided. Use classes instead.

The `struct` data types are allowed in C++ for C compatibility only. They are strictly equivalent to classes except that their fields without explicit scope are public by default (private by default in classes). So, there is no good reason to use `struct` in C++.

[Rule] Always declare a method of a class as `const` when it does not modify the object instance.

The contract is better defined. It asserts that the method will not modify the object. It also allows the method on `const` objects.

Example:

```
class C
{
public:
    void reset();        // this method will potentially modify the object
    void print() const;  // this method will not modify the object
};

void f(const C& c)
```

```
{
    c.print();   // OK
    c.reset();   // ERROR: cannot modify a const object
}
```

[Rule] Avoid public data members in class declarations.

Public data members give no control on the way the data members are used. Further maintenance is more difficult if you need to modify the internal structure of the class since you need to maintain a flawed contract.

Instead of publishing the data members, keep them private and export public getters and setters.

Good code:

```
class Foo
{
public:
    int getCount() const {return _count;}
    void setCount(int c) {_count = c;)
private:
    int _count = 0;
};
```

Bad code:

```
class Foo
{
public:
    int count = 0;
};
```

These two implementations have identical performances, thanks to the inlining of the setter and getter.

But the good code is much more maintainable. You may modify the internal structure of the class, you may completely remove the `_count` field, you may add bound checking in the setter, etc. But you will always be able to maintain the same contract (the two public methods `getCount` and `setCount`).

[Rule] When you provide a method with a `const std::string&` parameter, consider the advantages of providing an overload with `const char*` parameter. The same rule applies any other string type.

The type `std::string` provides a constructor from `const char*`. So, you may invoke a method with a `std::string` parameter using a null terminated C-string or a string literal. But, you should inspect how you use the `std::string` parameter within the method. If you simply use its null terminated C-string representation (method `std::string::c_str()`), then it is probably faster to overload the method.

The same rule applies to any string type, for instance `ts::UString`.

Slower code:

```
void f(const std::string& s)
{
    something(s.c_str());
    ...
}

std::string aString;
f(aString);
f("abcd"); // converted to std::string but
           // only used as C-string internally
```

Faster code:

```
void f(const char* s)
{
    something(s);
    ...
}

inline void f(const std::string& s)
{
    f(s.c_str());
}

std::string aString;
f(aString);
f("abcd"); // no conversion
```

[Recommendation] Try to define interface classes *when possible. An interface class describes the API for a service which can be implemented by concrete classes.*

The characteristics of an interface class are:

- All methods are public.
- All methods are pure virtual.
- There is no field.
- There is no static item.
- There is no constructor.
- There is an empty virtual destructor.

By convention, the name of an interface class ends with the suffix `Interface`.

The concept of *interface* class is not defined by the C++ standard. The C++ standard defines the concept of *abstract* class but it is less restrictive; an abstract class is a class with at least one pure virtual function.

The interface class is the C++ equivalent of the concept of *interface* in Java. The C++ interface class is interesting because it proposes a reasonable approach to multiple inheritance. Because of its characteristics, an interface class is typically limited to a header file.

Example:

```
class MutexInterface
{
public:
    virtual bool acquire() = 0;    // pure virtual method
    virtual bool release() = 0;    // pure virtual method
    virtual ~MutexInterface() {}   // empty virtual destructor
};
```

[Rule] Do not use actual multiple inheritance. A class should inherit from at most one non-interface class. A class may inherit from multiple interface classes.

Multiple inheritance is a powerful but dangerous concept. Using multiple inheritance from more than one concrete class often leads to structures which are complex and difficult to understand and maintain. After several levels of multiple inheritances, the result can be inconsistent sometimes.

The coding rules of some organizations even completely ban all forms of multiple inheritance.

Our coding rules propose a reasonable trade-off which is derived from Java: one "real" superclass and as many interface classes as necessary (the preceding recommendation defines the non-standard concept of interface class).

[Recommendation] At the beginning of the declaration of each subclass, define a type definition with the name `SuperClass` *for the superclass.*

This convention gives an easy and reliable way to explicitly invoke a method of the superclass. This is the C++ equivalent of the `super` keyword in Java.

If the class hierarchy is refactored and a new intermediate class is inserted, simply modify the type definition for `SuperClass`, there is no need to browse the entire implementation to track explicit invocations of the superclass.

Note that C++ allows multiple inheritance and it is not possible to define a single superclass in the general case. But the preceding rule limits the number of non-interface superclasses to one. This single non-interface superclass is considered as the "actual" superclass which is declared with the `SuperClass` type definition.

Exception: If a class has no non-interface superclass, the `SuperClass` type definition is omitted.

Example:

```
class A
{
public:
    void f();
};

class B: public A
{
public:
    // Standard name for superclass, modify it if you change the inheritance
    using SuperClass = A;

    // Override method
    void f()
    {
        // Explicit (and maintainable) invocation of superclass
        SuperClass::f();

        // Specific additional processing in subclass
        ....
    }
};
```

[Rule] In the documentation of a method of a class, clearly indicate if the method needs to be explicitly invoked by subclasses which override the method. If necessary, also indicate if the superclass method shall be invoked at the beginning or at the end of the subclass method.

This is entirely dependent on the implementation of the superclass. Only the developer of the superclass can decide what is necessary.

Example:

```
class File
{
public:
    // Invoke at beginning of overridden method in subclasses
    void open(const std::string& fileName);

    // Invoke at end of overridden method in subclasses
    void close();
};
```

When we specialize the class `File` to add buffering capabilities for instance, the methods `open()` and `close()` must be typically overridden to add the setup and flush of the buffer, respectively. But, while the buffer management is handled in the subclass, the file management shall remain in the superclass.

```
class BufferedFile: public File
{
public:
    using SuperClass = File;

    void open(const std::string& fileName)
    {
        SuperClass::open(fileName);
        // ... setup the buffer
    }

    void close()
    {
        // ... flush the buffer
        SuperClass::close();
    }
};
```

Exception: This rule does not apply to constructors and destructors. The compiler automatically invokes the superclass constructor at the beginning of the subclass constructor. It also automatically invokes the superclass destructor at the end of the subclass destructor.

[Rule] Never overload methods which differ only in the type of integer or pointer parameters.

In some contexts, the default integer type conversions give unexpected results. In case of maintenance of the code, you may even accidentally break the contract of the class.

Example:

```
class C
{
public:
    void f(int);
    void f(long); // BAD PRACTICE
};

int a = 1;
long b = 2;

C x:
x.f(a);
x.f(b);
```

In this example, `x.f(a)` invokes `C::f(int)` and `x.f(b)` invokes `C::f(long)`.

But let's assume that the initial version of the class `C` only had one method `C::f(int)`. The user code `x.f(b)` invoked `C::f(int)` since this was the only possible method at that time and the C++ compiler automatically downgraded `b` from a `long` to an `int` before the call.

Now, in the maintenance phase, assume that you add the `C::f(long)` method. When recompiling the existing user code, `x.f(b)` no longer invokes `C::f(int)`. Now it invokes `C::f(long)` since this is a better match. And the two methods may perform very different actions. So, by adding the overload `C::f(long)`, you break the contract and break the ascending compatibility of your class and the compiler does not even tell you it is different now.

Note 1: This rule is also valid for pointers and integers. Using the literal `0` can be indifferently interpreted by the compiler as an integer literal or as the null pointer.

Note 2: The terms *override* and *overload* shall not be confused. A method of a class *overrides* another when it redefines a method with the same name and same profile in a superclass. A method *overloads* another when it redefines a method with the same name and a different profile in the same context.

Example:

```
class A
{
public:
    void f(int i);
};
```

```
class B: public A
{
public:
    void f(int i);                 // override A::f(int)
    void g(int i);                 // overload B::g(const std::string&)
    void g(const std::string& s);  // overload B::g(int)
};
```

[Recommendation] When an overloaded method is overridden in a subclass, override all overloaded variants, unless there is a good reason to hide some of them or to add a new one.

Hum? This sounds a bit mysterious…

Counter-example: Explanations are always better with an example.

```
class A
{
public:
    void f(int i);
    void f(const std::string& s);
};

class B: public A
{
public:
    void f(int i);
};

A a;
B b;
a.f(1);       // OK, call A::f(int)
a.f("foo");   // OK, call A::f(const std::string&)
b.f(1);       // OK, call B::f(int)
b.f("foo");   // ERROR, does not compile, B::f(const std::string&) is hidden
```

In class `A`, the method `f()` is overloaded. There is one version taking an integer as parameter and one version using a character string. In class `B`, `f()` is overridden but only one version is declared, the integer one. As a consequence, the string one is hidden.

In C++, when you override a method in a subclass, all overloaded versions (methods with the same name in the superclass) are hidden. Only the explicit declarations in the subclass can be used. This is why `B::f(const std::string&)` is hidden in the above example. You cannot use it.

As a general rule, a subclass specializes a contract. It adds or replaces services but does not remove any. This rule is broken in the example.

Exception: In some cases, it can be legitimate to remove (hide) or add overloaded versions of a method. But this is tightly linked to the semantics of the object.

[Rule] Never change the default values for the parameters of a function after the function is published.

Changing the default values changes the contract.

Consider the following function:

```
void paint(Color c = BLACK);
```

All users which invoke `paint()` without argument are confident that the color will be black. If you later change the default value to `BLUE` and recompile the code, the users of the contract will obtain a different result.

[Rule] When overriding a virtual method with default parameters in a derived class, use the same default values for the parameters.

Counter-example: Let's review the consequences of breaking this rule. Assume that all shapes should be painted in black by default, except the rectangles which should be painted in blue by default.

See the following WRONG implementation:

```
class Shape
{
public:
    // "I see a red door and I want it painted black"
    virtual void paint(Color c = BLACK);
};

class Rectangle: public Shape
{
public:
    // WRONG: overriden method with other defaults
    virtual void paint(Color c = BLUE);
};
```

See why this implementation gives incorrect results:

```
Rectangle r1, r2;
Rectangle* p1 = &r1;
Shape* p2 = &r2;

p1->paint();  // Rectangle r1 is painted in BLUE
p2->paint();  // Rectangle r2 is painted in BLACK !
```

The default values for the parameters are always evaluated in the context of the caller, not the callee. When an object is accessed through a pointer to a superclass, the context of the caller is the superclass and the superclass' default parameters are applied.

This is why we impose to keep the same default values for parameters.

To solve the specific problem of the example where actual defaults should be different between the superclass and the subclass, a safe alternative would be to use overloading instead of default parameters as illustrated below.

```
class Shape
{
public:
    virtual void paint(Color c);
    virtual void paint() {paint(BLACK);}
};

class Rectangle: public Shape
{
public:
    virtual void paint(Color c);
    virtual void paint() {paint(BLUE);}
};
```

In this case, the parameter `BLUE` is evaluated in the context of the callee, which is `Rectangle::paint()`.

5.4.1.9. C++ constructors and destructors

[Rule] For a class type, always declare both default constructor and copy constructor, or none of them. If any of them should be disabled, explicitly delete them both.

The default and copy constructors are implicit if not defined by the developer. But the default implementation may not be adequate to your class. So, you shall either provide an explicit version or explicitly disable it.

If you don't and if you do not want a default or copy constructor because they are meaningless in your context, the compiler will implicitly generate wrong one for you. If any default object declaration is used, the buggy implicit constructor will be used.

Example: Disabling default and copy constructor:

```
class C
{
public:
    // Only my explicit constructors are valid (definition required)
    C(const std::string&);
    C(int, int);
private:
    // Default and copy constructors are explicitly deleted
    C() = delete;
    C(const C&) = delete;
};
```

[Rule] Always free all resources which are private to the class in a destructor.

Well-designed classes safely prevent resource leaks: memory, open files, locked resources, etc.

[Rule] Always provide a virtual destructor.

If your destructor is not virtual, it will not be invoked when an object is destructed using a delete of a pointer to one of its super-classes.

Counter-example: Class `C1` is a super-class. Subclass `C2` has a non-virtual destructor while class `VC2` has a virtual destructor. This example illustrates how instances of class `C2` can be incorrectly destructed, leading to potential resource leaks.

```
class C1 { ... };
class C2:  public C1 { public: ~C2(); };
class VC2: public C1 ( public: virtual ~VC2(); };

// later in the code:
const C1* x = new C2();
const C1* y = new VC2();
delete x;  // destructor ~C2 is NOT executed !
delete y;  // destructor ~VC2 is executed
```

[Rule] All destructors shall be idempotent.

In rare obscure cases, a destructor may be invoked twice on the same object. This may especially happen if the destructor is explicitly invoked once and will be likely implicitly invoked later when the object goes out of scope.

Thus, the implementation of a destructor shall be coded so that it is safe when invoked more than once. This is the meaning of idempotent.

Counter-example: Non-idempotent destructor:

```
class C
{
private:
    int* _p;
public:
    C(): _p(new int[8])
    {
    }
    virtual ~C()
    {
        delete[] _p; // double deallocation if invoked twice
    }
};
```

Example: The following alternative destructor is idempotent:

```
virtual ~C()
{
    if (_p != nullptr) {
        delete[] _p;
        _p = nullptr;
    }
}
```

[Rule] In a constructor or destructor, never invoke any virtual method of the class.

During the execution of the constructor and destructor, the *vtable* of an object instance points to the virtual methods of the class which defines the current constructor or destructor. On the other hand, during the rest of the lifetime of the object, the *vtable* points to the virtual methods of the actual class of the object. This can be very misleading and hard to debug.

Counter-example: Consider the following class `C1` which incorrectly invokes a virtual method in the constructor and destructor.

```
class C1
{
public:
    virtual void vf(const std::string& s);

    C1() // constructor
    {
        vf("in constructor");
    }

    ~C1() // destructor
    {
        vf("in destructor");
    }

    void f() // some standard method
    {
        vf("in method f()");
    }
};
```

Now consider a subclass C2 which overwrites the virtual method.

```
class C2: public C1
{
public:
    virtual void vf(const std::string& s);
};
```

Given the nature of virtual methods, for a given instance of C1 or any of its subclasses, it may be expected that the same virtual method vf() will be used in all contexts. But this is not true. Let's assume that vf() simply displays a message:

```
void C1::vf(const std::string& s)
{
    std::cout << "C1::f: " << s << std::endl;
}

void C2::vf(const std::string& s)
{
    std::cout << "C2::f: " << s << std::endl;
}
```

Consider the following applications code:

```
int main()
{
    C2 c;
    c.f();
}
```

The virtual method vf() of the instance c is invoked three times. But this is not the same vf() in all cases. The application displays this:

```
C1::f: in constructor
C2::f: in method f()
C1::f: in destructor
```

Thus, invoking virtual methods in constructors and destructors is error-prone. This may even fail if the methods are pure virtual in the superclass.

Debugging hint: If an application fails with a message similar to "pure virtual method was called", look for invocations of virtual methods in constructors and destructors.

[Rule] All constructors shall properly initialize all its super-classes and member fields.

This is an application of a previous rule: all data shall be initialized.

[Rule] In the definition of a constructor, the field initializers must be in the same order as their declarations in the class declaration.

The order is significant since the field initializers can make a reference to each other. In the recommended paranoid warning mode, some compilers even reject the constructor definition when the field initializers are in a different order from the class declaration.

Example:

```
class C
{
public:
    C(int x);
private:
    int a;
    int b;
    int c;
};

C::C(int x):
    a(0),
    b(x),
    c(x + 1)
{
    ...
}
```

[Recommendation] If a constructor has only one argument or if the second and subsequent arguments have default values, this constructor shall be prefixed by the `explicit` keyword.

Invoking a constructor with one argument is implicitly used by the compiler to perform an automatic type conversion. In some cases, this is the right thing to do. But in most cases, this isn't. As a general rule, you must analyze the semantic of a constructor with one argument. Does it make sense to convert back and forth between the class of the constructor and the class of the first argument? If the answer is yes, then leave the constructor alone. If the answer is no, use `explicit`. In this context, the keyword `explicit` means that type conversions shall be explicit and the compiler will never user this constructor for implicit type conversions.

Example: In the following code, we declare two classes, `City` and `House`, which implement two different concepts. It does not make sense to implicitly convert a `House` into a `City` or vice-versa. But it is legitimate that an instance of `House` has a property of class `City` because a house is located into a city. So, we provide a `House` constructor with an argument of type `City`, specifying the location of the constructed house.

```
class City { ... };

class House
{
public:
    explicit House(const City& location, const char* name = "");
};
```

Since the second argument of the constructor has a default value, it can be invoked with only one argument of type `City`. To avoid the accidental implicit conversion of a `City` into a `House`, we use the keyword `explicit`.

Let's examine the effect of the absence of `explicit` keyword using the following incorrect code.

```
void selectHouse(const House& h);

City paris;
selectHouse(paris);  // incorrect usage, we want to select a house, not a city
```

The function `selectHouse()` is probably used to select a specific existing house. So, calling `selectHouse()` with an argument of type `City` is incorrect and should be rejected by the compiler. Indeed, there is a compilation error here, thanks to the `explicit` keyword. Without this keyword, however, the compiler would implicitly use the constructor to convert the object of type `City` into `House` and `selectHouse()` is called with a new `House` object with all default values in the city of paris. This does not make sense.

Exception: When the class of the constructor and the class of its first argument have equivalent semantics with different representations, it makes sense to allow conversions between the two types. The standard type `std::string` has a constructor with one argument of type `const char*` (i.e. a C-string). The two types are semantically identical, they are character strings. So, converting between the two is both legitimate and useful. When a function uses a parameter of type `std::string`, it is useful to be able to call it with a string literal such as `"foo"` (which has type `const char*`). The compiler implicitly creates a new object of type `std::string` from the string literal. This would not be possible if the `std::string` constructor had the keyword `explicit`.

5.4.1.10. C++ operators

[Rule] Always provide an assignment operator with the same scope as the copy constructor.

The two operations, assignment operator and copy constructor, are closely related. They should be both enabled and implemented or both disabled.

Both enabled:

```
class C
{
public:
    // implementation required
    C(const C&);
    C& operator=(const C&);
};
```

Both disabled:

```
class C
{
private:
    // explicitly deleted
    C(const C&) = delete;
    C& operator=(const C&) = delete;
};
```

[Rule] When provided, the copy constructor and the assignment operator shall have consistent implementations.

The following two sequences shall have identical results.

Copy constructor:

```
C a;
C b(a);
```

Assignment operator:

```
C a;
C b;
b = a;
```

But beware that, although consistent, the two implementations are usually not identical. The copy constructor works on an uninitialized object with no pre-existent state. The assignment operator, on the contrary, works on an initialized object which may need some clean-up before copy.

[Rule] Prevent self-assignment in an assignment operator.

A self-assignment `a = a` is a valid but void operation which must be filtered specifically.

Example:

```
class C
{
public:
    C& operator=(const C&);
    ...
};

C& C::operator=(const C& other)
```

```
{
    if (this != &other) {
        // implement actual assignment here
        ...
    }
    return *this;
}
```

Consider a class which encapsulates complex dynamic data structures. Assume that you decide that the semantics of the assignment operator is a deep copy. In the assignment operator, you must first free the previous content of the object and then duplicate the new content from the assigned value. If the test `if (this != &other)` is not present, the effect of the self-assignment would be a *reuse-after-free* bug.

[Rule] In the assignment operator of a derived class, make sure that the superclass fields are properly assigned using an explicit invocation to the superclass assignment operator.

Failing to do this may leave the superclass fields in an inconsistent state.

Example:

```
class D: public C
{
public:
    using SuperClass = C;
    D& operator=(const D&);
    ...
};

D& D::operator=(const D& other)
{
    if (this != &other) {
        // assignment of superclass fields through explicit invocation of superclass
        SuperClass::operator=(other);
        // implement actual assignment of subclass fields here
        ...
    }
    return *this;
}
```

[Rule] All assignment operators (`=`, `+=`, `-=`, etc.) shall return `*this`.

This is required to be consistent with the semantic of these operators.

[Rule] The comparison operators, when implemented in a class, must be consistent with each other.

If you implement the operator `==`, be sure to also implement `!=` and ensure that for any instances `a` and `b` of the class:

```
(a != b) == !(a == b)
```

Similarly, if you implement any of `<`, `<=`, `>`, or `>=`, you must implement them all and ensure that for any instances `a` and `b` of the class:

```
(a < b)  == !(a == b) && !(a > b)
(a <= b) == !(a > b)
(a > b)  == !(a == b) && !(a < b)
(a >= b) == !(a < b)
```

In practice, it is only necessary to provide a deep implementation for operators `==` and `<`. All other operators can be implemented using references to the first two.

In C++20, it is no longer possible to define the operator `!=` when you have defined `==`. The `!=` is automatically derived from `==`. To face the challenge of compiling the same code with C++17 and C++20, TSDuck defines the macro `TS_UNEQUAL_OPERATOR` for `!=`.

Example:

```
class C
{
public:
    bool operator==(const C& other) const;
    TS_UNEQUAL_OPERATOR(C)
};
```

When compiled in C++17 mode, the macro `TS_UNEQUAL_OPERATOR` declare an inline function which returns `!operator==(other)`. When compiled in C++20 mode, the macro does nothing.

[Rule] When you do not use the final result of an expression using the increment (`++`) or decrement (`--`) unary operators, prefer the prefixed notation (`++i`, `--i`) to the postfix notation (`i++`, `i--`).

The postfix notation needs to create a temporary object holding the previous value of the object and use this temporary object as the result of the expression.

For integer or pointer types, this is usually optimized away by the compiler. But for more complex object classes which redefine these operators, the useless construction and destruction of the temporary object cannot be avoided. And this can cost some time for nothing.

This is especially the case for the iterators in the standard template library. The following example illustrates two functionally identical ways of walking through a list of int but the second way is uselessly slower.

```
std::list<int> x;

// Use ++i on iterator: good
for (std::list<int>::iterator i = x.begin(); i != x.end(); ++i) {
    ....
}

// Use i++ on iterator: identical but uselessly slow
for (std::list<int>::iterator i = x.begin(); i != x.end(); i++) {
    ....
}
```

5.4.1.11. C++ object management

[Rule] Never use `malloc()` and `free()` in C++, use the `new` and `delete` operators.

The C memory allocation routines return raw memory, not objects. Casting the result of `malloc()` to a pointer to an object instance is incorrect. This memory area is not a properly initialized object.

On the contrary, the C++ allocation operators manipulate objects. They invoke the proper constructors and destructors.

[Rule] Objects which were allocated using `new[ ]` must be deallocated using `delete[ ]`.

This is a really annoying feature of C++. The operators for allocating single objects and arrays are different and the corresponding `delete` operator must be used. But when a pointer is declared as `Foo*`, there is no implicit indication how the pointed object was allocated. The developer must take care of this.

[Recommendation] Never use dynamically allocated arrays (operator `new[ ]`).

The preceding rule demonstrates that using the operator `new[ ]` may create confusion when it comes to deallocation. Moreover, a previous recommendation explicitly recommends avoiding arrays of C++ objects for other reasons, whether they are statically or dynamically allocated.

Use standard containers from the C++ Standard Template Library (STL) instead of dynamically allocated arrays. The standard container `std::vector`, for instance, is a much better alternative.

If you think that you need an array because you will pass it to a C routine which requires the address of an array, a `std::vector` is still better than a dynamically allocated array. The C++ Standard specifies than the underlying representation of the current content of a `std::vector` must match the representation of an array. Thus, the following sample code is both legal and safe.

```
// A C routine which expects an "array"
void legacyFunction(int* buffer, size_t intCount);

std::vector<int> v;
v.resize(100);  // or fill the vector the way you like
legacyFunction(&v[0], v.size());
```

[Rule] Check the size of an instance of `std::vector` before using the index operator `[ ]`.

To reference an element in a vector, the C++ standard specifies that the method `std::vector::at()` performs bound checking while `std::vector::operator[]` does not. The difference is typically for performance reason. It is up to the developer to choose between safety and speed of code for each access.

Example:

```
int x = 0;
std::vector<int> v(4);  // declare a vector with 4 elements
x = v.at(6);            // throw exception std::out_of_range
x = v[6];               // read an invalid value in uninitialized memory
```

Some implementations of the C++ STL add an assertion on the index value in `std::vector::operator[]`. This means that, in case of invalid index value, the application is aborted when compiled in debug mode (assertions on). In production mode, the assertions are off and the invalid index value is unnoticed.

This is normally fine. You should not use invalid index values anyway and the assertion helps the developer in debug mode. But this is not always the case. There are rare cases where using an invalid index is correct and the application should not fail in debug mode. See the following counter-example.

Counter-example:

```
void legacyFunction(int* buffer, size_t intCount);

std::vector<int> v;
legacyFunction(&v[0], v.size());
```

If the vector `v` is empty, `0` is an invalid index value and simply getting the address of `v[0]` may fail in debug mode. However, this is valid code since `v.size()` is zero and any address value, even an incorrect one, is fine as first parameter for `legacyFunction()` because the function will not access the memory area anyway.

This is the case for the Microsoft Visual C++ library.

Be sure to identify that kind of usage and rewrite the code as follow:

```
legacyFunction(v.empty() ? nullptr : &v[0], v.size());
```

[Rule] In function declarations, when a parameter is not an elementary type, not a pointer type or is a template type, always use a reference. If the contract is to not modify the object, use a `const` qualifier.

Not using a reference forces a copy of the object. This can be costly and this may even not compile in the case of an actual template parameter type without copy constructor.

Good code:

```
void f(const Foo& x);

template <typename T>
void g(const T& x);
```

Bad code:

```
void f(Foo x);   // force a copy of a Foo
object

template <typename T>
void g(T x);     // call will not compile if
actual type
                 // for T has no copy
constructor
```

[Rule] When catching an exception, always use a reference to avoid a copy of the object.

This is an application of the preceding rule to the exception handling.

Example:

```
try {
    ....
}
catch (const std::exception& e) {  // reference to const exception object
    ....
}
```

[Rule] Multiple exception handlers in a `try` / `catch` structure must be ordered properly.

In a `try` / `catch` structure with multiple exception handlers, an exception is checked against all `catch` clauses until a matching one is found. Only the first matching one is used. So, when exception classes are derived from each other, the most specific `catch` clauses must come first. Otherwise, they are useless.

Example:

```
class A: public std::exception {...};
class B: public A {...};
```

Good code:

Bad code:

```
try {
    ...
}
catch (const B& e) {
    // catch all B exceptions
}
catch (const A& e) {
    // catch all A exceptions, including
subclasses
    // of A, except B which are handled above
}
```

```
try {
    ...
}
catch (const A& e) {
    // catch all A exceptions,
    // including all subclasses of A
}
catch (const B& e) {
    // never used since B is a subclass of A
    // and was already caught in previous
handler
}
```

[Rule] Use the *guard* ***design pattern to fail safely. Define guard classes for resources. Use destructors to fail safely.***

It is sometimes necessary to reserve, lock or allocate a resource temporarily. This means that something shall be done at the end of a sequence of statements (release, unlock, deallocate). A problem occurs when an exception is thrown or a return is inadvertently executed in the middle of the sequence. In this case, the resource does not get cleaned up.

The *guard* design pattern is a technique to avoid this. For a given type of resource which needs to be locally reserved, define an associated guard class which has basically only two operations: a constructor which reserves the resource and a destructor which releases the resource (whatever *reserve* and *release* means for the given resource). For each sequence of code which reserves the resource, simply create a block of `{ }` where a local guard object is defined.

Example: This is how the guard pattern is used.

Unsafe code:

```
Resource myResource;
...
myResource.reserve();
...
...
myResource.release();
```

Equivalent safe code using a guard class:

```
Resource myResource;
...
{
    ResourceGuard(myResource); // implicit
reserve()
    ...
    // some exception or return here
    ...
} // implicit release() even on exception or
return
```

Example: This is how the guard pattern can be implemented.

```
class Mutex
{
public:
    void acquire() {...}
    void release() {...}
};

class MutexGuard
{
public:
    MutexGuard(Mutex& mutex): _mutex(mutex) {_mutex.acquire();}
    ~MutexGuard() {_mutex.release();}
private:
    Mutex& _mutex;
};
```

And this is how this implementation is used.

```
Mutex globalMutex;
...
{
    MutexGuard guard (globalMutex); // implicit globalMutex.acquire()
    ...
    // some exception or return here
    ...
} // implicit globalMutex.release() even on exception or return
```

Application: Use the standard guard type `std::lock_guard` on `std::mutex` and `std::recursive_mutex`.

[Rule] Use the safe pointer design pattern to prevent memory leaks and complex memory tracking.

When comparing C++ to Java, the main C++ nightmare is the memory management: when should I free memory?

A *safe pointer* (or *smart pointer*, or *shared pointer*) is a design pattern which replaces the usage of plain pointers by a template class which tracks all accesses to a resource. When no more active reference to the object exists, the object is automatically reclaimed.

Great care shall be taken when designing the safe pointer class (especially the multi-threading aspect). But once available, C++ memory management becomes almost as easy as Java.

In early versions of TSDuck, before using C++11, a dedicated template class was designed. Now, TSDuck uses the standard template type `std::shared_ptr` for all complex pointer managements.

Based on `valgrind` results on Linux platforms, TSDuck has proven to be memory-safe when using safe pointers and zero explicit deallocation on any arbitrary large number of dynamically allocated objects with a limited life-time.

Warning: There are pathological cases where the safe pointer design pattern is ineffective. For instance, when two objects reference each other but are no longer referenced anywhere else, they are lost. They are not automatically freed by the safe pointers because references exist. They should be both freed at the same time. But this situation is typically the symptom of a poorly designed data model. So, the safe pointer design pattern removes the burden of the technical aspects of the memory management but not the requirement for a correct design.

[Recommendation] Take care of reentrancy in reusable components. Use an abstract mutex interface and template reusable components.

If a class is not thread-safe, it is possible to make it thread-safe on the user’s request without much effort and without performance penalty when thread-safety is not required.

Solution 1: Compile-time selection of the thread-safety model.

Define two mutex classes with identical services.

```
class NullMutex
{
public:
    // acquire() and release() are
    // empty and inlined as no code
    void acquire() {}
    void release() {}
};
```

```
class RealMutex
{
public:
    // Implement acquire() and release()
    // in the .cpp file
    void acquire();
    void release();
};
```

The second class implements the actual locking features on one or more environments. Now consider that the reusable component to implement is a safe pointer (see preceding rule). Define the safe pointer class as follow:

```
template <typename T, class MUTEX> class SafePointer {...};
```

In the implementation of the SafePointer class, define an object of the generic type `MUTEX` to implement the locking. Whenever you need a thread-safe or non-thread-safe pointer to objects of class Foo, use one of the following:

```
SafePointer <Foo, RealMutex> threadSafePointer;
SafePointer <Foo, NullMutex> nonThreadSafePointer;
```

The first pointer class uses thread-safe code while the second one uses fast code (the inline empty locking services are optimized away by the compiler). There is exactly zero overhead for the non-thread-safe version but the two versions share the same source code.

Solution 2: Run-time selection of the thread-safety model.

Define a general abstract mutex interface which declares the basic synchronization services as pure

```
virtual functions:
class MutexInterface
{
public:
    virtual void acquire() = 0;
    virtual void release() = 0;
    virtual ~MutexInterface() {}
};
```

Define two subclasses `NullMutex` and `RealMutex`. The first one implements inline empty services which do nothing. The second one implements the actual locking features on one or more environments.

A reusable component can, based on some run-time condition, allocate either a `NullMutex` instance or a `RealMutex` instance. In the non-thread-safe case, the `acquire()` and `release()` operations are simple indirect calls to an empty routine which returns immediately.

Comparison: Performances of the two versions are almost identical in the thread-safe case. The solution 1 offers the best performance in the non-thread-safe case. But it is slightly more constraining: the selection is done at compile-time, the mutex guard class must be a template one and the code footprint is larger when both thread-safe and non-thread-safe usages of the same reusable component are present in the same application.

In TSDuck, we use standard mutex types such as `std::mutex` or `std::recursive_mutex`. To implement the solution 1, TSDuck also defines the class `ts::null_mutex` which declare the standard methods `lock()`, `unlock()`, and `try_lock()` as empty inline methods. An instance of `ts::null_mutex` can be used wherever a `std::mutex` or `std::recursive_mutex` could be used, but doing nothing at zero cost.

[Rule] Do not use C-style casts. Use C++ cast operators `reinterpret_cast`, `dynamic_cast`, `static_cast` and `const_cast`.

The C++ cast operators provide a better control over which type of casting you want. More appropriate checks are performed by the compiler or even at run-time (`dynamic_cast`). The code is also more readable, your intention is more clearly explained to the maintainer.

The most general cast operator is `reinterpret_cast` and is equivalent to a C-style cast. There is no check at all. It is dangerous and most of the time inappropriate. Usually, we need to cast between related types and `static_cast` or `dynamic_cast` is a better option.

Beware of `const_cast` when it is used to remove the "constness" of an object. By doing so, you break the API contract if the `const` object is provided by your caller. Is the `const_cast` required because you invoke another API which is badly defined (an argument is not declared as `const` but not modified anyway)? Or is the `const_cast` required because you will actually modify the object? While the former situation is legitimate, the latter is not.

[Recommendation] Use `dynamic_cast` to check a subclass type.

If a general method working on a superclass needs some specific processing when the object belongs to a given subclass, this is usually the symptom of a bad design. A virtual method should be defined in the super-class.

But sometimes it is not possible to modify the superclass and it is necessary to use specific code. In this case, `dynamic_cast` is the only reliable and safe way to test the subclass of the object.

Example:

```
class C {...};
class C1 : public C {...};
class C2 : public C {...};

void f(const C& c)
{
    // generic processing on super-class view "c" of the object
    c.someMethod();

    // is this object an instance of C2?
    const C2* p2 = dynamic_cast<const C2*>(&c);
    if (p2 != 0) {
        // yes, the object is an instance of C2
        // specific processing on C2 sub-class view "*p2" of the object
        p2->someMethodOfC2();
    }
}
```

Note 1: Be aware that `dynamic_cast` is allowed only if the superclass is polymorphic, i.e. if it has at least one virtual method. A virtual destructor is sufficient.

Note 2: Using `dynamic_cast` requires that the C++ compiler supports Run-Time Type Information (RTTI). All modern C++ compilers on desktop and server platforms support RTTI. But, on some constrained embedded platforms, the C++ compiler may not support RTTI and `dynamic_cast` is rejected by the compiler. In that case, you have to determine the actual subclass by some other way and then use `static_cast` after verifying the actual subclass type.

[Rule] Do not use `exit()` or `std::exit()`, except in case of emergency termination.

In C++, `exit()` only guarantees the proper destruction of global objects. The destructors of pending local objects are not invoked. If those destructors have external effects (flushing a cache, closing a file, etc.), the state of some external resources may not be consistent.

To perform an early exit of the program, throw an exception. Use a dedicated application-defined exception and ensure that no function catches this exception (beware of generic `catch (…)` structures). In multi-threaded applications, this is a little bit more complicated; you need to synchronize the termination of all threads.

In the main program, to return an error code to the operating system, do not call `exit()`, perform a return instruction with the appropriate exit code.

[Rule] Do not use `va_start`, `va_arg` and `va_end`, especially on class types.

These macros were designed for C and not C++. They do not properly invoke constructors.

In C++, variable argument lists are handled in a much safer way using `std::initializer_list` and variadic templates.

In TSDuck, the class `ts::ArgMix` and its subclasses are designed to transparently support type-safe heterogeneous variable argument lists. Sample usages can be found in methods `ts::UString::format()` and `ts::UString::scan()`.

[Rule] Never declare a function with an argument being an array of objects when subclasses exist for this class.

This practice has very dangerous side effects on subclasses. This introduces bugs which are very hard to track.

Example:

```
class A
{
public:
    int a;
    A(): a(1) {}
};

class B: public A
{
public:
    int b;
    B(): b(2) {}
};

void f(A array[], size_t elemCount)
{
    // update second element of array
    if (elemCount >= 2) {
        array[1].a = 47;
    }
}

B x[2];
f(x, 2);  // do you think that x[1].a is updated ?

std::cout << x[0].a << ", " << x[0].b << ", "
          << x[1].a << ", " << x[1].b << std::endl;
```

The last instruction prints "1, 47, 1, 2". The function `f()` has corrupted the subclass fields of `x[0]`.

The same buggy effect is obtained when the function is defined using a pointer instead of an array:

```
void f(A* array, size_t elemCount)
```

Alternatives: If you really need that function, you must overload it for all possible subclasses of A. And if new subclasses of A are created in the future, you must remember to overload the function for all new subclasses. Since this is a maintenance challenge, this kind of function should really be avoided anyway.

[Recommendation] Avoid using arrays of C++ objects. Use standard containers from the STL instead.

The preceding rule is a good illustration of the danger of arrays of objects in C++.

There is almost no case where using an array is better than using the standard container `std::vector`, neither in performance nor in functionalities.

[Rule] Never assign objects through dereferencing a pointer to a base class.

The assignment operator cannot be virtual (at least in its strict definition). By using the assignment through dereferencing a pointer to a base class, the actual assignment operator is the one from the superclass. The assignment of the object will be partial (*slicing effect*) and the object can be left in an inconsistent state.

Example:

```
class Fruit
{
public:
    Fruit& operator=(const Fruit&);
    ...
};

class Apple: public Fruit
{
public:
    Apple& operator=(const Apple&);
    ...
};

class Tomato: public Fruit
{
public:
    Tomato& operator=(const Tomato&);
    ...
};

Apple  apple;
Tomato tomato;

Fruit* p1 = &apple;
Fruit* p2 = &tomato;

*p1 = *p2;  // Legal but WRONG !
```

The above assignment is legal and compiles. However, since the assignment operator cannot be virtual, it invokes the assignment operator of the superclass, i.e. the method `Fruit::operator=(const Fruit&)`. In practice, the common `Fruit` fields of an `Apple` object will be assigned with the common `Fruit` fields of a `Tomato` object. The specialized `Apple` fields of the object are left unmodified and, thus, possibly inconsistent with the new values of the common `Fruit` fields.

Honestly, this is weird to assign a tomato into an apple, even if both are fruits!

We reach here the limitation of the virtual vs. non-virtual method model. This model works fine as long as the virtual / non-virtual method works on one object only (this object). But when the method is designed to work on two objects globally, as it is the case for the assignment, there is no good solution.

5.4.2. C++ coding conventions

This section is present to fulfil the required separation of immutable *rules* and *recommendations* from potentially replaceable *conventions*, as explained in section 5.2.

5.4.2.1. Source code formatting

[Convention] The file name extensions by file type are `.h` for C++ header files and `.cpp` for C++ source files.

For C++ files, several conventions exist: `.h`, `.hpp`, `.hxx`, `.H` for headers, `.cpp`, `.cxx`, `.C` for source files. The selected convention has a large adoption and is good enough.

[Convention] Indentation: use 4 space characters without tabulation.

Using less than 4 spaces is not clear enough. Using 8 spaces moves too fast to the right.

5.4.2.2. Modularity

[Convention] When a module is specialized in the management of one data type (object-oriented design), the base name of the module files is the data type name, using the same lower/upper case letters.

Example: The class `ts::UString` is declared in file `tsUString.h` and defined in file `tsUString.cpp`.

[Convention] The file name of a module containing a C++ class is built from the concatenation of all nested namespaces and the class name.

This way, the class declaration can be easily found.

Example: The class named `ts::xml::Element` is declared in file `tsxmlElement.h` and defined in file `tsxmlElement.cpp`.

[Convention] Non-inline template methods or methods of template classes are grouped at the end of the header file, in a clearly identified section.

The portability of the C++ templates is a challenge. Most compilers require the definition of the template methods to be available during the compilation of the modules which use them. So, we need to have them in the header file.

5.4.2.3. Naming conventions

There are many naming conventions in the C and C++ communities. This is sometimes both the most sensitive part for the developers and the less important for the quality of the code. Discussing which naming convention is the best one is a waste of time. There is no best one. But many are good enough. The quality of the code only depends on the strict and consistent application of one single good enough naming convention.

This section describes the naming conventions for the C++ language in the TSDuck project. Simply use them.

[Convention] All entities which are defined within the project shall be declared within the namespace named `ts`.

This is a trade-off between readability and usability. Using a more significant but longer namespace such as `tsduck` would appear as painful to developers who could be tempted to disobey the *"no using namespace directive"* rule.

[Rule] Multiple nested namespaces may be defined within the namespace ts.

Typically, there is one inner namespace per subproject or logical group of classes.

Developers are encouraged to use short but meaningful namespaces (4 or 5 characters maximum). Using acronyms is acceptable.

Typical nested namespaces are `ts::xml`, `ts::json`, `ts::tlv`.

[Convention] Namespaces are composed of lowercase letters or digits only.

No capital letter, no underscore.

As mentioned in the preceding rules, all entities are in the namespace `ts` or in one of its inner namespaces.

[Convention] Preprocessing macro names (`#define`) are composed of uppercase letters and digits only. Words are separated by underscores. All macro names start with the prefix `TS_`.

Example:

```
#define TS_FOO_VERSION 47
#define TS_INCR(x) (...)
```

[Convention] Use a verb as the central component of the name for functions that do something. Use an imperative form for the name of functions returning a boolean value. Such functions typically start with `is` or `has`, depending on what they return.

Example:

```
namespace ts {
    class Foo
    {
    public:
        Foo(...);
        ~Foo();
        void load(...);
        void save(...) const;
        bool isEmpty() const;
        bool hasChildren() const;
    private:
        // ... internal fields
    };
}
```

[Convention] Class names, type names, public static functions, non-member (global) functions use a mixed case convention, without underscore, starting with an uppercase letter.

See examples below.

[Convention] Non-static public member fields and functions use a mixed case convention, without underscore, starting with a lowercase letter.

When reading code, it is useful to differentiate static and non-static members. Static members are class-wide; their name starts with an uppercase letter. Non-static members applies to one instance; their name starts with an lowercase letter.

See examples below.

[Convention] Public member fields and local variables (in functions) use either a mixed case convention, without underscore, starting with a lowercase letter, or an all-lowercase convention with underscores.

The dual convention is unfortunate in TSDuck. This is the result of history. However, as mentioned in section 5.3.1.7, *"do not rewrite existing code for the sole purpose of applying coding guidelines"*.

[Convention] Public constants use uppercase letters with underscores to separate words.

See examples below.

[Convention] Private entities of any sort use the same convention as their public counterpart but start with an underscore.

When reading code, it is important to understand the scope of an action, public or private. Modifying a public field may break the interface contract of the class while modifying a private field only requires a local analysis. Private fields are identified by their leading underscore.

Example:

```
namespace ts {
    void GlobalFunction();                    // global function, not in a class
    typedef Foo* FooPtr;                      // type name
    class ClassName                           // class name
    {
    public:
        static const size_t MAX_SIZE = ...;   // constant
        int somePublicField;                  // public member field
        void someMemberFunction();            // public member function
        static void SomeStaticFunction();     // public static function
    private:
        int _somePrivateField;                // private member field
        int _some_private_field;              // also acceptable
        void _somePrivateMemberFunction();    // private member function
        static void _SomePrivateStaticFunc(); // private static function
    };
}

void ts::ClassName::someMemberFunction()
{
    uint32_t messageIndex;                    // local variable
    uint32_t message_index;                   // also acceptable
}
```

[Convention] In all mixed case conventions, acronyms are all-uppercase, or all-lowercase at the beginning of the name.

Example:

```
typedef uint16_t TCPOrUDPPort;
TCPOrUDPPort tcpDefaultPort = 80;
static TCPOrUDPPort _udpDefaultPort = 80;
```

[Convention] Values in enum types are composed of uppercase letters and digits only. Words are separated by underscores. In pure enum types (ie. not enum classes), a common prefix which is derived from the type name is prepended to all values.

Example:

```
namespace ts {
    enum Counter {
        COUNTER_ONE,
        COUNTER_TWO
    };
}
```

[Convention] Conditional compilation of debug code is allowed using the symbol `DEBUG`. When `DEBUG` is

undefined, no debug code shall be compiled.

This symbol shall not be defined in any header file. It shall be defined by the compilation environment (makefile, IDE compilation profile, etc.)

As an exception, the macro `DEBUG` does not start with the required `TS_` prefix because `DEBUG` is a common symbol which is recognized by many libraries to trigger debug-specific code.

5.4.2.4. Syntax formatting conventions

Just like naming conventions, the syntax formatting conventions are subject to discussion. And, similarly, there is no best one. We must simply adopt one and use it consistently in all code. Most IDE's can be configured to automatically enforce these settings when typing code. This is possible with Emacs, Eclipse, Microsoft Visual Studio or Qt Creator.

[Convention] Add a space character in the following locations: before and after binary or ternary operators, before a group of one or more `(` in an expression, after a group of one or more `)` or `]` in an expression, after a `,`.

Exception: no space before `,` or `;`.

Example:

```
a = (b + c) * ((e / 4) % 3);
x = y > size ? y : size;
p = f(a, b, c[i] + 1);
```

[Convention] Do not use any space character before `(` in a function declaration, definition or call. Similarly, do not use any space character before `[` in an array declaration or reference.

Example:

```
void foo(char* name, size_t size);
char buffer[200];
buffer[0] = 'a';
foo(buffer, sizeof(buffer));
```

[Convention] Comment lines must be aligned on the code they refer to.

Good code:

```
void f(int x)
{
    // about the test
    if (x > 1) {
        // about the zero
        x = 0;
    }
}
```

Bad code:

```
void f(int x)
{
 // about the test
    if (x > 1) {
            // about the zero
        x = 0;
    }
}
```

[Convention] Multi-lines comments shall be formatted using `//` on each line. All lines which belong to the same logical comment shall be aligned.

A previous rule explains why single-line C++-style comments are safer than multi-line C-style comments.

Example:

```
// This is a comment
// on several lines.
```

Exception: For self-documentation (Doxygen for instance), use the required conventions for the corresponding documentation extraction tool.

[Convention] The function definition has its initial* `{` *on a new line. The parameters are listed on one line, if this makes the line not too long.

Example:

```
void FunctionCode(const char* name, size_t size)
{
    ...
}
```

[Convention] In a function declaration, definition or call, when the parameter list is too long to fit on one line, the parameters are aligned on the opening parenthesis and there must be exactly one parameter per line.

Example:

```
MyFunction("abcd", size, 47);  // short line

MyFunction(tsCallingSomethingVeryVeryLongAndUnreadable(a + 45 * x, "title"),
           size,
           47);    // long statement: one parameter per line
```

In the first example, using one parameter per line would be counter-productive. The code would be bloated and less readable. So, if everything fit on one line, use one line.

In the second example above, putting the two parameters `size, 47` on the same line could make the careless reader think that there are only two parameters to the function.

So, either put all parameters on one line or one parameter per line, but not a mixture of the two.

[Convention] The conditional and loop statements have their initial* `{` *on the same line. The closing* `}` *is on its own line. The* `else if` *construction is on one line.

Example:

```
if (x == a) {
    ...
}
else if (y < b) {
    ...
}
else {
    ...
}
```

[Convention] The switch statement has its initial* `{` *on the same line. The various* `case` *entries and the optional default entry are on individual lines. They are indented from the* `switch`*.

Example:

```
switch (state) {
    case TS_START:
        print("started");
        break;
    case TS_CONTINUE:
    case TS_END:
        print("ok");
        break;
    default:
        print("error");
        break;
}
```

[Convention] The namespace declaration has its initial `{` on the same line. The closing `}` is on its own line. The content of the namespace is indented.

Example:

```
namespace ts {
    namespace proj {
        void foo();
    }
}
```

[Convention] The `class` declaration has its initial `{` on a new line. The `public`, `protected` and `private` keywords are aligned on the `{`.

Example:

```
class C
{
public:
    void init();
private:
    void _reset();
};
```

Remember that the closing `}` of a class declaration must be followed by a `;`. This is a typical C++ oddity and a common source of compilation syntax errors.

[Convention] In template class declarations and definitions, the template part is on a separate line with the same alignment as the declaration or definition.

Example:

```
template <typename T, class MUTEX>
class SafePointer
{
     ...
};

template <typename T, class MUTEX>
ts::SafePointer<T,MUTEX>& operator=(T* p)
{
     ...
}
```

[Convention] In the definition of a constructor, the field initializers are indented and separated one per line.

Example:

```
class C
{
public:
    C(int x);
private:
    int _a;
    int _b;
    int _c;
};

C::C(int x):
    _a(0),
    _b(x),
    _c(x + 1)  // be careful, no comma on last field initializer
{
    ...
}
```

[Convention] If a sequence of stream output operators `<<` is too long to fit on one line, the `<<` operators are aligned on multiple lines.

Example:

```
std::cout << "title: " << title
          << ", message: " << message
          << std::endl;
```

5.4.2.5. Doxygen self-documentation

[Convention] All Doxygen commands in embedded comments in source files shall be introduced with the `@` character (e.g. `@file`, `@param`, `@return`, etc.)

There are two syntaxes for Doxygen commands, starting with a `@` or with a `\`. For consistency, all developers shall use one single common convention. The syntax using `@` is preferred since it is compatible with Javadoc.

[Convention] In C++ source files, the Doxygen comment blocks shall be formatted using `//!` on each line. All lines which belong to the same logical documentation block shall be aligned.

Example:

```
 //!
 //! ... Doxygen documentation commands
 //!
```

There are two main syntaxes for Doxygen comment blocks in C++, one using the C-style comment `/**` and one using the C++-style comment `//!`. For consistency, all developers shall use one single common convention.

A previous rule explains why single-line C++-style comments are safer than multi-line C-style comments.

[Rule] When an entity is in a Doxygen-documented source file, all its components shall be documented without exception.

The brief description of the entity (`@brief`) is mandatory. The detailed description (`@details`) is optional if the entity is so simple that the brief description is sufficient.

Functions and methods shall document all their parameters and their returned value (if any).

Enumeration types shall document all their values individually in addition to the documentation of the type.

Hint 1: The mandatory documentation of all fields, functions, methods, and their parameters can be enforced using the following directive in the `Doxyfile`:

```
WARN_NO_PARAMDOC = YES
```

Hint 2: The `@brief` and `@details` commands do not need to be explicitly present. The first item is implicitly the brief description and the detailed description is implicitly everything that follows a blank line after the brief description. The implicit notation is more readable in the source code for developers and produces the same documentation.

Example: The following two Doxygen blocks are equivalent.

```
//!
//! @brief This is the brief description of the next item.
//! @details This is the long and detailed description.
//! The Doxygen commands can be omitted if a blank line
//! separates the brief and detailed description.
//!

//!
//! This is the brief description of the next item.
//! This is the long and detailed description.
//! The Doxygen commands can be omitted if a blank line
//! separates the brief and detailed description.
//!
```

This behavior is allowed when the `Doxyfile` contains the following directive:

```
JAVADOC_AUTOBRIEF = YES
```

Hint 3: Individual parameters or values can be documented on one line after the element itself if the documentation contains only a brief description.

Example: The following two notations are equivalent. Note that the first one is more compact and probably more readable.

```
//!
//! Flags for the @c Hexa family of functions
//!
enum HexaFlags {
    HEX_HEXA  = 0x0001,  //!< Dump hexa values
    HEX_ASCII = 0x0002,  //!< Dump ascii values
    ....
};

//!
//! Flags for the @c Hexa family of functions
```

```
//!
enum HexaFlags {
    //!
    //! Dump hexa values
    //!
    HEX_HEXA  = 0x0001,
    //!
    //! Dump ascii values
    //!
    HEX_ASCII = 0x0002,
    ....
};
```

Appendix A: PSI/SI Signalization Reference

All signalization tables and descriptors which are supported by TSDuck are documented in the TSDuck user’s guide, appendix D "PSI/SI XML Reference Model".

A.1. PSI/SI tables

The table below summarize all available PSI/SI tables in TSDuck and the reference of the standard which specifies them.

XML name	C++ class	Defining document
`AIT`	`AIT`	ETSI TS 101 812, 10.4.6
`ATSC_EIT`	`ATSCEIT`	ATSC A/65, 6.5
`BAT`	`BAT`	ETSI EN 300 468, 5.2.2
`BIT`	`BIT`	ARIB STD-B10, Part 2, 5.2.13
`cable_emergency_alert_table`	`CableEmergencyAlertTable`	ANSI/SCTE 18, 5
`CAT`	`CAT`	ISO/IEC 13818-1, ITU-T H.222.0, 2.4.4.6
`CDT`	`CDT`	ARIB STD-B21, 12.2.2.2
`CIT`	`CIT`	ETSI TS 102 323, 12.2
`CVCT`	`CVCT`	ATSC A/65, 6.3.2
`DCCSCT`	`DCCSCT`	ATSC A/65, 6.8
`DCCT`	`DCCT`	ATSC A/65, 6.7
`discontinuity_information_table`	`DiscontinuityInformationTable`	ETSI EN 300 468, 7.1.1
`DSMCC_stream_descriptors_table`	`DSMCCStreamDescriptorsTable`	ISO/IEC 13818-6, 9.2.2 and 9.2.7
`EIT`	`EIT`	ETSI EN 300 468, 5.2.4
`ERT`	`ERT`	ARIB STD-B10, Part 3, 5.1.2
`ETT`	`ETT`	ATSC A/65, 6.6
`INT`	`INT`	ETSI EN 301 192, 8.4.3
`ITT`	`ITT`	ARIB STD-B10, Part 3, 5.1.3
`LDT`	`LDT`	ARIB STD-B10, Part 2, 5.2.15
`LIT`	`LIT`	ARIB STD-B10, Part 3, 5.1.1
`MGT`	`MGT`	ATSC A/65, 6.2
`NBIT`	`NBIT`	ARIB STD-B10, Part 2, 5.2.14
`NIT`	`NIT`	ETSI EN 300 468, 5.2.1
`PAT`	`PAT`	ISO/IEC 13818-1, ITU-T H.222.0, 2.4.4.3
`PCAT`	`PCAT`	ARIB STD-B10, Part 2, 5.2.12
`PMT`	`PMT`	ISO/IEC 13818-1, ITU-T H.222.0, 2.4.4.8
`RCT`	`RCT`	ETSI TS 102 323, 10.4.2
`RNT`	`RNT`	ETSI TS 102 323, 5.2.2

XML name	C++ class	Defining document
RRT	RRT	ATSC A/65, 6.4
RST	RST	ETSI EN 300 468, 5.2.7
SAT	SAT	ETSI EN 300 468, 5.2.11
SDT	SDT	ETSI EN 300 468, 5.2.3
SDTT	SDTT	ARIB STD-B21, 12.2.1.1
selection_information_table	SelectionInformationTable	ETSI EN 300 468, 7.1.2
splice_information_table	SpliceInformationTable	ANSI/SCTE 35, 9.2
STT	STT	ATSC A/65, 6.1
TDT	TDT	ETSI EN 300 468, 5.2.5
TOT	TOT	ETSI EN 300 468, 5.2.6
TSDT	TSDT	ISO/IEC 13818-1, ITU-T H.222.0, 2.4.4.12
TVCT	TVCT	ATSC A/65, 6.3.1
UNT	UNT	ETSI TS 102 006, 9.4.1

A.2. PSI/SI descriptors

The table below summarize all available PSI/SI desciptors in TSDuck and the reference of the standard which specifies them.

XML name	C++ class	Defining document
AAC_descriptor	AACDescriptor	ETSI EN 300 468, H.2.1
adaptation_field_data_descriptor	AdaptationFieldDataDescriptor	ETSI EN 300 468, 6.2.1
af_extensions_descriptor	AFExtensionsDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.99
ancillary_data_descriptor	AncillaryDataDescriptor	ETSI EN 300 468, 6.2.2
announcement_support_descriptor	AnnouncementSupportDescriptor	ETSI EN 300 468, 6.2.3
application_descriptor	ApplicationDescriptor	ETSI TS 102 809, 5.3.5.3
application_icons_descriptor	ApplicationIconsDescriptor	ETSI TS 102 809, 5.3.5.6.2
application_name_descriptor	ApplicationNameDescriptor	ETSI TS 101 812, 10.7.4.1
application_recording_descriptor	ApplicationRecordingDescriptor	ETSI TS 102 809, 5.3.5.4
application_signalling_descriptor	ApplicationSignallingDescriptor	ETSI TS 102 809, 5.3.5.1
application_storage_descriptor	ApplicationStorageDescriptor	ETSI TS 102 809, 5.3.10.1
application_usage_descriptor	ApplicationUsageDescriptor	ETSI TS 102 809, 5.3.5.5
area_broadcasting_information_descriptor	AreaBroadcastingInformationDescriptor	ARIB STD-B10, Part 2, 6.2.55
association_tag_descriptor	AssociationTagDescriptor	ISO/IEC 13818-6 (DSM-CC), 11.4.2
ATSC_AC3_audio_stream_descriptor	ATSCAC3AudioStreamDescriptor	ATSC A/52, A.4.3

XML name	C++ class	Defining document
ATSC_EAC3_audio_descriptor	ATSCEAC3AudioDescriptor	ATSC A/52, G.3.5
ATSC_stuffing_descriptor	ATSCStuffingDescriptor	ATSC A/65, 6.9.8
ATSC_time_shifted_service_descriptor	ATSCTimeShiftedServiceDescriptor	ATSC A/65, 6.9.6
audio_component_descriptor	AudioComponentDescriptor	ARIB STD-B10, Part 2, 6.2.26
audio_preselection_descriptor	AudioPreselectionDescriptor	ETSI EN 300 468, 6.4.1
audio_stream_descriptor	AudioStreamDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.4
australia_logical_channel_descriptor	AustraliaLogicalChannelDescriptor	Free TV Australia Operational Practice OP-41, 2.2
auxiliary_video_stream_descriptor	AuxiliaryVideoStreamDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.74 and ISO/IEC 23002-3
AV1_video_descriptor	AV1VideoDescriptor	https://aomediacodec.github.io/av1-mpeg2-ts/
AVC_timing_and_HRD_descriptor	AVCTimingAndHRDDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.66
AVC_video_descriptor	AVCVideoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.64
AVS3_video_descriptor	AVS3VideoDescriptor	T/AI 109-6
basic_local_event_descriptor	BasicLocalEventDescriptor	ARIB STD-B10, Part 3, 5.2.1
board_information_descriptor	BoardInformationDescriptor	ARIB STD-B10, Part 2, 6.2.39
bouquet_name_descriptor	BouquetNameDescriptor	ETSI EN 300 468, 6.2.4
broadcaster_name_descriptor	BroadcasterNameDescriptor	ARIB STD-B10, Part 2, 6.2.36
C2_bundle_delivery_system_descriptor	C2BundleDeliverySystemDescriptor	ETSI EN 300 468, 6.4.6.4
C2_delivery_system_descriptor	C2DeliverySystemDescriptor	ETSI EN 300 468, 6.4.6.1
CA_contract_info_descriptor	CAContractInfoDescriptor	ARIB STD-B25, Part 1, 4.7.2
CA_descriptor	CADescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.16
CA_EMM_TS_descriptor	CAEMMTSDescriptor	ARIB STD-B25, Part 1, 4.7.1
CA_identifier_descriptor	CAIdentifierDescriptor	ETSI EN 300 468, 6.2.5
CA_service_descriptor	CAServiceDescriptor	ARIB STD-B25, Part 1, 4.7.3
cable_delivery_system_descriptor	CableDeliverySystemDescriptor	ETSI EN 300 468, 6.2.13.1
caption_service_descriptor	CaptionServiceDescriptor	ATSC A/65, 6.9.2
carousel_compatible_composite_descriptor	CarouselCompatibleCompositeDescriptor	ARIB STD-B10, Part 2, 6.2.46
carousel_identifier_descriptor	CarouselIdentifierDescriptor	ISO/IEC 13818-6 (DSM-CC), 11.4.1
cell_frequency_link_descriptor	CellFrequencyLinkDescriptor	ETSI EN 300 468, 6.2.6
cell_list_descriptor	CellListDescriptor	ETSI EN 300 468, 6.2.7
CI_ancillary_data_descriptor	CIAncillaryDataDescriptor	ETSI EN 300 468, 6.4.1
component_descriptor	ComponentDescriptor	ETSI EN 300 468, 6.2.8
component_name_descriptor	ComponentNameDescriptor	ATSC A/65, 6.9.7

XML name	C++ class	Defining document
conditional_playback_descriptor	ConditionalPlaybackDescriptor	ARIB STD-B25, Part 2, 2.3.2.6.4
content_advisory_descriptor	ContentAdvisoryDescriptor	ATSC A/65, 6.9.3
content_availability_descriptor	ContentAvailabilityDescriptor	ARIB STD-B10, Part 2, 6.2.45
content_descriptor	ContentDescriptor	ETSI EN 300 468, 6.2.9
content_identifier_descriptor	ContentIdentifierDescriptor	ETSI TS 102 323, 12.1
content_labelling_descriptor	ContentLabellingDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.56
copyright_descriptor	CopyrightDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.24
country_availability_descriptor	CountryAvailabilityDescriptor	ETSI EN 300 468, 6.2.10
CP_descriptor	CPDescriptor	ETSI EN 300 468, 6.4.2
CP_identifier_descriptor	CPIdentifierDescriptor	ETSI EN 300 468, 6.4.3
cpcm_delivery_signalling_descriptor	CPCMDeliverySignallingDescriptor	ETSI TS 102 825-9, 4.1.5 and ETSI TS 102 825-4, 5.4.5
cue_identifier_descriptor	CueIdentifierDescriptor	ANSI/SCTE 35, 8.2
CUVV_video_stream_descriptor	UWAVideoStreamDescriptor	T/UWA 005-2.1
data_broadcast_descriptor	DataBroadcastDescriptor	ETSI EN 300 468, 6.2.11
data_broadcast_id_descriptor	DataBroadcastIdDescriptor	ETSI EN 300 468, 6.2.12
data_component_descriptor	DataComponentDescriptor	ARIB STD-B10, Part 2, 6.2.20
data_content_descriptor	DataContentDescriptor	ARIB STD-B10, Part 2, 6.2.28
data_stream_alignment_descriptor	DataStreamAlignmentDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.10
dcc_arriving_request_descriptor	DCCArrivingRequestDescriptor	ATSC A/65, 6.9.11
dcc_departing_request_descriptor	DCCDepartingRequestDescriptor	ATSC A/65, 6.9.10
default_authority_descriptor	DefaultAuthorityDescriptor	ETSI TS 102 323, 6.3.3 and, 5.2.2 for interpretation
deferred_association_tags_descriptor	DeferredAssociationTagsDescriptor	ISO/IEC 13818-6 (DSM-CC), 11.4.3
digital_copy_control_descriptor	DigitalCopyControlDescriptor	ARIB STD-B10, Part 2, 6.2.23
DII_location_descriptor	DIILocationDescriptor	ETSI TS 101 812, 10.8.3.3
download_content_descriptor	DownloadContentDescriptor	ARIB STD-B21, 12.2.1.1
DSNG_descriptor	DSNGDescriptor	ETSI EN 300 468, 6.2.14
dtg_guidance_descriptor	DTGGuidanceDescriptor	The D-Book 7 Part A (DTG), 8.5.3.20
dtg_HD_simulcast_logical_channel_descriptor	DTGHDSimulcastLogicalChannelDescriptor	The D-Book 7 Part A (DTG), 8.5.3.23
dtg_logical_channel_descriptor	DTGLogicalChannelDescriptor	The D-Book 7 Part A (DTG), 8.5.3.6
dtg_preferred_name_identifier_descriptor	DTGPreferredNameIdentifierDescriptor	The D-Book 7 Part A (DTG), 8.5.3.8
dtg_preferred_name_list_descriptor	DTGPreferredNameListDescriptor	The D-Book 7 Part A (DTG), 8.5.3.7

XML name	C++ class	Defining document
dtg_service_attribute_descriptor	DTGServiceAttributeDescriptor	The D-Book 7 Part A (DTG), 8.5.3.9
dtg_short_service_name_descriptor	DTGShortServiceNameDescriptor	The D-Book 7 Part A (DTG), 8.5.3.10
DTS_descriptor	DTSDescriptor	ETSI EN 300 468, G.2.1
DTS_HD_descriptor	DTSHDDescriptor	ETSI EN 300 468, G.3.1
DTS_neural_descriptor	DTSNeuralDescriptor	ETSI EN 300 468, L.1
DTS_UHD_descriptor	DVBDTSUHDDescriptor	ETSI EN 300 468, annex G
DVB_AC3_descriptor	DVBAC3Descriptor	ETSI EN 300 468, D.3
DVB_AC4_descriptor	DVBAC4Descriptor	ETSI EN 300 468, D.7
DVB_enhanced_AC3_descriptor	DVBEnhancedAC3Descriptor	ETSI EN 300 468, D.5
dvb_html_application_boundary_descriptor	DVBHTMLApplicationBoundaryDescriptor	ETSI TS 101 812, 10.10.3
dvb_html_application_descriptor	DVBHTMLApplicationDescriptor	ETSI TS 101 812, 10.10.1
dvb_html_application_location_descriptor	DVBHTMLApplicationLocationDescriptor	ETSI TS 101 812, 10.10.2
dvb_j_application_descriptor	DVBJApplicationDescriptor	ETSI TS 101 812, 10.9.1
dvb_j_application_location_descriptor	DVBJApplicationLocationDescriptor	ETSI TS 101 812, 10.9.2
DVB_stuffing_descriptor	DVBStuffingDescriptor	ETSI EN 300 468, 6.2.40
DVB_time_shifted_service_descriptor	DVBTimeShiftedServiceDescriptor	ETSI EN 300 468, 6.2.45
eacem_logical_channel_number_descriptor	EacemLogicalChannelNumberDescriptor	EACEM Technical Report Number TR-030, 9.2.11.2
eacem_preferred_name_identifier_descriptor	EacemPreferredNameIdentifierDescriptor	EACEM Technical Report Number TR-030, 9.2.11.2
eacem_preferred_name_list_descriptor	EacemPreferredNameListDescriptor	EACEM Technical Report Number TR-030, 9.2.11.2
eacem_stream_identifier_descriptor	EacemStreamIdentifierDescriptor	EACEM Technical Report Number TR-030, 9.2.11.2
EAS_audio_file_descriptor	EASAudioFileDescriptor	ANSI/SCTE 18, 5.1.3
EAS_inband_details_channel_descriptor	EASInbandDetailsChannelDescriptor	ANSI/SCTE 18, 5.1.1
EAS_inband_exception_channels_descriptor	EASInbandExceptionChannelsDescriptor	ANSI/SCTE 18, 5.1.2
EAS_metadata_descriptor	EASMetadataDescriptor	ANSI/SCTE 164, 5.0
ECM_repetition_rate_descriptor	ECMRepetitionRateDescriptor	ETSI EN 301 192, 9.7
emergency_information_descriptor	EmergencyInformationDescriptor	ARIB STD-B10, Part 2, 6.2.24
EVC_timing_and_HRD_descriptor	EVCTimingAndHRDDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.135
EVC_video_descriptor	EVCVideoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.133
event_group_descriptor	EventGroupDescriptor	ARIB STD-B10, Part 2, 6.2.34

XML name	C++ class	Defining document
extended_broadcaster_descriptor	ExtendedBroadcasterDescriptor	ARIB STD-B10, Part 2, 6.2.43
extended_channel_name_descriptor	ExtendedChannelNameDescriptor	ATSC A/65, 6.9.4
extended_event_descriptor	ExtendedEventDescriptor	ETSI EN 300 468, 6.2.15
external_application_authorization_descriptor	ExternalApplicationAuthorizationDescriptor	ETSI TS 102 809, 5.3.5.7
external_ES_ID_descriptor	ExternalESIdDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.46
FMC_descriptor	FMCDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.44
frequency_list_descriptor	FrequencyListDescriptor	ETSI EN 300 468, 6.2.17
FTA_content_management_descriptor	FTAContentManagementDescriptor	ETSI EN 300 468, 6.2.18
genre_descriptor	GenreDescriptor	ATSC A/65, 6.9.13
graphics_constraints_descriptor	GraphicsConstraintsDescriptor	ETSI TS 102 809, 5.3.5.8
green_extension_descriptor	GreenExtensionDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.104
HEVC_hierarchy_extension_descriptor	HEVCHierarchyExtensionDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.102
HEVC_operation_point_descriptor	HEVCOperationPointDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.100
HEVC_subregion_descriptor	HEVCSubregionDescriptor	ISO/IEC 13818-1, 2.6.138
HEVC_tile_substream_descriptor	HEVCTileSubstreamDescriptor	ISO/IEC 13818-1 clasue 2.6.122
HEVC_timing_and_HRD_descriptor	HEVCTimingAndHRDDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.97
HEVC_video_descriptor	HEVCVideoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.95
hierarchical_transmission_descriptor	HierarchicalTransmissionDescriptor	ARIB STD-B10, Part 2, 6.2.22
hierarchy_descriptor	HierarchyDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.6
hybrid_information_descriptor	HybridInformationDescriptor	ARIB STD-B10, Part 2, 6.2.58
IBP_descriptor	IBPDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.34
image_icon_descriptor	ImageIconDescriptor	ETSI EN 300 468, 6.4.7
ip_signalling_descriptor	IPSignallingDescriptor	ETSI TS 101 812, 10.8.2
IPMAC_generic_stream_location_descriptor	IPMACGenericStreamLocationDescriptor	ETSI EN 301 192, 8.4.5.15
IPMAC_platform_name_descriptor	IPMACPlatformNameDescriptor	ETSI EN 301 192, 8.4.5.2
IPMAC_platform_provider_name_descriptor	IPMACPlatformProviderNameDescriptor	ETSI EN 301 192, 8.4.5.3
IPMAC_stream_location_descriptor	IPMACStreamLocationDescriptor	ETSI EN 301 192, 8.4.5.14
ISDB_access_control_descriptor	ISDBAccessControlDescriptor	ARIB STD-B10, Part 2, 6.2.54
ISDB_component_group_descriptor	ISDBComponentGroupDescriptor	ARIB STD-B10, Part 2, 6.2.37
ISDB_connected_transmission_descriptor	ISDBConnectedTransmissionDescriptor	ARIB STD-B10, Part 2, 6.2.41
ISDB_hyperlink_descriptor	ISDBHyperlinkDescriptor	ARIB STD-B10, Part 2, 6.2.29

XML name	C++ class	Defining document
ISDB_LDT_linkage_descriptor	ISDBLDTLinkageDescriptor	ARIB STD-B10, Part 2, 6.2.40
ISDB_network_identifier_descriptor	ISDBNetworkIdentifierDescriptor	ARIB STD-B21, Part 2, 9.1.8.3
ISDB_target_region_descriptor	ISDBTargetRegionDescriptor	ARIB STD-B10, Part 2, 6.2.27
ISDB_terrestrial_delivery_system_descriptor	ISDBTerrestrialDeliverySystemDescriptor	ARIB STD-B10, Part 2, 6.2.31
ISO_639_language_descriptor	ISO639LanguageDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.18
ISP_access_mode_descriptor	ISPAccessModeDescriptor	ETSI EN 301 192, 8.4.5.16
J2K_video_descriptor	J2KVideoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.80
JPEG_XS_video_descriptor	JPEGXSVideoDescriptor	ISO/IEC 13818-1 2.6.127, ITU-T H.222.0
LCEVC_linkage_descriptor	LCEVCLinkageDescriptor	ISO/IEC 13818-1 (Amd.1) 2.6.137, ITU-T H.222.0
LCEVC_video_descriptor	LCEVCVideoDescriptor	ISO/IEC 13818-1 (Amd.1) 2.6.137, ITU-T H.222.0
linkage_descriptor	LinkageDescriptor	ETSI EN 300 468, 6.2.19
linkage_descriptor	SSULinkageDescriptor	ETSI EN 300 468, 6.2.19
local_time_offset_descriptor	LocalTimeOffsetDescriptor	ETSI EN 300 468, 6.2.20
logo_transmission_descriptor	LogoTransmissionDescriptor	ARIB STD-B10, Part 2, 6.2.44
m4mux_timing_descriptor	M4MuxTimingDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.54
M4MuxBufferSize_descriptor	M4MuxBufferSizeDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.50
maximum_bitrate_descriptor	MaximumBitrateDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.26
Media_service_kind_descriptor	MediaServiceKindDescriptor	ISO/IEC 13818-1 (Amd.1) 2.6.141
message_descriptor	MessageDescriptor	ETSI EN 300 468, 6.4.7
metadata_descriptor	MetadataDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.60
metadata_pointer_descriptor	MetadataPointerDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.58
metadata_STD_descriptor	MetadataSTDDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.62
mosaic_descriptor	MosaicDescriptor	ETSI EN 300 468, 6.2.21
MPEG2_AAC_audio_descriptor	MPEG2AACAudioDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.68
MPEG2_stereoscopic_video_format_descriptor	MPEG2StereoscopicVideoFormatDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.84
MPEG4_audio_descriptor	MPEG4AudioDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.38
MPEG4_text_descriptor	MPEG4TextDescriptor	ITU-T H.222.0, 2.6.70 and ISO/IEC 14496-17
MPEG4_video_descriptor	MPEG4VideoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.36
MPEGH_3D_audio_descriptor	MPEGH3DAudioDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.106
MPEGH_3D_audio_multi_stream_descriptor	MPEGH3DAudioMultiStreamDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.114
MPEGH_3D_audio_scene_descriptor	MPEGH3DAudioSceneDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.110
MPEGH_3D_audio_text_label_descriptor	MPEGH3DAudioTextLabelDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.112

XML name	C++ class	Defining document
multilingual_bouquet_name_descriptor	MultilingualBouquetNameDescriptor	ETSI EN 300 468, 6.2.22
multilingual_component_descriptor	MultilingualComponentDescriptor	ETSI EN 300 468, 6.2.23
multilingual_network_name_descriptor	MultilingualNetworkNameDescriptor	ETSI EN 300 468, 6.2.24
multilingual_service_name_descriptor	MultilingualServiceNameDescriptor	ETSI EN 300 468, 6.2.25
multiplex_buffer_descriptor	MultiplexBufferDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.52
multiplex_buffer_utilization_descriptor	MultiplexBufferUtilizationDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.22
MuxCode_descriptor	MuxcodeDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.48 and ISO/IEC 14496-1, 7.4.2.5
MVC_extension_descriptor	MVCExtensionDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.78
MVC_operation_point_descriptor	MVCOperationPointDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.82
network_change_notify_descriptor	NetworkChangeNotifyDescriptor	ETSI EN 300 468, 6.4.9
network_download_content_descriptor	NetworkDownloadContentDescriptor	ARIB STD-B21, 12.2.1.1
network_name_descriptor	NetworkNameDescriptor	ETSI EN 300 468, 6.2.27
node_relation_descriptor	NodeRelationDescriptor	ARIB STD-B10, Part 3, 5.2.3
nordig_logical_channel_descriptor_v1	NorDigLogicalChannelDescriptorV1	NorDig Unified Requirements ver. 3.1.1, 12.2.9.2
nordig_logical_channel_descriptor_v2	NorDigLogicalChannelDescriptorV2	NorDig Unified Requirements ver. 3.1.1, 12.2.9.3
NPT_endpoint_descriptor	NPTEndpointDescriptor	ISO/IEC 13818-6, 8.1.5
NPT_reference_descriptor	NPTReferenceDescriptor	ISO/IEC 13818-6, 8.1.1
NVOD_reference_descriptor	NVODReferenceDescriptor	ETSI EN 300 468, 6.2.26
parental_rating_descriptor	ParentalRatingDescriptor	ETSI EN 300 468, 6.2.28
partial_reception_descriptor	PartialReceptionDescriptor	ARIB STD-B10, Part 2, 6.2.32
partial_transport_stream_descriptor	PartialTransportStreamDescriptor	ETSI EN 300 468, 7.2.1
partialTS_time_descriptor	PartialTSTimeDescriptor	ARIB STD-B21, 9.1.8.3 (3)
PDC_descriptor	PDCDescriptor	ETSI EN 300 468, 6.2.30
prefetch_descriptor	PrefetchDescriptor	ETSI TS 101 812, 10.8.3.2
private_data_indicator_descriptor	PrivateDataIndicatorDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.28
private_data_specifier_descriptor	PrivateDataSpecifierDescriptor	ETSI EN 300 468, 6.2.31
protection_message_descriptor	ProtectionMessageDescriptor	ETSI TS 102 809, 9.3.3
quality_extension_descriptor	QualityExtensionDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.119 and ISO/ISC 23001-10

XML name	C++ class	Defining document
RAR_over_DVB_stream_descriptor	RARoverDVBstreamDescriptor	ETSI TS 102 323, 5.3.5
RAR_over_IP_descriptor	RARoverIPDescriptor	ETSI TS 102 323, 5.3.6
redistribution_control_descriptor	RedistributionControlDescriptor	ATSC A/65, 6.9.12
reference_descriptor	ReferenceDescriptor	ARIB STD-B10, Part 3, 5.2.2
registration_descriptor	RegistrationDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.8
related_content_descriptor	RelatedContentDescriptor	ETSI TS 102 323, 10.3
RNT_scan_descriptor	RNTScanDescriptor	ETSI TS 102 323, 5.3.7
S2_satellite_delivery_system_descriptor	S2SatelliteDeliverySystemDescriptor	ETSI EN 300 468, 6.2.13.3
S2X_satellite_delivery_system_descriptor	S2XSatelliteDeliverySystemDescriptor	ETSI EN 300 468, 6.4.6.5
S2Xv2_satellite_delivery_system_descriptor	S2Xv2SatelliteDeliverySystemDescriptor	ETSI EN 300 468, 6.4.6.5.3
satellite_delivery_system_descriptor	SatelliteDeliverySystemDescriptor	ETSI EN 300 468, 6.2.13.2
scheduling_descriptor	SchedulingDescriptor	ETSI TS 102 006, 9.5.2.9
scrambling_descriptor	ScramblingDescriptor	ETSI EN 300 468, 6.2.32
series_descriptor	SeriesDescriptor	ARIB STD-B10, Part 2, 6.2.33
service_availability_descriptor	ServiceAvailabilityDescriptor	ETSI EN 300 468, 6.2.34
service_descriptor	ServiceDescriptor	ETSI EN 300 468, 6.2.33
service_group_descriptor	ServiceGroupDescriptor	ARIB STD-B10, Part 2, 6.2.49
service_identifier_descriptor	ServiceIdentifierDescriptor	ETSI TS 102 809, 6.2.1
service_list_descriptor	ServiceListDescriptor	ETSI EN 300 468, 6.2.35
service_location_descriptor	ServiceLocationDescriptor	ATSC A/65, 6.9.5
service_move_descriptor	ServiceMoveDescriptor	ETSI EN 300 468, 6.2.34
service_prominence_descriptor	DVBServiceProminenceDescriptor	ETSI EN 300 468, 6.4.18
service_relocated_descriptor	ServiceRelocatedDescriptor	ETSI EN 300 468, 6.4.9
SH_delivery_system_descriptor	SHDeliverySystemDescriptor	ETSI EN 300 468, 6.4.6.2
short_event_descriptor	ShortEventDescriptor	ETSI EN 300 468, 6.2.37
short_node_information_descriptor	ShortNodeInformationDescriptor	ARIB STD-B10, Part 3, 5.2.4
short_smoothing_buffer_descriptor	ShortSmoothingBufferDescriptor	ETSI EN 300 468, 6.2.38
SI_parameter_descriptor	SIParameterDescriptor	ARIB STD-B10, Part 2, 6.2.35
SI_prime_TS_descriptor	SIPrimeTSDescriptor	ARIB STD-B10, Part 2, 6.2.38
simple_application_boundary_descriptor	SimpleApplicationBoundaryDescriptor	ETSI TS 102 809, 5.3.8
simple_application_location_descriptor	SimpleApplicationLocationDescriptor	ETSI TS 102 809, 5.3.7

XML name	C++ class	Defining document
SL_descriptor	SLDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.42
smoothing_buffer_descriptor	SmoothingBufferDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.30
splice_avail_descriptor	SpliceAvailDescriptor	ANSI/SCTE 35, 10.3.1
splice_DTMF_descriptor	SpliceDTMFDescriptor	ANSI/SCTE 35, 10.3.2
splice_segmentation_descriptor	SpliceSegmentationDescriptor	ANSI/SCTE 35, 10.3.3
splice_time_descriptor	SpliceTimeDescriptor	ANSI/SCTE 35, 10.3.4
SSU_enhanced_message_descriptor	SSUEnhancedMessageDescriptor	ETSI TS 102 006, 9.5.2.14
SSU_event_name_descriptor	SSUEventNameDescriptor	ETSI TS 102 006, 9.5.2.11
SSU_location_descriptor	SSULocationDescriptor	ETSI TS 102 006, 9.5.2.7
SSU_message_descriptor	SSUMessageDescriptor	ETSI TS 102 006, 9.5.2.12
SSU_subgroup_association_descriptor	SSUSubgroupAssociationDescriptor	ETSI TS 102 006, 9.5.2.8
SSU_uri_descriptor	SSUURIDescriptor	ETSI TS 102 006, 9.5.2.15
STC_reference_descriptor	STCReferenceDescriptor	ARIB STD-B10, Part 3, 5.2.5
STD_descriptor	STDDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.32
stereoscopic_program_info_descriptor	StereoscopicProgramInfoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.86
stereoscopic_video_info_descriptor	StereoscopicVideoInfoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.88
stream_event_descriptor	StreamEventDescriptor	ISO/IEC 13818-6, 8.3
stream_identifier_descriptor	StreamIdentifierDescriptor	ETSI EN 300 468, 6.2.39
stream_mode_descriptor	StreamModeDescriptor	ISO/IEC 13818-6, 8.2
subtitling_descriptor	SubtitlingDescriptor	ETSI EN 300 468, 6.2.41
supplementary_audio_descriptor	SupplementaryAudioDescriptor	ETSI EN 300 468, 6.4.11
SVC_extension_descriptor	SVCExtensionDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.76
system_clock_descriptor	SystemClockDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.20
system_management_descriptor	SystemManagementDescriptor	ARIB STD-B10, Part 2, 6.2.21
T2_delivery_system_descriptor	T2DeliverySystemDescriptor	ETSI EN 300 468, 6.4.6.3
T2MI_descriptor	T2MIDescriptor	ETSI EN 300 468, 6.4.14
target_background_grid_descriptor	TargetBackgroundGridDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.12
target_IP_address_descriptor	TargetIPAddressDescriptor	ETSI EN 301 192, 8.4.5.8
target_IP_slash_descriptor	TargetIPSlashDescriptor	ETSI EN 301 192, 8.4.5.9
target_IP_source_slash_descriptor	TargetIPSourceSlashDescriptor	ETSI EN 301 192, 8.4.5.10
target_IPv6_address_descriptor	TargetIPv6AddressDescriptor	ETSI EN 301 192, 8.4.5.11
target_IPv6_slash_descriptor	TargetIPv6SlashDescriptor	ETSI EN 301 192, 8.4.5.12
target_IPv6_source_slash_descriptor	TargetIPv6SourceSlashDescriptor	ETSI EN 301 192, 8.4.5.13

XML name	C++ class	Defining document
target_MAC_address_descriptor	TargetMACAddressDescriptor	ETSI EN 301 192, 8.4.5.6
target_MAC_address_range_descriptor	TargetMACAddressRangeDescriptor	ETSI EN 301 192, 8.4.5.7
target_region_descriptor	TargetRegionDescriptor	ETSI EN 300 468, 6.4.12
target_region_name_descriptor	TargetRegionNameDescriptor	ETSI EN 300 468, 6.4.13
target_serial_number_descriptor	TargetSerialNumberDescriptor	ETSI EN 301 192, 8.4.5.4
target_smartcard_descriptor	TargetSmartcardDescriptor	ETSI EN 301 192, 8.4.5.5
telephone_descriptor	TelephoneDescriptor	ETSI EN 300 468, 6.2.42
teletext_descriptor	TeletextDescriptor	ETSI EN 300 468, 6.2.43
terrestrial_delivery_system_descriptor	TerrestrialDeliverySystemDescriptor	ETSI EN 300 468, 6.2.13.4
time_shifted_event_descriptor	TimeShiftedEventDescriptor	ETSI EN 300 468, 6.2.44
time_slice_fec_identifier_descriptor	TimeSliceFECIdentifierDescriptor	ETSI EN 301 192, 9.5
transport_profile_descriptor	TransportProfileDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.93
transport_protocol_descriptor	TransportProtocolDescriptor	ETSI TS 101 812, 10.8.1
transport_stream_descriptor	TransportStreamDescriptor	ETSI EN 300 468, 6.2.46
TS_information_descriptor	TSInformationDescriptor	ARIB STD-B10, Part 2, 6.2.42
TTML_subtitling_descriptor	TTMLSubtitlingDescriptor	ETSI EN 303 560, 5.2.1.1
TVA_id_descriptor	TVAIdDescriptor	ETSI TS 102 323, 11.2.4
update_descriptor	UpdateDescriptor	ETSI TS 102 006, 9.5.2.6
URI_linkage_descriptor	URILinkageDescriptor	ETSI TS 101 162
VBI_data_descriptor	VBIDataDescriptor	ETSI EN 300 468, 6.2.47
VBI_teletext_descriptor	VBITeletextDescriptor	ETSI EN 300 468, 6.2.48
video_decode_control_descriptor	VideoDecodeControlDescriptor	ARIB STD-B10, Part 2, 6.2.30
video_depth_range_descriptor	VideoDepthRangeDescriptor	ETSI EN 300 468, 6.4.16
video_stream_descriptor	VideoStreamDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.2
video_window_descriptor	VideoWindowDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.14
virtual_segmentation_descriptor	VirtualSegmentationDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.120
vvc_subpictures_descriptor	VVCSubpicturesDescriptor	ETSI EN 300 468, 6.4.17
VVC_timing_and_HRD_descriptor	VVCTimingAndHRDDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.131
VVC_video_descriptor	VVCVideoDescriptor	ISO/IEC 13818-1, ITU-T H.222.0, 2.6.129

Appendix B: License

TSDuck is released under the terms of the license which is commonly referred to as "BSD 2-Clause License" or "Simplified BSD License" or "FreeBSD License". See http://opensource.org/licenses/BSD-2-Clause.

Appendix C: References

C.1. Acronyms and abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
BOM	Byte Order Mark (a feature of UTF-16 and UTF-32)
DRY	Don't Repeat Yourself (a good practice)
FOSS	Free and Open Source Software
GCC	GNU Compiler Collection
GPL	GNU General Public License
IDE	Integrated Development Environment (e.g. Eclipse, MS Visual Studio)
LGPL	Lesser GPL
MSC	Microsoft C/C++ Compiler
MSVC	Microsoft Visual C/C++
NIH	Not Invented Here (a bad practice)
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
RTFM	The Most Important Acronym For Developers (yes, it is)
RTTI	Run-Time Type Information (C++)
SFINAE	Substitution Failure Is Not An Error (C++)
STL	Standard Template Library (C++)
TDD	Test-Driven Development
UTF	Unicode Transformation Format (UTF-8, UTF-16, UTF-32, etc.)
XP	eXtreme Programming

Bibliography

- [CERT-C] "SEI CERT C Coding Standard, Rules for Developing Safe, Reliable and Secure Systems", Software Engineering Institute, Carnegie Mellon University, 2016 Edition.
- [CERT-CPP] "SEI CERT C Coding Standard, Rules for Developing Safe, Reliable and Secure Systems in C", Aaron Ballman, Software Engineering Institute, Carnegie Mellon University, 2016 Edition.
- [CPPREF] http://en.cppreference.com/, Online reference of the C++ language and standard library.
- [GAMMA] "Design Patterns, Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995.
- [ISO-14882] ISO/IEC 14882, "Programming Languages - C", 1998 (the C98 standard).
- [ISO-14882-11] ISO/IEC 14882:2011, "Programming Languages - C", 2011 (the C11 standard).
- [ISO-14882-14] ISO/IEC 14882:2014, "Programming Languages - C", 2014 (the C14 standard).
- [ISO-14882-17] ISO/IEC 14882:2017, "Programming Languages - C", 2017 (the C17 standard).

- [JOSUTTIS] "The C++ Standard Library, A Tutorial and Reference", Nicolai M. Josuttis, Addison-Wesley, 1999.
- [MEYERS-EFF] "Effective C++, Third Edition, 55 Specific Ways to Improve Your Programs and Designs", Scott Meyers, Addison-Wesley, 2005.
- [MEYERS-MORE] "More Effective C++, 35 New Ways to Improve Your Programs and Designs", Scott Meyers, Addison-Wesley, 2008.
- [MEYERS-STL] "Effective STL, 50 Specific Ways to Improve Your Use of the Standard Template Library", Scott Meyers, Addison-Wesley, 2001.
- [STROUSTRUP] "The C++ Programming Language, Special Edition", Bjarne Stroustrup, Addison-Wesley, 2000.
- [TSDuck] TSDuck Web site, https://tsduck.io/
- [TSDuck-Issues] TSDuck issues tracker and discussion forum, https://github.com/tsduck/tsduck/issues
- [TSDuck-Source] TSDuck source code repository, https://github.com/tsduck/tsduck