SMPTE VC-2 Conformance Software

Release v1.0.0

BBC

Apr 01, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

This is the manual for the VC-2 conformance testing software. This software is used to test implementations of the VC-2 video codec.

Specifically, this software tests conformance with the following SMPTE standards and recommended practices:

- SMPTE ST 2042-1:2017[1] (VC-2)
- SMPTE ST 2042-2:2017[2] (VC-2 Level Definitions)
- SMPTE RP 2047-1:2009[3] (VC-2 Mezzanine Compression of 1080P High Definition Video Sources)
- SMPTE RP 2047-3:2016[4] (VC-2 Level 65 Compression of High Definition Video Sources for Use with a Standard Definition Infrastructure)
- SMPTE RP 2047-5:2017[5] (VC-2 Level 66 Compression of Ultra High Definition Video Sources for use with a High Definition Infrastructure)

Note: Throughout this software and documentation, perenthesised references of the form '(1.2.3)' refer to section numbers within the SMPTE ST 2042-1:2017 specification unless otherwise indicated.

This manual is split into two parts.

The first (shorter) part, *User's manual* (page 5), is aimed at codec testers who wish to test a VC-2 codec implementation. This part includes software installation and codec testing procedures along with reference documentation for the tools provided.

The second (longer) part, *Maintainer's manual* (page 57), is aimed at developers tasked with maintaining this software; codec testers can disregard this part. This section includes a general overview of the conformance software internals followed by detailed reference documentation on its various components.

Finally, you can find the source code for `vc2_conformance` (page 59) on GitHub[6].

Note: This documentation is also available to browse online in HTML format[7].

[1] https://ieeexplore.ieee.org/document/7967896
[2] https://ieeexplore.ieee.org/document/8187792
[3] https://ieeexplore.ieee.org/document/7290342
[4] https://ieeexplore.ieee.org/document/7565453
[5] https://ieeexplore.ieee.org/document/8019813
[6] https://github.com/bbc/vc2_conformance/
[7] https://bbc.github.io/vc2_conformance/

Part I

User’s manual

CHAPTER

TWO

USER'S GUIDE (FOR CODEC TESTERS)

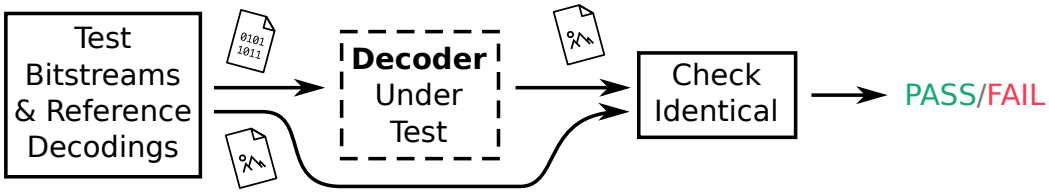
In the following sections we'll walk through the process of installing, configuring and using the VC-2 conformance testing software.

2.1 Introduction

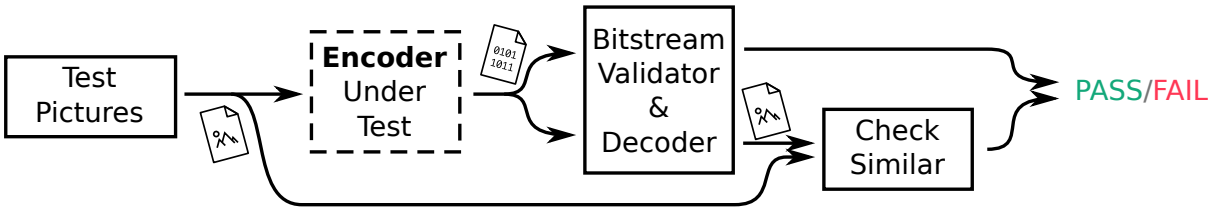
In this guide we'll walk through the steps involved in testing a VC-2 implementation for conformance to SMPTE ST 2042-family of specifications.

2.1.1 Testing procedure overview

The conformance testing processes for decoders and encoders is outlined below.



Decoders are tested using a series of test bitstreams (generated using *vc2-test-case-generator* (page 43)). The pictures produced by the decoder are then compared against reference decodings. If the decoded pictures are bit-for-bit identical (as checked by *vc2-picture-compare* (page 46)), and the decoder did not crash, the decoder passes the test.



Encoders are tested using a series of test pictures (generated using *vc2-test-case-generator* (page 43)). The encoded bitstreams are then fed to a bitstream validator (*vc2-bitstream-validator* (page 45)) which simultaneously validates the bitstream against the specification and decodes the encoded pictures. If the bitstream is free from technical errors, the decoded pictures are then compared with the input pictures (both visually and using *vc2-picture-compare* (page 46)). If the decoded pictures are sufficiently similar to the inputs, the encoder passes the test.

2.1.2 Guide outline

In *Conformance Software Installation* (page 6) we will walk through the process of installing the VC-2 conformance software on your computer.

In *Video file format* (page 8) the simple planar raw video file format used by the conformance software is introduced. You are responsible for converting between this format and the format natively accepted by the codec under test.

In *Generating test cases* (page 15) we will use the *vc2-test-case-generator* (page 43) tool to generate a set of test pictures and bitstreams. Because VC-2 supports such a wide variety of video formats and coding behaviours, test cases are generated on demand to suit the particular features of a codec under test.

In *VC-2 decoder conformance testing procedure* (page 22) and *VC-2 encoder conformance testing procedure* (page 31) we describe how the test pictures and bitstreams should be processed by the codec under test. We also describe the procedures for verifying that the codecs behaved as expected. Additionally, in *Testing additional bitstreams' conformance* (page 37) we explain how bitstreams produced outside of the conformance testing procedures can also be tested for conformance.

In *Codec debugging suggestions* (page 39) we provide some advice on how to approach the problem of debugging failing tests.

Finally, in *Conformance test limtations* (page 41) some of the limitations of the conformance test cases and procedures are enumerated.

So, lets move on to *Conformance Software Installation* (page 6)...

2.2 Conformance Software Installation

These steps will help you install the VC-2 conformance software, along with its dependencies. The VC-2 conformance software is cross platform and should run on any system with a Python interpreter, however these instructions will only cover the process under Linux.

2.2.1 Python interpreter

The VC-2 conformance software is compatible with both Python 2.7 and Python 3.6 and later. If in doubt, you should prefer Python 3.x. You should also make sure that the `pip` Python package manager is also installed.

Under Debian-like Linux distributions (e.g. Ubuntu), Python and `pip` can be installed using:

```
# apt install python3 python3-pip
```

Note: We strongly recommend running the VC-2 conformance software under the standard Python interpreter ('CPython') as opposed to other Python implementations[8] (such as PyPy[9]). These alternative implementations are often less stable and we have not tested this software running under them. If you're not sure which Python interpreter you've got on your system you'll almost certainly have the (correct) standard Python interpreter so there is no need to take any action.

[8] https://www.python.org/download/alternatives/
[9] https://www.pypy.org/

2.2.2 Installation

You can install the VC-2 conformance software using any of the methods below.

Via `pip` (recommended)

Before installing the VC-2 conformance software you must download and install the data files it requires as follows:

```
$ python -m pip install --user "vc2_conformance_data @ https://github.com/bbc/vc2_
↪conformance_data/releases/download/v1.0.0/vc2_conformance_data.tar.gz"
```

Note: We are currently working[10] to make the download and installation of these data files automatic during the conformance software installation. In the future you will be able to skip the step above.

Next, the VC-2 conformance software, along with all its dependencies, can be installed as follows:

```
$ python -m pip install --user vc2_conformance
```

The `--user` argument can be omitted for a system-wide installation (strongly *not* recommended) or when installing in a Python virtual environment[11].

Note: If installation fails on Debian and Ubuntu systems, you might need to execute the following line prior to the above:

```
$ export PIP_IGNORE_INSTALLED=0
```

From `.tar.gz` packages

If you have received a copy of the VC-2 conformance software as a collection of `.tar.gz` packages, these can be installed as follows (replacing `X.Y.Z` with the version numbers from the files supplied):

```
$ python -m pip install --user vc2_data_tables-X.Y.Z.tar.gz
$ python -m pip install --user vc2_bit_widths-X.Y.Z.tar.gz
$ python -m pip install --user vc2_conformance_data-X.Y.Z.tar.gz
$ python -m pip install --user vc2_conformance-X.Y.Z.tar.gz
```

Note: The installation order must be as shown above.

The `--user` argument can be omitted for a system-wide installation (strongly *not* recommended) or when installing in a Python virtual environment[12].

Note: If installation fails on Debian and Ubuntu systems, you might need to execute the following line prior to the above:

```
$ export PIP_IGNORE_INSTALLED=0
```

[10] https://github.com/bbc/vc2_conformance/issues/7
[11] https://docs.python.org/3/tutorial/venv.html
[12] https://docs.python.org/3/tutorial/venv.html

From source (advanced)

The latest VC-2 conformance software can be installed from the source as follows.

First, you must checkout (or download a snapshot of) the following repositories:

- https://github.com/bbc/vc2_data_tables
- https://github.com/bbc/vc2_bit_widths
- https://github.com/bbc/vc2_conformance_data
- https://github.com/bbc/vc2_conformance

Next, each package should be installed (in the order shown above) using the following steps:

```
$ cd path/to/repo/
$ pip install --user .
```

The `--user` argument can be omitted for a system-wide installation (not recommended).

All other dependencies will be downloaded automatically during the installation of these packages.

2.2.3 Verifying installation

To verify installation was successful, try running:

```
$ vc2-test-case-generator --version
```

This command should print a version number and then exit immediately. If the command cannot be found, check your `PATH` includes the directory the conformance software was installed into.

Tip: Under Linux, Python usually installs programmes into `$HOME/.local/bin`. This can be temporarily added to your path using:

```
$ export PATH="$HOME/.local/bin:$PATH"
```

Next, lets move on to *Video file format* (page 8)...

2.3 Video file format

The VC-2 conformance software uses a simple video and metadata format to represent uncompressed pictures consisting of a raw video file and associated JSON metadata file. This format is described below and it is left to the codec implementer to perform any translation necessary between this format and the format expected by the codec under test.

Below we'll describe the file format before introducing the `vc2-picture-explain` utility which can aid in understanding and displaying videos in this format.

2.3.1 Format description

Each picture in a sequence is stored as a pair of files: a file containing only raw sample values (`.raw`) and a metadata file containing a JSON description of the video format and picture number (`.json`). Both files are necessary in order to correctly interpret the picture data.

File names

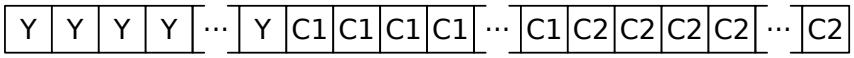
The following naming convention is used for sequences of pictures: `<name>_<number>.raw` and `<name>_<number>.json` where `<name>` is the same for every picture in the sequence and `<number>` starts at `0` and increments contiguously. For example, a four picture sequence might use the following file names:

- `my_sequence_0.raw`
- `my_sequence_0.json`
- `my_sequence_1.raw`
- `my_sequence_1.json`
- `my_sequence_2.raw`
- `my_sequence_2.json`
- `my_sequence_3.raw`
- `my_sequence_3.json`

Note: The `<number>` part of the filename can optionally include leading zeros.

`.raw` (picture data) file format

The raw picture file (`*.raw`) contains sample values in 'planar' form where the values for each picture component are stored separately as illustrated below:



Sample values are stored in raster scan order, starting with the top-left sample and working left-to-right then top-to-bottom.

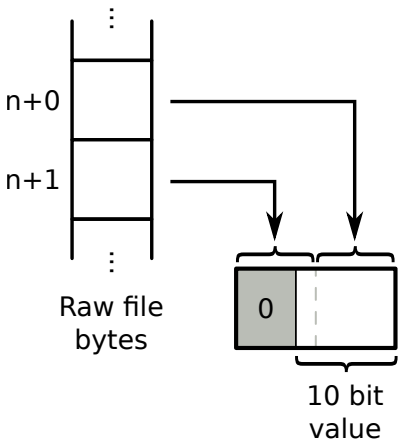
Samples are stored as unsigned integers in the smallest power-of-two number of bytes in which they fit. For example:

- 1 to 8 bit formats use one byte per sample
- 9 to 16 bit formats use two bytes per sample
- 17 to 32 bit formats use four bytes per sample
- And so on...

Note: The luma (Y) and color difference (C1, C2) components might have different bit depths, and therefore use a different number of bytes per sample in the raw format.

Sample values are stored in little-endian byte order, least-significant-bit aligned and zero padded.

For example, a 10 bit sample is stored as two bytes. The first byte contains the least significant eight bits of the sample value. The two least significant bits of the second byte contain the two most significant bits of the sample value. The six most significant bits of the second byte are all zero. This is illustrated below:



`.json` (metadata) file format

Each raw picture file is accompanied by a metadata file with the same name but a `.json` extension. This file is a UTF-8 encoded JSON (ECMA-404)[13] with the following structure:

```
{
    "picture_number": <string>,
    "picture_coding_mode": <int>,
    "video_parameters": <video-parameters>
}
```

The `picture_number` field gives the picture number (see section (12.2) of the VC-2 standard) as a string. This might not be the same as the number used in the file name.

Note: A string is used for the `picture_number` field because JSON implementations handle large integers inconsistently.

The `picture_coding_mode` indicates whether each picture corresponds to a frame (0) or a field (1) in the video (see section (11.5)).

Note: Note that the scan format flag defined in the `source_sampling` field of the `video_parameters` (11.4.5) does *not* control whether pictures correspond to frames or fields.

The `video_parameters` field contains an object of the following form:

```
<video-parameters> = {
    "frame_width": <int>,
    "frame_height": <int>,
    "color_diff_format_index": <int>,
    "source_sampling": <int>,
    "top_field_first": <bool>,
    "frame_rate_numer": <int>,
    "frame_rate_denom": <int>,
    "pixel_aspect_ratio_numer": <int>,
    "pixel_aspect_ratio_denom": <int>,
    "clean_width": <int>,
    "clean_height": <int>,
    "left_offset": <int>,
    "top_offset": <int>,
    "luma_offset": <int>,
    "luma_excursion": <int>,
```

(continues on next page)

[13] https://www.json.org/

(continued from previous page)

```
    "color_diff_offset": <int>,
    "color_diff_excursion": <int>,
    "color_primaries_index": <int>,
    "color_matrix_index": <int>,
    "transfer_function_index": <int>
}
```

This is the same structure described in section (11.4) of the VC-2 standard and populated by the `source_parameters` pseudocode function.

Computing picture component dimensions and depths

The dimensions of the Y, C1 and C2 components of each picture in the raw file can be computed from the metadata as specified in the `picture_dimensions` pseudocode function from section (11.6.2) of the VC-2 standard:

```
picture_dimensions(video_parameters, picture_coding_mode):
    state[luma_width] = video_parameters[frame_width]
    state[luma_height] = video_parameters[frame_height]
    state[color_diff_width] = state[luma_width]
    state[color_diff_height] = state[luma_height]
    color_diff_format_index = video_parameters[color_diff_format_index]
    if (color_diff_format_index == 1):
        state[color_diff_width] //= 2
    if (color_diff_format_index == 2):
        state[color_diff_width] //= 2
        state[color_diff_height] //= 2
    if (picture_coding_mode == 1):
        state[luma_height] //= 2
        state[color_diff_height] //= 2
```

The sample value bit depth is computed by the `video_depth` pseudocode function given in section (11.6.3) of the VC-2 standard:

```
video_depth(video_parameters):
    state[luma_depth] = intlog2(video_parameters[luma_excursion]+1)
    state[color_diff_depth] = intlog2(video_parameters[color_diff_excursion]+1)
```

2.3.2 `vc2-picture-explain` utility

The VC-2 conformance software provides the *vc2-picture-explain* (page 49) command line utility which produces informative explanations of the raw format used by a particular video, along with commands to display the video directly, if possible.

For example, given a typical raw 1080i60, 10-bit 4:2:2 video file as input:

```
$ vc2-picture-explain picture_0.raw
Normative description
=====================

Picture coding mode: pictures_are_fields (1)

Video parameters:

* frame_width: 1920
* frame_height: 1080
* color_diff_format_index: color_4_2_2 (1)
* source_sampling: interlaced (1)
* top_field_first: True
```

(continues on next page)

(continued from previous page)

```
* frame_rate_numer: 30000
* frame_rate_denom: 1001
* pixel_aspect_ratio_numer: 1
* pixel_aspect_ratio_denom: 1
* clean_width: 1920
* clean_height: 1080
* left_offset: 0
* top_offset: 0
* luma_offset: 64
* luma_excursion: 876
* color_diff_offset: 512
* color_diff_excursion: 896
* color_primaries_index: hdtv (0)
* color_matrix_index: hdtv (0)
* transfer_function_index: tv_gamma (0)

Explanation (informative)
=========================

Each raw picture contains a single field. The top field comes first.

Pictures contain three planar components: Y, Cb and Cr, in that order, which are
4:2:2 subsampled.

The Y component consists of 1920x540 10 bit values stored as 16 bit (2
byte) values (with the 6 most significant bits set to 0) in little-endian
byte order.  Expressible values run from 0 (video level -0.07) to 1023
(video level 1.09).

The Cb and Cr components consist of 960x540 10 bit values stored as 16 bit
(2 byte) values (with the 6 most significant bits set to 0) in
little-endian byte order. Expressible values run from 0 (video level -0.57)
to 1023 (video level 0.57).

The color model uses the 'hdtv' primaries (ITU-R BT.709), the 'hdtv' color
matrix (ITU-R BT.709) and the 'tv_gamma' transfer function (ITU-R BT.2020).

The pixel aspect ratio is 1:1 (not to be confused with the frame aspect ratio).

Example FFMPEG command (informative)
====================================

The following command can be used to play back this video format using FFMPEG:

    $ ffplay \
        -f image2 \
        -video_size 1920x540 \
        -framerate 60000/1001 \
        -pixel_format yuv422p10le \
        -i picture_%d.raw \
        -vf weave=t,yadif

Where:

* `-f image2` = Read pictures from individual files
* `-video_size 1920x540` = Picture size (not frame size).
* `-framerate 60000/1001` = Picture rate (not frame rate)
* `-pixel_format` = Specifies raw picture encoding.
* `yuv` = Y C1 C2 color.
* `422` = 4:2:2 color difference subsampling.
* `p` = Planar format.
```

(continues on next page)

(continued from previous page)

```
* `10le` = 10 bit little-endian values, LSB-aligned within 16 bit words.
* `-i /tmp/picture_%d.raw` = Input raw picture filename pattern
* `-vf` = define a pipeline of video filtering operations
* `weave=t` = interleave pairs of pictures, top field first
* `yadif` = (optional) apply a deinterlacing filter for display purposes

This command is provided as a minimal example for basic playback of this raw
video format.  While it attempts to ensure correct frame rate, pixel aspect
ratio, interlacing mode and basic pixel format, color model options are omitted
due to inconsistent handling by FFMPEG.

Example ImageMagick command (informative)
=========================================

No ImageMagick command is available for this raw picture format (Unsupported bit
depth: 10 bits).
```

Here, the 'explanation' section provides a human readable description of the raw format. This might be of help when trying to interpret the raw video data.

Example invocations of FFmpeg's[14] `ffplay` command and ImageMagick's[15] `convert` command are provided, when possible, for displaying the raw picture data directly.

Tip: The sample `ffplay` commands generated by `vc2-picture-explain` assume the number in each filename does not contain leading zeros. If your filenames contain leading zeros, replace the `%d` in the picture filenames in the generated commands with `%02d` (or with `2` set to however many digits are used) to handle this situation.

2.3.3 Tip: Splitting and combining picture data files

Many codec implementations natively produce or expect a raw video format where picture data is stored concatenated in a single file rather than as individual files. If individual pictures within a concatenated video format use the same representation as the conformance software, the following commands can be used to convert picture data between single-file and file-per-picture forms.

Note: All of the commands below assume you are using a Bash shell and GNU implementations of standard POSIX tools.

Warning: The commands described below only deal with picture data (`*.raw`) files. You will still need to process the metadata (`*.json` files) by other means.

[14] https://ffmpeg.org/
[15] https://imagemagick.org/

Combining pictures

To concatenate a series of (for example) 8 picture data (**.raw*) files numbered 0 to 7 into a single file, `cat` can be used:

```
$ cat picture_{0..7}.raw > video.raw
```

Warning: The explicit use of the Bash `{0..7}` range specifier is preferred over using a simple wildcard (e.g. `*`). This is because the order in which the individual pictures are listed by the wildcard expansion is not well defined.

Splitting concatenated pictures

To split a series of pictures concatenated together in a single file into individual pictures, `split` can be used:

```
$ split \
    video.raw \
    -b 12345 \
    -d \
    --additional-suffix=".raw" \
    picture_
```

- The file to be split is given as the first argument (`video.raw` in this example)
- The `-b 12345` argument defines the number of bytes in each picture and `12345` should be replaced with the correct number for the format used.
- The `-d` argument causes `split` to number (rather than letter) each output file.
- The `--additional-suffix` argument ensures the output filenames end with `.raw`.
- Final argument gives the start of the output filenames (`picture_` in this example)

Tip: An easy way to determine the picture size for a given video format is to use the `wc` command to get the size of a picture file generated by the conformance software. For example:

```
$ wc -c path/to/picture_0.raw
12345
```

Tip: The `split` command adds leading zeros in the picture numbers of the output files. These will not be found by the sample `ffplay` commands generated by `vc2-picture-explain`. Replace the `%d` in the picture filenames in the generated commands with `%02d` to handle this situation.

Next, let's walk through the process of generating test cases in *Generating test cases* (page 15).

2.4 Generating test cases

Test cases for VC-2 encoders and decoders are generated by the *vc2-test-case-generator* (page 43) program. Below we'll walk through the process of defining the capabilities of our codec in a 'codec features' file and then generating the test cases accordingly.

2.4.1 Defining codec features

In order to generate an appropriate set of test cases for a VC-2 encoder or decoder we must first enumerate the capabilities of that codec in a codec features file. This file must contain a table, in Comma Separated Values (CSV) format enumerating codec configurations to be tested. An example table is given below:

name	hd_lossy	hd_fragmented	hd_lossless	custom_8_bit_rgb
# (11.2.1)				
level	unconstrained	unconstrained	unconstrained	unconstrained
profile	high_quality	high_quality	high_quality	high_quality
# (11.1)				
base_video_format	hd1080p_50	hd1080p_50	hd1080p_50	hd1080p_50
picture_coding_mode	pictures_are_frames	pictures_are_frames	pictures_are_frames	pictures_are_frames
# (11.4.3)				
frame_width	default	default	default	640
frame_height	default	default	default	480
# (11.4.4)				
color_diff_format_index	default	default	default	color_4_4_4
# (11.4.5)				
source_sampling	default	default	default	progressive
top_field_first	default	default	default	TRUE
# (11.4.6)				
frame_rate_numer	default	default	default	25
frame_rate_denom	default	default	default	1
# (11.4.7)				
pixel_aspect_ratio_numer	default	default	default	1
pixel_aspect_ratio_denom	default	default	default	1
# (11.4.8)				
clean_width	default	default	default	640
clean_height	default	default	default	480
left_offset	default	default	default	0
top_offset	default	default	default	0
# (11.4.9)				
luma_offset	default	default	default	0
luma_excursion	default	default	default	255
color_diff_offset	default	default	default	0
color_diff_excursion	default	default	default	255
# (11.4.10)				

continues on next page

Table 1 – continued from previous page

color_primaries_index	default	default	default	hdtv
color_matrix_index	default	default	default	rgb
transfer_function_index	default	default	default	tv_gamma
# (12.4.1) and (12.4.4.1)				
wavelet_index	le_gall_5_3	le_gall_5_3	le_gall_5_3	le_gall_5_3
wavelet_index_ho	le_gall_5_3	le_gall_5_3	le_gall_5_3	le_gall_5_3
dwt_depth	2	2	2	2
dwt_depth_ho	0	0	0	0
# (12.4.5.2)				
slices_x	240	240	240	80
slices_y	135	135	135	60
# Slice size, etc.				
lossless	FALSE	FALSE	TRUE	FALSE
picture_bytes	1296000	1296000		230400
fragment_slice_count	0	8	0	0
# (12.4.5.3)				
quantization_matrix	default	default	default	4 2 2 0 4 4 2

A CSV version of this table can be downloaded here[16].

The first row should provide a unique name for each codec configuration for which test cases are to be generated with the left-most cell containing the text `name`. The remaining rows specify the parameters which define the codec configurations. Empty rows and rows whose first column starts with a `#` are ignored (i.e. treated as comments).

The following parameters must be given for each codec configuration.

`level` Integer or alias. The VC-2 level number (see annex (C.3)) to report in the bit stream. If not `0` (the 'unconstrained' level), the remaining parameters below must be set to values compatible with this level, otherwise the test case generator will reject the codec configuration.

The following aliases can be used in place of an integer for additional readability.

Level	Alias
0	`unconstrained`
1	`sub_sd`
2	`sd`
3	`hd`
4	`d_cinema_2k`
5	`d_cinema_4k`
6	`uhdtv_4k`
7	`uhdtv_8k`
64	`progressive_hd_over_sdi`
65	`hd_over_sd_sdi`
66	`uhd_over_hd_sdi`

`profile` Integer or alias. The VC-2 profile number (see annex (C.2)). `0` for 'low-delay' or `3` for 'high quality'.

The following aliases can be used in place of an integer for additional readability.

[16] https://bbc.github.io/vc2_conformance/_static/user_guide/sample_codec_features.csv

Profile	Alias
0	`low_delay`
3	`high_quality`

`picture_coding_mode` Integer or alias. The picture coding mode to use (see section (11.5)).

The following aliases can be used in place of an integer for additional readability.

Picture Coding Mode	Alias
0	`pictures_are_frames`
1	`pictures_are_fields`

`base_video_format` Integer or alias. The base video format index to use (see section (11.3) and annex (B)).

The following aliases can be used in place of an integer for additional readability.

Base Video Format	Alias
0	`custom_format`
1	`qsif525`
2	`qcif`
3	`sif525`
4	`cif`
5	`foursif525`
6	`foursif`
7	`sd_480i_60`
8	`sd576i_50`
9	`hd720p_60`
10	`hd720p_50`
11	`hd1080i_60`
12	`hd1080i_50`
13	`hd1080p_60`
14	`hd1080p_50`
15	`dc2k`
16	`dc4k`
17	`uhdtv4k_60`
18	`uhdtv4k_50`
19	`uhdtv8k_60`
20	`uhdtv8k_50`
21	`hd1080p_24`
22	`sd_pro486`

Note: This parameter is provided as a convenient way of setting the video parameters below to common sets of options. This parameter, and the options below, do not directly define *how* the video format is encoded in bitstreams – this will be determined automatically by the test case generator. In fact, the test case generator will produce test cases with several different encodings when possible.

`frame_width` and `frame_height` Integer or `default`. The dimensions of frames (not pictures) of video (see section (11.4.3)). If `default`, uses the dimensions specified by the `base_video_format`.

`color_diff_format_index` Integer, alias or `default`. The color difference subsampling mode to use (see section (11.4.4)). If `default`, uses the mode specified by the `base_video_format`.

The following aliases can be used in place of an integer for additional readability.

Index	Alias
0	`color_4_4_4`
1	`color_4_2_2`
2	`color_4_2_0`

`source_sampling` Integer, alias or `default`. The scan format to use (see section (11.4.5)). If `default`, uses the mode specified by the `base_video_format`.

The following aliases can be used in place of an integer for additional readability.

Index	Alias
0	`progressive`
1	`interlaced`

Note: This parameter is used as metadata only. It should not be confused with the `picture_coding_mode` parameter which determines whether each picture in a sequence contains a whole frame or a field of video.

`top_field_first` `TRUE`, `FALSE` or `default`. Indicates, for interlaced formats, whether the earlier field in a sequence contains the top field of a frame (`TRUE`) or bottom field (`FALSE`) (see section (11.4.5)). If `default`, uses the mode specified by the `base_video_format`.

`frame_rate_numer` and `frame_rate_denom` Integers or `default`. The frame rate (see section (11.4.6)). If `default`, uses the mode specified by the `base_video_format`.

`pixel_aspect_ratio_numer` and `pixel_aspect_ratio_denom` Integers or `default`. The pixel aspect ratio (see section (11.4.7)). If `default`, uses the mode specified by the `base_video_format`.

`clean_width`, `clean_height`, `left_offset` and `top_offset` Integers or `default`. The clean area (see section (11.4.8)). If `default`, uses the mode specified by the `base_video_format`.

`luma_offset`, `luma_excursion`, `color_diff_offset` and `color_diff_excursion` Integers or `default`. The luma and color difference picture component signal ranges (see section (11.4.9)). If `default`, uses the mode specified by the `base_video_format`.

`color_primaries_index`, `color_matrix_index` and `transfer_function_index` Integers, aliases or `default`. color specification options (see section (11.4.10)). If `default`, uses the mode specified by the `base_video_format`.

The following aliases can be used in place of an integer for additional readability.

Color Primaries Index	Alias
0	`hdtv`
1	`sdtv_525`
2	`sdtv_625`
3	`d_cinema`
4	`uhdtv`

Color Matrix Index	Alias
0	`hdtv`
1	`sdtv`
2	`reversible`
3	`rgb`
4	`uhdtv`

Transfer Function Index	Alias
0	`tv_gamma`
1	`extended_gamut`
2	`linear`
3	`d_cinema`
4	`perceptual_quantizer`
5	`hybrid_log_gamma`

`wavelet_index` and `wavelet_index_ho` Integers or aliases. Wavelet transform types to use vertically and horizontally, respectively (see sections (11.4.1) and (11.4.4.1)). For symmetric transforms, these values must be the same.

The following aliases can be used in place of an integer for additional readability.

Index	Alias
0	`deslauriers_dubuc_9_7`
1	`le_gall_5_3`
2	`deslauriers_dubuc_13_7`
3	`haar_no_shift`
4	`haar_with_shift`
5	`fidelity`
6	`daubechies_9_7`

`dwt_depth` and `dwt_depth_ho` Integers. Wavelet transform depths to use for both dimensions and horizontally only, respectively (see sections (11.4.1) and (11.4.4.1)). For symmetric transforms, `dwt_depth_ho` must be `0`.

`slices_x` and `slices_y` Integers. The number of horizontal and vertical picture slices to use (see section (12.4.5.2)).

`lossless` Boolean. If `FALSE`, test cases will be generated for a constant bit rate (lossy) codec. If `TRUE` test cases will be generated for a lossless (variable bit rate) codec. Lossless mode is only supported by the high quality profile.

`picture_bytes` Integer or blank. The number of bytes to use to encode the slices in each picture. Must be an integer when `lossless` is `FALSE` and blank when `lossless` is `TRUE`.

For the low delay profile, this sets the `slice_bytes_numerator` and `slice_bytes_denominator` values used by the stream (see section (13.5.3.2)) to the value `picture_bytes` divided by the number of slices per picture.

For the high quality profile, when `lossless` is `FALSE`, slices are assigned sizes using the same formula as used for the low delay profile. When `lossless` is `TRUE`, slices are sized as small as possible for the data they hold.

Note: This value only accounts for picture slice data, i.e. the data read by the `slice` pseudocode function in section (13.5.3). It does not take into account other stream overheads (e.g. sequence headers and transform parameters). As such the resulting stream will have a slightly higher bit rate than `picture_bytes` bytes per picture.

`fragment_slice_count` Integer.

If zero, non-fragmented picture coding is used: each picture will be coded as a single picture parse data unit.

If greater than zero, fragmented picture mode will be used (see section (14)). Pictures will be coded as several fragment parse data units containing at most `fragment_slice_count` slices each.

`quantization_matrix` List of space-separated integers or `default`. Specifies the quantization matrix to be used.

If `default`, the default quantisation matrix for the wavelet transform specified by `wavelet_index`, `wavelet_index_ho`, `dwt_depth` and `dwt_depth_ho` will be used (see annexe (D.2)).

If a list of space separated integers are provided defining a quantisation matrix, these will be used instead and encoded as a custom quantisation matrix in the stream (see (12.4.5.3)).

Quantisation matrix values, if provided, should be given in the same order they would appear in the stream as defined by the `quant_matrix` pseudocode function (12.4.5.3). For example for a transform with dwt_depth = 1 and dwt_depth_ho = 2, the following value:

```
0 1 2 3 4 5
```

Describes the following quantization matrix:

```
{
    0: {"L": 0},
    1: {"H": 1},
    2: {"H": 2},
    3: {"HL": 3, "LH": 4, "HH": 5},
}
```

If a non `default` value is given, the majority (though not all) generated test cases will use the supplied quantization matrix (with the `custom_quant_matrix` flag set (12.4.5.3)).

2.4.2 Generating test cases

Once a codec features CSV has been created, with columns covering the major operating modes of the codec to be tested, the *vc2-test-case-generator* (page 43) command can be used to generate test cases.

In the simplest case, the command should be provided with the filename of your codec features CSV:

```
$ vc2-test-case-generator path/to/codec_features.csv
```

By default, a `test_cases` directory will be created into which the test cases are written. This can be changed using the `--output <path>` argument. The `--verbose` option can be used to keep track of progress.

If only test cases for an encoder are required, the `--encoder-only` option can be given. Alternatively if only decoder test cases are needed `--decoder-only` can be used. By default, test cases are generated for both encoders and decoders.

Before any test cases are generated, the test case generator internally generates and then validates a simple test stream for each column of the codec features table. This step ensures that the codec features specified are not in conflict with themselves or the VC-2 standard. If this step fails, an error message is produced indicating the problem and test case generation is aborted.

If you are using a wavelet transform combination or depth for which a default quantization matrices are not provided in the VC-2 specification (see annexe (D.2)), the test case generator will produce the following warning:

```
WARNING:root:No static analysis available for the wavelet used by codec '<name>'.␣
↪Signal range test cases cannot be generated.
```

See *Generating static wavelet filter analyses* (page 38) for instructions on this specific case.

Warning messages are otherwise only produced for degenerate codec configurations. It is very unlikely a useful codec configuration will result in a warning. If any are produced, check the values in your codec features CSV if warnings are encountered.

Test case generation typically requires several hours, depending on the codec feature sets provided.

Note: The slow runtime performance of the VC-2 conformance software is an unfortunate side effect of it being based on the pseudocode published in the VC-2 specification. This design gives a high degree of confidence that it is consistent with the specification at the cost of slow execution.

2.4.3 Parallel test case generation

To speed up test case generation on multi-core systems, independent test cases can be generated in parallel. To do this, the `--parallel` argument is used. Instead of generating test cases, when `--parallel` is used, the test case generator will print a series of commands which can be executed in parallel to generate the test cases, for example using GNU Parallel[17]:

```
$ # Write test case generation commands to 'commands.txt'
$ vc2-test-case-generator path/to/codec_features.csv --parallel > commands.txt

$ # Run test case generation in parallel using GNU Parallel
$ parallel -a commands.txt
```

Warning: Some test cases require relatively large quantities of RAM during test case generation. You might need to reduce the number of commands run in parallel if your system runs out of memory. If you're using GNU parallel, the `-j N` argument can be used to set the number of parallel jobs to `N` (with the default being however many CPU cores are available).

2.4.4 Directory structure

The test case generator produces a directory structure as outlined below:

- **test_cases/**
  - **<codec feature set name>/**
    - **decoder/ – Test VC-2 bitstreams for decoders.**
      - `<test-case-name>.vc2` – VC-2 bitstream to be decoded.
      - `<test-case-name>_metadata.json` – Optional metadata file provided for some tests
      - **<test-case-name>_expected/ – Reference decoding of the bitstream.**
        - `picture_<N>.raw`
        - `picture_<N>.json`
    - **encoder/ – Test raw video streams for encoders.**
      - `<test-case-name>_metadata.json` – Optional metadata file provided for some tests
      - **<test-case-name>/ – Raw video to be encoded**
        - `picture_<N>.raw`
        - `picture_<N>.json`

The testing procedures for decoders and encoders are described in the next two sections:

- *VC-2 decoder conformance testing procedure* (page 22)
- *VC-2 encoder conformance testing procedure* (page 31)

[17] https://www.gnu.org/software/parallel/

2.5 VC-2 decoder conformance testing procedure

The VC-2 decoder conformance testing procedure is described below. In summary, each of the bitstreams generated in the previous step (*Generating test cases* (page 15)), we will be decoded using the candidate decoder and the resulting raw video compared with a reference decoding.

Note: Whilst it is possible to carry out the decoder testing procedure manually, we recommend producing a script to automate the steps required for the particular decoder being tested.

2.5.1 Decoding the reference bitstreams

For each codec feature set provided, a set of bitstreams which test different decoder behaviours are produced. These are located in files whose names match the pattern `<codec feature set name>/decoder/<test-case-name>.vc2`. The specific test cases generated will vary depending on the codec features specified.

Each bitstream must be decoded independently by the decoder under test. The decoded pictures must then be stored as raw video as described in the *Video file format* (page 8) section.

2.5.2 Checking the decoded pictures

Each bitstream has an associated reference decoding in the `<codec feature set name>/decoder/<test-case-name>_expected/` directory. The output of the decoder under test must be identical to the reference decoding.

The *vc2-picture-compare* (page 46) tool is provided for comparing decoder outputs with the reference decodings. It takes as argument either the names of two raw picture files, or the names of two directories containing numbered raw picture files. Differences between pictures are then reported.

For example:

```
$ vc2-picture-compare expected/ actual/
Comparing expected/picture_0.raw and actual/picture_0.raw
  Pictures are identical
Comparing expected/picture_1.raw and actual/picture_1.raw
  Pictures are identical
Comparing expected/picture_2.raw and actual/picture_2.raw
  Pictures are identical
Summary: 3 identical, 0 different
```

Note: When provided with two directories to compare the *vc2-picture-compare* (page 46) tool will ignore all but the numercial part of the filenames when matching pictures together. Differing names or uses (and non-uses) of leading zeros are ignored. For example, it would compare two files named `expected/picture_12.raw` and `actual/image_0012.raw`.

For a test case to pass:

- The *vc2-picture-compare* (page 46) tool must report `Pictures are identical`, with no warnings, for every picture in the reference decoding.
- No additional pictures must have been decoded by the decoder under test.
- The decoder under test must not have crashed or indicated an error condition while decoding the bitstream.

For a decoder to pass the conformance test, all test cases, for all supported codec feature sets must pass. If any tests fail, this indicates that the decoder is non-conformant to the VC-2 specification.

The section below outlines the purpose of each test case and gives advice on what that case failing could indicate. Alternatively, once all decoder tests have passed, we can continue onto *VC-2 encoder conformance testing procedure* (page 31).

2.5.3 Decoder Test Cases

The purpose of each test case (or group of test cases), along with advice on debugging failing tests is provided below. In all test cases, the bitstream provided is a valid bitstream permitted by the spec.

Decoder test case: `absent_next_parse_offset`

Tests handling of missing 'next parse offset' field.

The 'next parse offset' field of the `parse_info` header (see (10.5.1)) can be set to zero (i.e. omitted) for pictures. This test case verifies that decoders are still able to decode streams with this field absent.

Decoder test case: `concatenated_sequences`

Tests that streams containing multiple concatenated sequences can be decoded.

A stream consisting of the concatenation of two sequences (10.3) with one frame each, the first picture is given picture number zero in both sequences.

Decoder test case: `custom_quantization_matrix`

Tests that a custom quantization matrix can be specified.

A series of bitstreams with different custom quantisation matrices are generated as follows:

`custom_quantization_matrix[zeros]` Specifies a custom quantisation matrix with all matrix values set to zero.

`custom_quantization_matrix[arbitrary]` Specifies a custom quantisation matrix with all matrix values set to different, though arbitrary, values.

`custom_quantization_matrix[default]` Specifies a custom quantisation matrix containing the same values as the default quantisation matrix. This test case is only generated when a default quantization matrix is defined for the codec.

These test cases are only generated when permitted by the VC-2 level in use.

Note: For lossy coding modes, the encoded picture will contain a noise signal (see the *`static_noise`* (page 30) test case).

For lossless coding modes, the encoded picture will be the test pattern used by the *`lossless_quantization`* (page 26) test case. This test pattern is designed to be losslessly encodable when some quantization is applied.

Decoder test case: `dangling_bounded_block_data`

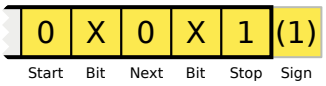
Tests that transform values which lie beyond the end of a bounded block are read correctly.

Picture slices (13.5.3) and (13.5.4) contain transform values in bounded blocks (A.4.2). These test cases include bounded blocks in which some encoded values lie off the end of the block. Specifically, the following cases are tested:

`dangling_bounded_block_data[zero_dangling]`

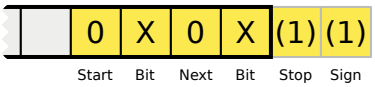
A zero value (1 bit) is encoded entirely beyond the end of the bounded block.

`dangling_bounded_block_data[sign_dangling]`



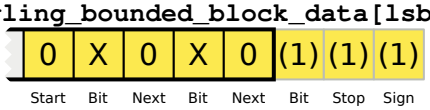
The final bit (the sign bit) of a non-zero exp-golomb value is dangling beyond the end of the bounded block.

`dangling_bounded_block_data[stop_and_sign_dangling]`



The final two bits (the stop bit and sign bit) of a non-zero exp-golomb value are dangling beyond the end of the bounded block.

`dangling_bounded_block_data[lsb_stop_and_sign_dangling]`



The final three bits (the least significant bit, stop bit and sign bit) of a non-zero exp-golomb value are dangling beyond the end of the bounded block.

Note: The value and magnitudes of the dangling values are chosen depending on the wavelet transform in use and might differ from the illustrations above.

Decoder test case: `default_quantization_matrix`

Tests that the default quantization matrix can be used.

This test case is only generated when a non `default` value is specified for the `quantization_matrix` codec features CSV entry but when a default quantization matrix is defined.

Note: This is the only test case which sets the `custom_quant_matrix` flag (12.4.5.3) to 0 when a `quantization_matrix` is supplied in the codec features CSV.

Note: For lossy coding modes, the encoded picture will contain a noise signal (see the *`static_noise`* (page 30) test case).

For lossless coding modes, the encoded picture will be the test pattern used by the *`lossless_quantization`* (page 26) test case. This test pattern is designed to be losslessly encodable when some quantization is applied.

Decoder test case: `extended_transform_parameters`

Tests that extended transform parameter flags are handled correctly.

Ensures that extended transform parameters fields (12.4.4) are correctly handled by decoders for symmetric transform modes.

`extended_transform_parameters[asym_transform_index_flag]` Verifies that `asym_transform_index_flag` can be set to `1`.

`extended_transform_parameters[asym_transform_flag]` Verifies that `asym_transform_flag` can be set to `1`.

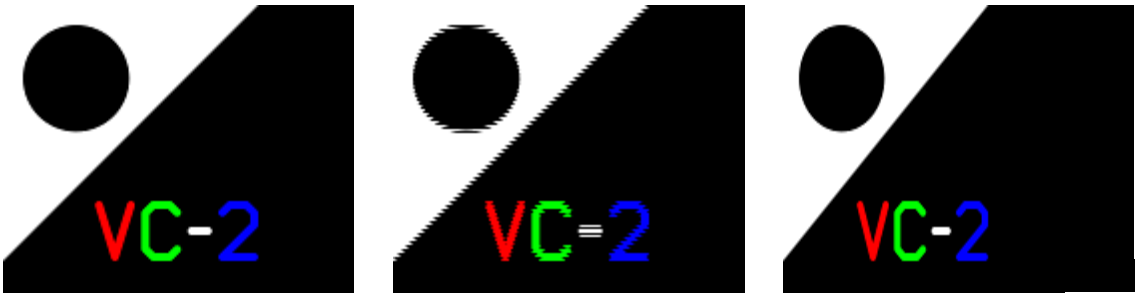
These test cases are skipped for streams whose major version is less than 3 (which do not support the extended transform parameters header). Additionally, these test cases are skipped for asymmetric transforms when the flag being tested must already be `1`.

Decoder test case: `interlace_mode_and_pixel_aspect_ratio`

Tests that the interlacing mode and pixel aspect ratio is correctly decoded.

These tests require that the decoded pictures are observed using the intended display equipment for the decoder to ensure that the relevant display metadata is passed on.

`interlace_mode_and_pixel_aspect_ratio[static_sequence]` A single frame containing a stationary graphic at the top-left corner on a black background, as illustrated below.



Correct | Wrong Field Order | Wrong Pixel Aspect Ratio

If the field ordering (i.e. top field first flag, see (7.3) and (11.3)) has been decoded correctly, the edges should be smooth. If the field order has been reversed the edges will appear jagged.

If the pixel aspect ratio (see (11.4.7)) has been correctly decoded, the white triangle should be as wide as it is tall and the 'hole' should be circular.

`interlace_mode_and_pixel_aspect_ratio[moving_sequence]` A sequence of 10 frames containing a graphic moving from left to right along the top of the frame. In each successive frame, the graphic moves 16 luma samples to the right (i.e. 8 samples every field, for interlaced formats).

Progressive (Frame 0) Progressive (Frame 1) Progressive (Frame 2)

Interlaced (Frame 0) Interlaced (Frame 1) Interlaced (Frame 2)

For progressive formats, the graphic should appear with smooth edges in each frame.

For interlaced formats, the graphic should move smoothly when displayed on an interlaced monitor. If displayed as progressive frames (as in the illustration above), the pictures will appear to have ragged edges.

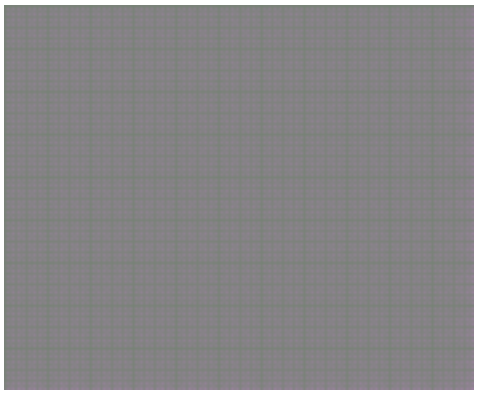
Decoder test case: `lossless_quantization`

Tests support for quantization in lossless decoders.

Quantization can, in principle, be used in lossless coding modes in cases where all transform coefficients are divisible by the same factor. This test case contains a synthetic test pattern with this property.

This test case is only generated for lossless codecs.

Note: For informational purposes, an example decoded test pattern is shown below:



Note the faint repeating pattern.

Decoder test case: `padding_data`

Tests that the contents of padding data units are ignored.

This test case consists of a sequence containing two blank frames in which every-other data unit is a padding data unit (10.4.5) of various lengths and contents (described below).

`padding_data[empty]` Padding data units containing zero padding bytes (i.e. just consisting of a parse info header).

`padding_data[zero]` Padding data units containing 32 bytes set to 0x00.

`padding_data[non_zero]` Padding data units containing 32 bytes containing the ASCII encoding of the text `Ignore this padding data please!`.

`padding_data[dummy_end_of_sequence]` Padding data units containing 32 bytes containing an encoding of an end of sequence data unit (10.4.1).

Where padding data units are not permitted by the VC-2 level in use, these test cases are omitted.

Decoder test case: `picture_numbers`

Tests picture numbers are correctly read from the bitstream.

Each test case contains 8 blank pictures numbered in a particular way.

`picture_numbers[start_at_zero]` The first picture has picture number 0.

`picture_numbers[non_zero_start]` The first picture has picture number 1000.

`picture_numbers[wrap_around]` The first picture has picture number 4294967292, with the picture numbers wrapping around to 0 on the 4th picture in the sequence.

`picture_numbers[odd_first_picture]` The first picture has picture number 7. This test case is only included when the picture coding mode is 0 (i.e. pictures are frames) since the first field of each frame must have an even number when the picture coding mode is 1 (i.e. pictures are fields) (11.5).

Decoder test case: `real_pictures`

Tests real pictures are decoded correctly.

A series of three still photographs.

Frame 0

Frame 1

Frame 2

Note: The images encoded in this sequence are generated from 4256 by 2832 pixel, 4:4:4, 16 bit, standard dynamic range, RGB color images with the ITU-R BT.709 gamut. As such, the decoded pictures might be of reduced technical quality compared with the capabilities of the format. The rescaling, color conversion and encoding algorithms used are also basic in nature, potentially further reducing the picture quality.

Decoder test case: `repeated_sequence_headers`

Tests the decoder can handle a stream with repeated sequence headers.

This test case consists of a sequence containing two frames in which the sequence header is repeated before every picture.

This test will be omitted if the VC-2 level prohibits the repetition of the sequence header.

Decoder test case: `signal_range`

Tests that a decoder has sufficient numerical dynamic range.

These test cases contain a series of pictures containing test patterns designed to produce extreme signals within decoders. During these test cases, no integer clamping (except for final output clamping) or integer overflows must occur.

A test case is produced for each picture component:

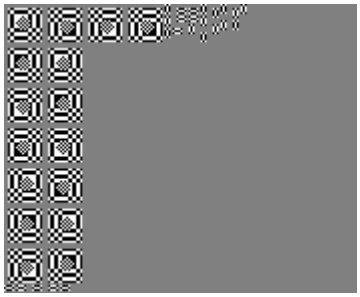
`signal_range[Y]` Luma component test patterns.

`signal_range[C1]` Color difference 1 component test patterns.

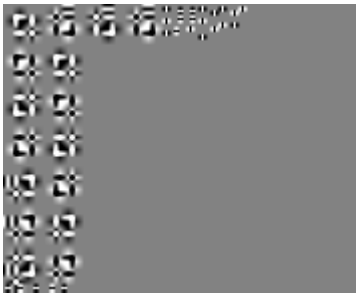
`signal_range[C2]` Color difference 2 component test patterns.

These test cases are produced by encoding pictures consisting test patterns made up of entirely of legal (in range) signal values. Nevertheless, the resulting bitstreams produce large intermediate values within a decoder, though these are not guaranteed to be worst-case.

Note: For informational purposes, an example of a set of test patterns before and after encoding and quantisation is shown below:



Original Image



Decoder Output

Note: The quantization indices used for lossy codecs are chosen to maximise the peak signal range produced by the test patterns. These are often higher than a typical VC-2 encoder might pick for a given bit rate but are nevertheless valid.

An informative metadata file is provided along side each test case which gives, for each picture in the bitstream, the parts of a decoder which are being tested by the test patterns. See `vc2_bit_widths.helpers.TestPoint` ([vc2_bit_widths], page 40) for details.

Decoder test case: `slice_padding_data`

Tests that padding bits in picture slices are ignored.

Picture slices (13.5.3) and (13.5.4) might contain padding bits beyond the end of the transform coefficients for each picture component. These test cases check that decoders correctly ignore these values. Padding values will be filled with the following:

`slice_padding_data[slice_?_all_zeros]` Padding bits are all zero.

`slice_padding_data[slice_?_all_ones]` Padding bits are all one.

`slice_padding_data[slice_?_alternating_1s_and_0s]` Padding bits alternate between one and zero, starting with one.

`slice_padding_data[slice_?_alternating_0s_and_1s]` Padding bits alternate between zero and one, starting with zero.

`slice_padding_data[slice_?_dummy_end_of_sequence]` Padding bits will contain bits which encode an end of sequence data unit (10.6).

The above cases are repeated for the luma and color difference picture components as indicated by the value substituted for `?` in the test case names above. For low-delay pictures these will be `Y` (luma) and `C` (interleaved color difference). For high quality pictures these will be `Y` (luma), `C1` (color difference 1) and `C2` (color difference 2).

Decoder test case: `slice_prefix_bytes`

Tests the decoder can handle a non-zero number of slice prefix bytes.

Produces test cases with a non-zero number of slice prefix bytes containing the following values:

`slice_prefix_bytes[zeros]` All slice prefix bytes are 0x00.

`slice_prefix_bytes[ones]` All slice prefix bytes are 0xFF.

`slice_prefix_bytes[end_of_sequence]` All slice prefix bytes contain bits which encode an end of sequence data unit (10.4).

These test cases apply only to the high quality profile and are omitted when the low delay profile is used.

Decoder test case: `slice_size_scaler`

Tests that the 'slice_size_scaler' field is correctly handled.

This test case generates a sequence which sets slice_size_scaler value (13.5.4) 1 larger than it otherwise would be.

This test case is only generated for the high quality profile, and levels which permit a slice size scaler value greater than 1.

Decoder test case: `source_parameters_encodings`

Tests the decoder can decode different encodings of the video format metadata.

This series of test cases each contain the same source parameters (11.4), but in different ways.

`source_parameters_encodings[custom_flags_combination_?_base_video_format_?]` For these test cases, the base video format which most closely matches the desired video format is used. Each test case incrementally checks that source parameters can be explicitly set to their desired values (e.g. by setting `custom_*_flag` bits to 1).

`source_parameters_encodings[base_video_format_?]` These test cases, check that other base video formats can be used (and overridden) to specify the desired video format. Each of these test cases will explicitly specify as few video parameters as possible (e.g. setting as many `custom_*_flag` fields to 0 as possible).

Tip: The *vc2-bitstream-viewer* (page 51) can be used to display the encoding used in a given test case as follows:

```
$ vc2-bitstream-viewer --show sequence_header path/to/test_case.vc2
```

Note: Some VC-2 levels constrain the allowed encoding of source parameters in the bit stream and so fewer test cases will be produced in this instance.

Note: Not all base video formats can be used as the basis for encoding a specific video format. For example, the 'top field first' flag (11.3) set by a base video format cannot be overridden. As a result, test cases will not include every base video format index.

Decoder test case: `static_gray`

Tests that the decoder can decode a maximally compressible sequence.

This sequence contains an image in which every transform coefficient is zero. For most color specifications (11.4.10), this decodes to a mid-gray frame.

This special case image is maximally compressible since no transform coefficients need to be explicitly coded in the bitstream. For lossless coding modes, this will also produce produce the smallest possible bitstream.

Decoder test case: `static_noise`

Tests that decoder correctly decodes a noise plate.

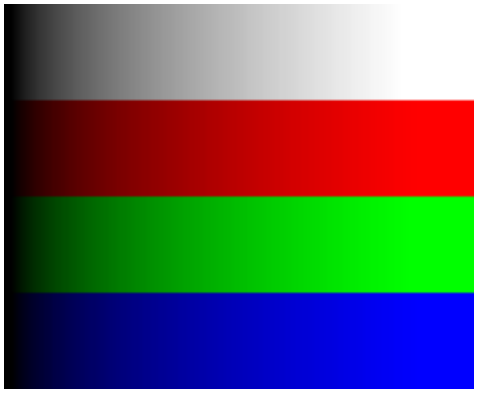
A static frame containing pseudo-random uniform noise as illustrated below:

Decoder test case: `static_ramps`

Tests that decoder correctly reports color encoding information.

This test requires that the decoded pictures are observed using the intended display equipment for the decoder to ensure that the relevant color coding metadata is passed on.

A static frame containing linear signal ramps for white and primary red, green and blue (in that order, from top-to-bottom) as illustrated below:



The color bands must be in the correct order (white, red, green, blue from top to bottom). If not, the color components might have been ordered incorrectly.

The red, green and blue colors should correspond to the red, green and blue primaries for the color specification (11.4.10.2).

Note: When D-Cinema primaries are specified (preset color primaries index 3), red, green and blue are replaced with CIE X, Y and Z respectively. Note that these might not represent physically realisable colors.

The left-most pixels in each band are notionally video black and the right-most pixels video white, red, green and blue (respectively). That is, oversaturated signals (e.g. 'super-blacks' and 'super-white') are not included.

Note: For lossy codecs, the decoded signal values might vary due to coding artefacts.

The value ramps in the test picture are linear, meaning that the (linear) pixel values increase at a constant rate from left (black) to right (saturated white/red/green/blue). Due to the non-linear response of human vision, this will produce a non-linear brightness ramp which appears to quickly saturate. Further, when a non-linear transfer function is specified (11.4.10.4) the raw decoded picture values will not be linearly spaced.

Note: When the D-Cinema transfer function is specified (preset transfer function index 3), the saturated signals do not correspond to a non-linear signal value of 1.0 but instead approximately 0.97. This is because the D-Cinema transfer function allocates part of its nominal output range to over-saturated signals.

2.6 VC-2 encoder conformance testing procedure

The VC-2 encoder conformance testing procedure is described below. In summary, the raw video test sequences generated in the *Generating test cases* (page 15) step will be encoded using the candidate encoder. The resulting bitstream will then be checked using the *vc2-bitstream-validator* (page 45) and encoded pictures compared with the originals for similarity.

Note: Whilst it is possible to carry out the encoder testing procedure manually, we recommend producing a script to automate most of the steps required for the particular encoder being tested. You must still take care to manually inspect and compare all decoded pictures, however.

2.6.1 Encoding the reference bitstreams

For each codec feature set provided, a set of raw videos are produced. These are located in directories matching the pattern `<codec feature set name>/encoder/<test-case-name>/`. The specific test cases generated will vary depending on the codec features specified.

Each test case must be encoded independently by the encoder under test and the encoded bitstreams stored on disk. The encoder must not crash or produce any warnings when encoding these test sequences.

2.6.2 Checking the encoded bitstreams

Each encoded bitstream must be checked with the *vc2-bitstream-validator* (page 45). This tool simultaneously verifies that the bitstream meets the requirements of the VC-2 specification and provides a reference decoding of the stream.

The tool takes a VC-2 bitstream as argument and optionally an output name for the decoded pictures, as illustrated below:

```
$ mkdir real_pictures_decoded
$ vc2-bitstream-validator \
    real_pictures.vc2 \
    --output real_pictures_decoded/picture_%d.raw
```

If the bitstream is valid, the following message will be produced:

```
No errors found in bitstream. Verify decoded pictures to confirm conformance.
```

Otherwise, if a conformance error is found, processing will stop and a detailed error message will be produced explaining the problem.

Once a bitstream has been validated and decoded using *vc2-bitstream-validator* (page 45), the *vc2-picture-compare* (page 46) command is used to compare the output against the original pictures. The script must be provided with two raw picture filenames, or two directory names containing raw pictures. One should contain the original images, and the other its encoded then decoded counterpart. The similarity of the images will be reported. For example:

```
$ vc2-picture-compare real_pictures/ real_pictures_decoded/
Comparing real_pictures/picture_0.raw and real_pictures_decoded/picture_0.raw
  Pictures are different:
    Y: Different: PSNR = 55.6 dB, 1426363 pixels (68.8%) differ
    C1: Different: PSNR = 57.7 dB, 662607 pixels (63.9%) differ
    C2: Different: PSNR = 56.8 dB, 703531 pixels (67.9%) differ
Comparing real_pictures/picture_1.raw and real_pictures_decoded/picture_1.raw
  Pictures are different:
    Y: Different: PSNR = 55.6 dB, 1426363 pixels (68.8%) differ
    C1: Different: PSNR = 57.7 dB, 662607 pixels (63.9%) differ
    C2: Different: PSNR = 56.8 dB, 703531 pixels (67.9%) differ
Comparing real_pictures/picture_2.raw and real_pictures_decoded/picture_2.raw
  Pictures are different:
    Y: Different: PSNR = 55.6 dB, 1426363 pixels (68.8%) differ
    C1: Different: PSNR = 57.7 dB, 662607 pixels (63.9%) differ
    C2: Different: PSNR = 56.8 dB, 703531 pixels (67.9%) differ
Summary: 0 identical, 3 different
```

Note: When provided with two directories to compare the *vc2-picture-compare* (page 46) tool will ignore all but the numercial part of the filenames when matching pictures together. Differing names or uses (and non-uses) of leading zeros are ignored. For example, it would compare two files named `expected/picture_12.raw` and `actual/image_0012.raw`.

For a test case to pass:

- The encoder must not raise an error condition during encoding.
- The *vc2-bitstream-validator* (page 45) must not find any errors in the bit stream.
- For lossless encoders, *vc2-picture-compare* (page 46) tool must report `Pictures are identical`, with no warnings, for every picture in the reference decoding.
- For lossy encoders, *vc2-picture-compare* (page 46) tool might report a difference and the quoted PSNR figure should be checked to ensure it is appropriate for the intended application of the codec.
- Input and output pictures must be visually compared and should be sufficiently similar as to be suitable for the intended application of the codec.
- No additional pictures must have been decoded.

Tip: When viewing pictures using the `ffplay` commands suggested by *vc2-picture-explain* (page 49) you might sometimes find it helpful to use a very low frame rate or playback the sequence in a loop.

To reduce the frame rate such that each frame is shown for 5 seconds, replace the value after `-framerate` with `1/5`.

To loop the sequence indefinately add `-loop 0` to the command.

For an encoder to pass the conformance test, all test cases, for all supported codec feature sets must pass. If any tests fail, this indicates that the encoder is non-conformant to the VC-2 specification.

The section below outlines the purpose of each test case and gives advice on what that case failing might indicate.

2.6.3 Encoder test cases

The purpose of each test case (or group of test cases), along with advice on debugging failing tests is provided below.

Encoder test case: `real_pictures`

Tests real pictures are encoded sensibly.

This test contains a series of three still photographs of natural scenes with varying levels of high and low spatial frequency content.

Frame 0

Frame 1

Frame 2

Note: The source images for this test case are generated from 4256 by 2832 pixel, 4:4:4, 16 bit, standard dynamic range, RGB color images with the ITU-R BT.709 gamut. As such, the pictures might be of reduced technical quality compared with the capabilities of the format. The rescaling and color conversion algorithms used are also basic in nature, potentially further reducing the picture quality.

Encoder test case: `signal_range`

Tests that an encoder has sufficient numerical dynamic range.

These test cases contain test patterns designed to produce extreme signals within encoders. During these test cases, no integer clamping or overflows must occur.

A test case is produced for each picture component:

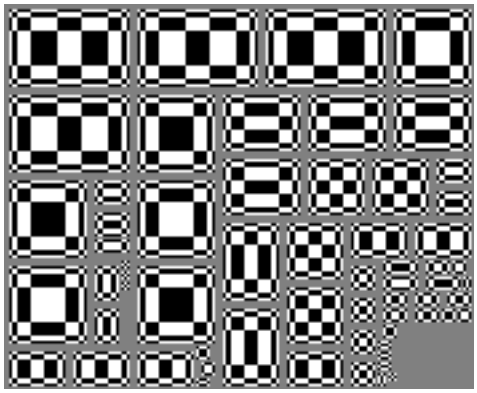
`signal_range[Y]` Luma component test patterns.

`signal_range[C1]` Color difference 1 component test patterns.

`signal_range[C2]` Color difference 2 component test patterns.

Though the test patterns produce near worst case signal levels, they are not guaranteed to produce the largest values possible.

Note: For informational purposes, an example of a set of test patterns are shown below:



An informative metadata file is provided along side each test case which gives, for each picture in the bitstream, the parts of a encoder which are being tested by the test patterns. See `vc2_bit_widths.helpers.TestPoint` ([vc2_bit_widths], page 40) for details.

Encoder test case: `synthetic_gray`

Tests that the encoder can encode a maximally compressible sequence.

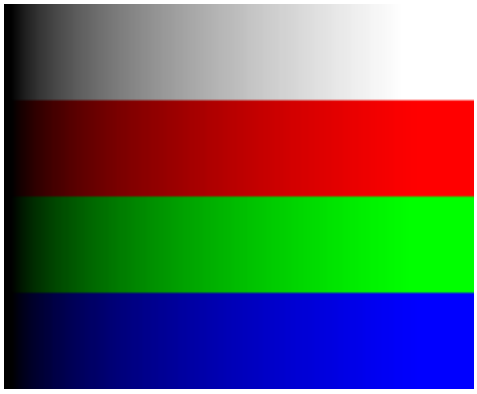
This sequence contains an image in which every transform coefficient is zero. For most color specifications (11.4.10), this decodes to a mid-gray frame.

This special case image is maximally compressible since no transform coefficients need to be explicitly coded in the bitstream. For lossless coding modes, this should also produce produce the smallest possible bitstream.

Encoder test case: `synthetic_linear_ramps`

Tests that an encoder correctly encodes color specification information.

A static frame containing linear signal ramps for white and primary red, green and blue (in that order, from top-to-bottom) as illustrated below:



The red, green and blue colors correspond to the red, green and blue primaries for the color specification (11.4.10.2).

Note: When D-Cinema primaries are specified (preset color primaries index 3), red, green and blue are replaced with CIE X, Y and Z respectively. Note that these might not represent physically realisable colors.

The left-most pixels in each band are video black and the right-most pixels video white, red, green and blue (respectively). That is, oversaturated signals (e.g. 'super-blacks' and 'super-white') are not included.

The value ramps in the test picture are linear meaning that the (linear) pixel values increase at a constant rate from left (black) to right (saturated white/red/green/blue). Due to the non-linear response of human vision, this will produce a non-linear brightness ramp which appears to quickly saturate. Further, when a non-linear transfer function is specified (11.4.10.4) the raw picture values will not be linearly spaced.

Note: When the D-Cinema transfer function is specified (preset transfer function index 3), the saturated signals do not correspond to a non-linear signal value of 1.0 but instead approximately 0.97. This is because the D-Cinema transfer function allocates part of its nominal output range to over-saturated signals.

Encoder test case: `synthetic_moving_sprite`

Tests that an encoder produces sensible results for motion.

A sequence of 10 frames containing a graphic moving from left to right along the top of the frame. In successive each frame, the graphic moves 16 luma samples to the right (i.e. 8 samples every field, for interlaced formats).

Progressive (Frame 0) Progressive (Frame 1) Progressive (Frame 2)

Interlaced (Frame 0) Interlaced (Frame 1) Interlaced (Frame 2)

For progressive formats, the graphic should appear with smooth edges in each frame.

For interlaced formats, the graphic should move smoothly when displayed on an interlaced monitor. If displayed as progressive frames (as in the illustration above), the pictures will appear to have ragged edges.

Encoder test case: `synthetic_noise`

Tests that an encoder correctly encodes a noise plate.

A static frame containing pseudo-random uniform noise as illustrated below:

Note: It is likely that lossy encoders will be unable to compress this test case without a fairly significant loss of fidelity. As such, it is acceptable for this test case for an encoder to produce only visually similar results.

2.7 Testing additional bitstreams' conformance

the *vc2-bitstream-validator* (page 45) tool can be used to test any VC-2 bitstreams, including those produced outside of the conformance testing procedures. For example, you might wish to test how an encoder behaves with your own test materials. Alternatively you might have existing bitstreams which you would like to validate, perhaps because a decoder you're testing is having trouble with them.

2.7.1 Bitstream validation

Any existing bitstream can be validated following a similar manner to the *VC-2 encoder conformance testing procedure* (page 31) using the *vc2-bitstream-validator* (page 45) command like so:

```
$ vc2-bitstream-validator \
    my_bitstream.vc2 \
    --output my_decoded_picture_%d.raw
```

Note: The bitstream validator always produces a decoded output. If `--output` is not given, the decoded pictures will still be produced but given the default name `picture_%d.raw`.

If the bitstream is valid, the following message will be produced:

```
No errors found in bitstream. Verify decoded pictures to confirm conformance.
```

Otherwise, if a conformance error is found, processing will stop and a detailed error message will be produced explaining the problem.

Once a bitstream has been validated by *vc2-bitstream-validator* (page 45), you can view the decoded pictures to verify that the bitstream contained the pictures and metadata you expected. You might wish to use *vc2-picture-explain* (page 49) to produce an ffmpeg or ImageMagick command to do this.

2.7.2 Decoded picture validation

The *vc2-bitstream-validator* (page 45) tool will correctly decode any valid VC-2 bitstream. You can therefore use its output to validate that a decoder has correctly decoded a given bitstream. The vc2-bitstream-compare command can be used to compare a decoder's output with that of the bitstream validator in a manner similar to the *VC-2 decoder conformance testing procedure* (page 22):

```
$ vc2-picture-compare expected/ actual/
Comparing expected/picture_0.raw and actual/picture_0.raw
  Pictures are identical
Comparing expected/picture_1.raw and actual/picture_1.raw
  Pictures are identical
Comparing expected/picture_2.raw and actual/picture_2.raw
  Pictures are identical
Summary: 3 identical, 0 different
```

A conformant decoder must produce identical results to the *vc2-bitstream-validator* (page 45) command.

2.8 Generating static wavelet filter analyses

Note: This section does not apply to most codecs. You can skip this section if your codec only uses combinations of wavelet transforms and depths for which a default quantisation matrix is defined in annex (D) (even if you don't use the default quantisation matrix).

The generation of certain test cases requires a mathematical analysis of the wavelet filter used by the codec under test. The VC-2 conformance software is supplied with analyses for all wavelet filter configurations for which a default quantisation matrix is defined in annex (D). If your codec uses a wavelet and depth combination for which no default quantisation matrix is defined, a suitable analysis must be produced to enable tests to be generated for this codec. The steps below walk through the process of using the `vc2-static-filter-analysis` tool to produce the required analyses.

2.8.1 Step 1: Generating static analyses

Static analyses must be created for every wavelet transform used by your codec (including those with a default quantisation matrix).

The `vc2-static-filter-analysis` command (provided by the `vc2_bit_widths` ([vc2_bit_widths], page 33) package) is used to generate a mathematical analysis of arbitrary VC-2 filter configurations.

For example, to analyse a filter which:

- Uses the Haar (with shift) filter (wavelet index 4) vertically
- Uses the Le Gall (5, 3) filter (wavelet index 1) horizontally
- With a 2 level 2D transform depth
- And 1 level horizontal-only transform depth

The following command is used:

```
$ vc2-static-filter-analysis \
    --wavelet-index 4 \
    --wavelet-index-ho 1 \
    --dwt-depth 2 \
    --dwt-depth-ho 1 \
    --output filter_analysis.json
```

This command will compute the static analysis and write them to `filter_analysis.json`.

For modest transform depths, this process should take a few seconds. For larger transforms, this command can take several minutes or even hours to execute so the `--verbose` option can be added to track progress. For extremely large filters, see the `vc2_bit_widths` ([vc2_bit_widths], page 33) package's user guide for further guidance.

2.8.2 Step 2: Bundling static analyses

Once static analyses have been produced for all required wavelets, these must be combined into an analysis bundle file as follows:

```
$ vc2-bundle create bundle.zip \
      --static-filter-analysis path/to/analyses/*.json
```

2.8.3 Step 3: Run test case generation

Finally, to use the generated filter analyses during test case generation, the `VC2_BIT_WIDTHS_BUNDLE` environment variale must be set to the location of the bundle file generated in step 2. For example:

```
$ export VC2_BIT_WIDTHS_BUNDLE="path/to/bundle.zip"
$ vc2-test-case-generator path/to/codec_features.csv
```

2.9 Codec debugging suggestions

The following sections provide suggestions for tackling some of the issues which can be found in encoders and decoders.

2.9.1 Encoder bitstream conformance issues

When validating an encoded bitstream with the *vc2-bitstream-validator* (page 45) tool, conformance issues often relate to syntactic features of the bitstream format itself. Consequently, it can be useful to inspect the bitstream in a human-readable form using the *vc2-bitstream-viewer* (page 51) tool.

Every conformance error message includes a 'Suggested bitstream viewer commands' section which provides a sample invocation of the *vc2-bitstream-viewer* (page 51) command intended to display just the problematic region in the bitstream. See the *vc2-bitstream-viewer documentation* (page 51) for a more detailed guide to the bitstream viewer.

In addition, error messages also provide a traceback of the pseudocode functions in the VC-2 specification which were running at the time of the error. It might be helpful to compare the logic in the relevant pseudocode functions side-by-side with the corresponding logic in an encoder when tracing an issue.

2.9.2 Decoder bitstream syntax issues

Many decoder test cases are designed to exercise specific syntactic features of the VC-2 bitstream. If a decoder is encountering difficulties with a test case, it might be helpful to use the *vc2-bitstream-viewer* (page 51) tool to display a human-readable version of the test case bitstream.

The `--hide slice` argument can be used to suppress the printing of picture slice data when this is not relevant. This dramatically reduces the quantity of output produced by the tool.

The `--offset <bit offset>` argument can be used to display only the part of a bitstream within a few bits of the specified bit (not byte) offset into the file. This might be useful when you know where in a decoder had read up to when it failed.

The `--show <pseudocode function name>` argument can be used to filter the bitstream viewer output. This argument takes the name of a pseudocode function in the VC-2 specification and shows only parts of the bitstream read by that function.

The `--show-internal-state` argument causes the bitstream viewer to print (a subset of) the contents of the `state` variable used by the VC-2 pseudocode. This might be useful if a decoder appears to be correctly deserialising a stream but is interpreting its meaning differently.

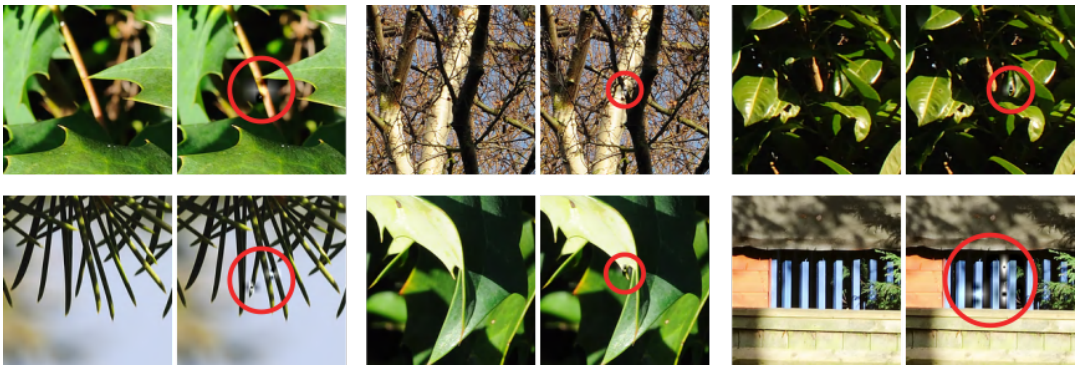
See the *vc2-bitstream-viewer documentation* (page 51) for a complete guide to using this tool.

2.9.3 Signal range test cases

The *decoder signal_range* (page 28) and *encoder signal_range* (page 34) test cases are designed to result in large, near worst case, numerical values within a codec's wavelet transform. These tests are intended to determine if a codec has used integers of sufficient size for their wavelet transform implementation.

If these test cases do not pass, it is likely that the codec being tested has used integers too small for the wavelet transform and picture bit depths in use. Try increasing the size of the integers used to hold intermediate wavelet transform values.

Typical symptoms of codecs with insufficiently large integers are the infrequent production of easily visible artefacts such as illustrated below:



Warning: It is extremely easy to under-estimate the size of integers required within a codec, and within decoders especially. In particular, the peak signal levels produced within a codec can differ by multiple orders of magnitude between different pictures. Even large suites of test material (whether pictures or noise) are unlikely to reach worst-case signal levels within every wavelet transform stage. Because peak levels vary so much between pictures, the possibility of encountering occasional near-worst case signals in production is actually relatively high.

The test signals used in these test cases are designed to achieve near worst case signal levels in every intermediate value of VC-2's wavelet transform but true worst-case signal levels might be larger still. This limitation arises from a number of factors:

- The full VC-2 encoder-decoder signal chain is a non-linear system and so it is not possible to directly compute worst-case inputs or signal levels. Instead, the test patterns in these test cases are derived from heuristics and so cannot guarantee worst-case behaviour.
- The decoder test cases make some assumptions about encoder behaviours, since VC-2 does not specify an encoder design. Consequently it is possible that an encoder might produce outputs which yield higher worst-case signal levels.
- These test signals assume the wavelet transform is carried out as outlined in the VC-2 pseudocode. Alternative implementations might have different worst-case characteristics.

For more in-depth information on codec integer bit width selection for VC-2 implementations, see the `vc2_bit_widths` ([vc2_bit_widths], page 33) package documentation.

2.10 Conformance test limtations

The conformance testing procedures outlined in the previous sections are unable to guarantee the overall conformance of an implementation – only their conformance with respect to particular bitstreams or pictures. Furthermore, not every possible combination of features are verified, though care has been taken to include any combinations likely to uncover issues.

There are also a small number of VC-2 features which are only tested to a limited extent by the generated test cases. Some of these are listed below.

Longer sequences The conformance test cases all consist of relatively short test sequences. Since VC-2 is an intra-frame only codec, codecs should only maintain limited state between pictures and so be insensitive to sequence length and so this limitation of the test suite should be immaterial.

Mixing fragmented and non-fragmented picture data units The VC-2 standard does not prohibit the use of fragmented and non-fragmented pictures within the same sequence. No test cases are generated to test this esoteric case, however, since this is unlikely to be used in practice.

Quantisation in lossless formats Lossless formats can use quantization where transform coefficients happen to be multiples of the quantisation factor. Because quantisation can, in the general case, result in larger intermediate signals within a decoder, it is not appropriate to use lossily encoded test signals to test a lossless decoder's support for quantisation. As a result, a special test case is provided for lossless decoders which uses quantisation but ensures safe signal ranges.

Auxiliary data-units Auxiliary data units (10.4.4) are not included in any tests. This is because the contents of such units are not defined by the VC-2 standard and so in the event that a particular codec supported some form of auxiliary data stream, its format would not be known to the conformance software. Likewise, the contents of any auxiliary data units produced by an encoder under test will be ignored by this software.

Variable slice sizes In all non-lossless mode test cases, high quality picture slices are sized to (approximately) the same number of bytes each. Although in principle other modes of operation are possible (e.g. buffer-based byte allocation), these are not the intended mode of operation for VC-2 and would be too numerous to comprehensively test.

Changing wavelet transform mid sequence All test cases use the same wavelet transform and slice parameters for every picture in the sequence. Though the VC-2 specification permits these parameters to change mid-sequence, this is not the intended mode of operation for VC-2. Because of this, and the number of test cases which would be required, this scenario is not tested.

Degenerate formats The test case generator cannot generate test cases for all degenerate video formats. For example, picture component bit depths of greater than 32 bits or absurd transform depths are not supported.

Differences from specification The SMPTE ST 2042-1:2017 specification contains a small number of minor errors. In these cases, this software assumes the intention of the specification.

CHAPTER

THREE

SOFTWARE TOOLS REFERENCE

This chapter contains reference documentation and basic usage examples for each of the command line tools included with the VC-2 conformance software. In summary, these are:

vc2-test-case-generator **(page 43)** Generates test cases (i.e. pictures and bitstreams) for testing the conformance of VC-2 encoders and decoders.

vc2-bitstream-validator **(page 45)** Validates the conformance of a VC-2 bitstream and produces a reference decoding.

vc2-picture-compare **(page 46)** Compares pairs of pictures.

vc2-picture-explain **(page 49)** Produces informative explanations of the video formats written and accepted by the VC-2 conformance software. (See *Video file format* (page 8)).

vc2-bitstream-viewer **(page 51)** Produces a human-readable low-level description of VC-2 bitstreams.

3.1 `vc2-test-case-generator`

This application generates test cases for VC-2 encoders and decoders given a set of codec features to target.

- For a guide to the codec features CSV file format see *Defining codec features* (page 15)
- For a guide to encoder test cases see *Encoder test cases* (page 33)
- For a guide to decoder test cases see *Decoder Test Cases* (page 23)

3.1.1 Usage

A codec features CSV file should be created (as described in *Generating test cases* (page 15)). This can then be processed by `vc2-test-case-generator` to generate test cases for the codecs supporting the specific codec features described.

Test case generation is performed as follows:

```
$ vc2-test-case-generator path/to/codec_features.csv
```

The test cases are written to a `test_cases` in the current working directory. The `--output` argument can be given to choose an alternative location. By default, if any existing test cases are present in the output directory, the test case generator will not run. The `--force` argument can be added to run the test case generator anyway.

If only a subset of test cases are required, the following arguments can also be used:

- `--encoder-only`: Only generate test cases for encoders
- `--decoder-only`: Only generate test cases for decoders
- `--codec <regexp>`: Only generate test cases for codec feature sets whose names match a supplied pattern.

3.1.2 Parallel execution

The test case generator can take many hours to run. To speed up execution, test cases can be generated in parallel on multi-core machines. To do this the `--parallel` argument is added to `vc2-test-case-generator`. Rather than performing test case generation, this command outputs a series of newline-separated commands which can be executed in parallel to perform test case generation. These can then be executed, for example using GNU Parallel[18]:

```
$ # Write test case generation commands to 'commands.txt'
$ vc2-test-case-generator path/to/codec_features.csv --parallel > commands.txt

$ # Run test case generation in parallel using GNU Parallel
$ parallel -a commands.txt
```

3.1.3 Static wavelet filter analyses

Some test cases require a precomputed mathematical analysis of the wavelet transform used.

The VC-2 conformance software includes suitable analyses for all wavelet transforms for which a default quantisation matrix is defined in the VC-2 specification. To support additional wavelet transform combinations and depths, additional analyses must be generated as described in *Generating static wavelet filter analyses* (page 38).

If the `VC2_BIT_WIDTHS_BUNDLE` environment variable is defined, the test case generator will attempt to read wavelet filter analyses from the file it specifies. This must be a filter analysis bundle file created by `vc2-bundle`.

3.1.4 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-test-case-generator [-h] [--version] [--verbose] [--parallel]
                               [--output OUTPUT] [--force]
                               [--encoder-only | --decoder-only]
                               [--codecs REGEX]
                               codec_configurations

Generate test inputs for VC-2 encoder and decoder implementations.

positional arguments:
  codec_configurations  CSV file containing the set of codec configurations to
                        generate test cases for.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --verbose, -v         Show additional status information during execution.
  --parallel, -p        If given, don't generate test cases but instead
                        produce a series of commands on stdout which can be
                        executed in parallel to generate the test cases.
  --output OUTPUT, -o OUTPUT
                        Directory name to write test cases to. Will be created
                        if it does not exist. Defaults to './test_cases/'.
  --force, -f           Force the test case generator to run even if the
                        output directory is not empty.
  --encoder-only, -e    If set, only generate test cases for VC-2 encoders.
  --decoder-only, -d    If set, only generate test cases for VC-2 decoders.
  --codecs REGEX, -c REGEX
                        If given, a regular expression which selects which
                        codec configurations to generate test cases for. If
```

(continues on next page)

[18] https://www.gnu.org/software/parallel/

(continued from previous page)

```
                        not given, test cases will be generated for all codec
                        configurations.
```

3.2 vc2-bitstream-validator

A command-line utility for validating VC-2 bitstreams' conformance with the VC-2 specification and providing a reference decoding of the pictures within.

3.2.1 Usage

This command should be passed a filename containing a candidate VC-2 bitstream. For example, given a valid stream:

```
$ vc2-bitstream-validator path/to/bitstream.vc2 --output decoded_picture_%d.raw
No errors found in bitstream. Verify decoded pictures to confirm conformance.
```

Here the `--output` argument specifies the printf-style template for the decoded picture filenames. The decoded pictures are written as raw files (see *Video file format* (page 8)).

If a conformance error is detected, a detailed explanation of the problem is displayed:

```
$ vc2-bitstream-validator invalid.vc2
Conformance error at bit offset 104
==================================

Non-zero previous_parse_offset, 5789784, in the parse info at the start of
a sequence (10.5.1).


Details
-------

Was this parse info block copied from another stream without the
previous_parse_offset being updated?

Does this parse info block incorrectly include an offset into an adjacent
sequence?


Suggested bitstream viewer commands
-----------------------------------

To view the offending part of the bitstream:

    vc2-bitstream-viewer invalid.vc2 --offset 104


Pseudocode traceback
--------------------

Most recent call last:

* parse_stream (10.3)
  * parse_sequence (10.4.1)
    * parse_info (10.5.1)

vc2-bitstream-validator: error: non-conformant bitstream (see above)
```

Errors include an explanation of the conformance problem (along with references to the VC-2 standards documents) along with possible causes of the error. Additionally, a sample invocation of *vc2-bitstream-viewer* (page 51) is given which can be used to display the contents of the bitstream at the offending position. Finally, a stack trace for the VC-2 pseudocode functions involved in parsing the stream at the point of failure is also printed.

3.2.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-bitstream-validator [-h] [--version] [--no-status] [--verbose]
                               [--output OUTPUT]
                               bitstream

Validate a bitstream's conformance with the VC-2 specifications.

positional arguments:
  bitstream             The filename of the bitstream to validate.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --no-status, --quiet, -q
                        Do not display a status line on stderr while
                        validating the bitstream.
  --verbose, -v         Show full Python stack-traces on failure.
  --output OUTPUT, -o OUTPUT
                        The filename pattern for decoded picture data and
                        metadata. The supplied pattern should a 'printf' style
                        template with (e.g.) '%d' where an index will be
                        substituted. The first decoded picture will be
                        assigned index '0', the second '1' and so on -- i.e.
                        these indices are unrelated to the picture number. The
                        file extension supplied will be stripped and two files
                        will be written for each decoded picture: a '.raw'
                        planar image file and a '.json' JSON metadata file.
                        (Default: picture_%d.raw).
```

3.3 vc2-picture-compare

A command-line utility which compares pairs of raw pictures (see *Video file format* (page 8)), or pairs of directories containing a series of raw pictures.

3.3.1 Usage

Given a pair of images in raw format (see *Video file format* (page 8)), these images can be compared as follows:

```
$ vc2-picture-compare image_a.raw image_b.raw
Pictures are different:
  Y: Different: PSNR = 55.6 dB, 1426363 pixels (68.8%) differ
  C1: Different: PSNR = 57.7 dB, 662607 pixels (63.9%) differ
  C2: Different: PSNR = 56.8 dB, 703531 pixels (67.9%) differ
```

Alternatively a pair of directories containing images whose filenames end with a number (and the extensions `.raw` and `.json`). For example:

```
$ vc2-picture-compare ./directory_a/ ./directory_b/
Comparing ./directory_a/picture_0.raw and ./directory_b/picture_0.raw:
  Pictures are identical
Comparing ./directory_a/picture_1.raw and ./directory_b/picture_1.raw:
  Pictures are different:
    Y: Different: PSNR = 55.6 dB, 1426363 pixels (68.8%) differ
    C1: Different: PSNR = 57.7 dB, 662607 pixels (63.9%) differ
    C2: Different: PSNR = 56.8 dB, 703531 pixels (67.9%) differ
Comparing ./directory_a/picture_2.raw and ./directory_b/picture_2.raw:
  Pictures are identical
Summary: 2 identical, 1 different
```

Differences in the encoded values are reported separately for each picture component.

Peak Signal to Noise Ratio (PSNR) figures give the PSNR of the raw integer signal values (and not, for example, linear light levels).

Note: PSNR is computed as:

$$\text{MeanSquareError} = \sum_{i,j}(E_{i,j} - A_{i,j})^2$$

$$\text{MaxE} = \max_{i,j}(E_{i,j})$$

$$\text{PSNR} = 20\log_{10}(\text{MaxE}) - 10\log_{10}(\text{MeanSquareError})$$

Where $E_{i,j}$ is pixel (i, j) of the raw integer signal value of the test image and $A_{i,j}$ is pixel (i, j) of the raw integer values of the decoded image.

The number of pixels which are not bit-for-bit identical is also reported.

The tool also compares the metadata of the raw images and will flag up differences here too, for example:

```
$ vc2-picture-compare picture_hd.raw picture_cif.raw
Picture numbers are different:
Video parameters are different:
  - frame_width: 1920
  + frame_width: 352
  - frame_height: 1080
  + frame_height: 288
  - color_diff_format_index: color_4_2_2 (1)
  + color_diff_format_index: color_4_2_0 (2)
    source_sampling: progressive (0)
    top_field_first: True
  - frame_rate_numer: 50
  + frame_rate_numer: 25
  - frame_rate_denom: 1
  + frame_rate_denom: 2
  - pixel_aspect_ratio_numer: 1
  + pixel_aspect_ratio_numer: 12
  - pixel_aspect_ratio_denom: 1
  + pixel_aspect_ratio_denom: 11
  - clean_width: 1920
  + clean_width: 352
  - clean_height: 1080
  + clean_height: 288
    left_offset: 0
    top_offset: 0
  - luma_offset: 64
  + luma_offset: 0
  - luma_excursion: 876
  + luma_excursion: 255
```

(continues on next page)

(continued from previous page)

```
  - color_diff_offset: 512
  + color_diff_offset: 128
  - color_diff_excursion: 896
  + color_diff_excursion: 255
  - color_primaries_index: hdtv (0)
  + color_primaries_index: sdtv_625 (2)
  - color_matrix_index: hdtv (0)
  + color_matrix_index: sdtv (1)
    transfer_function_index: tv_gamma (0)
```

When picture metadata differs, the pixel values will not be compared since the difference in metadata typically means a difference in format making the pictures incomparable.

3.3.2 Generating difference masks

The `vc2-picture-compare` tool can optionally output a simple difference mask image using `--difference-mask`/`-D`. The generated image contains white pixels wherever the inputs differed and black pixels wherever they were identical. The generated difference mask is output as a raw file of the same format as the two input files. This mode is only supported when two individual files are provided, not two directories.

For example:

```
$ vc2-picture-compare \
      image_a.raw \
      image_b.raw \
      --difference-mask difference_mask.raw
Pictures are different:
  Y: Different: PSNR = 55.6 dB, 1426363 pixels (68.8%) differ
  C1: Different: PSNR = 57.7 dB, 662607 pixels (63.9%) differ
  C2: Different: PSNR = 56.8 dB, 703531 pixels (67.9%) differ
```

3.3.3 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-picture-compare [-h] [--version] [--difference-mask FILENAME]
                           filename_a filename_b

Compare a pair of pictures in the raw format used by the VC-2 conformance
software.

positional arguments:
  filename_a            The filename of a .raw or .json raw picture, or a
                        directory containing a series of .raw and .json files
                        whose names end in a number.
  filename_b            The filename of a .raw or .json raw picture, or a
                        directory containing a series of .raw and .json files
                        whose names end in a number.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit

difference image options:
  --difference-mask FILENAME, -D FILENAME
                        Output a difference mask image to the specified file.
                        This mask will contain white pixels wherever the input
                        images differ and black pixels where they match. Only
                        available when comparing files (not directories).
```

3.4 vc2-picture-explain

A command-line utility which provides informative descriptions the raw video format (see *Video file format* (page 8)) used by the VC-2 conformance software. As well as providing an explanation of the format, where possible sample invocations of FFmpeg[19] and ImageMagick[20] are provided for viewing the raw files directly.

3.4.1 Example usage

An example invocation is shown below:

```
$ vc2-picture-explain path/to/raw/picture_0.json
Normative description
=====================

Picture coding mode: pictures_are_fields (1)

Video parameters:

* frame_width: 720
* frame_height: 576
* color_diff_format_index: color_4_2_2 (1)
* source_sampling: interlaced (1)
* top_field_first: True
* frame_rate_numer: 25
* frame_rate_denom: 1
* pixel_aspect_ratio_numer: 12
* pixel_aspect_ratio_denom: 11
* clean_width: 704
* clean_height: 576
* left_offset: 8
* top_offset: 0
* luma_offset: 16
* luma_excursion: 219
* color_diff_offset: 128
* color_diff_excursion: 224
* color_primaries_index: sdtv_625 (2)
* color_matrix_index: sdtv (1)
* transfer_function_index: tv_gamma (0)

Explanation (informative)
=========================

Each raw picture contains a single field. The top field comes first.

Pictures contain three planar components: Y, Cb and Cr, in that order, which are
4:2:2 subsampled.

The Y component consists of 720x288 8 bit values. Expressible values run from 0
(video level -0.07) to 255 (video level 1.09).

The Cb and Cr components consist of 360x288 8 bit values. Expressible values run
from 0 (video level -0.57) to 255 (video level 0.57).

The color model uses the 'sdtv_625' primaries (ITU-R BT.601), the 'sdtv' color
matrix (ITU-R BT.601) and the 'tv_gamma' transfer function (ITU-R BT.2020).

The pixel aspect ratio is 12:11 (not to be confused with the frame aspect
ratio).
```

(continues on next page)

[19] https://ffmpeg.org/
[20] https://imagemagick.org/

(continued from previous page)

```
Example FFMPEG command (informative)
====================================

The following command can be used to play back this video format using FFMPEG:

    $ ffplay \
        -f image2 \
        -video_size 720x288 \
        -framerate 50 \
        -pixel_format yuv422p \
        -i path/to/raw/picture_%d.raw \
        -vf weave=t,yadif,scale='trunc((iw*12)/11):ih'

Where:

* `-f image2` = Read pictures from individual files
* `-video_size 720x288` = Picture size (not frame size).
* `-framerate 50` = Picture rate (not frame rate)
* `-pixel_format` = Specifies raw picture encoding.
* `yuv` = Y C1 C2 color.
* `422` = 4:2:2 color difference subsampling.
* `p` = Planar format.
* `-i path/to/raw/picture_%d.raw` = Input raw picture filename pattern
* `-vf` = define a pipeline of video filtering operations
* `weave=t` = interleave pairs of pictures, top field first
* `yadif` = (optional) apply a deinterlacing filter for display purposes
* `scale='trunc((iw*12)/11):ih'` = rescale non-square pixels for display with
  square pixels

This command is provided as a minimal example for basic playback of this raw
video format.  While it attempts to ensure correct frame rate, pixel aspect
ratio, interlacing mode and basic pixel format, color model options are omitted
due to inconsistent handling by FFMPEG.

Example ImageMagick command (informative)
=========================================

The following command can be used to convert a single raw picture into PNG
format for viewing in a conventional image viewer:

    $ convert \
        -size 720x288 \
        -depth 8 \
        -sampling-factor 4:2:2 \
        -interlace plane \
        -colorspace sRGB \
        yuv:path/to/raw/picture_0.raw \
        -resize 109.0909090909091%,100% \
        png24:path/to/raw/picture_0.raw

Where:

* `-size 720x288` = Picture size.
* `-depth 8` = 8 bit values.
* `-sampling-factor 4:2:2` = 4:2:2 color difference subsampling.
* `-interlace plane` = Planar format (not related to video interlace mode).
* `-colorspace sRGB` = Display as if using sRGB color.
* `yuv:` = Input is Y C1 C2 picture.
* `path/to/raw/picture_0.raw` = Input filename.
* `-resize 109.0909090909091%,100%` = rescale non-square pixels for display with
```

(continues on next page)

(continued from previous page)

```
  square pixels
* `png24:path/to/raw/picture_0.png` = Save as 24-bit PNG (e.g. 8 bit channels)

This command is provided as a minimal example for basic viewing of pictures.
Interlacing and correct color model conversion are not implemented.
```

3.4.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-picture-explain [-h] [--version] filename

Informatively explain the video format used a raw video file generated by the
VC-2 conformance software.

positional arguments:
  filename    The filename of a .raw or .json raw video file.

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
```

3.5 vc2-bitstream-viewer

A command-line utility for displaying VC-2 bitstreams in a human readable form.

The following tutorials give a higher-level introduction to this tool. Complete usage information can be obtained by running `vc2-bitstream-viewer --help`.

Warning: This program prioritises correctness over performance. Internally the pseudocode from the VC-2 specification is used to drive the bitstream parser. As a consequence this program is relatively slow, e.g. taking many seconds per HD frame.

3.5.1 Basic usage

In its simplest form, this command can be used to create a complete textual representation of a VC-2 bitstream:

```
$ vc2-bitstream-viewer path/to/bitstream.vc2
                                                    +- data_units:
                                                    | +- 0:
                                                    | | +- parse_info:
000000000000:                                       | | | +- padding: 0b
000000000000: 01000010010000100100001101000100     | | | +- parse_info_prefix:␣
↪Correct (0x42424344)
000000000032: 00010000                              | | | +- parse_code: end_of_
↪sequence (0x10)
000000000040: 00000000000000000000000000000000     | | | +- next_parse_offset: 0
000000000072: 00000000000000000000000000000000     | | | +- previous_parse_offset: 0
```

In the printed output, each value read from the bitstream is shown on its own line:

- The number at the start of the line is the offset (in bits) from the start of the file that that value begins.

- The next value is the value in the bitstream expressed as a binary number. If values are being read from the bounded block in the bitstream, a (*) indicates that any remaining bits in the value shown were read from beyond the end of the block (and read as '1').
- The final part of the line indicates the name and decoded value of the read data. Where possible, the names used match the names of variables in the VC-2 pseudocode.

By default, every bit and every field in the bitstream will be visible, including padding bits.

3.5.2 Displaying an area of interest

It is possible to restrict the output to only parts of the bitstream surrounding a particular offset. For example, the `--offset`/`-o` argument can be used to restrict display to only the parts of the bitstream surrounding a particular bit offset:

```
$ vc2-bitstream-viewer bitstream.vc2 --offset 40000
```

See the `--context`/`-C`, `--context-after`/`-A` and `--context-before`/`-B` arguments to control the number of bits before and after the supplied offset to be shown. Alternatively, you can also specify an explicit range to display using the `--from-offset`/`-f` and `--to-offset`/`-t`.

Note: Though values before the specified bit offset will not be displayed, they must still be read and parsed by the bitstream viewer in order to correctly parse what comes later in the bitstream.

3.5.3 Filtering displayed values

It is possible to filter the displayed values according to the VC-2 pseudocode function which read them. A common case might be to show only the parse_info headers from a stream:

```
$ vc2-bitstream-viewer bitstream.vc2 --show parse_info
```

Alternatively, you might wish to show everything in a bitstream except for the transform coefficients (slice data):

```
$ vc2-bitstream-viewer bitstream.vc2 --hide slice
```

For this particular action, the convenience alias `-S` for `--hide slice` is also provided.

3.5.4 Showing VC-2 decoder state

As well as displaying a textual representation of the VC-2 bitstream, this tool can also display (a subset of) the internal state of a VC-2 decoder based on the pseudocode in the specification at different points during bitstream processing. Use the `--show-internal-state` option to enable this. This can be useful when debugging the encoding of transform data whose organisation depends on various computed values earlier in the bitstream.

3.5.5 Malformed bitstream handling

The bitstream parser is as tolerant of malformed bitstreams as is practical and will accept, and blindly tolerate values which are out-of-spec so long as they do not lead to undefined behaviour. In the event that the parser encounters an anomaly that prevents it preceding further, an error will be shown and parsing will terminate.

By default, error messages show only limited information about the failure. Adding the `--verbose`/`-v` option will cause additional details (such as the next few bits in the bitstream and details of what was being parsed at the time.

3.5.6 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-bitstream-viewer [-h] [--version] [--no-status] [--verbose]
                            [--show-internal-state]
                            [--ignore-parse-info-prefix]
                            [--num-trailing-bits NUM_TRAILING_BITS]
                            [--from-offset BIT_OFFSET]
                            [--to-offset BIT_OFFSET] [--offset BIT_OFFSET]
                            [--after-context NUM_BITS]
                            [--before-context NUM_BITS] [--context NUM_BITS]
                            [--show FUNCTION] [--hide FUNCTION] [--hide-slice]
                            bitstream

Display VC-2 bitstreams in a human-readable form.

positional arguments:
  bitstream             The filename of the bitstream to read.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --no-status, --quiet, -q
                        Do not display a status line on stderr while reading
                        the bitstream.
  --verbose, -v         Increase the verbosity of error messages. Used once:
                        also show bitstream offset, bitstream target value and
                        next --num-trailing-bits bits in the bitstream at the
                        time of the error. Used twice: also show the Python
                        stack trace for the error.
  --show-internal-state, -i
                        Print the internal state variable used by the VC-2
                        pseudo code after parsing each data unit. Parts of the
                        pseudocode state not directly related to bitstream
                        processing might or might not be included in this
                        printout.
  --ignore-parse-info-prefix, -p
                        By default, parsing is halted if an invalid parse_info
                        prefix is encountered. Giving this option suppresses
                        this check and allows parsing to continue.
  --num-trailing-bits NUM_TRAILING_BITS, -b NUM_TRAILING_BITS
                        When --verbose is used, this argument defines the
                        number of bits to show from the bitstream after the
                        point the error occurred. (Default: 128).

range options:
  --from-offset BIT_OFFSET, -f BIT_OFFSET
                        Don't display bitstream values until the specified bit
                        offset. If negative, gives an offset from the end of
                        the file. Default: 0.
  --to-offset BIT_OFFSET, -t BIT_OFFSET
                        Stop reading the bitstream at the specified bit
                        offset. If prefixed with '+', the offset will be
                        relative to the offset given to '--from-offset'. If
                        negative, gives an offset from the end of the file.
                        Default: the end of the file.
  --offset BIT_OFFSET, -o BIT_OFFSET
                        Shows the parts of the bitstream surrounding the
                        specified bit offset. (An alternative to manually
                        setting '--from-offset' and '--to-offset' to values
                        either-side of the chosen offset). If none of '--
```

(continues on next page)

(continued from previous page)

```
                        after-context', '--before-context' or '--context' are
                        given, 128 bits of context either side of this offset
                        will be shown.
  --after-context NUM_BITS, -A NUM_BITS
                        Sets the number of bits after '--around-offset' to be
                        shown.
  --before-context NUM_BITS, -B NUM_BITS
                        Sets the number of bits before '--around-offset' to be
                        shown.
  --context NUM_BITS, -C NUM_BITS
                        Sets the number of bits before and after '--around-
                        offset' to be shown.

filtering options:
  --show FUNCTION, -s FUNCTION
                        Display only parts of the bitstream which are
                        processed by the specified pseudo-code function as
                        described in the VC-2 specification. By default all
                        parts of the bitstream are displayed. Can be used
                        multiple times to show different parts of the
                        bitstream. Supported function names: auxiliary_data,
                        clean_area, color_diff_sampling_format, color_matrix,
                        color_primaries, color_spec,
                        extended_transform_parameters, fragment_data,
                        fragment_header, fragment_parse, frame_rate,
                        frame_size, hq_slice, ld_slice, padding, parse_info,
                        parse_parameters, parse_sequence, parse_stream,
                        picture_header, picture_parse, pixel_aspect_ratio,
                        quant_matrix, scan_format, sequence_header,
                        signal_range, slice, slice_parameters,
                        source_parameters, transfer_function, transform_data,
                        transform_parameters, wavelet_transform.
  --hide FUNCTION, -H FUNCTION
                        Omit parts of the bitstream which are processed by the
                        specified pseudo-code function as described in the
                        VC-2 specification. Accepts the same values as --show.
  --hide-slice, -S      Alias for '--hide slice'. Suppresses the printing of
                        all transform coefficients, greatly reducing the
                        quantity of output.
```

Part II

Maintainer’s manual

CHAPTER

FOUR

CONFORMANCE SOFTWARE DEVELOPMENT GUIDE

This part of the documentation provides a guide to the VC-2 conformance software internals. This documentation is aimed at maintainers of the VC-2 conformance software. Users of the VC-2 conformance software need not read this, or later sections.

The maintainer's manual aims to provide a 'big picture' view of the software architecture along with more detailed introductions to each of the main concepts and components within. It does not, however, provide an exhaustive reference including every minor component in the codebase – refer to the comments and docstrings within the code in these instances.

This introductory chapter provides instructions on setting up a suitable installation of this software for development purposes along with an overview of the structure of the codebase. Subsequent chapters will provide in-depth reference documentation for each of the main parts of this software.

4.1 Development setup

The following instructions outline the process for setting up a development environment for the VC-2 conformance software and the procedures for running tests and generating documentation.

4.1.1 Checking out repositories

The VC-2 conformance software is split across in the following Git[21] repositories, each containing a Python package of the same name:

https://github.com/bbc/vc2_conformance (*vc2_conformance* **(page 59))** The main conformance software repository on which this documentation focuses.

https://github.com/bbc/vc2_conformance_data (**vc2_conformance_data** (**[vc2_conformance_data], page 1**)) Data files (e.g. test pictures) used in the conformance testing process.

https://github.com/bbc/vc2_data_tables (**vc2_data_tables** (**[vc2_data_tables], page 3**)) Data tables and constant definitions from the VC-2 standard.

https://github.com/bbc/vc2_bit_widths (**vc2_bit_widths** (**[vc2_bit_widths], page 33**)) Mathematical routines for computing near worst case signals for VC-2 codecs.

The above repositories should be cloned into local directories, e.g. using:

```
$ git clone git@github.com:bbc/vc2_conformance.git
$ git clone git@github.com:bbc/vc2_conformance_data.git
$ git clone git@github.com:bbc/vc2_data_tables.git
$ git clone git@github.com:bbc/vc2_bit_widths.git
```

Pre-commit hooks[22] are used to enforce certain code standards in these repositories. These should be installed as follows:

[21] https://git-scm.com/
[22] https://pre-commit.com/

```
$ # For each cloned repository...
$ cd path/to/repo/
$ pre-commit install
```

4.1.2 Virtual environment

It is strongly recommended that development is carried out in a Python virtual environment[23] (see warning below). This can be setup using:

```
$ python -m virtualenv --python <PYTHON INTERPRETER> venv
```

This will create a virtual environment in the directory `venv` which uses the python interpreter `<PYTHON INTERPRETER>`, which should generally be one of `python2` or `python3`.

Once created, the virtual environment must be activated in any shell you use using:

```
$ source venv/bin/activate
```

Note: Python virtual environment provides an isolated environment in which packages can be installed without impacting on the rest of the system. Once activated, the `python` and `pip` commands will use the python version and packages setup within the virtual environment.

Warning: When working outside a virtual environment, Python packages included by some operating systems (e.g. Ubuntu) can be very out of date leading to problems during development. This is the result of certain development dependencies not correctly specifiying their version requirements and is outside of our control. By using a virtual environment, up-to-date versions of all dependencies will be installed which avoids these problems.

Note: The VC-2 conformance software itself *does* correctly specify its dependencies so these problems only apply during development and should not affect end users.

4.1.3 Development installation

A development installation of the conformance software can be performed directly from each of the cloned repositories as follows:

```
$ # Each repo should be installed as follows, in the following order:
$ # * vc2_data_tables
$ # * vc2_bit_widths
$ # * vc2_conformance_data
$ # * vc2_conformance
$ cd path/to/repo/

$ # Install in editable/development mode (so edits take effect immediately)
$ pip install -e .

$ # Install test suite dependencies
$ pip install -r requirements-test.txt

$ # Install documentation building dependencies (not present for all
$ # repositories)
$ pip install -r requirements-docs.txt
```

[23] https://virtualenv.pypa.io/en/stable/

After installation, the various `vc2-*` commands will be made available in your `$PATH` and the various `vc2_*` Python modules in your `$PYTHONPATH`. These will point directly to the cloned source code and so changes will take effect immediately.

4.1.4 Running tests

Test routines relating to the code in each repository can be found in the `tests/` directory of each repository. The test suites are built on pytest[24] and, once a development install has been completed, can be executed as follows:

```
$ py.test path/to/vc2_data_tables/tests/
$ py.test path/to/vc2_bit_widths/tests/
$ py.test path/to/vc2_conformance_data/tests/
$ py.test path/to/vc2_conformance/tests/
```

4.1.5 Building documentation

HTML documentation (including the documentation you're reading now) is built as follows (after a development install has been performed):

```
$ make -C path/to/vc2_data_tables/docs html
$ make -C path/to/vc2_bit_widths/docs html
$ make -C path/to/vc2_conformance_data/docs html
$ make -C path/to/vc2_conformance/docs html
```

HTML documentation will be written to the `docs/build/html/` directory (open the `index.html` file in a web browser to read it).

Alternatively, PDF documentation can be built by replacing `html` with `latexpdf` in the above commands. This will require a working installation of LaTeX[25] and Inkscape[26] to build. In addition, for cross-references between PDFs to be created correctly, the documentation must be built within the Python virtual environment where the various `vc2_*` Python packages are installed in development/editable mode.

4.2 `vc2_conformance` internals overview

The main VC-2 conformance software is contained within the *`vc2_conformance`* (page 59) module.

Note: Before going any further you should familiarize yourself with the *User's guide (for codec testers)* (page 5) which gives an introduction of the general tasks carried out by the conformance software.

Below we give a general overview of the design of the conformance software.

[24] https://docs.pytest.org/en/latest/
[25] https://www.latex-project.org/
[26] https://inkscape.org/

4.2.1 Main components

The VC-2 conformance software consists of four main components:

- A reference VC-2 decoder, including bitstream validation logic (*vc2_conformance.decoder* (page 69))
- A flexible VC-2 encoder (*vc2_conformance.encoder* (page 75))
- A VC-2 bitstream manipulation library (*vc2_conformance.bitstream* (page 81))
- A set of test case generation routines (*vc2_conformance.test_cases* (page 63))

The reference decoder and bitstream validator forms a key part of the conformance testing procedures (via the *vc2-bitstream-validator* (page 45) command). This decoder is based directly on the pseudocode published in the VC-2 standard. Refer to the *vc2_conformance.decoder* (page 69) module documentation for a full introduction to this component.

The encoder is used internally to generate decoder test cases. This encoder is flexible enough to support all of VC-2 features but otherwise simplistic in terms of runtime performance and picture quality. Further documentation on the encoder's design and functionality can be found in the *vc2_conformance.encoder* (page 75) module.

The bitstream manipulation library in the *vc2_conformance.bitstream* (page 81) module is used internally for three purposes. Firstly it is used during decoder test case generation to produce bitstreams with specific properties. Secondly it is used by the *vc2-bitstream-viewer* (page 51) command to produce human readable descriptions of bitstream contents. Finally it is used extensively by the VC-2 conformance software's own test suite to generate and check bitstreams.

Finally, the test case generation routines form the basis of the *vc2-test-case-generator* (page 43) tool which generates test pictures and bitstreams used during conformance testing procedures. These will be introduced in *vc2_conformance.test_cases: VC-2 codec test case generation* (page 63).

4.2.2 Use of VC-2 pseudocode

The VC-2 conformance software design prioritises correctness and consistency with the VC-2 specification. To achieve this, significant parts are built using the pseudocode within the VC-2 specification.

The VC-2 specification uses pseudocode to define the nominal operation of a VC-2 decoder. The pseudocode language used is sufficiently similar to Python that a translation from pseudocode into executable Python is trivial. Automated translation is also possible using the VC-2 Pseudocode Parser tool[27]. Once translated, the pseudocode may be used as the basis for correct-by-definition implementations of parts of a VC-2 codec.

The reference VC-2 decoder and bitstream validator (*vc2_conformance.decoder* (page 69)) consists of the VC-2 pseudocode (implementing the decoder behaviour) augmented with additional checks (implementing the validation logic).

The VC-2 encoder (*vc2_conformance.encoder* (page 75)), though not specified by the VC-2 standard, nevertheless makes substantial use of the VC-2 pseudocode functions. For example routines for computing slice dimensions are reused while other parts, such as the forward discrete wavelet transform are simple inversions of the decoder pseudocode. The correctness of these inversions is relatively easily verified in the test suite thanks to the invertability of VC-2's transforms.

The bitstream manipulation library (*vc2_conformance.bitstream* (page 81)) provides routines for serialising and deserialising binary bitstreams into easily manipulated Python data structures. With the exception of the Python data structure definitions, this library is based entirely on the VC-2 pseudocode, using the *serdes* (page 93) framework.

To ensure consistency with the VC-2 pseudocode, the conformance software's test suite automatically compares the software's source code with the pseudocode to verify equivalence. See the *verification* (page 147) module for details.

[27] https://github.com/bbc/vc2_pseudocode_parser

4.2.3 Performance

The major drawback of the pseudocode-driven approach used by this software is poor performance. The VC-2 pseudocode is structured with comprehensibility as its main priority and therefore often has poor algorithmic performance. Further, the use of Python – due to its similarity with the pseudocode and high level of abstraction – introduces an additional performance overhead.

Another source of slow performance is the use of infinite precision (i.e. native Python) integers whenever possible. This helps avoid the class of bugs relating to the use of insufficient integer bit width in most calculations. The dynamic range of signals passing through a VC-2 codec can grow dramatically (i.e. by many orders of magnitude) and vary significantly between superficially similar inputs. As a consequence, this class of bug is all too easy to introduce without infinite precision arithmetic.

As a result of the above factors, the conformance software may take on the order minutes to encode or decode each picture in a stream. While this would be unacceptable for a production encoder or decoder, the conformance software is intended only for use with only extremely short sequences, where correctness is the most significant factor.

The majority of test cases apply to decoders for which all necessary materials (test bitstreams and reference decodings) may be produced in an 'overnight' batch process after which the tools are not needed. The remaining (encoder) test cases generally amount to fewer than ten frames.

4.2.4 Test case generation

Since VC-2 supports a great variety of configurations, parametrised test case generators (*vc2_conformance.test_cases* (page 63)) are used rather than a 'universal' collection of bitstreams. While this means that users of the conformance software are required to configure and run the test case generator themselves, it also simplifies the testing process by ensuring only relevant test cases are produced. Furthermore, it allows certain tests to be highly tailored to a particular configuration. For example, signal range tests are targeted specifically at the specific wavelet transform configuration, bit width and quantization matrices used.

4.2.5 External packages

Some smaller, or more specialised aspects of the conformance testing software have been split into their own separate Python packages. These are:

vc2_conformance_data ([vc2_conformance_data], page 1) Larger data files (including pictures and pre-computed values) which are used to generate certain test cases.

vc2_data_tables ([vc2_data_tables], page 3) General VC-2 related constants (e.g. `PARSE_INFO_PREFIX` ([vc2_data_tables], page 3)), enumerated values (e.g. `ParseCodes` ([vc2_data_tables], page 3)) and tabular data used during coding (e.g. `PRESET_FRAME_RATES` ([vc2_data_tables], page 4)).

vc2_bit_widths ([vc2_bit_widths], page 33) Computes near worst-case inputs for VC-2 encoders and decoders which produce very large signal values. This package is used to generate test cases to verify codecs have used large enough integers in their implementations.

CHAPTER

FIVE

TEST CASE GENERATION

5.1 vc2_conformance.test_cases: VC-2 codec test case generation

The *vc2_conformance.test_cases* (page 63) module contains routines for generating test cases for VC-2 encoder and decoder implementations.

A description of each test case generator can be found in *Encoder test cases* (page 33) and *Decoder Test Cases* (page 23) in the user guide.

5.1.1 Test case generators

Test case generators are generator functions which take a *CodecFeatures* (page 65) dictionary as their only argument and produce picture sequences (for encoders) or bitstreams (for decoders). Test case generators may take one of the following forms:

- Function which returns a single test case
- Function which returns a single test case or None (indicating no test case was produced)
- Generator function which yields multiple test cases.

A test case can be one of:

- For encoder test cases, a picture sequence in the form of an *EncoderTestSequence* (page 64) object.
- For decoder test cases, a bitstream in the form of a *Stream* (page 86) object which can be serialised using *autofill_and_serialise_stream()* (page 107).
- A *TestCase* (page 63) object containing one of the above as its *value* (page 64).

Test case generators may prefer to output *TestCase* (page 63) objects when multiple test cases are produced so that each testcase can be given its own 'subcase' name. In addition, test cases may be accompanied by a freeform JSON serialisable metadata object when *TestCase* (page 63) objects are produced.

class TestCase(*value*, *subcase_name=None*, *case_name=None*, *metadata=None*)

A test case, produced by a test case generator function.

Parameters

value [*EncoderTestSequence* (page 64) or *Stream* (page 86)] A value containing the test case itself. An *EncoderTestSequence* (page 64) for encoder test cases or *Stream* (page 86) for decoder test cases.

subcase_name [str or None] An identifier for the sub-case if this test case has several sub-cases.

case_name [str or None] The name of the test case. If None, the `case_name` will be populated automatically with the name of the test case generator function which returned it.

metadata [object or None] Optional JSON-serialisable metadata associated with this test case. The meaning and formatting of of this metadata is left to the individual test case to define.

property name
The complete name of this test case. Constructed from the *case_name* (page 64) and *subcase_name* (page 64) attributes.

A string of the form `"case_name"` or `"case_name[subcase_name]"`.

property case_name

property subcase_name

property value

property metadata

All test case generator functions are run via *normalise_test_case_generator()* (page 64) which normalises the function into the form of a generator which yields *TestCase* (page 63) objects. This also populates the *case_name* (page 64) field of the generated *TestCase* (page 63) automatically with the test case generator's function name.

normalise_test_case_generator(*f*, **args*, ***kwargs*)
Call a test case generator, f, and, regardless of its native output, produces a generator of *TestCase* (page 63) objects.

If the function returns or yields *TestCase* (page 63) objects, their *TestCase.case_name* (page 64) attributes will be populated with the function name, if not already defined. If the function returns or generates other values, these will be wrapped in *TestCase* (page 63) objects automatically. If the function returns or generates None, no test case will be emitted.

5.1.2 Encoder test case generators

Encoder test case generators are located in `vc2_conformance.test_cases.encoder` must be decorated with the *vc2_conformance.test_cases.encoder_test_case_generator* (page 64) decorator, take a *CodecFeatures* (page 65) and produce *EncoderTestSequence* (page 64) objects.

encoder_test_case_generator
Decorator to use to register all encoder test case generators.

class EncoderTestSequence(*pictures*, *video_parameters*, *picture_coding_mode*)
A sequence of pictures to be encoded by a VC-2 encoder under test.

Parameters

pictures [[{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}, ...]] A `list`[28] of dictionaries containing raw picture data in 2D arrays for each picture component.

video_parameters [*VideoParameters* (page 142)] The video parameters associated with the test sequence.

picture_coding_mode [`PictureCodingModes` ([vc2_data_tables], page 4)] The picture coding mode associated with the test sequence.

[28] https://docs.python.org/3/library/stdtypes.html#list

5.1.3 Decoder test case generators

Decoder test case generators are located in `vc2_conformance.test_cases.decoder` must be decorated with the *vc2_conformance.test_cases.decoder_test_case_generator* (page 65) decorator, take a *CodecFeatures* (page 65) and produce *vc2_conformance.bitstream.Stream* (page 86) dictionaries which can be serialised using *autofill_and_serialise_stream()* (page 107).

decoder_test_case_generator
Decorator to use to register all decoder test case generators.

5.1.4 Test case generator registries

All test case generators are registered (by the *encoder_test_case_generator* (page 64) and *decoder_test_case_generator* (page 65) decorators) with one of two *Registry* (page 65) singletons:

ENCODER_TEST_CASE_GENERATOR_REGISTRY
Registry (page 65) singleton with which all VC-2 encoder test cases are registered.

DECODER_TEST_CASE_GENERATOR_REGISTRY
Registry (page 65) singleton with which all VC-2 decoder test cases are registered.

The *Registry* (page 65) class implements a registry of test case generators which the *vc2-test-case-generator* (page 43) script uses to generate a complete set of test cases.

class Registry
A registry of test case generating functions.

register_test_case_generator(*f*)
Register a test case generator function with this registry.

Returns the (unmodified) function allowing this method to be used as a decorator.

generate_test_cases(**args*, ***kwargs*)
Run every test case generator registered with this registry, passing each generator the supplied arguments. Generates all *TestCase* (page 63) objects.

iter_independent_generators(**args*, ***kwargs*)
Produce a series of generator functions which may be called in parallel. Each returned zero-argument function should be called and will generate a series of *TestCase* (page 63) objects.

iter_registered_functions()
Iterates over the raw functions registered with this registry.

Only intended for use during documentation generation.

5.2 vc2_conformance.codec_features: Codec feature definitions

The *vc2_conformance.codec_features* (page 65) module defines the *CodecFeatures* (page 65) *fixeddict* (page 157) which is used to describe codec and video format configurations. These are used to control the test case generators (*vc2_conformance.test_cases* (page 63)) and encoder (*vc2_conformance.encoder* (page 75)).

fixeddict CodecFeatures
A definition of a set of coding features supported by a video codec implementation. In practice, a particular codec's feature set may be defined by a collection of these, defining support for several picture formats.

Keys

name [str] A unique, human-readable name to identify this collection of features (e.g. `"uhd_over_hd_sdi"`).

level [`Levels` ([vc2_data_tables], page 8)] The VC-2 level. The user is responsible for choosing settings below which actually conform to this level.

profile [`Profiles` ([vc2_data_tables], page 7)] The VC-2 profile to use.

picture_coding_mode [`PictureCodingModes` ([vc2_data_tables], page 4)] The picture coding mode to use.

video_parameters [*`VideoParameters`* (page 142)] The video format to use.

wavelet_index, wavelet_index_ho [`WaveletFilters` ([vc2_data_tables], page 7)] The wavelet transform to use. For symmetric transforms, both values must be equal.

dwt_depth and dwt_depth_ho [int] The transform depths to use. For symmetric transforms, dwt_depth_ho must be zero.

slices_x and slices_y [int] The number of picture slices, horizontally and vertically.

fragment_slice_count [int] If fragmented pictures are in use, should be non-zero and contain the maximum number of slices to include in each fragment. Otherwise, should be zero.

lossless [bool] If True, lossless variable-bit-rate coding will be used. If false, fixed-rate lossy coding is used.

picture_bytes [int or None] When `lossless` is False, this gives the number of bytes per picture to use. Slices will be assigned (as close to) the same number of bytes each as possible. If `lossless` is True, this value should be None.

quantization_matrix [None or {level: {orient: value, ...}, ...}] None or a hierarchy of dictionaries as constructed by the `quant_matrix` pseudocode function (12.4.5.3). If None, the default quantization matrix will be used.

The *`read_codec_features_csv()`* (page 66) may be used to read these structures from a CSV. This functionality is used by the *vc2-test-case-generator* (page 43) script.

`read_codec_features_csv`(*csvfile*)

Read a set of *`CodecFeatures`* (page 65) from a CSV file in the format described by *Defining codec features* (page 15).

Parameters

csvfile [iterable] An iterable of lines from a CSV file, e.g. an open `file` object.

Returns

codec_feature_sets [OrderedDict([(name, *`CodecFeatures`* (page 65)), ...])]

Raises

***`InvalidCodecFeaturesError`* (page 66)** Raised if the provided CSV contains invalid or incomplete data.

Note: Validation largely extends only to syntactic issues (e.g. invalid integer values, 'picture_bytes' being specified for lossless formats etc). It does not include validation of 'deeper' issues such as too-small picture_bytes values or parameters not being permitted by the specified level.

---

exception `InvalidCodecFeaturesError`

Raised by *`read_codec_features_csv()`* (page 66) encounters a problem with the data presented in a codec features listing CSV file.

Finally, the following function may be used when determining level-related restrictions which apply to a set of *`CodecFeatures`* (page 65).

codec_features_to_trivial_level_constraints(*codec_features*)

Returns the some of the values a given *CodecFeatures* (page 65) would assign in a *level_constraints* (page 119) table.

Parameters

codec_features [*CodecFeatures* (page 65)]

Returns

constrained_values [{key: concrete_value, . . . }] A partial set of *level_constraints* (page 119), specifically containing the following keys:

- level
- profile
- picture_coding_mode
- wavelet_index
- dwt_depth
- slices_x
- slices_y
- slices_have_same_dimensions
- custom_quant_matrix

In addition for the low delay profile only:

- slice_bytes_numerator
- slice_bytes_denominator

In addition for the high quality profile only:

- slice_prefix_bytes

Note: In principle, more keys could be determined, however a line in the sand is required for what is considered 'simple' to determine and what requires re-implementing much of the codec. We draw the line at these values since all of them are straightforward to work out.

CHAPTER

SIX

VC2_CONFORMANCE.DECODER: REFERENCE DECODER AND BITSTREAM VALIDATOR

The *vc2_conformance.decoder* (page 69) module contains the components of a VC-2 decoder and bitstream validator. Along with the basic decoding logic, additional tests are included to check all conditions imposed on bitstreams by the VC-2 specification.

6.1 Usage

The bitstream decoder/validator is exposed to end-users via the *vc2-bitstream-validator* (page 45) command line utility.

This module may also be used directly. The following snippet illustrates how a VC-2 bitstream might be decoded and verified using this module:

```
>>> from vc2_conformance.string_utils import wrap_paragraphs
>>> from vc2_conformance.pseudocode import State
>>> from vc2_conformance.decoder import init_io, parse_stream, ConformanceError

>>> # Create a callback to be called with picture data whenever a picture
>>> # is decoded from the bitstream.
>>> def output_picture_callback(picture, video_parameters, picture_coding_mode):
>>>     print("A picture was decoded...")

>>> # Create an initial state object ready to read the bitstream
>>> state = State(_output_picture_callback=output_picture_callback)
>>> f = open("path/to/bitstream.vc2", "rb")
>>> init_io(state, f)

>>> # Decode and validate!
>>> try:
...     parse_stream(state)
...     print("Bitstream is valid!")
... except ConformanceError as e:
...     print("Bitstream is NOT valid:")
...     print(wrap_paragraphs(e.explain(), 80))
Bitstream is NOT valid:
An invalid parse code, 0x0A, was provided to a parse info header (10.5.1).

See (Table 10.1) for the list of allowed parse codes.

Perhaps this bitstream conforms to an earlier or later version of the VC-2
standard?
```

6.2 Overview

This decoder is based on the pseudocode published in the VC-2 specification and consequently follows the same structure as the pseudocode with the `parse_stream()` function being used to decode a complete stream.

The pseudocode is automatically verified for consistency with the VC-2 specification by the conformance software test suite. Verified pseudocode functions are annotated with the *ref_pseudocode()* (page 144) decorator. See *verification* (page 147) (in the `tests/` directory) for details on the automated verification process. All bitstream validation logic, which doesn't form part of the specified pseudocode, appears between `## Begin not in spec` and `## End not in spec` comments.

All global state is passed around via the *State* (page 139) dictionary. This dictionary is augmented with a number of additional entries not included in the VC-2 specification but which are necessary for a 'real' decoder implementation (e.g. an input file handle) and for validation purposes (e.g. recorded offsets of previous data units). See *vc2_conformance.pseudocode.state.State* (page 139) for a complete enumeration of these.

Underlying I/O operations are not specified by the VC-2 specification. This decoder reads streams from file-like objects. See the *vc2_conformance.decoder.io* (page 70) module for details. As illustrated in the example above, the *init_io()* (page 71) function is used to specify the file-like object the stream will be read from.

When conformance errors are detected, *ConformanceError* (page 72) exceptions are thrown. These exceptions provide in-depth human readable explanations of conformance issues along with suggested invocations of the *vc2-bitstream-viewer* (page 51) tool for diagnosing issues. See the *vc2_conformance.decoder.exceptions* (page 72) module for details. These exceptions are largely thrown directly by validation code spliced into the pseudocode routines. Some common checks are factored out into their own 'assertions' in the `vc2_conformance.decoder.assertions`.

The decoder logic is organised in line with the sections of the VC-2 specification:

- *vc2_conformance.decoder.io* (page 70): (A) Bitstream I/O
- `vc2_conformance.decoder.stream`: (10) Stream syntax
- `vc2_conformance.decoder.sequence_header`: (11) Sequence header
- `vc2_conformance.decoder.picture_syntax`: (12) Picture syntax
- `vc2_conformance.decoder.transform_data_syntax`: (13) Transform data syntax
- `vc2_conformance.decoder.fragment_syntax`: (14) Fragment syntax

The *vc2_conformance.decoder* (page 69) module only includes pseudocode functions which define additional behaviour not defined by the spec. Specifically, this includes performing I/O or additional checks for bitstream validation purposes. All other pseudocode routines are used 'verbatim' and can be found in *vc2_conformance.pseudocode* (page 133).

6.3 Stream I/O

The `vc2_conformance.io` module implements I/O functions for reading bitstreams from file-like objects. The exposed functions implement the interface specified in annex (A).

6.3.1 Initialisation

The *init_io()* (page 71) function must be used to initialise a *State* (page 139) dictionary so that it is ready to read a bitstream.

init_io(*state*, *f*)
(A.2.1) Initialise the I/O-related variables in state.

This function should be called exactly once to initialise the I/O-related parts of the state dictionary to their initial state as specified by (A.2.1):

> ... a decoder is deemed to maintain a copy of the current byte, state[current_byte], and an index to the next bit (in the byte) to be read, state[next_bit] ...

As well as initialising the state["current_byte"] and state["next_bit"] fields, this sets the (out-of-spec) state["_file"] entry to the provided file-like object.

Parameters

state [*State* (page 139)] The state dictionary to be initialised.

f [file-like object] The file to read the bitstream from.

6.3.2 Determining stream position

The *tell()* (page 71) function (below) is used by verification logic to report and check offsets of values within a bitstream. For example, it may be used to check `next_parse_offset` fields are correct (see `vc2_conformance.decoder.stream.parse_info()`).

tell(*state*)
Not part of spec; used to log bit offsets in the bitstream.

Return a (byte, bit) tuple giving the offset of the next bit to be read in the stream.

6.3.3 Bitstream recording

The VC-2 specification sometimes requires that the coded bitstream representation of a particular set of repeated fields is consistent within a bitstream (e.g. see `vc2_conformance.decoder.sequence_header.sequence_header()`). To facilitate this test, the *record_bitstream_start()* (page 71) and *record_bitstream_finish()* (page 71) functions may be used to capture the bitstream bytes read within part of the bitstream.

record_bitstream_start(*state*)
Not part of spec; used for verifying that repeated sequence_headers are byte-for-byte identical (11.1).

This function causes all future bytes read from the bitstream to be logged into state["_read_bytes"] until *record_bitstream_finish()* (page 71) is called.

Recordings must start byte aligned.

record_bitstream_finish(*state*)
See *record_bitstream_start()* (page 71).

Returns

bytearray The bytes read since *record_bitstream_start()* (page 71) was called. Any unread bits of the final byte will be set to zero.

6.4 Conformance exceptions

The *vc2_conformance.decoder.exceptions* (page 72) module defines a number of exceptions derived from *ConformanceError* (page 72) representing different conformance errors a bitstream may contain. These exceptions provide additional methods which return detailed human-readable information about the conformance error.

exception ConformanceError
Base class for all bitstream conformance failure exceptions.

explain()
Produce a detailed human readable explanation of the conformance failure.

Should return a string which can be re-linewrapped by *vc2_conformance.string_utils.wrap_paragraphs()* (page 164).

The first line will be used as a summary when the exception is printed using `str()`.

bitstream_viewer_hint()
Return a set of sample command line arguments for the vc2-bitstream-viewer tool which will display the relevant portion of the bitstream.

This string may include the following `str.format()`[29] substitutions which should be filled in before display:

- `{cmd}` The command name of the bitstream viewer (i.e. usually `vc2-bitstream-viewer`)
- `{file}` The filename of the bitstream.
- `{offset}` The bit offset of the next bit in the bitstream to be read.

This returned string should *not* be line-wrapped but should be de-indented by `textwrap.dedent()`[30].

offending_offset()
If known, return the bit-offset of the offending part of the bitstream. Otherwise return None (and the current offset will be assumed).

6.5 Sequence composition restrictions

Various restrictions may be imposed on choice and order of data unit types in a sequence. For example, all sequences must start with a sequence header and end with an end of sequence. Some levels impose additional restrictions such as prohibiting the mixing of fragments and pictures or requiring sequence headers to be interleaved between every picture.

Rather than using ad-hoc logic to enforce these restrictions, regular expressions are used to check the pattern of data unit types. See the *vc2_conformance.symbol_re* (page 122) module for details on the regular expression matching system.

[29] https://docs.python.org/3/library/stdtypes.html#str.format
[30] https://docs.python.org/3/library/textwrap.html#textwrap.dedent

6.6 Level constraints

VC-2's levels impose additional constraints on bitstreams, for example restricting some fields to particular ranges of values. Rather than including ad-hoc validation logic for each level, a 'constraints table' is used. Bitstream values which may be constrained are checked using `vc2_conformance.decoder.assertions.assert_level_constraint()`.

See the *vc2_conformance.constraint_table* (page 128) module for an introduction to constraint tables. See the *vc2_conformance.level_constraints* (page 119) module for level-related constraint data, including documentation on the entries included in the levels constraints table.

CHAPTER

SEVEN

VC2_CONFORMANCE.ENCODER: INTERNAL VC-2 ENCODER

The *vc2_conformance.encoder* (page 75) module implements a simple VC-2 encoder which is used to produce test streams for conformance testing purposes (see *vc2_conformance.test_cases* (page 63)). It is extremely slow and performs only simple picture compression, but is sufficiently flexible to support all VC-2 coding modes.

The encoder is principally concerned with carrying out the following tasks:

- Encoding the video and codec configuration into a sequence header (see *vc2_conformance.encoder.sequence_header* (page 76)).
- Transforming, slicing and quantizing pictures (i.e. compressing them) (see *vc2_conformance.encoder.pictures* (page 77)).
- Assembling sequences of data units comprising a complete VC-2 stream (see *vc2_conformance.encoder.sequence* (page 79)).

This module does not generate serialised VC-2 bitstreams in binary format. Instead, it generates a bitstream description data structure which may subsequently be serialised by the bitstream serialiser in the *vc2_conformance.bitstream* (page 81) module. This design allows the generated bitstream to be more easily manipulated prior to serialisation if required for a particular test case.

7.1 Usage

The encoder behaviour is controlled by a *CodecFeatures* (page 65) dictionary. This specifies picture/video format to be compressed (e.g. resolution etc.) along with the coding options (e.g. wavelet transform and bitrate). See the *vc2_conformance.codec_features* (page 65) module for more details. There are essentially two modes of operation, depending on the value of *CodecFeatures* (page 65)`["lossless"]`:

- Lossless mode: variable bitrate, qindex is always 0.
- Lossy mode: fixed bit rate, variable qindex.

A series of pictures may be encoded into a VC-2 sequence using the `vc2_conformance.encoder.make_sequence()` function, and then serialised into a binary bitstream as illustrated below:

```
>>> # Define the video format to be encoded and basic coding options
>>> from vc2_conformance.codec_features import CodecFeatures
>>> codec_features = CodecFeatures(...)

>>> # Encode a series of pictures
>>> from vc2_conformance.encoder import make_sequence
>>> pictures = [
...     {"Y": ..., "C1": ..., "C2": ..., "pic_num": 100},
...     # ...
... ]
>>> sequence = make_sequence(codec_features, pictures)

>>> # Serialise to a file
```

(continues on next page)

(continued from previous page)

```
>>> from vc2_conformance.bitstream import (
...     Stream,
...     autofill_and_serialise_stream,
... )
>>> with open("bitstream.vc2", "wb") as f:
...     autofill_and_serialise_stream(f, Stream(sequences=[sequence]))
```

7.2 Bitstream conformance

This encoder will produce bitstreams conformant with the VC-2 specification whenever the coding parameters specified represent a legal combination.

In some cases, invalid coding options will cause the encoder to fail and raise an exception in *vc2_conformance.encoder.exceptions* (page 76). For example, if lossless coding and the low delay profile are requested simultaneously.

In other cases, the encoder will produce a bitstream, however this bitstream will be non-conformant. For example, if the clean area defined is larger than the frame size, the encoder will ignore this inconsistency of metadata and produce a (non-conformant) bitstream anyway.

When conformant bitstreams are required, it is the responsibility of the user of the encoder to ensure that the provided *CodecFeatures* (page 65) are valid. In practice, the easiest way to do this is to check for exceptions when the encoder is used then use the bitstream validator (*vc2_conformance.decoder* (page 69)) to validate the generated bitstream.

7.3 Exceptions

The exceptions defined in *vc2_conformance.encoder.exceptions* (page 76) derive from *UnsatisfiableCodecFeaturesError* (page 76) and are thrown when the presented encoder configuration makes encoding impossible. These exceptions provide detailed explanations of why encoding was not possible.

exception UnsatisfiableCodecFeaturesError

Base class for exceptions thrown by the encoder when it is unable to generate a stream in the desired format due to some invalid *CodecFeatures* (page 65) configuration.

explain()

Produce a detailed human readable explanation of the failure.

Should return a string which can be re-linewrapped by *vc2_conformance.string_utils.wrap_paragraphs()* (page 164).

The first line will be used as a summary when the exception is printed using `str()`.

7.4 Sequence header generation

The *vc2_conformance.encoder.sequence_header* (page 76) module contains routines for encoding a set of video format and codec parameters into sequence headers.

The *make_sequence_header_data_unit()* (page 76) function is used to generate sequence headers by the encoder:

make_sequence_header_data_unit(*codec_features*)

Create a *DataUnit* (page 87) object containing a sequence header which sensibly encodes the features specified in *CodecFeatures* (page 65) dictionary provided.

Parameters

codec_features [*CodecFeatures* (page 65)]

Returns

data_unit [*DataUnit* (page 87)]

Raises

IncompatibleLevelAndVideoFormatError

In practice there are often many potential sequence header encodings for a given set of video parameters. For example, when a video format closely matches a predefined base video format, the various `custom_*_flag` overrides may largely be omitted. This is optional, however, and an encoder is free to use these overrides explicitly even when they're not required.

The *make_sequence_header_data_unit()* (page 76) function always attempts to use the most compact encoding it can. Some test cases, however may wish to use less compact encodings and so to support this the *iter_sequence_headers()* (page 77) function is provided:

iter_sequence_headers(*codec_features*)

Generate a series of *SequenceHeader* (page 87) objects which encode the video format specified in *CodecFeatures* (page 65) dictionary provided.

This generator will start with an efficient encoding of the required features, built on the most closely matched base video format. This will be followed by successively less efficient encodings (i.e. using more custom fields) but the same (best-matched) base video format. After this, encodings based on other base video formats will be produced (again starting with the most efficient encoding for each format first).

This generator may output no items if the VC-2 level specified does not permit the format given.

Parameters

codec_features [*CodecFeatures* (page 65)]

Yields

sequence_header [*SequenceHeader* (page 87)]

7.5 Picture encoding & compression

The *vc2_conformance.encoder.pictures* (page 77) module contains simple routines for compressing pictures in a VC-2 bitstream.

The picture encoding behaviour used by the encoder is encapsulated by the *make_picture_data_units()* (page 77) function which turns a series of pictures (given as raw pixel values) into a series of *DataUnits* (page 87):

make_picture_data_units(*codec_features*, *picture*, *minimum_qindex=0*, *minimum_slice_size_scaler=1*)

Create a seires of one or more *DataUnits* (page 87) containing a compressed version of the supplied picture.

When `codec_features["fragment_slice_count"]` is 0, a single picture parse data unit will be produced. otherwise a series of two or more fragment parse data units will be produced.

A simple wrapper around `make_picture_parse_data_unit()` and `make_fragment_parse_data_units()`.

Parameters

codec_features [*CodecFeatures* (page 65)]

picture [{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}] The picture to be encoded. This picture will be compressed using a simple VC-2 encoder implementation. It does not necessarily produce the most high-quality encodings. If `pic_num` is omitted, `picture_number` fields will be omitted in the output.

minimum_qindex [int] Specifies the minimum quantization index to be used. Must be 0 for lossless codecs.

minimum_slice_size_scaler [int] Specifies the minimum slice_size_scaler to be used for high quality pictures. Ignored in low delay mode.

Returns

data_units [[*vc2_conformance.bitstream.DataUnit* (page 87), ...]]

7.5.1 Encoding algorithm

Depending on the lossy/lossless coding mode chosen, one of two simple algorithms is used.

Lossless mode

In lossless mode, every slice's `qindex` will be set to 0 (no quantization) and all transform coefficients will be coded verbatim (though trailing zeros will be coded implicitly).

Slices will be sized as large as necessary, though as small as possible.

The smallest `slice_size_scaler` possible will be used for each coded picture independently.

Note: In principle, lossless modes may occasionally make use of quantization to achieve better compression. For example where all transform coefficients are a multiple of the same quantisation factor. This encoder, however, does not do this.

Lossy mode

In lossy mode the `qindex` for each slice is chosen on a slice-by-slice basis. The encoder tests quantization indices starting at zero and stopping when the transform coefficients fit into the slice.

Slices are sized such that the picture slice data in the bitstream totals *CodecFeatures* (page 65)`["picture_bytes"]`.

For the high quality profile, the smallest `slice_size_scaler` which can encode a slice where a single component consumes a whole slice is used for every picture.

Warning: The total size of picture slice data may differ from *CodecFeatures* (page 65)`["picture_bytes"]` by up to `slice_size_scaler` bytes (for high quality profile formats) or one byte (for low delay profile formats). This will occur when the number of bytes (or multiple of `slice_size_scaler` bytes) is not exactly divisible by the required number of picture bytes.

Warning: The total number of bytes used to encode each picture, once other coding overheads (such as headers) will be higher than *CodecFeatures* (page 65)`["picture_bytes"]`.

Note: This codec may not always produce highest quality pictures possible in lossy modes. For example, sometimes chosing higher quantisation indices can produce fewer coding artefacts, particularly in concatenated coding applications. Similarly, higher picture quality may sometimes be obtained by setting later transform coefficients to zero enabling a lower quantization index to be used. Other more sophisticated schemes may also directly tweak transform coefficients.

7.5.2 Use of pseudocode

This module uses the pseudocode-derived *vc2_conformance.pseudocode.picture_encoding* (page 136) module for its forward-DWT and *vc2_conformance.pseudocode.quantization* (page 137) for quantization. Other pseudocode routines are also used where possible, for example for computing slice dimensions.

7.6 Sequence generation

The *vc2_conformance.encoder.sequence* (page 79) module provides routines for constructing complete VC-2 sequences.

Principally, this module implements the *make_sequence()* (page 79) function which produces *vc2_conformance.bitstream.Sequence* (page 86) objects containing pictures compressed according to the required codec specifications. This is the main entry point to the encoder.

make_sequence(*codec_features*, *pictures*, **data_unit_patterns*, ***kwargs*)

Generate a complete VC-2 bitstream based on the provided set of codec features and containing compressed versions of the specified set of pictures.

This function also takes a small number of additional parameters which override certain encoder behaviours as may be required by some test case generators.

Parameters

codec_features [*CodecFeatures* (page 65)]

pictures [[{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}, ...]] The pictures to be encoded in the bitstream. If `pic_num` is omitted, `picture_number` fields will also be omitted in the output (and left for, e.g. `vc2_conformance.bitstream.autofill_and_serialise_stream()` to assign). See *vc2_conformance.encoder.pictures* (page 77) for details of the picture compression process.

***data_unit_patterns** [str] Force the generated sequence of data units to match a specified regular expression. For example, `"(. padding_data)+ end_of_sequence"` will force a padding data unit to be inserted between each data unit. See the *vc2_conformance.symbol_re* (page 122) module for details of the regular expression format.

A sequence of data units matching all specified patterns while meeting the requirements of the VC-2 standard will be generated. If this is not possible, `vc2_conformance.encoder.exceptions.IncompatibleLevelAndDataUnitError` will be raised.

minimum_qindex [int or [int, ...]] Keyword-only argument. Default 0. Specifies the minimum quantization index to be used for all picture slices. If a list is provided, specifies the minimum quantization index separately for each picture.

This option may be used by test cases where a particular (very high) quantization index must be used. Note that the encoder may still use larger quantization indices if a set of transform coefficients still do not fit into a slice so the caller must check that this has not occurred.

Must be 0 for lossless coding modes.

minimum_slice_size_scaler [int] Keyword-only argument. Default 1. Specifies the minimum slice size scaler to use.

For almost all sensible coding modes, the `slice_size_scaler` can be set to '1' – and this encoder will do so if possible. To facilitate the production of test cases verifying higher values are supported, this option may be used to pick a larger value. The encoder may still use larger `slice_size_scaler` values if this is necessary, however.

Only has an effect on high quality profile coding modes, will be ignored for the low delay profile modes.

Returns

sequence [*vc2_conformance.bitstream.Sequence* (page 86)] The VC-2 bitstream sequence. This may be serialised by encapsulating it in a *vc2_conformance.bitstream.Stream* (page 86) and serialising it with *autofill_and_serialise_stream()* (page 107).

Raises

UnsatisfiableCodecFeaturesError Raised if a sequence could not be generated according to the requirements given.

7.7 Level constraints

For the most part, all of the parameters which could be restricted by a VC-2 level are chosen in the supplied *CodecFeatures* (page 65). As such, choosing parameters which comply with the declared level is the responsibility of the caller (see comments above). However, some coding choices restricted by levels are left up to this encoder, such as how video parameters are coded in a sequence header. In these cases, the encoder makes choices which comply with the supplied level, a process which may require a constraint solving procedure.

In principle level constaints, as expressed by constraints tables (see *vc2_conformance.constraint_table* (page 128) and *vc2_conformance.level_constraints* (page 119)), could require a full global constraint solver to resolve. Fortunately, all existing VC-2 levels are specified such that, once the level (and a few other parameters) have been defined, almost all constrained parameters are independent meaning that global constraint solving is not required. The only case where constraint dependencies exist are the parameters relating to sequence headers. As a consequence the *sequence_header* (page 76) generation module uses a simple constraint solver internally.

Note: The constraint parameter independence property of the VC-2 levels mentioned above is essential for the encoder to generate level-conforming bitstreams. A test in `tests/encoder/test_level_constraints_assumptions.py` is provided which will fail should a future VC-2 level not have this property. See the detailed documentation at the top of this file for a more thorough introduction and discussion of this topic.

CHAPTER

EIGHT

VC2_CONFORMANCE.BITSTREAM: BITSTREAM MANIPULATION MODULE

The *vc2_conformance.bitstream* (page 81) module implements facilities for deserialising, displaying, manipulating and serialising VC-2 bitstreams, including non-conformant streams, at a low-level.

This documentation begins with an overview of how bitstreams can be serialised, deserialised and represented as Python data structures using this module. This is followed by an in-depth description of how the serialiser and deserialisers work internally.

8.1 How the serialiser/deserialiser module is used

This module is used by various parts of the VC-2 conformance software, for example:

- The *vc2-bitstream-viewer* (page 51) utility uses this module to produce human readable, hierarchical descriptions of bitstreams.
- The test case generators in *vc2_conformance.test_cases* (page 63) use this module to manipulate bitstreams, for example by tweaking values or filling padding bits with specific data.
- The VC-2 encoder in *vc2_conformance.encoder* (page 75) produces deserialised bitstreams directly for later serialisation by this module.
- The conformance software's own test suite makes extensive use of this module.

Note: This module is *not* used by the bitstream validator (*vc2_conformance.decoder* (page 69)) which instead operates directly on the binary bitstream instead.

This module consists of two main parts: the *serdes* (page 93) framework for building serialisers and deserialisers and a serialiser/deserialiser for VC-2.

The *serdes* (page 93) framework allows serialisers and deserialisers to be constructed *directly* from the VC-2 pseudocode, ensuring a high chance of correctness.

The VC-2 serialiser/deserialiser, implemented using the *serdes* (page 93) framework defines a series of Python data structures which may be used to describe a VC-2 bitstream.

8.2 Quick-start guide

Before diving into the details, we'll briefly give few quick examples which illustrate how this module is used below. We'll show how a small bitstream can be explicitly described as a Python data structure, serialised and then deserialised again. We'll skim over the details (and ignore a number of important but minor features) in the process.

8.2.1 Deserialised bitstream data structures

A VC-2 bitstream can be described hierarchically as a series of dictionaries and lists. For example, the following structure describes a minimal VC-2 bitstream containing just a single end-of-sequence data unit:

```
>>> from bitarray import bitarray

>>> # A minimal bitstream...
>>> bitstream_description = {
...     # ...consisting of a single sequence...
...     "sequences": [
...         {
...             # ...with a single data unit...
...             "data_units": [
...                 {
...                     # ...which is an end-of-sequence data unit
...                     "parse_info": {
...                         "padding": bitarray(),  # No byte alignment bits
...                         "parse_info_prefix": 0x42424344,
...                         "parse_code": 0x10,  # End of sequence
...                         "next_parse_offset": 0,
...                         "previous_parse_offset": 0,
...                     },
...                 },
...             ],
...         },
...     ],
... }
```

To make this somewhat clearer and more robust, a set of *fixeddicts* (page 157) are provided which may be used instead of bare Python dictionaries. The `vc2_data_tables` ([vc2_data_tables], page 3) package also includes many helpful constant definitions. Together, these make it easier to see what's going on while also eliminating simple mistakes like misspelling a field name.

Note: *fixeddicts* (page 157) are subclasses of Python's native `dict`[31] type with the following extra features:

- They allow only a permitted set of key names to be used.
- The `__str__` implementation produces an easier to read pretty-printed format.

Using these types, our example now looks like:

```
>>> from vc2_data_tables import PARSE_INFO_PREFIX, ParseCodes
>>> from vc2_conformance.bitstream import Stream, Sequence, DataUnit, ParseInfo

>>> bitstream_description = Stream(
...     sequences=[
...         Sequence(
...             data_units=[
...                 DataUnit(
```

(continues on next page)

[31] https://docs.python.org/3/library/stdtypes.html#dict

(continued from previous page)

```
...                         parse_info=ParseInfo(
...                             padding=bitarray(),  # No byte alignment bits
...                             parse_info_prefix=PARSE_INFO_PREFIX,
...                             parse_code=ParseCodes.end_of_sequence,
...                             next_parse_offset=0,
...                             previous_parse_offset=0,
...                         ),
...                     ),
...                 ],
...             ),
...         ],
... )
```

See *Deserialised VC-2 bitstream data types* (page 86) for details of the expected hierarchy of a deserialised bitstream (and the *fixeddict* (page 157) dictionary types provided).

8.2.2 Serialising bitstreams

To serialise our bitstream into binary form, we can use the following (which we'll unpick afterwards):

```
>>> from vc2_conformance.bitstream import BitstreamWriter, Serialiser, parse_stream
>>> from vc2_conformance.pseudocode import State

>>> with open("/path/to/bitstream.vc2", "wb") as f:
...     with Serialiser(BitstreamWriter(f), bitstream_description) as ser:
...         parse_stream(ser, State())
```

In the example above, `parse_stream` is (a special version of) the `parse_stream` VC-2 pseudocode function provided by the *vc2_conformance.bitstream* (page 81) module. This pseudocode function would normally decode a VC-2 stream (as per the VC-2 specification), however this modified version may be used to serialise (or deserialise) a bitstream. The modified function takes an extra first argument, a *Serialiser* (page 102) in this case, which it will use to produce the serialised bitstream.

The *Serialiser* (page 102) class takes two arguments in this example: a *BitstreamWriter* (page 104) and the data structure to serialise (`bitstream_description` in our example).

The *BitstreamWriter* (page 104) is a wrapper for file-like objects which provides additional bitwise I/O operations used during serialisation.

The second argument to *Serialiser* (page 102) is the deserialised data structure to be serialised. This may be an ordinary Python `dicts`[32] or a *fixeddict* (page 157).

Note: It is possible to serialise (or deserialise) components of a bitstream in isolation by using other pseudocode functions in place of `parse_stream`. In this case, the data structure provided to the *Serialiser* (page 102) must match the shape expected by the modified pseudocode function used. See *Deserialised VC-2 bitstream data types* (page 86) for an enumeration of the pseudocode functions available and the expected data structure.

---

[32] https://docs.python.org/3/library/stdtypes.html#dict

8.2.3 Autofilling bitstream values

In the example above we explicitly spelt out every field in the bitstream – including the empty padding field! If we had omitted this field, the serialiser will produce an error because it wouldn't know what padding bits we wanted it to use in the stream. However, often we don't care about details such as these and so the `Serialiser` can optionally 'autofill' certain values which weren't given in the deserialised data structure.

The `Serialiser` class takes an optional third argument which it uses to autofill missing values. A sensible set of a autofill values is provided in `vc2_conformance.bitstream.vc2_default_values` allowing us to rewrite our example like so:

```
>>> from vc2_conformance.bitstream import vc2_default_values

>>> concise_bitstream_description = Stream(
...     sequences=[
...         Sequence(
...             data_units=[
...                 DataUnit(
...                     parse_info=ParseInfo(
...                         parse_code=ParseCodes.end_of_sequence,
...                         next_parse_offset=0,
...                         previous_parse_offset=0,
...                     ),
...                 ),
...             ],
...         ),
...     ],
... )

>>> with open("/path/to/bitstream.vc2", "wb") as f:
...     with Serialiser(
...         BitstreamWriter(f),
...         concise_bitstream_description,
...         vc2_default_values,
...     ) as ser:
...         parse_stream(ser, State())
```

This time we were able to omit the byte-alignment padding value and parse info prefix which the serialiser autofilled with zeros and 0x42424344 respectively.

The default values for all fields are given in *Deserialised VC-2 bitstream data types* (page 86).

Unfortunately, when using the mechanism above, autofill values are not provided for every field in a bitstream (or for fields listed as autofilled with `<AUTO>` in the documentation). For instance, picture numbers and parse offset values must still be calculated and specified explicitly. For a more complete bitstream autofill solution the `vc2_conformance.bitstream.autofill_and_serialise_stream()` utility function is provided.

The `autofill_and_serialise_stream()` function can autofill most values including picture numbers, and parse offset fields (which marked as `<AUTO>` in the documentation). It also provides a more concise wrapper around the serialisation process.

Using `autofill_and_serialise_stream()` our example now becomes:

```
>>> from vc2_conformance.bitstream import autofill_and_serialise_stream

>>> very_concise_bitstream_description = Stream(
...     sequences=[
...         Sequence(
...             data_units=[
...                 DataUnit(
...                     parse_info=ParseInfo(
...                         parse_code=ParseCodes.end_of_sequence,
...                     ),
```

(continues on next page)

(continued from previous page)

```
...                     ),
...                 ],
...             ),
...         ],
... )

>>> with open("/path/to/bitstream.vc2", "wb") as f:
...     autofill_and_serialise_stream(f, very_concise_bitstream_description)
```

Notice that this time we could omit all but the `parse_code` field.

Note: The `autofill_and_serialise_stream()` function only supports serialisation of entire *Streams* (page 86) and cannot be used to serialise smaller pieces of a bitstream in isolation.

8.2.4 Deserialising bitstreams

To deserialise a bitstream again, the process is similar:

```
>>> from vc2_conformance.bitstream import BitstreamReader, Deserialiser

>>> with open("/tmp/bitstream.vc2", "rb") as f:
...     with Deserialiser(BitstreamReader(f)) as des:
...         parse_stream(des, State())
>>> deserialised_bitstream = des.context
```

This time, we pass a *Deserialiser* (page 102) (which takes a *BitstreamReader* (page 103) as argument) into `parse_stream`. The deserialised bitstream is placed into `des.context` as a hierarchy of `fixeddicts`. We can then print or interact with the deserialised data structure just like any other Python object:

```
>>> # NB: Fixeddicts produce pretty-printed output when printed!
>>> print(deserialised_bitstream)
Stream:
  sequences:
    0: Sequence:
      data_units:
        0: DataUnit:
          parse_info: ParseInfo:
            padding: 0b
            parse_info_prefix: Correct (0x42424344)
            parse_code: end_of_sequence (0x10)
            next_parse_offset: 0
            previous_parse_offset: 0

>>> data_unit = deserialised_bitstream["sequences"][0]["data_units"][0]
>>> print(data_unit["parse_info"]["parse_code"])
16
```

8.3 Deserialised VC-2 bitstream data types

Deserialised VC-2 bitstreams are described by a hierarchy of *fixeddicts* (page 157), exported in *vc2_conformance.bitstream* (page 81). Each *fixeddict* (page 157) represents the data read by a particular VC-2 pseudocode function. Special implementations of these functions are provided in *vc2_conformance.bitstream* (page 81) which may be used to serialise and deserialise VC-2 bitstreams (or individual parts thereof).

Type	Pseudocode function
Stream (page 86)	parse_stream
Sequence (page 86)	parse_sequence
DataUnit (page 87)	
ParseInfo (page 87)	parse_info
SequenceHeader (page 87)	sequence_header
ParseParameters (page 87)	parse_parameters
SourceParameters (page 88)	source_parameters
FrameSize (page 88)	frame_size
ColorDiffSamplingFormat (page 88)	color_diff_sampling_format
ScanFormat (page 88)	scan_format
FrameRate (page 88)	frame_rate
PixelAspectRatio (page 88)	pixel_aspect_ratio
CleanArea (page 89)	clean_area
SignalRange (page 89)	signal_range
ColorSpec (page 89)	color_spec
ColorPrimaries (page 89)	color_primaries
ColorMatrix (page 89)	color_matrix
TransferFunction (page 90)	transfer_function
PictureParse (page 90)	picture_parse
PictureHeader (page 90)	picture_header
WaveletTransform (page 90)	wavelet_transform
TransformParameters (page 90)	transform_parameters
ExtendedTransformParameters (page 90)	extended_transform_parameters
SliceParameters (page 91)	slice_parameters
QuantMatrix (page 91)	quant_matrix
TransformData (page 91)	transform_data
LDSlice (page 91)	ld_slice
HQSlice (page 91)	hq_slice
FragmentParse (page 92)	fragment_parse
FragmentHeader (page 92)	fragment_header
TransformParameters (page 90)	transform_parameters
ExtendedTransformParameters (page 90)	extended_transform_parameters
SliceParameters (page 91)	slice_parameters
QuantMatrix (page 91)	quant_matrix
FragmentData (page 92)	fragment_data
LDSlice (page 91)	ld_slice
HQSlice (page 91)	hq_slice
AuxiliaryData (page 92)	auxiliary_data
Padding (page 93)	padding

fixeddict Stream

(10.3) A VC-2 stream.

Keys

sequences [[*Sequence* (page 86), ...]]

fixeddict Sequence

(10.4.1) A VC-2 sequence.

Keys

data_units [[*DataUnit* (page 87), ...]]

_state [*State* (page 139)] Computed value. The *State* (page 139) object being populated by the parser.

fixeddict DataUnit

A data unit (e.g. sequence header or picture) and its associated parse info. Based on the values read by parse_sequence() (10.4.1) in each iteration.

Keys

parse_info [*ParseInfo* (page 87)]

sequence_header [*SequenceHeader* (page 87)]

picture_parse [*PictureParse* (page 90)]

fragment_parse [*FragmentParse* (page 92)]

auxiliary_data [*AuxiliaryData* (page 92)]

padding [*Padding* (page 93)]

fixeddict ParseInfo

(10.5.1) Parse info header defined by `parse_info()`.

Keys

padding [`bitarray` (autofilled with bitarray())] Byte alignment padding bits.

parse_info_prefix [int (autofilled with 1111638852)]

parse_code [`ParseCodes` ([vc2_data_tables], page 3) (autofilled with <ParseCodes.end_of_sequence: 16>)]

next_parse_offset [int (autofilled with <AUTO>)]

previous_parse_offset [int (autofilled with <AUTO>)]

_offset [int] Computed value. The byte offset of the start of this parse_info block in the bitstream.

fixeddict SequenceHeader

(11.1) Sequence header defined by `sequence_header()`.

Keys

parse_parameters [*ParseParameters* (page 87)]

base_video_format [`BaseVideoFormats` ([vc2_data_tables], page 6) (autofilled with <BaseVideoFormats.custom_format: 0>)]

video_parameters [*SourceParameters* (page 88)]

picture_coding_mode [`PictureCodingModes` ([vc2_data_tables], page 4) (autofilled with <PictureCodingModes.pictures_are_frames: 0>)]

fixeddict ParseParameters

(11.2.1) Sequence header defined by `parse_parameters()`.

Keys

major_version [int (autofilled with <AUTO>)]

minor_version [int (autofilled with 0)]

profile [`Profiles` ([vc2_data_tables], page 7) (autofilled with <Profiles.high_quality: 3>)]

level [`Levels` ([vc2_data_tables], page 8) (autofilled with <Levels.unconstrained: 0>)]

fixeddict SourceParameters
(11.4.1) Video format overrides defined by `source_parameters()`.

Keys

frame_size [*FrameSize* (page 88)]

color_diff_sampling_format [*ColorDiffSamplingFormat* (page 88)]

scan_format [*ScanFormat* (page 88)]

frame_rate [*FrameRate* (page 88)]

pixel_aspect_ratio [*PixelAspectRatio* (page 88)]

clean_area [*CleanArea* (page 89)]

signal_range [*SignalRange* (page 89)]

color_spec [*ColorSpec* (page 89)]

fixeddict FrameSize
(11.4.3) Frame size override defined by `frame_size()`.

Keys

custom_dimensions_flag [bool (autofilled with False)]

frame_width [int (autofilled with 1)]

frame_height [int (autofilled with 1)]

fixeddict ColorDiffSamplingFormat
(11.4.4) Color-difference sampling override defined by `color_diff_sampling_format()`.

Keys

custom_color_diff_format_flag [bool (autofilled with False)]

color_diff_format_index [`ColorDifferenceSamplingFormats` ([vc2_data_tables], page 4) (autofilled with <ColorDifferenceSamplingFormats.color_4_4_4: 0>)]

fixeddict ScanFormat
(11.4.5) Scan format override defined by `scan_format()`.

Keys

custom_scan_format_flag [bool (autofilled with False)]

source_sampling [`SourceSamplingModes` ([vc2_data_tables], page 4) (autofilled with <SourceSamplingModes.progressive: 0>)]

fixeddict FrameRate
(11.4.6) Frame-rate override defined by `frame_rate()`.

Keys

custom_frame_rate_flag [bool (autofilled with False)]

index [`PresetFrameRates` ([vc2_data_tables], page 4) (autofilled with <PresetFrameRates.fps_25: 3>)]

frame_rate_numer [int (autofilled with 25)]

frame_rate_denom [int (autofilled with 1)]

fixeddict PixelAspectRatio
(11.4.7) Pixel aspect ratio override defined by `pixel_aspect_ratio()` (errata: also listed as `aspect_ratio()` in some parts of the spec).

Keys

custom_pixel_aspect_ratio_flag [bool (autofilled with False)]

index [PresetPixelAspectRatios ([vc2_data_tables], page 4) (autofilled with <PresetPixelAspectRatios.ratio_1_1: 1>)]

pixel_aspect_ratio_numer [int (autofilled with 1)]

pixel_aspect_ratio_denom [int (autofilled with 1)]

fixeddict CleanArea

(11.4.8) Clean areas override defined by `clean_area()`.

Keys

custom_clean_area_flag [bool (autofilled with False)]

clean_width [int (autofilled with 1)]

clean_height [int (autofilled with 1)]

left_offset [int (autofilled with 0)]

top_offset [int (autofilled with 0)]

fixeddict SignalRange

(11.4.9) Signal range override defined by `signal_range()`.

Keys

custom_signal_range_flag [bool (autofilled with False)]

index [PresetSignalRanges ([vc2_data_tables], page 4) (autofilled with <PresetSignalRanges.video_8bit_full_range: 1>)]

luma_offset [int (autofilled with 0)]

luma_excursion [int (autofilled with 1)]

color_diff_offset [int (autofilled with 0)]

color_diff_excursion [int (autofilled with 1)]

fixeddict ColorSpec

(11.4.10.1) Color specification override defined by `color_spec()`.

Keys

custom_color_spec_flag [bool (autofilled with False)]

index [PresetColorSpecs ([vc2_data_tables], page 6) (autofilled with <PresetColorSpecs.hdtv: 3>)]

color_primaries [*ColorPrimaries* (page 89)]

color_matrix [*ColorMatrix* (page 89)]

transfer_function [*TransferFunction* (page 90)]

fixeddict ColorPrimaries

(11.4.10.2) Color primaries override defined by `color_primaries()`.

Keys

custom_color_primaries_flag [bool (autofilled with False)]

index [PresetColorPrimaries ([vc2_data_tables], page 5) (autofilled with <PresetColorPrimaries.hdtv: 0>)]

fixeddict ColorMatrix

(11.4.10.3) Color matrix override defined by `color_matrix()`.

Keys

custom_color_matrix_flag [bool (autofilled with False)]

index [PresetColorMatrices ([vc2_data_tables], page 5) (autofilled with <PresetColorMatrices.hdtv: 0>)]

fixeddict TransferFunction

(11.4.10.4) Transfer function override defined by transfer_function().

Keys

custom_transfer_function_flag [bool (autofilled with False)]

index [PresetTransferFunctions ([vc2_data_tables], page 5) (autofilled with <PresetTransferFunctions.tv_gamma: 0>)]

fixeddict PictureParse

(12.1) A picture data unit defined by picture_parse()

Keys

padding1 [bitarray (autofilled with bitarray())] Picture header byte alignment padding bits.

picture_header [*PictureHeader* (page 90)]

padding2 [bitarray (autofilled with bitarray())] Wavelet transform byte alignment padding bits.

wavelet_transform [*WaveletTransform* (page 90)]

fixeddict PictureHeader

(12.2) Picture header information defined by picture_header().

Keys

picture_number [int (autofilled with <AUTO>)]

fixeddict WaveletTransform

(12.3) Wavelet parameters and coefficients defined by wavelet_transform().

Keys

transform_parameters [*TransformParameters* (page 90)]

padding [bitarray (autofilled with bitarray())] Byte alignment padding bits.

transform_data [*TransformData* (page 91)]

fixeddict TransformParameters

(12.4.1) Wavelet transform parameters defined by transform_parameters().

Keys

wavelet_index [WaveletFilters ([vc2_data_tables], page 7) (autofilled with <WaveletFilters.haar_with_shift: 4>)]

dwt_depth [int (autofilled with 0)]

extended_transform_parameters [*ExtendedTransformParameters* (page 90)]

slice_parameters [*SliceParameters* (page 91)]

quant_matrix [*QuantMatrix* (page 91)]

fixeddict ExtendedTransformParameters

(12.4.4.1) Extended (horizontal-only) wavelet transform parameters defined by extended_transform_parameters().

Keys

asym_transform_index_flag [bool (autofilled with False)]

wavelet_index_ho [WaveletFilters ([vc2_data_tables], page 7) (autofilled with <WaveletFilters.haar_with_shift: 4>)]

asym_transform_flag [bool (autofilled with False)]

dwt_depth_ho [int (autofilled with 0)]

fixeddict SliceParameters

(12.4.5.2) Slice dimension parameters defined by `slice_parameters()`.

Keys

slices_x [int (autofilled with 1)]

slices_y [int (autofilled with 1)]

slice_bytes_numerator [int (autofilled with 1)]

slice_bytes_denominator [int (autofilled with 1)]

slice_prefix_bytes [int (autofilled with 0)]

slice_size_scaler [int (autofilled with 1)]

fixeddict QuantMatrix

(12.4.5.3) Custom quantisation matrix override defined by `quant_matrix()`.

Keys

custom_quant_matrix [bool (autofilled with False)]

quant_matrix [[int, ...] (autofilled with 0)] Quantization matrix values in bitstream order.

fixeddict TransformData

(13.5.2) Transform coefficient data slices read by `transform_data()`.

Keys

ld_slices [[*LDSlice* (page 91), ...]]

hq_slices [[*HQSlice* (page 91), ...]]

_state [*State* (page 139)] Computed value. A copy of the *State* (page 139) dictionary held when processing this transform data. May be used to work out how the deserialised values correspond to transform components within the slices above.

fixeddict LDSlice

(13.5.3.1) The data associated with a single low-delay slice, defined by `ld_slice()`.

Keys

qindex [int (autofilled with 0)]

slice_y_length [int (autofilled with 0)]

y_transform [[int, ...] (autofilled with 0)] Slice luma transform coefficients in bitstream order.

c_transform [[int, ...] (autofilled with 0)] Slice interleaved colordifference transform coefficients in bitstream order.

y_block_padding [`bitarray` (autofilled with bitarray())] Unused bits from y_transform bounded block.

c_block_padding [`bitarray` (autofilled with bitarray())] Unused bits from c_transform bounded block.

_sx [int] Computed value. Slice coordinates.

_sy [int] Computed value. Slice coordinates.

fixeddict HQSlice

(13.5.4) The data associated with a single high-quality slice, defined by `hq_slice()`.

Keys

prefix_bytes [bytes (autofilled with b'')]

qindex [int (autofilled with 0)]

slice_y_length [int (autofilled with 0)]

slice_c1_length [int (autofilled with 0)]

slice_c2_length [int (autofilled with 0)]

y_transform [[int, ...] (autofilled with 0)] Slice luma transform coefficients in bitstream order.

c1_transform [[int, ...] (autofilled with 0)] Slice color difference 1 transform coefficients in bitstream order.

c2_transform [[int, ...] (autofilled with 0)] Slice color difference 2 transform coefficients in bitstream order.

y_block_padding [`bitarray` (autofilled with bitarray())] Unused bits in y_transform bounded block

c1_block_padding [`bitarray` (autofilled with bitarray())] Unused bits in c1_transform bounded block

c2_block_padding [`bitarray` (autofilled with bitarray())] Unused bits in c2_transform bounded block

_sx [int] Computed value. Slice coordinates.

_sy [int] Computed value. Slice coordinates.

fixeddict FragmentParse

(14.1) A fragment data unit defined by `fragment_parse()` containing part of a picture.

Keys

fragment_header [*FragmentHeader* (page 92)]

transform_parameters [*TransformParameters* (page 90)]

fragment_data [*FragmentData* (page 92)]

fixeddict FragmentHeader

(14.2) Fragment header defined by `fragment_header()`.

Keys

picture_number [int (autofilled with <AUTO>)]

fragment_data_length [int (autofilled with 0)]

fragment_slice_count [int (autofilled with 0)]

fragment_x_offset [int (autofilled with 0)]

fragment_y_offset [int (autofilled with 0)]

fixeddict FragmentData

(14.4) Transform coefficient data slices read by `fragment_data()`.

Keys

ld_slices [[*LDSlice* (page 91), ...]]

hq_slices [[*HQSlice* (page 91), ...]]

_state [*State* (page 139)] Computed value. A copy of the *State* (page 139) dictionary held when processing this fragment data. May be used to work out how the deserialised values correspond to transform components within the slices above.

fixeddict AuxiliaryData

(10.4.4) Auxiliary data block (as per auxiliary_data()).

Keys

bytes [bytes (autofilled with b")]

fixeddict Padding
(10.4.5) Padding data block (as per padding()).

Keys

bytes [bytes (autofilled with b")]

8.4 serdes: A serialiser/deserialiser framework

The *vc2_conformance.bitstream.serdes* (page 93) module provides a framework for transforming a set of functions designed to process a bitstream (e.g. the VC-2 specification's pseudocode) into general-purpose bitstream serialisers, deserialisers and analysers.

The following sections introduce the design and operation of this module in detail.

8.4.1 A basic bitstream serialiser

The VC-2 specification describes the bitstream and decoding process in a series of pseudocode functions such as the following:

```
frame_size(video_parameters):
    custom_dimensions_flag = read_bool()
    if(custom_dimensions_flag == True)
        video_parameters[frame_width] = read_uint()
        video_parameters[frame_height] = read_uint()
```

To see how this definition might be transformed into a general purpose bitstream serialiser we must transform this definition of a program which *reads* a VC-2 bitstream into one which *writes* one.

We start by replacing all of the `read_*` functions with equivalent `write_*` functions (which we define here as returning the value that they write):

```
frame_size(video_parameters):
    custom_dimensions_flag = write_bool(???)
    if(custom_dimensions_flag == True)
        video_parameters[frame_width] = write_uint(???)
        video_parameters[frame_height] = write_uint(???)
```

Next we need to define what values we'd like writing by replacing the `???` placeholders with a suitable global variables like so:

```
new_custom_dimensions_flag = True
new_frame_width = 1920
new_frame_height = 1080

frame_size(video_parameters):
    custom_dimensions_flag = write_bool(new_custom_dimensions_flag)
    if(custom_dimensions_flag == True)
        video_parameters[frame_width] = write_uint(new_frame_width)
        video_parameters[frame_height] = write_uint(new_frame_height)
```

We have now transformed the VC-2 pseudocode bitstream *reader* function into a *writer* function. What's more, by just changing the values of global variables we created it is possible to use this function as a general-purpose bitstream serialiser.

8.4.2 A basic deserialiser

Unfortunately, the original bitstream reader pseudocode from the VC-2 specification is not quite usable as a general-purpose bitstream deserialiser:

- The reader does not capture every value read from the bitstream in a variable we can later examine (e.g. the `custom_dimensions_flag` is kept in a local variable and not returned).
- The values which are captured are stored in a structure designed to aid decoding and not necessarily to faithfully describe a bitstream.

Lets create a new version of the reader function which overcomes these limitations. We redefine the `read_*` functions to take an additional argument naming a global variable where the read values will be stored, in addition to being returned, giving the following pseudocode:

```
read_custom_dimensions_flag = None
read_frame_width = None
read_frame_height = None

frame_size(video_parameters):
    custom_dimensions_flag = read_bool(read_custom_dimensions_flag)
    if(custom_dimensions_flag == True)
        video_parameters[frame_width] = read_uint(read_frame_width)
        video_parameters[frame_height] = read_uint(read_frame_height)
```

This small change ensures that every value read from the bitstream is captured in a global variable which we can later examine and which is orthogonal to whatever data structures the VC-2 pseudocode might otherwise use.

An introduction to the `SerDes` interface

The similarities between the transformations required to turn the VC-2 pseudocode into general purpose serialisers and deserialisers should be fairly clear. In fact, the only difference between the two is that in one the functions are called `read_*` and in the other they're called `write_*`. In both cases, the `read_*` and `write_*` functions take essentially the same arguments: a name of a global variable.

This module defines the *SerDes* (page 98) interface which can be used by the VC-2 pseudocode specifications to drive both bitstream serialisation and deserialisation. To use it, we replace the `read_*` or `write_*` calls with `serdes.*` calls.

Translating the `frame_size` function into valid Python and taking a *SerDes* (page 98) instance as an argument we arrive at the following code:

```
def frame_size(serdes, video_parameters):
    custom_dimensions_flag = serdes.bool("custom_dimensions_flag")
    if(custom_dimensions_flag == True)
        video_parameters["frame_width"] = serdes.uint("frame_width")
        video_parameters["frame_height"] = serdes.uint("frame_height")
```

To deserialise (read) a bitstream we use the *Deserialiser* (page 102) implementation of *SerDes* (page 98) like so:

```
>>> from vc2_conformance.bitstream import BitstreamReader, Deserialiser
>>> reader = BitstreamReader(open("frame_size_snippet.bin", "rb"))

>>> with Deserialiser(reader) as des:
...     frame_size(des, {})
>>> des.context
{"custom_dimensions_flag": True, "frame_width": 1920, "frame_height": 1080}
```

The *SerDes.context* (page 101) property is a `dict`[33] which contains each of the values read from the bitstream (named as per the calls to the various *SerDes* (page 98) methods).

[33] https://docs.python.org/3/library/stdtypes.html#dict

In the nomenclature of this module, this *context* dictionary holds values for each of the *target names* specified by *SerDes.bool()* (page 98), *SerDes.uint()* (page 99) etc.

Values to be serialised should be structured into a context dictionary of similar shape and passed to a *Serialiser* (page 102):

```
>>> from vc2_conformance.bitstream import BitstreamWriter, Serialiser
>>> writer = BitstreamWriter(open("frame_size_snippet.bin", "wb"))

>>> context = {"custom_dimensions_flag": True, "frame_width": 1920, "frame_height
↪": 1080}
>>> with Serialiser(writer, context) as ser:
...     frame_size(ser, {})
```

In this example a bitstream containing a '1' followed by the variable-length integers '1920' and '1080' would be written to the bitstream.

8.4.3 Verification

The *SerDes* (page 98) implementations perform various 'sanity checks' during serialisation and deserialisation to ensure that the values passed in or returned have a 1:1 correspondence with values in the bitstream.

- When values are read during deserialisation, *Deserialiser* (page 102) checks that names are not re-used, guaranteeing that if a value appears in the bitstream it also appears in the output context dictionary (and are not later overwritten).
- When values are written during serialisation, *Serialiser* (page 102) checks that every value in the context dictionary is used exactly once, ensuring that every value provided is represented in the bitstream.
- During serialisation, values are also checked to ensure they can be correctly represented by the bitstream encoding. For example, providing a negative value to *uint()* (page 99) will fail.

8.4.4 Representing hierarchy

The VC-2 bitstream does not represent a flat collection of values but rather a hierarchy. The *SerDes* (page 98) interface provides additional facilities to allow this structure to be recreated in the deserialised representation, making it easier to inspect and describes bitstreams in their deserialised form.

For example, in the VC-2 specification, the `source_parameters` function (11.4) is defined by a series of functions which each read the values relating to a particular video feature such as the `frame_size` function we've seen above. To collect together related values we can use *SerDes.subcontext()* (page 101) to create nested context dictionaries:

```
def source_parameters(serdes):
    video_parameters = {}
    with serdes.subcontext("frame_size"):
        frame_size(serdes, video_parameters)
    with serdes.subcontext("color_diff_sampling_format"):
        color_diff_sampling_format(serdes, video_parameters)
    # ...
    return video_parameters
```

This results in a nested dictionary structure:

```
>>> with Deserialiser(reader) as des:
...     video_parameters = source_parameters(des)

>>> from pprint import pprint
>>> pprint(des.context)
{
```

(continues on next page)

(continued from previous page)

```
    "frame_size": {
        "custom_dimensions_flag": True,
        "frame_width": 1920,
        "frame_height": 1080,
    },
    "color_diff_sampling_format": {
        "custom_color_diff_format_flag": False,
    },
    # ...
}
```

When used with *vc2_conformance.fixeddict* (page 157), *SerDes* (page 98) also makes it possible to define custom dictionary types for each part of the hierarchy using the *context_type()* (page 103) decorator. Benefits include:

- Improved 'pretty-printed' string representations.
- Additional checks that unexpected values are not used accidentally in the bitstream.

For example, here's how the `parse_info` header (10.5.1) might be represented:

```
from vc2_conformance.fixeddict import fixeddict, Entry
from vc2_conformance.formatters import Hex
from vc2_data_tables import ParseCodes, PARSE_INFO_PREFIX
ParseInfo = fixeddict(
    "ParseInfo",
    Entry("parse_info_prefix", formatter=Hex(8)),
    Entry("parse_code", enum=ParseCodes),
    Entry("next_parse_offset"),
    Entry("previous_parse_offset"),
)

@context_type(ParseInfo)
def parse_info(serdes, state):
    serdes.nbits(4*8, "parse_info_prefix")
    state["parse_code"] = serdes.nbits(8, "parse_code")
    state["next_parse_offset"] = serdes.nbits(32, "next_parse_offset")
    state["previous_parse_offset"] = serdes.nbits(32, "previous_parse_offset")
```

Using the above we can quickly create structures ready for serialisation:

```
>>> context = ParseInfo(
...     parse_info_prefix=PARSE_INFO_PREFIX,
...     parse_code=ParseCodes.end_of_sequence,
...     next_parse_offset=0,
...     previous_parse_offset=1234,
... )
>>> with Deserialiser(writer, context) as des:
...     parse_info(des, {})
```

We also benefit from improved string formatting when deserialising values:

```
>>> with Deserialiser(reader) as des:
...     parse_info(des, {})
>>> str(des.context)
ParseInfo:
  parse_info_prefix: 0x42424344
  parse_code: end_of_sequence (0x10)
  next_parse_offset: 0
  previous_parse_offset: 1234
```

8.4.5 Representing arrays

The VC-2 bitstream format includes a number of array-like fields, for example arrays of transform coefficients within slices. Rather than defining unique names for every array value, *SerDes* (page 98) allows values to be declared as lists. For example:

```
def list_example(serdes):
    serdes.declare_list("three_values")
    serdes.uint("three_values")
    serdes.uint("three_values")
    serdes.uint("three_values")
```

When deserialising, the result will look like:

```
>>> with Deserialiser(reader) as des:
...     list_example(des)
>>> des.context
{"three_values": [100, 200, 300]}
```

Likewise, when serialising, a list of values (of the correct length) should also be provided:

```
>>> context = {"three_values": [10, 20, 30]}
>>> with Deserialiser(writer, context) as des:
...     list_example(des)
```

As usual, the *SerDes* (page 98) classes will verify that the correct number of values is present and will throw exceptions when too many or too few are provided.

8.4.6 Computed values

In some circumstances, when interpreting a deserialised bitstream it may be necessary to know information computed by an earlier part of the bitstream. For example, the dimensions of a slice depend on numerous video formatting options. To avoid error-prone reimplementation of these calculations it is possible to use *SerDes.computed_value()* (page 101) to add values to the context dictionary which do not appear in the bitstream. For example:

```
def ld_slice(serdes, state, sx, sy):
    serdes.computed_value("_slices_x", state["slices_x"])
    serdes.computed_value("_slices_y", state["slices_y"])
    # ...
```

The computed value will be set in the context dictionary regardless of whether serialisation or deserialisation is taking place and any existing value is always ignored.

Note: It is recommended that by convention computed value target names are prefixed or suffixed with an underscore.

8.4.7 Default values during serialisation

As discussed above, the default behaviour of the *Serialiser* (page 102) is to require that every value in the bitstream is provided in the context dictionary to make it explicit what is being serialised. In certain cases, however, it may be desirable for certain values to be filled in automatically. For example:

- For pre-filling constants like the parse_info prefix.
- For use in unit tests where only certain bitstream fields' values are of importance (and assigning defaults for the remainder makes the code clearer).
- For providing default (e.g. zero) values for padding fields

To facilitate this, the *Serialiser* (page 102) class may be passed a default value lookup like so:

```
>>> default_values = {
...     ParseInfo: {
...         "parse_info_prefix": PARSE_INFO_PREFIX,
...         "parse_code": ParseCodes.end_of_sequence,
...         "next_parse_offset": 0,
...         "previous_parse_offset": 0,
...     },
... }

>>> writer = BitstreamWriter(open("frame_size_snippet.bin", "wb"))
>>> context = ParseInfo(
...     parse_code=ParseCodes.end_of_sequence,
...     previous_parse_offset=123,
... )
>>> with Serialiser(writer, context, default_values=default_values) as ser:
...     parse_info(ser, {})
```

The `default_values` lookup should provide a separate set of default values for each context dictionary type. See `vc2_conformance.bitstream.vc2_fixeddicts.fixeddict_default_values` for a complete example.

For arrays/lists of values, the default value provided will be usd to populate array elements and not to provide a default for the list as a whole.

Where a default value is not found in the lookup, a :py;exc:*KeyError* will be thrown as usual. This behaviour allows a partial set of default values to be provided (e.g. providing defaults only for padding values) while still validating that the provided input is correct.

API

class SerDes(*io*, *context=None*)

The base serialiser/deserialiser interface and implementation.

This base implementation includes all but the value writing/reading features of the serialisation and deserialisation process.

Attributes

io [*BitstreamReader* (page 103) or *BitstreamWriter* (page 104)] The I/O interface in use.

***context* (page 101)** [dict or None] Get the top-level context dictionary.

cur_context [dict or None] The context dictionary currently being populated.

bool(*target*)

Reads or writes a boolean (single bit) in a bitstream (as per (A.3.2) read_bool()).

Parameters

target [str] The target for the bit (as a *bool* (page 98)).

Returns

value [bool]

nbits (*target*, *num_bits*)
Reads or writes a fixed-length unsigned integer in a bitstream (as per (A.3.3) read_nbits()).

Parameters

target [str] The target for the value (as an `int`[34]).

num_bits [int] The number of bits in the value.

Returns

value [int]

uint_lit (*target*, *num_bytes*)
Reads or writes a fixed-length unsigned integer in a bitstream (as per (A.3.4) read_uint_lit()). Not to be confused with *uint()* (page 99).

Parameters

target [str] The target for the value (as an `int`[35]).

num_bytes [int] The number of bytes in the value.

Returns

value [int]

bitarray (*target*, *num_bits*)
Reads or writes a fixed-length string of bits from the bitstream as a `bitarray.bitarray`. This may be a more sensible type for holding unpredictably sized non-integer binary values such as padding bits.

Parameters

target [str] The target for the value (as a `bitarray.bitarray`).

num_bits [int] The number of bits in the value.

Returns

value [`bitarray.bitarray`]

bytes (*target*, *num_bytes*)
Reads or writes a fixed-length *bytes* (page 99) string from the bitstream. This is a more convenient alternative to *nbits()* (page 99) or *bitarray()* (page 99) when large blocks of data are to be read but not treated as integers.

Parameters

target [str] The target for the value (as a *bytes* (page 99)).

num_bits [int] The number of *bytes* (not bits) in the value.

Returns

value [*bytes* (page 99)]

uint (*target*)
A variable-length, unsigned exp-golomb integer in a bitstream (as per (A.4.3) read_uint()).

Parameters

target [str] The target for the value (as an `int`[36]).

Returns

value [int]

sint (*target*, *num_bits*)
A variable-length, signed exp-golomb integer in a bitstream (as per (A.4.4) read_sint()).

Parameters

target [str] The target for the value (as an `int`[37]).

Returns

value [int]

byte_align(*target*)

Advance in the bitstream to the next whole byte boundary, if not already on one (as per (A.2.4) byte_align()).

Parameters

target [str] The target for the padding bits (as a `bitarray.bitarray`).

bounded_block_begin(*length*)

Defines the start of a bounded block (as per (A.4.2)). Must be followed by a matching *bounded_block_end()* (page 100).

See also: *bounded_block()* (page 100).

Bits beyond the end of the block are always '1'. If a '0' is written past the end of the block a `ValueError`[38] will be thrown.

Parameters

length [int] The length of the bounded block in bits

bounded_block_end(*target*)

Defines the end of a bounded block (as per (A.4.2)). Must be proceeded by a matching *bounded_block_begin()* (page 100).

Parameters

target [str] The target name for any unused bits (as a `bitarray.bitarray`).

bounded_block(*target*, *length*)

A context manager defining a bounded block (as per (A.4.2)).

See also: *bounded_block_begin()* (page 100).

Example usage:

```
with serdes.bounded_block("unused_bits", 100):
    # ...
```

Parameters

target [str] The target name for any unused bits (as a `bitarray.bitarray`).

length [int] The length of the bounded block in bits

declare_list(*target*)

Declares that the specified target should be treated as a `list`[39]. Whenever this target is used in the future, values will be read/written sequentially from the list.

This method has no impact on the bitstream.

Parameters

target [str] The target name to be declared as a list.

set_context_type(*context_type*)

Set (or change) the type of the current context dictionary.

This method has no impact on the bitstream.

Parameters

context_type [dict[40]-like type] The desired type. If the context is already of the required type, no change will be made. If the context is currently of a different type, it will be passed to the `context_type` constructor and the new type used in its place.

subcontext_enter(*target*)
Creates and/or enters a context dictionary within the specified target of the current context dictionary. Must be followed later by a matching *subcontext_leave()* (page 101).

Parameters

target [str] The name of the target in the current context in which the new subcontext is/will be stored.

subcontext_leave()
Leaves the current nested context dictionary entered by *subcontext_enter()* (page 101). Verifies that the closed dictionary has no unused entries, throwing an appropriate exception if not.

subcontext(*target*)
A Python context manager alternative to ;py:meth:*subcontext_enter* and ;py:meth:*subcontext_leave*.

Example usage:

```
>>> with serdes.subcontext("target"):
...     # ...
```

Exactly equivalent to:

```
>>> serdes.subcontext_enter("target"):
>>> # ...
>>> serdes.subcontext_leave():
```

(But without the possibility of forgetting the *subcontext_leave()* (page 101) call).

Parameters

target [str] The name of the target in the current context in which the new subcontext is/will be stored.

computed_value(*target*, *value*)
Places a value into the named target in the current context, without reading or writing anything into the bitstream. Any existing value in the context will be overwritten.

This operation should be used sparingly to embed additional information in a context dictionary which might be required to sensibly interpret its contents and which cannot be trivially computed from the context dictionary. For example, the number of transform coefficients in a coded picture depends on numerous computations and table lookups using earlier bitstream values.

Parameters

target [str] The name of the target in the current context to store the value in.

value [any] The value to be stored.

is_target_complete(*target*)
Test whether a target in the current context is complete, i.e. has been fully used up. Returns True if so, False otherwise.

verify_complete()
Verify that all values in the current context have been used and that no bounded blocks or nested contexts have been left over.

Raises

UnusedTargetError

UnclosedNestedContextError

UnclosedBoundedBlockError

property context
Get the top-level context dictionary.

path(*target=None*)
Produce a 'path' describing the part of the bitstream the parser is currently processing.

If 'target' is None, only includes the path of the current nested context dictionary. If 'target' is a target name in the current target dictionary, the path to the last-used target will be included.

A path might look like:

```
['source_parameters', 'frame_size', 'frame_width']
```

describe_path(*target=None*)
Produce a human-readable description of the part of the bitstream the parser is currently processing.

If 'target' is None, prints only the path of the current nested context dictionary. If 'target' is a target name in the current target dictionary, this will be included in the string.

As a sample, a path might look like the following:

```
SequenceHeader['source_parameters']['frame_size']['frame_width']
```

class Serialiser(*io*, *context=None*, *default_values={}*)
Bases: *vc2_conformance.bitstream.serdes.SerDes* (page 98)

A bitstream serialiser which, given a populated context dictionary, writes the corresponding bitstream.

Parameters

io [*BitstreamWriter* (page 104)]

context [dict]

class Deserialiser(*io*, *context=None*)
Bases: *vc2_conformance.bitstream.serdes.SerDes* (page 98)

A bitstream deserialiser which creates a context dictionary based on a bitstream.

Parameters

io [*BitstreamReader* (page 103)]

class MonitoredSerialiser(*monitor*, **args*, ***kwargs*)
Bases: vc2_conformance.bitstream.serdes.MonitoredMixin, *vc2_conformance.bitstream.serdes.Serialiser* (page 102)

Like *Serialiser* (page 102) but takes a 'monitor' function as the first constructor argument. This function will be called every time bitstream value has been serialised (written).

Parameters

monitor [callable(ser, target, value)] A function which will be called after every primitive I/O operation completes. This function is passed this *MonitoredSerialiser* (page 102) instance and the target name and value of the target just serialised.

This function may be used to inform a user of the current progress of serialisation (e.g. using *SerDes.describe_path()* (page 102) or SerDes.io) or to terminate serialisation early (by throwing an exception).

***args, **kwargs** [see *Serialiser* (page 102)]

[34] https://docs.python.org/3/library/functions.html#int
[35] https://docs.python.org/3/library/functions.html#int
[36] https://docs.python.org/3/library/functions.html#int
[37] https://docs.python.org/3/library/functions.html#int
[38] https://docs.python.org/3/library/exceptions.html#ValueError
[39] https://docs.python.org/3/library/stdtypes.html#list
[40] https://docs.python.org/3/library/stdtypes.html#dict

class MonitoredDeserialiser(*monitor*, **args*, ***kwargs*)

Bases: vc2_conformance.bitstream.serdes.MonitoredMixin, *vc2_conformance.bitstream.serdes.Deserialiser* (page 102)

Like *Deserialiser* (page 102) but takes a 'monitor' function as the first constructor argument. This function will be called every time bitstream value has been deserialised (read).

Parameters

monitor [callable(des, target, value)] A function which will be called after every primitive I/O operation completes. This function is passed this *MonitoredDeserialiser* (page 102) instance and the target name and value of the target just serialised.

This function may be used to inform a user of the current progress of deserialisation (e.g. using *SerDes.context()* (page 101), *SerDes.describe_path()* (page 102) or SerDes.io) or to terminate deserialisation early (by throwing an exception).

***args, **kwargs** [see *Serialiser* (page 102)]

context_type(*dict_type*)

Syntactic sugar. A decorator for *SerDes* (page 98) which uses *SerDes.set_context_type()* (page 100) to set the type of the current context dict:

Example usage:

```
@context_type(FrameSize)
def frame_size(serdes):
    # ...
```

Exactly equivalent to:

```
def frame_size(serdes):
    serdes.set_context_type(FrameSize)
    # ...
```

The wrapped function must take a *SerDes* (page 98) as its first argument.

For introspection purposes, the wrapper function will be given a 'context_type' attribute holding the passed 'dict_type'.

8.5 Low-level bitstream IO

The *vc2_conformance.bitstream.io* (page 103) module contains low-level wrappers for file-like objects which facilitate bitwise read and write operations of the kinds used by VC-2's bitstream format.

The *BitstreamReader* (page 103) and *BitstreamWriter* (page 104) classes provide equivalent methods for the various read_* pseudocode functions defined in the VC-2 specification, along with a few additional utility methods.

Note: These methods are designed to be 'safe' meaning that if out-of-range values are provided an error will be produced (rather than an unexpected value being written/read).

---

class BitstreamReader(*file*)

An open file which may be read one bit at a time.

When the end-of-file is encountered, reads will result in a EOFError[41].

is_end_of_stream()

Check if we've reached the EOF. (A.2.5)

tell()

Report the current bit-position within the stream.

Returns

(bytes, bits) `bytes` is the offset of the current byte from the start of the stream. `bits` is the offset in the current byte (starting at 7 (MSB) and advancing towards 0 (LSB) as bits are read).

seek(*bytes*, *bits=7*)
Seek to a specific (absolute) position in the file.

Parameters

bytes [int] The byte-offset from the start of the file.

bits [int] The bit offset into the specified byte to start reading from.

property bits_remaining
The number of bits left in the current bounded block.

None, if not in a bounded block. Otherwise, the number of unused bits remaining in the block. If negative, indicates the number of bits read past the end of the block.

bounded_block_begin(*length*)
Begin a bounded block of the specified length in bits.

bounded_block_end()
Ends the current bounded block. Returns the number of unused bits remaining, but does not read them or seek past them.

read_bit()
Read and return the next bit in the stream. (A.2.3) Reads '1' for values past the end of file.

read_nbits(*bits*)
Read an 'bits'-bit unsigned integer (like read_nbits (A.3.3)).

read_uint_lit(*num_bytes*)
Read a 'num-bytes' long integer (like read_uint_lit (A.3.4)).

read_bitarray(*bits*)
Read 'bits' bits returning the value as a `bitarray.bitarray`.

read_bytes(*num_bytes*)
Read a number of bytes returning a `bytes`[42] string.

read_uint()
Read an unsigned exp-golomb code (like read_uint (A.4.3)) and return an integer.

read_sint()
Signed version of *read_uint()* (page 104) (like read_sint (A.4.4)).

try_read_bitarray(*bits*)
Attempt to read the next 'bits' bits from the bitstream file, leaving any bounded blocks we might be in if necessary). May read fewer bits if the end-of-file is encountered (but will not throw a `EOFError`[43] like other methods of this class).

Intended for the display of error messages (i.e. as the final use of a *BitstreamReader* (page 103) instance) only since this method may (or may not) exit the current bounded block as a side effect.

class BitstreamWriter(*file*)
An open file which may be written one bit at a time.

is_end_of_stream()
Always True. (A.2.5)

[41] https://docs.python.org/3/library/exceptions.html#EOFError
[42] https://docs.python.org/3/library/stdtypes.html#bytes
[43] https://docs.python.org/3/library/exceptions.html#EOFError

Note: Strictly speaking this should return False when seeking to an earlier part of the stream however this behaviour is not implemented here for simplicity's sake.

tell()
Report the current bit-position within the stream.

Returns

(bytes, bits) `bytes` is the offset of the current byte from the start of the stream. `bits` is the offset in the current byte (starting at 7 (MSB) and advancing towards 0 (LSB) as bits are written).

seek(*bytes*, *bits=7*)
Seek to a specific (absolute) position in the file. Seeking to a given byte will overwrite any bits already set in that byte to 0.

Parameters

bytes [int] The byte-offset from the start of the file.

bits [int] The bit offset into the specified byte to start writing to.

flush()
Ensure all bytes are committed to the file.

property bits_remaining
The number of bits left in the current bounded block.

None, if not in a bounded block. Otherwise, the number of unused bits remaining in the block. If negative, indicates the number of bits read past the end of the block.

bounded_block_begin(*length*)
Begin a bounded block of the specified length in bits.

bounded_block_end()
Ends the current bounded block. Returns the number of unused bits remaining, but does not write them or seek past them.

write_bit(*value*)
Write a bit into the bitstream. If in a bounded block, raises a `ValueError`[44] if a '0' is written beyond the end of the block.

write_nbits(*bits*, *value*)
Write an 'bits'-bit integer. The complement of read_nbits (A.3.3).

Throws an `OutOfRangeError` if the value is too large to fit in the requested number of bits.

write_uint_lit(*num_bytes*, *value*)
Write a 'num-bytes' long integer. The complement of read_uint_lit (A.3.4).

Throws an `OutOfRangeError` if the value is too large to fit in the requested number of bytes.

write_bitarray(*bits*, *value*)
Write the 'bits' from the :py;class:*bitarray.bitarray* 'value'.

Throws an `OutOfRangeError` if the value is longer than 'bits'. The value will be right-hand zero-padded to the required length.

write_bytes(*num_bytes*, *value*)
Write the provided `bytes`[45] or `bytearray`[46] in a python bytestring.

If the provided byte string is too long an `OutOfRangeError` will be raised. If it is too short, it will be right-hand zero-padded.

write_uint(*value*)
Write an unsigned exp-golomb code.

An `OutOfRangeError` will be raised if a negative value is provided.

write_sint(*value*)
Signed version of *write_uint()* (page 105).

The following utility functions are also provided for converting between offsets given as `(bytes, bits)` pairs and offsets given in bytes.

to_bit_offset(*bytes*, *bits=7*)
Convert from a (bytes, bits) tuple (as used by *BitstreamReader.tell()* (page 103) and *BitstreamWriter.tell()* (page 105)) into a total number of bits.

from_bit_offset(*total_bits*)
Convert from a bit offset into a (bytes, bits) tuple (as used by *BitstreamReader.tell()* (page 103) and *BitstreamWriter.tell()* (page 105)).

8.6 Fixeddicts and pseudocode

The *vc2_conformance.bitstream.vc2_fixeddicts* (page 106) module contains *fixeddict* (page 157) definitions for holding VC-2 bitstream values in a hierarchy which strongly mimics the bitstream structure. These names are re-exported in the *vc2_conformance.bitstream* (page 81) module for convenience. See *Deserialised VC-2 bitstream data types* (page 86) for a listing.

It also provides the following metadata structures:

vc2_fixeddict_nesting
A lookup `{fixeddict_type: [fixeddict_type, ...], ...}`.

This lookup enumerates the fixeddicts which may be directly contained by the fixed dictionary types in this module.

This hierarchical information is used by user-facing tools to allow (e.g.) recursive selection of a particular dictionary to display.

vc2_default_values
A lookup `{fixeddict_type: {key: default_value, ...}, ...}`

For each fixeddict type below, provides a sensible default value for each key. The defaults are generally chosen to produce a minimal, but valid, bitstream.

Where a particular fixeddict entry is a list, the value listed in this lookup should be treated as the default value to use for list entries.

> **Warning:** For default values containing a `bitarray.bitarray` or any other mutable type, users must take care to copy the default value before mutating it.

The *vc2_conformance.bitstream.vc2* (page 106) module contains a set of *SerDes* (page 98)-using (see *serdes* (page 93)) functions which follow the pseudo-code in the VC-2 specification as closely as possible. All pseudocode functions are re-exported by the *vc2_conformance.bitstream* (page 81) module.

See the table in *Deserialised VC-2 bitstream data types* (page 86) which relates these functions to their matching *fixeddicts* (page 157).

In this module, all functions are derived from the pseudocode by:

- Replacing `read_*` calls with *SerDes* (page 98) calls.
- Adding *SerDes* (page 98) annotations.
- Removing of decoding functionality (retaining only the code required for bitstream deserialisation).

Consistency with the VC-2 pseudocode is checked by the test suite (see *verification* (page 147)).

[44] https://docs.python.org/3/library/exceptions.html#ValueError
[45] https://docs.python.org/3/library/stdtypes.html#bytes
[46] https://docs.python.org/3/library/stdtypes.html#bytearray

8.7 Autofill

The *vc2_conformance.bitstream.vc2_autofill* (page 107) module provides auto-fill routines for automatically computing certain values for the context dictionaries used by the *vc2_conformance.bitstream.vc2* (page 106) *serdes* (page 93) functions. These values include the picture number and parse offset fields which can't default to a simple fixed value.

In the common case, the *autofill_and_serialise_stream()* (page 107) function may be used to serialise a complete *Stream* (page 86), with sensible defaults provided for all fields (including picture numbers and next/previous parse offsets).

autofill_and_serialise_stream(*file*, *stream*)

Given a `Stream` dictionary describing a VC-2 stream, serialise that into the supplied file.

Parameters

file [file-like object] A file open for binary writing. The serialised bitstream will be written to this file.

stream [`Stream`] The stream to be serialised. Unspecified values will be auto-filled if possible. See *Deserialised VC-2 bitstream data types* (page 86) for the default auto-fill values.

Note: Internally, auto-fill values are taken from *vc2_default_values_with_auto* (page 108).

Supported fields containing the special value *AUTO* (page 108) will be autofilled with suitably computed values. Specifically:

- Picture numbers will set to incrementing values (starting at 0, or continuing from the value used by the previous picture) by *autofill_picture_number()* (page 107).
- The `major_version` field will be populated by *autofill_major_version()* (page 107) and, if appropriate, extended transform parameters fields will be removed.
- Next and previous parse offsets will be calculated automatically by *autofill_parse_offsets()* (page 107) and *autofill_parse_offsets_finalize()* (page 108).

8.7.1 Autofill value routines

The following functions implement autofill routines for specific bitstream values.

autofill_picture_number(*stream*, *initial_picture_number=0*)

Given a `Stream`, find all picture_number fields which are absent or contain the *AUTO* (page 108) sentinel and automatically fill them with consecutive picture numbers. Numbering is restarted for each sequence.

autofill_major_version(*stream*)

Given a `Stream`, find all `major_version` fields which are set to the *AUTO* (page 108) sentinel and automatically set them to the appropriate version number for the features used by this stream.

As a side effect, this function will automatically remove the `ExtendedTransformParameters` field whenever it appears in `TransformParameters` if the major_version evaluates to less than 3. This change will only be made when `major_version` was set to AUTO in a proceeding sequence header, if the field was explicitly set to a particular value, no changes will be made to any transform parameters dicts which follow.

autofill_parse_offsets(*stream*)

Given a *Stream* (page 86), find and fill in all next_parse_offset and previous_parse_offset fields which are absent or contain the *AUTO* (page 108) sentinel.

In many (but not all) cases computing these field values is most straight-forwardly done post serialisation. In these cases, fields in the stream will be autofilled with '0'. These fields should then subsequently be ammended by *autofill_parse_offsets_finalize()* (page 108).

autofill_parse_offsets_finalize(*bitstream_writer*, *stream*, *next_parse_offsets_to_autofill*, *previous_parse_offsets_to_autofill*)

Finalize the autofillling of next and previous parse offsets by directly modifying the serialised bitstream.

Parameters

bitstream_writer [*BitstreamWriter* (page 104)] A *BitstreamWriter* (page 104) set up to write to the already-serialised bitstream.

stream [*Stream* (page 86)] The context dictionary used to serialies the bitstream. Since computed values added to these dictionaries by the serialisation process, it may be necessary to use the dictionary provided by `vc2_conformance.bitstream.Serialiser.context`, rather than the one passed into the Serialiser. This is because the Serialiser may have replaced some dictionaries during serialisation.

next_parse_offsets_to_autofill, previous_parse_offsets_to_autofill The arrays of parse info indices whose next and previous parse offsets remain to be auto-filled.

8.7.2 Autofill value dictionary

vc2_default_values_with_auto

Like `vc2_conformance.bitstreams.vc2_default_values` but with *AUTO* (page 108) set as the default value for all fields which support it.

AUTO

A constant which may be placed in a *vc2_fixeddicts* (page 106) fixed dictionary field to indicate that the various `autofill_*` functions in this module should automatically compute a value for that field.

8.8 Metadata

The *vc2_conformance.bitstream.metadata* (page 108) module contains metadata about the relationship between VC-2 pseudocode functions and deserialised *fixeddict* (page 157) structures. These are used by the *vc2-bitstream-viewer* (page 51) command to produce richer output and during documentation generation.

pseudocode_function_to_fixeddicts

For the subset of pseudocode functions in the VC-2 spec dedicated to serialisation/deserialisation, gives the corresponding fixeddict type in *vc2_conformance.bitstream.vc2_fixeddicts* (page 106).

A dictionary of the shape `{function_name: [type, ...], ...}`.

pseudocode_function_to_fixeddicts_recursive

Like *pseudocode_function_to_fixeddicts* (page 108) but each entry also recursively includes the fixeddict types of all contained entries.

fixeddict_to_pseudocode_function

Provides a mapping from *vc2_conformance.bitstream.vc2_fixeddicts* (page 106) types to the name of the corresponding pseudocode function which may be used to serialise/deserialise from/to it.

CHAPTER

NINE

TEST PICTURE GENERATION REFERENCE

9.1 vc2_conformance.picture_generators: Picture generators

The *vc2_conformance.picture_generators* (page 109) module contains routines for generating video sequences for arbitrary VC-2 video formats.

The picture generators in this module are used to generate encoder and decoder test cases (*vc2_conformance.test_cases* (page 63)).

All picture generator functions take a *VideoParameters* (page 142), and `PictureCodingModes` ([vc2_data_tables], page 4) as their arguments, defining the desired VC-2 video format. They then yield a sequence of `{"Y": [[int, ...], ...], "C1": [[int, ...], ...], "C2": [[int, ...], ...], "pic_num": int}` dictionaries containing integer picture component values, ready for encoding or writing to a file.

The following picture generators are provided.

moving_sprite (*video_parameters*, *picture_coding_mode*, *num_frames=10*)

A video sequence containing a simple moving synthetic image.

This sequence consists of a 128 by 128 pixel sprite (shown below) on a black background which traverses the screen from left-to-right moving 16 pixels to the right every frame (or 8 every field). By default the sequence is 10 frames long.

This test sequence may be used to verify that interlacing, pixel aspect ratio and frame-rate metadata is being correctly reported by a codec for display purposes.

For interlaced formats (see `scan_format` (11.4.5)), sequential fields will contain the sprite at different horizontal positions, regardless of whether pictures are fields or frames (see picture coding mode (11.5)). As a result, when frames are viewed as a set of raw interleaved fields, ragged edges will be visible.

Conversely, for progressive formats, sequential fields contain alternate lines from the same moment in time and when interleaved should produce smooth edges, regardless of the picture coding mode.

In the very first field of the sequence, the left edge of the white triangle will touch the edge of the frame. In interlaced formats, the top line of white pixels in the sprite will always be located on the top field. As a result, the line immediately below should always appear shifted to the right when top-field-first field order is used and shifted to the left when bottom-field-first order is used (see 'top field first parameter' (11.3)).

The sprite should be square with the white triangle having equal height and length and the hypotenuse lying at an angle of 45 degrees. The circular cut-out should be a perfect circle. This verifies that the pictures are displayed with the correct pixel aspect ratio (11.4.7).

The text in the sprite is provided to check that the correct picture orientation has been used.

The colors of the characters 'V', 'C' and '2' are colored saturated primary red, green, and blue for the color primaries used (11.4.10.2). This provides a basic verification that the color components have been provided to the display system and decoded correctly.

static_sprite (*video_parameters*, *picture_coding_mode*)
A video sequence containing a exactly one frame containing a synthetic image.

This sequence consists of a 128 by 128 pixel sprite (shown below) located at the top-left corner of the frame on a black background.

This test sequence may be used to verify that interlacing, pixel aspect ratio and frame-rate metadata is being correctly reported by a codec for display purposes.

For if incorrect field ordering is specified, edges will appear ragged and not smooth.

The sprite should be square with the white triangle having equal height and length and the hypotenuse lying at an angle of 45 degrees. The circular cut-out should be a perfect circle. This verifies that the pictures are displayed with the correct pixel aspect ratio (11.4.7).

The text in the sprite is provided to check that the correct picture orientation has been used.

The colors of the characters 'V', 'C' and '2' are colored saturated primary red, green, and blue for the color primaries used (11.4.10.2). This provides a basic verification that the color components have been provided to the display system and decoded correctly.

mid_gray (*video_parameters*, *picture_coding_mode*)
A video sequence containing exactly one empty mid-gray frame.

'Mid gray' is defined as having each color component set to the integer value exactly half-way along its range. When encoded using VC-2 all transform coefficients will be zero.

The actual color will differ depending on the color model used and the signal offsets specified, though typically a gray color is produced.

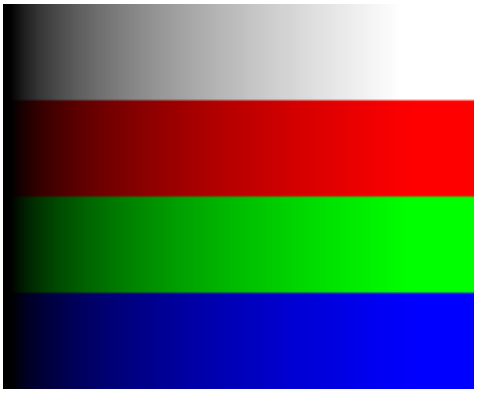
white_noise (*video_parameters*, *picture_coding_mode*, *num_frames=1*, *seed=0*)
A video sequence containing uniformly distributed pseudo-random full-range signal values.

Note: Because all picture components are filled with noise, the resulting pictures will contain 'color' noise rather than black-and-white noise. This may include out-of-gamut signals.

By default a single frame is produced, but the `num_frames` argument may be used to specify more.

By default a fixed seed is used, but alternative seeds may be provided in the `seed` argument.

linear_ramps (*video_parameters*, *picture_coding_mode*)
An video sequence containing exactly one frame with a series of linear color ramps (illustrated below).



The frame is split into horizontal bands which contain, top to bottom:

- A black-to-white linear ramp
- A black-to-red linear ramp
- A black-to-green linear ramp
- A black-to-blue linear ramp

The color ramps are linear in intensity. For most color formats (using a non-linear transfer function) this produces a non-linear ramp in coded pixel values. Further, the ramp will not be perceptually linear since human vision does not have a linear response.

This is provided for the purposes of checking that metadata related to color is correctly passed through for display purposes.

Longer sequences may be produced by repeating the pictures produced by the above generators:

repeat_pictures(*pictures*, *count*)
Repeat a sequence of pictures produced by a picture generator to produce a longer sequence.

Note: Where the picture coding mode indicates that pictures represents fields (not frames), an even number of pictures (i.e. a whole number of frames) is always generated. This means that, for example, a single-frame picture generator will generate *two* pictures for these formats.

9.2 vc2_conformance.dimensions_and_depths: Picture dimension and depth calculation

The *vc2_conformance.dimensions_and_depths* (page 111) module contains the *compute_dimensions_and_depths()* (page 111) function which returns the picture dimensions and bit depths implied by a VC-2 video format.

compute_dimensions_and_depths(*video_parameters*, *picture_coding_mode*)
Compute the dimensions, bit depth and bytes-per-sample of a picture.

Parameters

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

Returns

OrderedDict An ordered dictionary mapping from component name ("Y", "C1" and "C2") to a *DimensionsAndDepths* (page 111) (width, height, depth_bits, bytes_per_sample) namedtuple.

class DimensionsAndDepths(*width*, *height*, *depth_bits*, *bytes_per_sample*)
A set of picture component dimensions and bit depths.

Parameters

width, height [int] The dimensions of the picture.

depth_bits [int] The number of bits per pixel.

bytes_per_sample [int] The number of bytes used to store each pixel value in raw file formats (see *vc2_conformance.file_format* (page 117)).

9.3 vc2_conformance.color_conversion: Color conversion routines

The *vc2_conformance.color_conversion* (page 112) module implements color system related functions relating to the color formats supported by VC-2.

The primary use for this module is to provide routines for converting between colors specified by VC-2's various supported color systems (see Annex (E.1) of the VC-2 specification). This functionality is used during the generation of certain encoder and decoder test cases (*vc2_conformance.test_cases* (page 63)).

9.3.1 High-level API

This module implements simple color format conversion routines for converting between arbitrary VC-2 color formats via floating point CIE XYZ color. The process is implemented by the following high-level functions:

to_xyz(*y*, *c1*, *c2*, *video_parameters*)
Convert a picture from a native VC-2 integer Y C1 C2 format into floating point CIE XYZ format.

Parameters

y, c1, c2 [`numpy.array`] Three 2D `numpy.arrays` containing integer Y C1 C2 values for a picture.

video_parameters [*VideoParameters* (page 142)] The VC-2 parameters describing the video format in use. The following fields are required:

- `color_diff_format_index`
- `luma_offset`
- `luma_excursion`
- `color_diff_offset`
- `color_diff_excursion`
- `color_primaries`
- `color_matrix`
- `transfer_function`

Returns

yxz [`numpy.array`] A 3D `numpy.array` with dimensions `(height, width, 3)` containing floating point CIE XYZ values for a picture.

from_xyz(*xyz*, *video_parameters*)
Convert a picture from CIE XYZ format into a native VC-2 integer, chroma subsampled Y C1 C2 format.

Parameters

yxz [`numpy.array`] A 3D `numpy.array` with dimensions `(height, width, 3)` containing floating point CIE XYZ values for a picture.

video_parameters [*VideoParameters* (page 142)] The VC-2 parameters describing the video format to produce. The following fields are required:

- `color_diff_format_index`
- `luma_offset`
- `luma_excursion`
- `color_diff_offset`
- `color_diff_excursion`
- `color_primaries`
- `color_matrix`
- `transfer_function`

Returns

y, c1, c2 [`numpy.array`] A set of three 2D `numpy.array` containing integer Y C1 C2 values for a picture. If chroma subsampling is used, the C1 and C2 arrays may differ in size from the Y component.

Warning: Color format conversion is an extremely complex problem. The approach used by this module is simplistic in both its approach and implementation. While it will always produce plausible colors, it may not produce the best possible result. To give a few examples of limitations of this module:

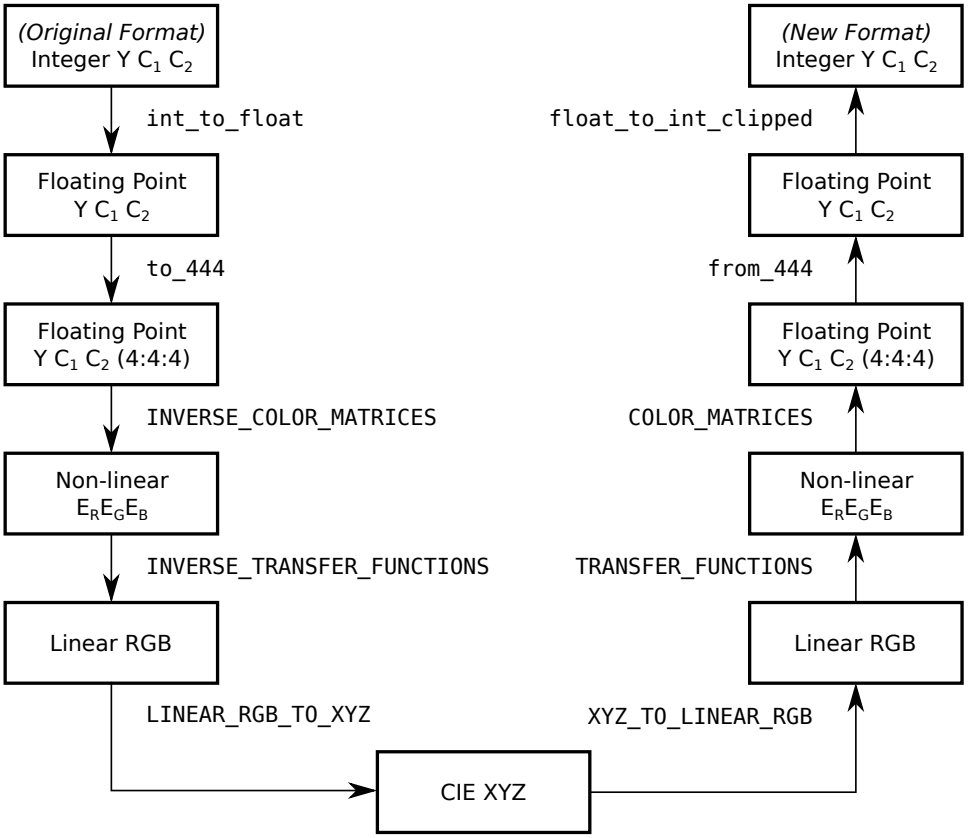
- Potential numerical stability issues are ignored (e.g. YCgCo conversions may be lossy)
- No white point correction is applied
- Out-of-gamut colors are crudely clipped
- Poor quality antialiasing filters for chroma subsampling/interpolation

Finally, this module should be considered a 'best effort' at a correct implementation and the resulting color conversion should largely be treated as informative.

Warning: Support *to_xyz()* (page 112) is limited to only formats using the `tv_gamma` transfer function. All formats are supported, however, by *from_xyz()* (page 112).

9.3.2 Low-level API

The conversion processes used by *to_xyz()* (page 112) and *from_xyz()* (page 112) is built on a series of lower-level transformations as described by the figure below. These lower-level primitives may be used directly to perform more specialised conversions.



These steps build on the following conversion functions and matrices. These are implemented based on the specifications cited by the VC-2 specification.

float_to_int_clipped(*a*, *offset*, *excursion*)
Convert (an array of) float sample values in the nominal range 0 to +1 or -0.5 to +0.5 to integers (with the specified offset and excursion).

Values which fall outside the range of the integer representation are clipped.

float_to_int(*a*, *offset*, *excursion*)
Convert (an array of) float sample values in the nominal range 0 to +1 or -0.5 to +0.5 to integers (with the specified offset and excursion).

Values which fall outside the range of the integer representation are *not* clipped. See *float_to_int_clipped()* (page 114).

int_to_float(*a*, *offset*, *excursion*)
Convert (an array of) integer sample values from integers (with the specified offset and excursion) to floating point values nominally in the range 0 to +1 or -0.5 to +0.5.

from_444(*chroma*, *subsampling*)
Subsample a chroma picture component into the specified `ColorDifferenceSamplingFormats` ([vc2_data_tables], page 4).

> **Warning:** This function uses an extremely crude low-pass filter during downsampling which is likely to produce aliasing artefacts. As such, pictures produced by this function should not be used for anything where high fidelity is required.

to_444 (*chroma*, *subsampling*)
Given a chroma picture subsampled according to the specified `ColorDifferenceSamplingFormats` ([vc2_data_tables], page 4), return an upsampled chroma signal.

> **Warning:** This function uses an extremely crude anti-aliasing filter during upsampling which is likely to produce artefacts. As such, pictures produced by this function should not be used for anything where high fidelity is required.

COLOR_MATRICES = {<color matrix index>: <3x3 matrix>, ...}
For each color matrix supported by VC-2, a 3×3 matrix which transforms from non-linear RGB ($E_RE_GE_B$) to Y C1 C2.

INVERSE_COLOR_MATRICES = {<color matrix index>: <3x3 matrix>, ...}
For each color matrix supported by VC-2, a 3×3 matrix which transforms from Y C1 C2 to non-linear RGB ($E_RE_GE_B$).

TRANSFER_FUNCTIONS = {<transfer function index>: <function>, ...}
For each set of VC-2's, supported transfer functions, a Numpy implementation of that function. These functions implement the transform from linear to non-linear RGB, $E_RE_GE_B$. These functions expect and returns a single value or Numpy array of values.

INVERSE_TRANSFER_FUNCTIONS = {<transfer function index>: <function>, ...}
For (a subset of) VC-2's, supported transfer functions, a Numpy implementation of the inverse function. These functions implement the transform from non-linear to linear RGB. These functions expect and returns a single value or Numpy array of values.

> **Warning:** An inverse transfer function is currently only provided for `tv_gamma` because this is all that was required at the time of development.

XYZ_TO_LINEAR_RGB = {<color primaries index>: <3x3 matrix>, ...}
For each set of color primaries in `PresetColorPrimaries` ([vc2_data_tables], page 5), a 3×3 matrix which converts from CIE XYZ into linear RGB.

LINEAR_RGB_TO_XYZ = {<color primaries index>: <3x3 matrix>, ...}
For each set of color primaries in `PresetColorPrimaries` ([vc2_data_tables], page 5), a 3×3 matrix which converts from linear RGB into CIE XYZ.

9.3.3 Additional utility functions

The following additional utility functions are provided for the manual evaluation of certain transform steps.

matmul_colors (*matrix*, *array*)
Given a (height, width, 3) 3D array, return a new 3D array where each triple in the first array has been multiplied by the specified 3×3 matrix.

swap_primaries (*xyz*, *video_parameters_before*, *video_parameters_after*)
Given an image defined in terms of one set of primaries, return a new image defined in terms of a different set of primaries but with the same numerical R, G and B values under the new set of primaries.

This transformation is useful when an image is defined not by absolute colors but rather colors relative to whatever primaries are in use. For example, a test pattern designed to show swatches of pure color primaries may be given relative to a particular set of primaries but needs to be adapted for use with another set of primaries.

Parameters

xyz [3×3 array (height, width, 3)]

video_parameters_before [*VideoParameters* (page 142)]

video_parameters_after [*VideoParameters* (page 142)]

Returns

xyz [3×3 array (height, width, 3)]

9.3.4 Color parameter sanity checking

The *sanity_check_video_parameters()* (page 116) function is provided which can check a given VC-2 video format is 'sane' – that is it might plausibly be able to represent some colors.

sanity_check_video_parameters(*video_parameters*)

Given a set of *VideoParameters* (page 142), check that a set of video parameters could plausibly be used to encode a color signal (regardless of whether the color specification itself is sensible).

Specifically, the following checks are carried out:

- Are the luma and color difference signals at least 8 bits?
- Can white, black and saturated primary red, green and blue be encoded?
- **When the RGB color matrix is used:**
  - Is the color difference sampling mode 4:4:4?
  - Are the luma and chroma components the same depth?

Returns a *ColorParametersSanity* (page 116) as a result.

class ColorParametersSanity(*luma_depth_sane=True*, *color_diff_depth_sane=True*, *black_sane=True*, *white_sane=True*, *red_sane=True*, *green_sane=True*, *blue_sane=True*, *color_diff_format_sane=True*, *luma_vs_color_diff_depths_sane=True*)

Result of *sanity_check_video_parameters()* (page 116). Indicates the sanity (or insanity) of a set of video parameters.

Truthy if sane, falsey otherwise.

Use the various properties to determine what is and is not sane.

Use the *explain()* (page 117) function to return a string with a human readable explanation.

property luma_depth_sane

True iff the luma component has been assigned at least 8 bits.

property color_diff_depth_sane

True iff the color difference components have been assigned at least 8 bits.

property black_sane

If True, the format can represent (video) black.

property white_sane

If True, the format can represent (video) white.

property red_sane

If True, the format can represent (video) primary red.

property green_sane

If True, the format can represent (video) primary green.

property blue_sane

If True, the format can represent (video) primary blue.

property color_diff_format_sane

True iff the color difference sampling format is appropriate for the color format.

False when non-4:4:4 sampling is used for RGB formats.

property luma_vs_color_diff_depths_sane
True iff the relative offsets/excursions of luma and color difference components are appropriately matched.

False when not identical for RGB formats.

explain()
Return a human-readable explanation of why a video format is not sane (or simply state that it is sane, if it is).

9.4 vc2_conformance.file_format: Picture file format I/O

The *vc2_conformance.file_format* (page 117) module contains functions for reading and writing raw pictures and their metadata as files.

Picture and metadata files must come in pairs with the raw planar picture data file having the extension '.raw' and the metadata file having the extension '.json'. See *Video file format* (page 8) for a description of the file format.

The following functions may be used to read and write picture/metadata files:

read(*filename*)
Read a picture from a data and metadata file.

Convenience wrapper around *read_picture()* (page 118) and *read_metadata()* (page 117).

Parameters

filename [str] The filename of either the picture data file (.raw) or metadata file (.json). The name of the other file will be inferred automatically.

Returns

picture [{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}]

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

write(*picture*, *video_parameters*, *picture_coding_mode*, *filename*)
Write a picture to a data and metadata file.

Convenience wrapper around *write_picture()* (page 118) and *write_metadata()* (page 118).

Parameters

picture [{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}]

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

filename [str] The filename of either the picture data file (.raw) or metadata file (.json). The name of the other file will be inferred automatically.

The above functions are just wrappers around the following functions which read and write picture and metadata files in isolation:

read_metadata(*file*)
Read a JSON picture metadata file.

Parameters

file [file] A file open for binary reading.

Returns

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

picture_number [int]

read_picture(*video_parameters*, *picture_coding_mode*, *picture_number*, *file*)
Read a picture from a file.

Parameters

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

picture_number [int]

file [file] A file open for binary reading.

Returns

picture [{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}]

write_metadata(*picture*, *video_parameters*, *picture_coding_mode*, *file*)
Write the metadata associated with a decoded picture to a file as JSON.

Parameters

picture [{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}]

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

file [file] A file open for binary writing.

write_picture(*picture*, *video_parameters*, *picture_coding_mode*, *file*)
Write a decoded picture to a file as a planar image.

Parameters

picture [{"Y": [[s, ...], ...], "C1": ..., "C2": ..., "pic_num": int}]

video_parameters [*VideoParameters* (page 142)]

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)]

file [file] A file open for binary writing.

Finally, the following function may be used to get the filenames for both parts of a picture/metadata file pair:

get_metadata_and_picture_filenames(*filename*)
Given either the filename of a saved picture (.raw) or metadata file (.json), return a (metadata_filename, picture_filename) tuple with the names of the two corresponding files.

CHAPTER

TEN

LEVEL CONSTRAINT CHECKING/SOLVING REFERENCE

10.1 vc2_conformance.level_constraints: Level constraint definitions

The *vc2_conformance.level_constraints* (page 119) module contains definitions of constraints imposed on VC-2 bitstreams by the various VC-2 level specifications.

10.1.1 Sequence data unit ordering restrictions

Levels may restrict the ordering or choice of data unit types within a bitstream. These restrictions are described using *symbol_re* (page 122) regular expressions provided in *LEVEL_SEQUENCE_RESTRICTIONS* (page 119).

LEVEL_SEQUENCE_RESTRICTIONS = {level: LevelSequenceRestrictions, ...}
A lookup from `Levels` to *LevelSequenceRestrictions* (page 119) (loaded from `vc2_conformance/level_sequence_restrictions.csv`) describing the restrictions on sequences imposed by each VC-2 level.

class LevelSequenceRestrictions(*sequence_restriction_explanation*, *sequence_restriction_regex*)
Restrictions on sequence orderings for a VC-2 level.

Parameters

sequence_restriction_explanation [str] A human readable explanation of the restriction imposed (informative).

sequence_restriction_regex [str] A regular expression describing the sequence ordering allowed which can be matched using a *Matcher* (page 124). Each symbol is a `ParseCodes` ([vc2_data_tables], page 3) name string.

10.1.2 Coding parameter restrictions

Levels impose various restrictions on bitstream parameters and values. These restrictions are collected into a constraint table (see *constraint_table* (page 128)) in *LEVEL_CONSTRAINTS* (page 119).

LEVEL_CONSTRAINTS = <constraint table>
A constraint table (see *vc2_conformance.constraint_table* (page 128)) loaded from `vc2_conformance/level_constraints.csv`.

Constraints which apply due to levels. Keys correspond to particular bitstream values or properties and are enumerated below:

- **(11.2.1)**
  - `level`: int (from the `Levels` enum)
  - `profile`: int (from the `Profiles` enum)

  - major_version: int
  - minor_version: int
- **(11.1)**
  - base_video_format: int (from the BaseVideoFormats enum)
- **(11.4.3)**
  - custom_dimensions_flag: bool
  - frame_width: int
  - frame_height: int
- **(11.4.4)**
  - custom_color_diff_format_flag: bool
  - color_diff_format_index: int (from the ColorDifferenceSamplingFormats enum)
- **(11.4.5)**
  - custom_scan_format_flag: bool
  - source_sampling: int (from the SourceSamplingModes enum)
- **(11.4.6)**
  - custom_frame_rate_flag: bool
  - frame_rate_index: int (from the PresetFrameRates enum, or 0)
  - frame_rate_numer: int
  - frame_rate_denom: int
- **(11.4.7)**
  - custom_pixel_aspect_ratio_flag: bool
  - pixel_aspect_ratio_index: int (from the PresetPixelAspectRatios enum, or 0)
  - pixel_aspect_ratio_numer: int
  - pixel_aspect_ratio_denom: int
- **(11.4.8)**
  - custom_clean_area_flag: bool
  - clean_width: int
  - clean_height: int
  - left_offset: int
  - top_offset: int
- **(11.4.9)**
  - custom_signal_range_flag: bool
  - custom_signal_range_index: int (from the PresetSignalRanges enum, or 0)
  - luma_offset: int
  - luma_excursion: int
  - color_diff_offset: int
  - color_diff_excursion: int
- **(11.4.10)**

  - `custom_color_spec_flag`: bool
  - `color_spec_index`: int (from the `PresetColorSpecs` enum)
  - `custom_color_primaries_flag`: bool
  - `color_primaries_index`: int (from the `PresetColorPrimaries` enum)
  - `custom_color_matrix_flag`: bool
  - `color_matrix_index`: int (from the `PresetColorMatrices` enum)
  - `custom_transfer_function_flag`: bool
  - `transfer_function_index`: int (from the `PresetTransferFunctions` enum)
- **(11.1)**
  - `picture_coding_mode`: int (from the `PictureCodingModes` enum)
- **(12.4.1)**
  - `wavelet_index`: int (from the `WaveletFilters` enum)
  - `dwt_depth`: int
- **(12.4.4.1)**
  - `asym_transform_index_flag`: bool
  - `wavelet_index_ho`: int (from the `WaveletFilters` enum)
  - `asym_transform_flag`: bool
  - `dwt_depth_ho`: int
- **(12.4.5.2)**
  - `slices_x`: int (giving the allowed number of slices in the x dimension)
  - `slices_y`: int (giving the allowed number of slices in the y dimension)
  - `slices_have_same_dimensions`: bool. True iff all slices contain exactly the same number of transform components.
  - `slice_bytes_numerator`: int
  - `slice_bytes_denominator`: int
  - `slice_prefix_bytes`: int
  - `slice_size_scaler`: int
- **(12.4.5.3)**
  - `custom_quant_matrix`: bool
  - `quant_matrix_values`: int (giving the allowed values within a custom quantisation matrix).
- **(13.5.3)**
  - `qindex`: int (the allowed qindex values as defined by individual slices)
- **(13.5.3.2)**
  - `total_slice_bytes`: int (total number of bytes allowed in a high quality picture slice, including all prefix bytes and slice size fields.

See also: *LEVEL_CONSTRAINT_ANY_VALUES* (page 121).

LEVEL_CONSTRAINT_ANY_VALUES = {key: ValueSet, ...}

For keys in *LEVEL_CONSTRAINTS* (page 119) which may hold *AnyValue* (page 130), defines an explicit *ValueSet* (page 129) defining all valid values, for example when the key refers to an enumerated value. Where the range of allowed values is truly open ended, no value is provided in this dictionary.

10.2 vc2_conformance.symbol_re: Regular expressions for VC-2 sequences

The *vc2_conformance.symbol_re* (page 122) module contains a regular expression matching system for sequences of data unit types in VC-2 bitstreams.

This module has two main applications: checking the order of data units during validation and generating valid sequences during bitstream generation.

During bitstream validation (see *vc2_conformance.decoder* (page 69)) the validator must check that sequences contain the right pattern of data unit types. For example, some levels might require bitstreams to include a sequence header between each picture while others may require fragmented and non-fragmented picture types are not used in the same stream. These rules are described by regular expressions which are evaluated by this module.

During bitstream generation, the encoder (see *vc2_conformance.encoder* (page 75)) must produce sequences conforming to the above patterns. To facilitate this, this module can also *generate* sequence orderings which fulfil the rules described by regular expressions.

10.2.1 Regular expressions of symbols

Unlike string-matching regular expression libraries (e.g. `re`[47]) which match sequences of characters, this module is designed to match sequences of 'symbols' where each symbol is just a string like `"sequence_header"` or `"high_quality_picture"`.

As an example, we might write the following regular expression to describe the rule 'all sequences must start with a sequence header and end with an end of sequence data unit':

```
sequence_header .* end_of_sequence
```

In the regular expression syntax used by this module, whitespace is ignored but otherwise follows the typical form for regular expressions. For example, `.` is a wildcard which matches any symbol and `*` means 'zero or more occurrences of the previous pattern'.

This regular expression would match the following sequence of symbols (data unit type names):

```
"sequence_header"
"high_quality_picture"
"sequence_header"
"high_quality_picture"
"end_of_sequence"
```

But would not match the following sequence (because it does not start with a sequence header):

```
"high_quality_picture"
"sequence_header"
"high_quality_picture"
"end_of_sequence"
```

[47] https://docs.python.org/3/library/re.html#module-re

10.2.2 Matching VC-2 sequences

The *Matcher* (page 124) class implements a regular expression matching state machine, taking the regular expression to be matched as its argument. By convention, we use the parse code names defined in `ParseCodes` ([vc2_data_tables], page 3) as symbols to represent each type of data unit in a VC-2 sequence.

Data unit parse code	Symbol
0	`sequence_header`
16	`end_of_sequence`
32	`auxiliary_data`
48	`padding_data`
200	`low_delay_picture`
232	`high_quality_picture`
204	`low_delay_picture_fragment`
236	`high_quality_picture_fragment`

In this example below we'll create a *Matcher* (page 124) which checks if a sequence consists of alternating sequence headers and high quality picture data units:

```
>>> from vc2_conformance.symbol_re import Matcher

>>> m = Matcher("(sequence_header high_quality_picture)* end_of_sequence")
```

This *Matcher* (page 124) instance is then used to check if a sequence matches the required pattern by feeding it symbols one at a time via the *match_symbol()* (page 125) method. This returns True so long as the sequence matches to expression:

```
>>> m.match_symbol("sequence_header")
True
>>> m.match_symbol("high_quality_picture")
True
>>> m.match_symbol("sequence_header")
True
>>> m.match_symbol("high_quality_picture")
True
```

When we reach the end of the sequence, the *is_complete()* (page 125) method will check to see if the regular expression has completely matched the sequence (i.e. it isn't expecting any other data units):

```
>>> # Missing the required end_of_sequence!
>>> m.is_complete()
False

>>> # Now complete
>>> m.match_symbol("end_of_sequence")
True
>>> m.is_complete()
True
```

When a non-matching sequence is encountered, the *Matcher.valid_next_symbols()* (page 125) method may be used to enumerate which symbols the regular expression matcher was expecting. For example:

```
>>> # NB: Each Matcher can only be used once so we must create a new one
>>> # for this new sequence!
>>> m = Matcher("(sequence_header high_quality_picture)* end_of_sequence")

>>> m.match_symbol("sequence_header")
True
>>> m.match_symbol("high_quality_picture")
True
```

(continues on next page)

(continued from previous page)

```
>>> m.match_symbol("high_quality_picture")  # Not allowed!
False
>>> m.valid_next_symbols()
{"sequence_header", "end_of_sequence"}
```

10.2.3 Generating VC-2 sequences

This module provides the *make_matching_sequence()* (page 125) function which can generate minimal sequences matching a set of regular expressions. For example, say we wish to generate a sequence containing three pictures matching the regular expression we used in our previous example:

```
>>> from vc2_conformance.symbol_re import make_matching_sequence

>>> from pprint import pprint
>>> pprint(make_matching_sequence(
...     ["high_quality_picture"]*3,
...     "(sequence_header high_quality_picture)* end_of_sequence",
... ))
['sequence_header',
 'high_quality_picture',
 'sequence_header',
 'high_quality_picture',
 'sequence_header',
 'high_quality_picture',
 'end_of_sequence']
```

Here, the sequence headers and the final end of sequence data units have been added automatically.

10.2.4 API

class Matcher(*pattern*)

Test whether a sequence of symbols (alpha-numeric strings with underscores, e.g. `"foo"` or `"bar_123"`) conforms to a pattern described by a regular expression.

match_symbol() (page 125) should be called for each symbol in the sequence. If `False` is returned, the sequence does not match the specified regular expression. *valid_next_symbols()* (page 125) may be used to list what symbols *would* have been allowed at this stage of the sequence. Once the entire sequence has been passed to *match_symbol()* (page 125), *is_complete()* (page 125) should be used to check that a complete pattern has been matched.

Note: Each instance of *Matcher* (page 124) can only be used to match a single sequence. To match another sequence a new *Matcher* (page 124) must be created.

Parameters

pattern [str] The regular expression describing the pattern this *Matcher* (page 124) should match.

Regular expressions may be specified with a syntax similar (but different to) that used by common string-matching regular expression libraries.

- An alpha-numeric-with-underscore expression matches a single instance of the specified symbol. For example `foo_123` will match the symbol `"foo_123"`.
- A dot (`.`) will match any (singular) symbol.
- A dollar (`$`) will match only at the end of the sequence.

- Two expressions (separated by any amount of whitespace) will match the first expression followed by the second. For example `. foo` will match a sequence `"anything", "foo"`.
- Two expressions separated by a bar (`|`) will match either the first expression or the second expression.
- A question mark (`?`) suffix to an expression will match zero or one instances of that expression. For example `foo?` will match an empty sequence or a single `"foo"`.
- An asterisk (`*`) suffix to an expression will match zero or more instances of that expression. For example `foo*` will match an empty sequence, a sequence of a single `"foo"` symbol or a sequence of many `"foo"` symbols.
- A plus (`+`) suffix to an expression will match one or more instances of that expression. For example `foo+` will match a sequence of a single `"foo"` symbol or a sequence of many `"foo"` symbols.
- Parentheses (`(` and `)`) may be used to group expressions together into a single logical expression.

The expression suffixes bind tightly to their left-hand expression. Beyond this, consider operator precedence undefined: be explicit to help readability!

match_symbol(*symbol*)
Attempt to match the next symbol in the sequence.

Returns True if the symbol matched and False otherwise.

If no symbol was matched, the state machine will not be advanced (i.e. you can try again with a different symbol as if nothing happened).

is_complete()
Is it valid for the sequence to terminate at this point?

valid_next_symbols()
Return the `set`[48] of valid next symbols in the sequence.

If a wildcard is allowed, *WILDCARD* (page 126) will be returned as one of the symbols in addition to any concretely allowed symbols.

If it is valid for the sequence to end at this point, *END_OF_SEQUENCE* (page 126) will be in the returned set.

make_matching_sequence(*initial_sequence*, **patterns*, ***kwargs*)
Given a sequence of symbols, returns a new sequence based on this which matches the supplied set of patterns. The new sequence will be a copy of the supplied sequence with additional symbols inserted where necessary.

Parameters

initial_sequence [[symbol, ...]] The minimal set of entries which must be included in the sequence, in the order they are required to appear.

patterns [str] A series of one or more regular expression specifications (as accepted by *Matcher* (page 124)) which the generated sequence must simultaneously satisfy.

depth_limit [int] Keyword-only argument specifying the maximum number of consecutive symbols to try inserting before giving up on finding a matching sequence. Defaults to 4.

symbol_priority [[symbol, ...]] Keyword-only argument. If supplied, orders possible symbols from most to least preferable. Though this function will always return a sequence of the shortest possible length, when several equal-length sequences would be valid, this argument may be used to influence which is returned. Any symbols not appearing in the list will be given the lowest priority, and sorted in alphabetical order. If

[48] https://docs.python.org/3/library/stdtypes.html#set

this argument is not supplied (or is empty), whenever 'wildcard' value (`.`) is required by a regular expression, the *WILDCARD* (page 126) sentinel will be inserted into the output sequence rather than a concrete symbol.

Returns

matching_sequence [[symbol, ...]] The shortest sequence of symbols which satisfies all of the supplied regular expression patterns. This will contain a superset of the sequence in `initial_sequence`, that is one where additional symbols may have been inserted but none are removed or reordered.

Raises

ImpossibleSequenceError Thrown if no sequence of symbols could be found which matches all of the supplied patterns.

WILDCARD = '.'
A constant representing a wildcard match.

END_OF_SEQUENCE = ''
A constant representing the end-of-sequence.

exception SymbolRegexSyntaxError
Thrown when a regular expression string is provided which could not be parsed.

exception ImpossibleSequenceError
Thrown when *make_matching_sequence()* (page 125) is unable to find a suitable sequence of symbols.

10.2.5 Internals

Beyond the parts exposed by the public API above, this module internally is built on top of two main parts:

- A parser which parses the regular expression syntax accepted by *Matcher* (page 124) into an Abstract Syntax Tree (AST)
- A Non-deterministic Finite-state Automaton (NFA) representation which is constructed from the AST using Thompson's constructions[49].

The parser is broken into two stages: a simple tokenizer/lexer (*tokenize_regex()* (page 126)) and a recursive descent parser (*parse_expression()* (page 126)). A utility function combining these steps is provided by *parse_regex()* (page 127).

tokenize_regex(*regex_string*)
A generator which tokenizes a sequence regular expression specification into (token_type, token_value, offset) 3-tuples.

Token types are:

- `"string"` (value is the string)
- `"modifier"` (value is one of `?*+`)
- `"wildcard"` (value is `.`)
- `"end_of_sequence"` (value is `$`)
- `"bar"` (value is `|`)
- `"parenthesis"` (value is one of `()`)

Throws a *SymbolRegexSyntaxError* (page 126) if an invalid character is encountered.

parse_expression(*tokens*)
A recursive-descent parser which parses an Abstract Syntax Tree (AST) from the regex specification.

[49] https://en.wikipedia.org/wiki/Thompson%27s_construction

The parsed tokens will be removed from the provided token list. Tokens are consumed right-to-left making implementing tight binding of modifiers (i.e. '?', '*' and '+') easy.

This function will return as soon as it runs out of tokens or reaches an unmatched opening parenthesis.

Returns an AST node: one of *Symbol* (page 127), *Star* (page 127), *Concatenation* (page 127), *Union* (page 127) or `None`.

parse_regex(*regex_string*)
Parse a sequence regular expression specification into an Abstract Syntax Tree (AST) consisting of None (empty), *Symbol* (page 127), *Star* (page 127) *Concatenation* (page 127) and *Union* (page 127) objects.

The parser outputs an AST constructed from the following elements:

class Symbol(*symbol*)
Leaf AST node for a symbol.

class Star(*expr*)
AST node for a Kleene Star pattern ($\star$).

class Concatenation(*a*, *b*)
AST node for a concatenation of two expressions.

class Union(*a*, *b*)
AST node for a union (`|`) of two expressions.

An NFA can be constructed from an AST using the *NFA.from_ast()* (page 127) class method.

class NFA(*start=None*, *final=None*)
A Non-deterministic Finite-state Automaton (NFA) with a labelled 'start' and 'final' state.

Attributes

start [*NFANode* (page 127)]

final [*NFANode* (page 127)]

classmethod from_ast(*ast*)
Convert a regular expression AST node into a new *NFA* (page 127) object using Thompson's constructions[50].

class NFANode
A node (i.e. state) in a Non-deterministic Finite-state Automaton (NFA).

Attributes

transitions [{symbol: set([*NFANode* (page 127), ...]), ...}] The transition rules from this node.

Empty transitions are listed under the symbol `None` and are always bidirectional.

add_transition(*dest_node*, *symbol=None*)
Add a transition rule from this node to the specified destination.

If no symbols are specified, a (bidirectional) empty transition between the two nodes will be added.

equivalent_nodes()
Iterate over the set of *NFANode* (page 127) nodes connected to this one by only empty transitions (includes this node).

follow(*symbol*)
Iterate over the *NFANodes* (page 127) reachable from this node following the given symbol.

[50] https://en.wikipedia.org/wiki/Thompson%27s_construction

10.3 vc2_conformance.constraint_table: A simple constraints model

The *vc2_conformance.constraint_table* (page 128) module describes a constraint system which is used to describe restrictions imposed by VC-2 levels. See *vc2_conformance.level_constraints.LEVEL_CONSTRAINTS* (page 119) for the actual level constraints table.

10.3.1 Tutorial

Constraint tables enumerate allowed combinations of values as a list of dictionaries containing *ValueSet* (page 129) objects. Each dictionary describes a valid combination of values. In the contrived running example below we'll define valid food-color combinations (rather than VC-2 codec options):

```
>>> real_foods = [
...     {"type": ValueSet("tomato"), "color": ValueSet("red")},
...     {"type": ValueSet("apple"), "color": ValueSet("red", "green")},
...     {"type": ValueSet("beetroot"), "color": ValueSet("purple")},
... ]
```

We can check dictionaries of values against this permitted list of combinations using *is_allowed_combination()* (page 131):

```
>>> # Allowed combinations
>>> is_allowed_combination(real_foods, {"type": "tomato", "color": "red"})
True
>>> is_allowed_combination(real_foods, {"type": "apple", "color": "red"})
True
>>> is_allowed_combination(real_foods, {"type": "apple", "color": "green"})
True

>>> # Purple apples? I don't think so...
>>> is_allowed_combination(real_foods, {"type": "apple", "color": "purple"})
False
```

But we don't have to check a complete set of values. For example, we can check if a particular color is valid for any foodstuff:

```
>>> is_allowed_combination(real_foods, {"color": "red"})
True
>>> is_allowed_combination(real_foods, {"color": "yellow"})
False
```

This behaviour allows us to detect the first non-constraint-satisfying value when values are obtained sequentially (as they are for a VC-2 bitstream). The bitstream validator (*vc2_conformance.decoder* (page 69)) uses this functionality to check bitstream values conform to the constraints imposed by a specified VC-2 level.

Given an incomplete set of values, we can use *allowed_values_for()* (page 131) to discover what values are permissible for values we've not yet assigned. For example:

```
>>> # If we have an apple, what colors can it be?
>>> allowed_values_for(real_foods, "color", {"type": "apple"})
ValueSet('red', 'green')
```

```
>>> # If we have something red, what might it be?
>>> allowed_values_for(real_foods, "type", {"color": "red"})
ValueSet('apple', 'tomato')
```

This functionality is used by the test case generators and encoder (*vc2_conformance.test_cases* (page 63) and *vc2_conformance.encoder* (page 75)) to discover combinations of bitstream features which satisfy particular level requirements.

10.3.2 ValueSet

class ValueSet(**values_and_ranges*)

Represents a set of allowed values. May consist of anything from a single value, a range of values or a combination of several of these.

__init__(**values_and_ranges*)

Create a *ValueSet* (page 129) containing the specified set of values:

```
>>> no_values = ValueSet()
>>> 100 in no_values
False

>>> single_value = ValueSet(100)
>>> 100 in single_value
True
>>> 200 in single_value
False

>>> range_of_values = ValueSet((10, 20))
>>> 9 in range_of_values
False
>>> 10 in range_of_values
True
>>> 11 in range_of_values
True
>>> 20 in range_of_values  # NB: Range is inclusive
True
>>> 21 in range_of_values
False

>>> many_values = ValueSet(100, 200, (300, 400))
>>> 100 in many_values
True
>>> 200 in many_values
True
>>> 300 in many_values
True
>>> 350 in many_values
True
>>> 500 in many_values
False

>>> non_numeric = ValueSet("foo", "bar", "baz")
>>> "foo" in non_numeric
True
>>> "nope" in non_numeric
False
```

Parameters

***values_and_ranges** [value, or (lower_value, upper_value)] Sets the initial set of values and (inclusive) ranges to be matched

add_value(*value*)

Add a single value to the set.

add_range(*lower_bound*, *upper_bound*)

Add the range of values between the two inclusive bounds to the set.

__contains__(*value*)

Test if a value is a member of this set. For example:

```
>>> value_set = ValueSet(1, 2, 3)
>>> 1 in value_set
True
>>> 100 in value_set
False
```

is_disjoint(*other*)
Test if this *ValueSet* (page 129) is disjoint from another – i.e. they share no common values.

__eq__(*other*)
Return self==value.

__add__(*other*)
Combine two *ValueSet* (page 129) objects into a single object containing the union of both of their values.

For example:

```
>>> a = ValueSet(123)
>>> b = ValueSet((10, 20))
>>> a + b
ValueSet(123, (10, 20))
```

__iter__()
Iterate over the values and (lower_bound, upper_bound) tuples in this value set in no particular order.

iter_values()
Iterate over the values (including the enumerated values of ranges) in this value set in no particular order.

__str__()
Produce a human-readable description of the permitted values.

For example:

```
>>> print(ValueSet())
{<no values>}
>>> print(ValueSet(1, 2, 3, (10, 20)))
{1, 2, 3, 10-20}
```

class AnyValue
Like *ValueSet* (page 129) but represents a 'wildcard' set of values which contains all possible values.

10.3.3 Constraint tables

A 'constraint table' is defined as a list of 'allowed combination' dictionaries.

An 'allowed combination' dictionary defines a *ValueSet* (page 129) for every permitted key.

For example, the following is a constraint table containing three allowed combination dictionaries:

```
>>> real_foods = [
...     {"type": ValueSet("tomato"), "color": ValueSet("red")},
...     {"type": ValueSet("apple"), "color": ValueSet("red", "green")},
...     {"type": ValueSet("beetroot"), "color": ValueSet("purple")},
... ]
```

A set of values satisfies a constraint table if there is at least one allowed combination dictionary which contains the specified combination of values. For example, the following two dictionaries satisfy the constraint table:

```
{"type": "apple", "color": "red"}

{"color": "red"}
```

The first satisfies the constraint table because the combination of values given appears in the second entry of the constraint table.

The second satisfies the constraint table because, even though it does not define a value for every key, the key it does define is included in both the first and second entries.

Meanwhile, the following dictionaries do *not* satisfy the constraint table:

```
{"type": "apple", "color": "purple"}

{"type": "beetroot", "color": "purple", "pickleable": True}
```

The first of these contains values which, in isolation, would be permitted by the second and third entries of the table but which are not present together in any table entries. Consequently, this value does not satisfy the table.

The second contains a 'pickleable' key which is not present in any of the allowed combinations in constraint table and so does not satisfy the table.

The functions below may be used to interrogate a constraint table.

filter_constraint_table(*constraint_table*, *values*)
Return the subset of `constraint_table` entries which match all of the values in `values`. That is, with the entries whose constraints are not met by the provided values removed.

is_allowed_combination(*constraint_table*, *values*)
Check to see if the `values` dictionary holds an allowed combination of values according to the provided constraint table.

Note: A candidate containing only a subset of the keys listed in the constraint table is allowed if the fields it does define are a permitted combination.

Parameters

constraint_table: [{key: :py:class:`ValueSet`, ...}, ...]

values [{key: value}]

allowed_values_for(*constraint_table*, *key*, *values={}*, *any_value=AnyValue()*)
Return a *ValueSet* (page 129) which matches all allowed values for the specified key, given the existing values defined in `values`.

Parameters

constraint_table [[{key: *ValueSet* (page 129), ...}, ...]]

key [key]

values [{key: value, ...}] Optional. The values already chosen. (Default: assume nothing chosen).

any_value [*ValueSet* (page 129)] Optional. If provided and *AnyValue* (page 130) is allowed, this value will be substituted instead. This may be useful when *AnyValue* (page 130) is being used as a short-hand for a more concrete set of values.

10.3.4 CSV format

A Constraint table can be read from CSV files using the following function:

read_constraints_from_csv(*csv_filename*)
Reads a table of constraints from a CSV file.

The CSV file should be arranged with each row describing a particular value to be constrained and each column defining an allowed combination of values.

Empty rows and rows containing only '#' prefixed values will be skipped.

The first column will be treated as the keys being constrained, remaining columns should contain allowed combinations of values. Each of these values will be converted into a *ValueSet* (page 129) as follows:

- Values which contain integers will be converted to `int`
- Values which contain 'TRUE' or 'FALSE' will be converted to `bool`
- Values containing a pair of integers separated by a − will be treated as an incusive range.
- Several comma-separated instances of the above will be combined into a single *ValueSet* (page 129). (Cells containing comma-separated values will need to be enclosed in double quotes (`"`) in the CSV).
- The value 'any' will be substituted for *AnyValue* (page 130).
- Empty cells will be converted into empty *ValueSets* (page 129).
- Cells which contain only a pair of quotes (e.g. `""`, i.e. ditto) will be assigned the same value as the column to their left. (This is encoded using four double quotes (`""""`) in CSV format).

The read constraint table will be returned as a list of dictionaries (one per column) as expected by the functions in *vc2_conformance.constraint_table* (page 128).

CHAPTER

ELEVEN

VC2_CONFORMANCE.PSEUDOCODE: VC-2 PSEUDOCODE FUNCTION IMPLEMENTATIONS

The *vc2_conformance.pseudocode* (page 133) module contains implementations of many of the VC-2 pseudocode functions and associated data structures. In particular, it includes everything which can be used as-is in the construction of encoders, decoders and so on without any alterations.

The VC-2 pseudocode is augmented by a modest number of additional functions or data structure fields. For instance, additional fields are added to *State* (page 139) which are used by conformance checking routines and forward wavelet transform routines.

11.1 vc2_conformance.pseudocode.arrays

VC-2 style 2D array functions (5.5).

new_array(**dimensions*)
(5.4.2) Makes an N-dimensional array out of nested lists. Dimensions are given in the same order as they are indexed, e.g. `new_array(height, width)`, like `array[y][x]`.

width(*a*)
(5.5.4)

height(*a*)
(5.5.4)

row(*a*, *k*)
(15.4.1) A 1D-array-like view into a row of a (2D) nested list as returned by *new_array()* (page 133).

class column(*a*, *k*)
(15.4.1) A 1D-array-like view into a column of a (2D) nested list as returned by *new_array()* (page 133).

delete_rows_after(*a*, *k*)
(15.4.5) Delete rows ‘k’ and after in ‘a’.

delete_columns_after(*a*, *k*)
(15.4.5) Delete columns ‘k’ and after in ‘a’.

11.2 vc2_conformance.pseudocode.offsetting

Picture component offsetting routines (15)

This module collects picture component value offsetting functions from (15) and augments these with complementary de-offsetting functions (inferred from, but not defined by the standard).

offset_picture(*state*, *current_picture*)
(15.5) Remove picture value offsets (added during encoding).

Parameters

state [*vc2_conformance.pseudocode.state.State* (page 139)] Where `luma_depth` and `color_diff_depth` are defined.

current_picture [{comp: [[pixel_value, ...], ...], ...}] Will be mutated in-place.

offset_component(*state*, *comp_data*, *c*)
(15.5) Remove picture value offsets from a single component.

remove_offset_picture(*state*, *current_picture*)
Inverse of offset_picture (15.5). Centers picture values around zero.

remove_offset_component(*state*, *comp_data*, *c*)
Inverse of offset_component (15.5). Centers picture values around zero.

Parameters

state [*vc2_conformance.pseudocode.state.State* (page 139)] Where `luma_depth` and `color_diff_depth` are defined.

current_picture [{comp: [[pixel_value, ...], ...], ...}] Will be mutated in-place.

11.3 vc2_conformance.pseudocode.parse_code_functions

Parse code classification functions (Table 10.2)

These functions all take a 'state' dictionary containing at least `parse_code` which should be an int or `ParseCodes` enum value.

is_seq_header(*state*)
(Table 10.2)

is_end_of_sequence(*state*)
(Table 10.2)

is_auxiliary_data(*state*)
(Table 10.2)

is_padding_data(*state*)
(Table 10.2)

is_ld(*state*)
(Table 10.2)

is_hq(*state*)
(Table 10.2)

is_picture(*state*)
(Table 10.2)

is_fragment(*state*)
(Table 10.2)

using_dc_prediction(*state*)
(Table 10.2)

11.4 vc2_conformance.pseudocode.picture_decoding

This module contains the wavelet synthesis filters and associated functions defined in the pseudocode of the VC-2 standard (15).

See also *vc2_conformance.pseudocode.picture_encoding* (page 136).

inverse_wavelet_transform(*state*)
(15.3)

idwt(*state*, *coeff_data*)
(15.4.1)

Inverse Discrete Wavelet Transform.

Parameters

state [*State* (page 139)] A state dictionary containing at least the following:

- `wavelet_index`
- `wavelet_index_ho`
- `dwt_depth`
- `dwt_depth_ho`

coeff_data [{level: {orientation: [[coeff, ...], ...], ...}, ...}] The complete (power-of-two dimensioned) transform coefficient data.

Returns

picture [[[pixel_value, ...], ...]] The synthesized picture.

h_synthesis(*state*, *L_data*, *H_data*)
(15.4.2) Horizontal-only synthesis.

vh_synthesis(*state*, *LL_data*, *HL_data*, *LH_data*, *HH_data*)
(15.4.3) Interleaved vertical and horizontal synthesis.

oned_synthesis(*A*, *filter_index*)
(15.4.4.1) and (15.4.4.3). Acts in-place on 'A'

filter_bit_shift(*state*)
(15.4.2) Return the bit shift for the current horizontal-only filter.

lift1(*A*, *L*, *D*, *taps*, *S*)
(15.4.4.1) Update even, add odd.

lift2(*A*, *L*, *D*, *taps*, *S*)
(15.4.4.1) Update even, subtract odd.

lift3(*A*, *L*, *D*, *taps*, *S*)
(15.4.4.1) Update odd, add even.

lift4(*A*, *L*, *D*, *taps*, *S*)
(15.4.4.1) Update odd, subtract even.

idwt_pad_removal(*state*, *pic*, *c*)
(15.4.5)

offset_picture(*state*, *current_picture*)
(15.5) Remove picture value offsets (added during encoding).

Parameters

state [*vc2_conformance.pseudocode.state.State* (page 139)] Where `luma_depth` and `color_diff_depth` are defined.

current_picture [{comp: [[pixel_value, ...], ...], ...}] Will be mutated in-place.

offset_component(*state*, *comp_data*, *c*)
(15.5) Remove picture value offsets from a single component.

clip_picture(*state*, *current_picture*)
(15.5)

clip_component(*state*, *comp_data*, *c*)
(15.5)

SYNTHESIS_LIFTING_FUNCTION_TYPES = {...}
Lookup from lifting function ID to function implementation for implementing wavelet synthesis.

11.5 vc2_conformance.pseudocode.picture_encoding

This module is the inverse of the *vc2_conformance.pseudocode.picture_decoding* (page 135) module and contains functions for performing wavelet analysis filtering.

This functionality is not specified by the standard but is used to generate simple bitstreams (and test cases) in this software (and its test suite).

picture_encode(*state*, *current_picture*)
Inverse of picture_decode (15.2).

Parameters

state [*vc2_conformance.pseudocode.state.State* (page 139)] A state dictionary containing at least:

- `luma_width`
- `luma_height`
- `color_diff_width`
- `color_diff_height`
- `luma_depth`
- `color_diff_depth`
- `wavelet_index`
- `wavelet_index_ho`
- `dwt_depth`
- `dwt_depth_ho`

The following entries will be added/replaced with the encoded picture.

- `y_transform`
- `c1_transform`
- `c3_transform`

current_picture [{component: [[value, ...], ...], ...}] The picture to be encoded.

forward_wavelet_transform(*state*, *current_picture*)
(15.3)

dwt(*state*, *picture*)
Discrete Wavelet Transform, inverse of idwt (15.4.1)

Parameters

state [*State* (page 139)] A state dictionary containing at least the following:

- `wavelet_index`

- `wavelet_index_ho`
- `dwt_depth`
- `dwt_depth_ho`

picture [[[pixel_value, ...], ...]] The synthesized picture.

Returns

coeff_data [{level: {orientation: [[coeff, ...], ...], ...}, ...}] The complete (power-of-two dimensioned) transform coefficient data.

h_analysis (*state*, *data*)
Horizontal-only analysis, inverse of h_synthesis (15.4.2).

Returns a tuple (L_data, H_data)

vh_analysis (*state*, *data*)
Interleaved vertical and horizontal analysis, inverse of vh_synthesis (15.4.3).

Returns a tuple (LL_data, HL_data, LH_data, HH_data)

oned_analysis (*A*, *filter_index*)
Inverse of oned_synthesis (15.4.4.1) and (15.4.4.3). Acts in-place on 'A'

dwt_pad_addition (*state*, *pic*, *c*)
Inverse of idwt_pad_removal (15.4.5): pads a picture to a size compatible with wavelet filtering at the level specified by the provided *State* (page 139).

Extra values are obtained by copying the final pixels in the existing rows and columns.

remove_offset_picture (*state*, *current_picture*)
Inverse of offset_picture (15.5). Centers picture values around zero.

remove_offset_component (*state*, *comp_data*, *c*)
Inverse of offset_component (15.5). Centers picture values around zero.

Parameters

state [*vc2_conformance.pseudocode.state.State* (page 139)] Where `luma_depth` and `color_diff_depth` are defined.

current_picture [{comp: [[pixel_value, ...], ...], ...}] Will be mutated in-place.

ANALYSIS_LIFTING_FUNCTION_TYPES = {...}
Lookup from lifting function ID to function implementation for implementing wavelet analysis.

11.6 vc2_conformance.pseudocode.quantization

Quantization-related VC-2 pseudocode routines (13.3).

inverse_quant (*quantized_coeff*, *quant_index*)
(13.3.1)

forward_quant (*coeff*, *quant_index*)
(13.3.1) Based on the informative note 1.

quant_factor (*index*)
(13.3.2)

quant_offset (*index*)
(13.3.2)

11.7 vc2_conformance.pseudocode.slice_sizes

Slice size computation functions (13)

All of the functions below take a 'state' argument which should be a dictionary-like object containing the following entries:

- `"luma_width"`
- `"luma_height"`
- `"color_diff_width"`
- `"color_diff_height"`
- `"dwt_depth"`
- `"dwt_depth_ho"`

The *slice_bytes()* (page 138), *slice_top()* (page 138), *slice_bottom()* (page 138), *slice_left()* (page 138), *slice_right()* (page 138) and *slices_have_same_dimensions()* (page 138) functions all additionally require the following entries:

- `"slices_x"`
- `"slices_y"`

Finally *slice_bytes()* (page 138) function requires the following further values:

- `"slice_bytes_numerator"`
- `"slice_bytes_denominator"`

The 'c' or 'comp' arguments should be set to one of the following strings:

- `"Y"`
- `"C1"`
- `"C2"`

The *slices_have_same_dimensions()* (page 138) utility is added beyond the VC-2 pseudocode functions which determines if all slices will contain the same number of samples or not.

subband_width(*state*, *level*, *comp*)
(13.2.3)

subband_height(*state*, *level*, *comp*)
(13.2.3)

slice_bytes(*state*, *sx*, *sy*)
(13.5.3.2) Compute the number of bytes in a low-delay picture slice.

slice_left(*state*, *sx*, *c*, *level*)
(13.5.6.2) Get the x coordinate of the LHS of the given slice.

slice_right(*state*, *sx*, *c*, *level*)
(13.5.6.2) Get the x coordinate of the RHS of the given slice.

slice_top(*state*, *sy*, *c*, *level*)
(13.5.6.2) Get the y coordinate of the top of the given slice.

slice_bottom(*state*, *sy*, *c*, *level*)
(13.5.6.2) Get the y coordinate of the bottom of the given slice.

slices_have_same_dimensions(*state*)
Utility function, not part of the standard. Tests whether all picture slices will have the same dimensions.

11.8 vc2_conformance.pseudocode.state

A *fixeddict* (page 157) for the 'state' object used by the pseudocode in the VC-2 spec.

fixeddict State

The global state variable type.

Entries prefixed with an underscore (_) are not specified by the VC-2 pseudocode but may be used by the conformance software.

Keys

video_parameters [*VideoParameters* (page 142)] Set by (10.4.1) parse_sequence

parse_code [ParseCodes ([vc2_data_tables], page 3)] Set by (10.5.1) parse_info

next_parse_offset [int] Set by (10.5.1) parse_info

previous_parse_offset [int] Set by (10.5.1) parse_info

picture_coding_mode [PictureCodingModes ([vc2_data_tables], page 4)] Set by (11.1) sequence_header

major_version [int] Set by (11.2.1) parse_parameters

minor_version [int] Set by (11.2.1) parse_parameters

profile [Profiles ([vc2_data_tables], page 7)] Set by (11.2.1) parse_parameters

level [Profiles ([vc2_data_tables], page 7)] Set by (11.2.1) parse_parameters

luma_width [int] Set by (11.6.2) picture_dimensions

luma_height [int] Set by (11.6.2) picture_dimensions

color_diff_width [int] Set by (11.6.2) picture_dimensions

color_diff_height [int] Set by (11.6.2) picture_dimensions

luma_depth [int] Set by (11.6.3) video_depth

color_diff_depth [int] Set by (11.6.3) video_depth

picture_number [int] Set by (12.2) picture_header and (14.2) fragment_header

wavelet_index [WaveletFilters ([vc2_data_tables], page 7)] Set by (12.4.1) transform_parameters

dwt_depth [int] Set by (12.4.1) transform_parameters

wavelet_index_ho [WaveletFilters ([vc2_data_tables], page 7)] Set by (12.4.4.1) extended_transform_parameters

dwt_depth_ho [int] Set by (12.4.4.1) extended_transform_parameters

slices_x [int] Set by (12.4.5.2) slice_parameters

slices_y [int] Set by (12.4.5.2) slice_parameters

slice_bytes_numerator [int] Set by (12.4.5.2) slice_parameters

slice_bytes_denominator [int] Set by (12.4.5.2) slice_parameters

slice_prefix_bytes [int] Set by (12.4.5.2) slice_parameters

slice_size_scaler [int] Set by (12.4.5.2) slice_parameters

quant_matrix [{level: {orient: int, ...}, ...}] Set by (12.4.5.3) quant_matrix

quantizer [{level: {orient: int, ...}, ...}] Set by (13.5.5) slice_quantizers

y_transform [{level: {orient: [[int, ...], ...], ...}, ...}] Set by (13.5.6.3) slice_band. The dequantised luma transform data read from the bitstream. A 2D array for each subband, indexed as [level][orient][y][x].

c1_transform [{level: {orient: [[int, ...], ...], ...}, ...}] Set by (13.5.6.3) slice_band and (13.5.6.4) color_diff_slice_band. The dequantised color difference 1 transform data read from the bitstream. A 2D array for each subband, indexed as [level][orient][y][x].

c2_transform [{level: {orient: [[int, ...], ...], ...}, ...}] Set by (13.5.6.3) slice_band and (13.5.6.4) color_diff_slice_band. The dequantised color difference 2 transform data read from the bitstream. A 2D array for each subband, indexed as [level][orient][y][x].

fragment_data_length [int] Set by (14.2) fragment_header

fragment_slice_count [int] Set by (14.2) fragment_header

fragment_x_offset [int] Set by (14.2) fragment_header

fragment_y_offset [int] Set by (14.2) fragment_header

fragment_slices_received [int] Set by (14.4) fragment_data

fragmented_picture_done [int] Set by (14.4) fragment_data

current_picture [{'pic_num': int, 'Y': [[int, ...], ...], 'C1': [[int, ...], ...], 'C2': [[int, ...], ...]}] Set by (15.2) picture_decode, contains the decoded picture.

next_bit [int] Set by (A.2.1) read_*

current_byte [int] Set by (A.2.1) read_*

bits_left [int] Bits left in the current bounded block (A.4.2)

_generic_sequence_matcher [*vc2_conformance.symbol_re.Matcher* (page 124)] Not in spec, used by *vc2_conformance.decoder* (page 69). A *Matcher* (page 124) which checks that the sequence follows the specified general pattern of data units (10.4.1) (e.g. start with a sequence header, end with end of sequence). Other matchers will test for, e.g. level-defined patterns.

_output_picture_callback [function({'pic_num': int, 'Y': [[int, ...], ...], 'C1': [[int, ...], ...], 'C2': [[int, ...], ...]}, *VideoParameters* (page 142), PictureCodingModes ([vc2_data_tables], page 4))] Not in spec, used by *vc2_conformance.decoder* (page 69). A callback function to call when output_picture (15.2) is called. This callback (if defined) will be passed the picture, video parameters and picture coding mode.

_num_pictures_in_sequence [int] Not in spec, used by *vc2_conformance.decoder* (page 69). (10.4.3) and (12.2) A counter of how many pictures have been encountered in the current sequence. Used to determine if all fields have been received (when pictures are fields) and that the earliest fields have an even picture number. Initialised in parse_sequence (10.4.1) and incremented in vc2_conformance.decoder.assertions.assert_picture_number_incremented_as_expected(), called in picture_header (12.2), and fragment_header (14.2).

_last_parse_info_offset [int] Not in spec, used by *vc2_conformance.decoder* (page 69). (10.5.1) parse_info related state. The byte offset of the previously read parse_info block.

_recorded_bytes [bytearray[51]] Not in spec, used by *vc2_conformance.decoder* (page 69). (11.1) sequence_header requires that repeats must be byte-for-byte identical. To facilitate this, the *vc2_conformance.decoder.io.record_bitstream_start()* (page 71) and *vc2_conformance.decoder.io.record_bitstream_finish()* (page 71) functions are used to make a recording. When this entry is absent, read_byte works as usual. If it is a bytearray[52], whenever a byte has been completely read, it will be placed into this array.

_last_sequence_header_bytes [bytearray[53]] Not in spec, used by *vc2_conformance.decoder* (page 69). (11.1) the bitstream bytes of the data which encoded the previous sequence_header in the sequence. Not present if no previous sequence_header has appeared.

_last_sequence_header_offset [int] Not in spec, used by *vc2_conformance.decoder* (page 69). (11.1) the bitstream offset in bytes of the data which encoded the previous sequence_header in the sequence. Not present if no previous sequence_header has appeared.

_expected_major_version [int] Not in spec, used by *vc2_conformance.decoder* (page 69). Used to record the expected major_version (11.2.1) for this sequence according to the constraints listed in (11.2.2). This field is set by `log_version_lower_bound()`.

_last_picture_number_offset [(byte offset, next bit)] Not in spec, used by *vc2_conformance.decoder* (page 69). (12.2) picture_header and (14.2) fragment_header: The offset of the last picture number encountered in the sequence (or absent if no pictures have yet been encountered).

_last_picture_number [int] Not in spec, used by *vc2_conformance.decoder* (page 69). (12.2) picture_header and (14.2) fragment_header: The last picture number encountered in the sequence (or absent if no pictures have yet been encountered).

_picture_initial_fragment_offset [(byte offset, next bit)] Not in spec, used by *vc2_conformance.decoder* (page 69). (14) The offset in the bitstream of the last fragment with fragment_slice_count==0.

_fragment_slices_remaining [int] Not in spec, used by *vc2_conformance.decoder* (page 69). (14) The number of fragment slices which remain to be received in the current picture. Initialised to zero by parse_sequence (10.4.1), reset to the number of slices per picture by initialize_fragment_state (14.3) and decremented whenever a slice is received by fragment_data (14.4).

_file [file-like] The Python file-like object from which the bitstream will be read by read_byte (A.2.1) .

_level_sequence_matcher [*vc2_conformance.symbol_re.Matcher* (page 124)] Not in spec, used by *vc2_conformance.decoder* (page 69). A *Matcher* (page 124) which checks that the sequence follows the pattern dictated by the current level. Populated after the first (11.2.1) parse_parameters is encountered (and should therefore be ignored until that point).

_level_constrained_values [{key: value, ...}] Not in spec, used by *vc2_conformance.decoder* (page 69). A dictionary which will be incrementally populated with values read or computed from the bitstream which are constrained by the level constraints table (see *vc2_conformance.level_constraints.LEVEL_CONSTRAINTS* (page 119) and `vc2_conformance.decoder.assertions.assert_level_constraint()`).

reset_state(*state*)

Reset a *State* (page 139) dictionary to only include values retained between sequences in a VC-2 stream. Modifies the dictionary in place.

[51] https://docs.python.org/3/library/stdtypes.html#bytearray
[52] https://docs.python.org/3/library/stdtypes.html#bytearray
[53] https://docs.python.org/3/library/stdtypes.html#bytearray

11.9 vc2_conformance.pseudocode.vc2_math

Mathematical functions and operators defined in the spec (5.5.3)

intlog2_float(*n*)
(5.5.3) Implemented as described in the spec, requiring floating point arithmetic.

In practice, the alternative implementation in *intlog2()* (page 142) should be used instead since it does not rely on floating point arithmetic (and is therefore faster and has unlimited precision).

intlog2(*n*)
(5.5.3) Implemented via pure integer, arbitrary-precision operations.

sign(*a*)
(5.5.3)

clip(*a*, *b*, *t*)
(5.5.3)

mean(**S*)
(5.5.3)

11.10 vc2_conformance.pseudocode.video_parameters

Video parameter computation functions (11)

The functions in this module make up the purely functional (i.e. non bitstream-reading) logic for computing video parameters. This predominantly consists of preset-loading and component dimension calculation routines.

For functions below which take a 'state' argument, this should be a dictionary-like object (e.g. *State* (page 139)) with the following entries:

- `"frame_width"`
- `"frame_height"`
- `"luma_width"`
- `"luma_height"`
- `"color_diff_width"`
- `"color_diff_height"`
- `"luma_depth"`
- `"color_diff_depth"`
- `"picture_coding_mode"`

fixeddict VideoParameters
(11.4) Video parameters struct.

Keys

frame_width [int] Set by (11.4.3) frame_size

frame_height [int] Set by (11.4.3) frame_size

color_diff_format_index [`ColorDifferenceSamplingFormats` ([vc2_data_tables], page 4)] Set by (11.4.4) color_diff_sampling_format

source_sampling [`SourceSamplingModes` ([vc2_data_tables], page 4)] Set by (11.4.5) scan_format

top_field_first [bool] Set by (11.4.5) set_source_defaults

frame_rate_numer [int] Set by (11.4.6) frame_rate

frame_rate_denom [int] Set by (11.4.6) frame_rate

pixel_aspect_ratio_numer [int] Set by (11.4.7) aspect_ratio

pixel_aspect_ratio_denom [int] Set by (11.4.7) aspect_ratio

clean_width [int] Set by (11.4.8) clean_area

clean_height [int] Set by (11.4.8) clean_area

left_offset [int] Set by (11.4.8) clean_area

top_offset [int] Set by (11.4.8) clean_area

luma_offset [int] Set by (11.4.9) signal_range

luma_excursion [int] Set by (11.4.9) signal_range

color_diff_offset [int] Set by (11.4.9) signal_range

color_diff_excursion [int] Set by (11.4.9) signal_range

color_primaries_index [`PresetColorPrimaries` ([vc2_data_tables], page 5)] Set by (11.4.10.2) color_primaries

color_matrix_index [`PresetColorMatrices` ([vc2_data_tables], page 5)] Set by (11.4.10.3) color_matrix

transfer_function_index [`PresetTransferFunctions` ([vc2_data_tables], page 5)] Set by (11.4.10.4) transfer_function

set_source_defaults(*base_video_format*)
(11.4.2) Create a *VideoParameters* (page 142) object with the parameters specified in a base video format.

set_coding_parameters(*state*, *video_parameters*)
(11.6.1) Set picture coding mode parameter.

picture_dimensions(*state*, *video_parameters*)
(11.6.2) Compute the picture component dimensions in global state.

video_depth(*state*, *video_parameters*)
(11.6.3) Compute the bits-per-sample for the decoded video.

preset_frame_rate(*video_parameters*, *index*)
(11.4.6) Set frame rate from preset.

preset_pixel_aspect_ratio(*video_parameters*, *index*)
(11.4.7) Set pixel aspect ratio from preset.

preset_signal_range(*video_parameters*, *index*)
(11.4.7) Set signal range from preset.

preset_color_primaries(*video_parameters*, *index*)
(11.4.10.2) Set the color primaries parameter from a preset.

preset_color_matrix(*video_parameters*, *index*)
(11.4.10.3) Set the color matrix parameter from a preset.

preset_transfer_function(*video_parameters*, *index*)
(11.4.10.4) Set the transfer function parameter from a preset.

preset_color_spec(*video_parameters*, *index*)
(11.4.10.1) Load a preset color specification.

11.11 vc2_conformance.pseudocode.metadata

The *vc2_conformance.pseudocode.metadata* (page 144) module is used to record the relationship between the functions in the *vc2_conformance* (page 59) software and the pseudocode functions in the VC-2 specification documents. This information has two main uses:

1. To produce more helpful error messages which include cross-references to published specifications.
2. To enable automatic verification that the behaviour of this software exactly matches the published specifications (see *verification* (page 147)).

11.11.1 Implementing pseudocode functions

Functions in the codebase which implement a pseudocode function in the specification must be labelled as such using the *ref_pseudocode()* (page 144) decorator:

```
>>> from vc2_conformance.pseudocode.metadata import ref_pseudocode

>>> @ref_pseudocode
... def parse_info(state):
...     '''(10.5.1) Read a parse_info header.'''
...     byte_align()
...     read_uint_lit(state, 4)
...     state["parse_code"] = read_uint_lit(state, 1)
...     state["next_parse_offset"] = read_uint_lit(state, 4)
...     state["previous_parse_offset"] = read_uint_lit(state, 4)
```

The *ref_pseudocode()* (page 144) decorator will log the existence of the decorated function, along with the reference to the spec at the start of its docstring.

By default, it is implied that a decorated function does *not* deviate from the pseudocode – a fact which is verified by the *verification* (page 147) module in the test suite. When a function does deviate (for example for use in serialisation/deserialisation) or is not defined in the spec, the `deviation` argument should be provided. This is used by the *verification* (page 147) logic to determine how the function will be verified. See *Pseudocode deviations* (page 147) for the allowed values. For example:

```
>>> @ref_pseudocode(deviation="serdes")
... def parse_parameters(serdes, state):
...     '''(11.2.1)'''
...     state["major_version"] = serdes.uint("major_version")
...     state["minor_version"] = serdes.uint("minor_version")
...     state["profile"] = serdes.uint("profile")
...     state["level"] = serdes.uint("level")
```

ref_pseudocode(**args*, ***kwargs*)

Decorator for marking functions which are derived from the VC-2 pseudocode.

Example usage:

```
@ref_pseudocode
def parse_info(state):
    '''(10.5.1) Read a parse_info header.'''
    # ...

@ref_pseudocode("10.5.1", deviation="alternative_implementation")
def parse_info(state):
    '''Some alternative implementation of parse info.'''
```

Parameters

deviation [str or None] If this function deviates from the pseudocode in any way, indicates the nature of this deviation. None indicates that this function exactly matches the pseudocode.

Automatic checks for pseudocode equivalence in the test suite (see *verification* (page 147)) will use this value to determine how exactly this function must match the pseudocode for tests to pass. See *Pseudocode deviations* (page 147) for details of values this field may hold.

document, section [str or None] The name of the document and section defining this function. If None, will be extracted from the function docstring. Defaults to the main VC-2 specification. If not provided, and not found in the docstring, a `TypeError`[54] will be thrown.

When extracted from a docstring, a reference of one of the following forms must be used at the start of the docstring:

- `(1.2.3)`: Reference a section in the main VC-2 spec
- `(SMPTE ST 20402-2:2017: 1.2.3)`: Reference a section in another spec

name [str or None] The name of the pseudocode function this function implements. If None, the provided function's name will be used.

11.11.2 Accessing the metadata

After all submodules of *vc2_conformance* (page 59) have been loaded, the *pseudocode_derived_functions* (page 146) list will be populated with *PseudocodeDerivedFunction* (page 145) instances for every pseudocode derived function annotated with *ref_pseudocode()* (page 144).

class PseudocodeDerivedFunction(*function*, *deviation=None*, *document=None*, *section=None*, *name=None*)

A record of a pseudocode-derived function within this codebase.

Parameters

function [function] The Python function which was derived from a pseudocode function.

deviation [str or None] If this function deviates from the pseudocode in any way, indicates the nature of this deviation. None indicates that this function exactly matches the pseudocode.

Automatic checks for pseudocode equivalence in the test suite (see *verification* (page 147)) will use this value to determine how exactly this function must match the pseudocode for tests to pass. See *Pseudocode deviations* (page 147) for details of values this field may hold.

document, section [str or None] The name of the document and section defining this function. If None, will be extracted from the function docstring. Defaults to the main VC-2 specification. If not provided, and not found in the docstring, a `TypeError`[55] will be thrown.

When extracted from a docstring, a reference of one of the following forms must be used at the start of the docstring:

- `(1.2.3)`: Reference a section in the main VC-2 spec
- `(SMPTE ST 20402-2:2017: 1.2.3)`: Reference a section in another spec

name [str or None] The name of the pseudocode function this function implements. If None, the provided function's name will be used.

[54] https://docs.python.org/3/library/exceptions.html#TypeError

property citation
A human-readable citation string for this pseudocode function.

pseudocode_derived_functions = [PseudocodeDerivedFunction, ...]
The complete set of *PseudocodeDerivedFunctions* (page 145) in this code base, in no particular order.

> **Warning:** This array is only completely populated once every module of *vc2_conformance* (page 59) has been loaded.

11.11.3 Pseudocode tracebacks

The *make_pseudocode_traceback()* (page 146) function may be used to convert a Python traceback into a pseudocode function traceback.

make_pseudocode_traceback(*tb*)
Given a `traceback.extract_tb()`[56] generated traceback description, return a list of *PseudocodeDerivedFunction* (page 145) objects for the stack trace, most recently called last. Entries in the traceback which don't have a corresponding pseudocode-derived function are omitted.

[55] https://docs.python.org/3/library/exceptions.html#TypeError
[56] https://docs.python.org/3/library/traceback.html#traceback.extract_tb

CHAPTER

TWELVE

AUTOMATED STATIC CODE VERIFICATION

The `tests/verification` (page 147) module (and embedded tests) implement automatic checks that the code in `vc2_conformance` (page 59) matches the pseudocode definitions in the VC-2 specification.

The `tests/verification/test_equivalence.py` test script (which is part of the normal Pytest test suite) automatically finds functions in the `vc2_conformance` (page 59) codebase (using `vc2_conformance.pseudocode.metadata` (page 144)) and checks they match the equivalent function in the VC-2 specification (which are copied out verbatim in `tests/verification/reference_pseudocode.py`).

Note: To ensure that `vc2_conformance.pseudocode.metadata` (page 144) contains information about all submodules of `vc2_conformance` (page 59), the `conftest.py` file in this directory ensures all submodules of vc2_conformance are loaded.

12.1 Pseudocode deviations

In some cases, a limited set of well-defined differences are allowed to exist between the specification and the code used in `vc2_conformance` (page 59). For example, docstrings need not match and in some cases, extra changes may be allowed to facilitate, e.g. bitstream deserialisation. The specific comparator used depends on the `deviation` parameter given in the metadata as follows:

- `deviation=None`: `verification.comparators.Identical` (page 151)
- `deviation="serdes"`: `verification.comparators.SerdesChangesOnly` (page 151)

Functions marked with the following additional `deviation` values will not undergo automatic verification, but are used to indicate other kinds of pseudocode derived function:

- `deviation="alternative_implementation"`: An alternative, orthogonal implementation intended to perform the same role as an existing pseudocode function.
- `deviation="inferred_implementation"`: A function whose existence is implied or whose behaviour is explained in prose and therefore has no corresponding pseudocode definition.

12.2 Amendment comments

In some cases it is necessary for an implementation to differ arbitrarily from the standard (i.e. to make 'amendments'). For example, additional type checks may be added or picture decoding functions disabled when not required. Such amendments must be marked by special 'amendment comments' which start with either two or three # characters, as shown by the snippet below:

```
def example_function(a, b):
    # The following lines are not part of the standard and so are marked by
    # an amendment comment to avoid the pseudocode equivalence checking
```

(continues on next page)

(continued from previous page)

```
    # logic complaining about them
    ## Begin not in spec
    if b <= 0:
        raise Exception("'b' cannot be zero or negative!")
    ## End not in spec

    # For single-line snippets which are not in the standard you can use
    # the following end-of-line amendment comment
    assert b > 0  ## Not in spec

    # The following code is part of the standard but is disabled in this
    # example. Even though it is commented out, it will still be checked
    # against the standard. If the standard changes this check ensures that
    # the maintainer must revisit the commented-out code and re-evaluate
    # the suitability of any amendments made.
    ### if do_something(a):
    ###     do_something_else(b)

    return a / b
```

More details of the amendment comment syntax can be found in *verification.amendment_comments* (page 152).

12.3 Internals

This module does *not* attempt to perform general purpose functional equivalence checking – a known uncomputable problem. Instead, comparisons are made at the Abstract Syntax Tree (AST) level. By performing checks at this level semantically insignificant differences (e.g. whitespace and comments) are ignored while all other changes are robustly identified. The Python built-in `ast`[57] module is used to produce ASTs ensuring complete support for all Python language features.

To compare ASTs, this module provides the *verification.node_comparator.NodeComparator* (page 149). Instances of this class can be used to compare pairs of ASTs and report differences between them.

The subclasses in *verification.comparators* (page 151) are similar but allow certain well-defined differences to exist between ASTs. As an example, *verification.comparators.SerdesChangesOnly* (page 151) will allow calls to the `read_*` functions to be swapped for their equivalent *vc2_conformance.bitstream.serdes.SerDes* (page 98) method calls with otherwise identical arguments.

To allow differences between function implementations and the specification, functions are pre-processed according to the amendment comment syntax described above. This preprocessing step is implemented in *verification.amendment_comments* (page 152) and uses the built-in Python `tokenize`[58] module to ensure correct interoperability with all other Python language features.

Finally the *verification.compare* (page 154) module provides the *compare_functions()* (page 154) function which ties all of the above components together and produces human-readable reports of differences between functions.

All of the above functionality is tested by the other `test_*.py` test scripts in this module's directory.

[57] https://docs.python.org/3/library/ast.html#module-ast
[58] https://docs.python.org/3/library/tokenize.html#module-tokenize

12.3.1 verification.node_comparator: AST Comparison Framework

The *NodeComparator* (page 149) class is intended to form the basis of comparison routines which allow controlled differences between two ASTs to be ignored.

For exapmle, the following can be used to compare two ASTs, ignoring docstrings at the start of functions:

```
from itertools import dropwhile

from verification.node_comparator import NodeComparator

class SameExceptDocstrings(NodeComparator):

    def compare_FunctionDef(self, n1, n2):
        def without_docstrings(body):
            return dropwhile(
                lambda e: isinstance(e, ast.Expr) and isinstance(e.value, ast.Str),
                body,
            )

        return self.generic_compare(
            n1, n2, filter_fields={"body": without_docstrings}
        )
```

This can then be used like so:

```
>>> func_1 = "def func(a, b):\n    '''Add a and b'''\n    return a + b"
>>> func_2 = "def func(a, b):\n    return a + b"

>>> import ast
>>> c = SameExceptDocstrings()
>>> c.compare(ast.parse(func_1), ast.parse(func_2))
True
```

NodeComparator API

class NodeComparator

An `ast.AST`[59] visitor object (similar to `ast.NodeVisitor`[60] which simultaneously walks two ASTs, testing them for equivalence.

The *compare()* (page 149) method of instances of this class may be used to recursively compare two AST nodes.

get_row_col()

Find the current row and column offsets for the tokens currently being compared with *compare()* (page 149).

compare(*n1*, *n2*)

Recursively compare two AST nodes.

If `n1` has the type named N1Type and `n2` has the type named N2Type, this function will try to call one of the following methods:

- `compare_N1Type` (if N1Type is the same as N2Type)
- `compare_N1Type_N2_type`
- `compare_N1Type_ANY`
- `compare_ANY_N2Type`
- `generic_compare`

The first method to be found will be called and its return value returned. The various `compare_*` methods may be overridden by subclasses of *NodeComparator* (page 149) and should implement the same interface as this method.

Parameters

n1, n2 [`ast.AST`[61]] The nodes to compare

Returns

result [True or *NodesDiffer* (page 150)] True if the ASTs are equal and *NodesDiffer* (page 150) (which is faslsey) otherwise.

generic_compare(*n1*, *n2*, *ignore_fields=[]*, *filter_fields={}*)

Base implementation of recurisive comparison of two AST nodes.

Compare the type of AST node and recursively compares field values. Recursion is via calls to *compare()* (page 149).

Options are provided for ignoring differences in certain fields of the passed AST nodes. Authors of custom `compare_*` methods may wish to use these arguments when calling *generic_compare()* (page 150) to allow certain fields to differ while still reporting equality.

Parameters

n1, n2 [`ast.AST`[62]] The nodes to compare

ignore_fields [[str, ...]] A list of field names to ignore while comparing the AST nodes.

filter_fields [{fieldname: fn or (fn, fn) ...}] When a list-containing field is encountered, functions may be provided for pre-filtering the entries of the lists being compared. For example, one might supply a filtering function which removes docstrings from function bodies.

Entries in this dictionary may be either:

- Functions which take the list of values and should return a new list of values to use in the comparison. This function must not mutate the list passed to it.
- A pair of functions like the one above but the first will be used for filtering n1's field and the second for n2's field. Either may be None if no filtering is to take place for one of the nodes.

Returns

result [True or *NodesDiffer* (page 150)] True if the ASTs are equal and *NodesDiffer* (page 150) (which is faslsey) otherwise.

`NodesDiffer` types

class NodesDiffer(*n1*, *n1_row_col*, *n2*, *n2_row_col*, *reason=None*)

A result from *NodeComparator* (page 149) indicating that two ASTs differ.

This object is 'falsy' (i.e. calling `bool()` on a *NodeComparator* (page 149) instance returns False).

Attributes

n1, n2 [`ast.AST`[63]] The two ASTs being compared.

n1_row_col, n2_row_col [(row, col) or (None, None)] The row and column of the 'n1' and 'n2' tokens, or the values for the row and column of the nearest token with a known position.

[59] https://docs.python.org/3/library/ast.html#ast.AST
[60] https://docs.python.org/3/library/ast.html#ast.NodeVisitor
[61] https://docs.python.org/3/library/ast.html#ast.AST
[62] https://docs.python.org/3/library/ast.html#ast.AST

reason [str or None] A string describing how the two nodes differ with a human-readable message.

class NodeTypesDiffer(*n1*, *n1_row_col*, *n2*, *n2_row_col*)
A pair of nodes have different types.

class NodeFieldsDiffer(*n1*, *n1_row_col*, *n2*, *n2_row_col*, *field*)
A pair of nodes differ in the value of a particular field.

Attributes

field [str] The field name where the AST nodes differ.

class NodeFieldLengthsDiffer(*n1*, *n1_row_col*, *n2*, *n2_row_col*, *field*, *v1*, *v2*)
A pair of nodes differ in the length of the list of values in a particular field.

Attributes

field [str] The field name where the AST nodes differ.

v1, v2 [list] The values of the fields (after any filtering has taken place).

class NodeListFieldsDiffer(*n1*, *n1_row_col*, *n2*, *n2_row_col*, *field*, *index*, *v1*, *v2*)
A pair of nodes differ in the value of a list field entry.

Attributes

field [str] The field name where the AST nodes differ.

index [int] The index of the value in the v1 and v2 lists.

v1, v2 [list] The values of the fields (after any filtering has taken place).

12.3.2 verification.comparators

A series of *verification.node_comparator.NodeComparator* (page 149) based comparators for checking the equivalence of VC-2 pseudocode functions from the spec with their implementations in the *vc2_conformance* (page 59) package.

class Identical
Bases: *verification.node_comparator.NodeComparator* (page 149)

Compares two function implementations only allowing the following differences:

1. Their docstrings may be different.
2. A *vc2_conformance.pseudocode.metadata.ref_pseudocode()* (page 144) decorator may be used.
3. Constants from the `vc2_data_tables` ([vc2_data_tables], page 3) module may be used in place of their numerical literal equivalents.

class SerdesChangesOnly
Bases: *verification.node_comparator.NodeComparator* (page 149)

Compares two function implementations where the first is a VC-2 pseudocode definition and the second is a function for use with the *vc2_conformance.bitstream.serdes* (page 93) framework. The following differences are allowed:

1. Differing docstrings. (Justification: has no effect on behaviour.)
2. The addition of a *vc2_conformance.pseudocode.metadata.ref_pseudocode()* (page 144) decorator to the second function. (Justification: has no effect on behaviour.)
3. The addition of a *vc2_conformance.bitstream.serdes.context_type()* (page 103) decorator to the second function. (Justification: has no effect on behaviour.)

[63] https://docs.python.org/3/library/ast.html#ast.AST

4. The addition of `serdes` as a first argument to the second function. (Justification: required for use of the serdes framework, has no effect on behaviour.)
5. The wrapping of statements in `with serdes.subcontext` context managers in the second function will be ignored. (Justification: these context managers have no effect on behaviour but are required to set the serdes state.)
6. The addition of the following methods calls in the second function (Justification: these method calls have no effect on behaviour but are required to set the serdes state):
   - *vc2_conformance.bitstream.serdes.SerDes.subcontext_enter()* (page 101)
   - *vc2_conformance.bitstream.serdes.SerDes.subcontext_leave()* (page 101)
   - *vc2_conformance.bitstream.serdes.SerDes.set_context_type()* (page 100)
   - *vc2_conformance.bitstream.serdes.SerDes.declare_list()* (page 100)
   - *vc2_conformance.bitstream.serdes.SerDes.computed_value()* (page 101)
7. The substitution of an assignment to `state.bits_left` with a call to *vc2_conformance.bitstream.serdes.SerDes.bounded_block_begin()* (page 100) in the second function, taking the assigned value as argument. (Justification: this has the equivalent effect in the bitstream I/O system).
8. The following I/O function substitutions in the second function are allowed with an additional first argument (for the target name). (Justification: these functions have the equivalent effect in the bitstream I/O system).
   - `read_bool` -> `serdes.bool`
   - `read_nbits` -> `serdes.nbits`
   - `read_uint_lit` -> `serdes.uint_lit` or `serdes.bytes`
   - `read_uint` or `read_uintb` -> `serdes.uint`
   - `read_sint` or `read_sintb` -> `serdes.sint`
   - `byte_align` -> `serdes.byte_align`
   - `flush_inputb` -> `serdes.bounded_block_end`
9. Substitution of empty dictionary creation for creation of *vc2_conformance.pseudocode.state.State* (page 139) or *vc2_conformance.pseudocode.video_parameters.VideoParameters* (page 142) fixed dicts is allowed. (Justification: These are valid dictionary types, but provide better type checking and pretty printing which is valuable here).

12.3.3 verification.amendment_comments

In the *vc2_conformance* (page 59) module it is sometimes necessary to make amendments to the pseudocode. For example, validity checks may be added or unneeded steps removed (such as performing a wavelet transform while simply deserialising a bitstream).

To make changes made to a VC-2 function implementations explicit, the following conventions are used:

- When code present in the spec is removed or disabled, it is commented out using triple-hash comments like so:

```
def example_1(a, b):
    # Code as-per the VC-2 spec
    c = a + b

```

(continues on next page)

(continued from previous page)

```
    # No need to actually perform wavelet transform
    ### if c > 1:
    ###     wavelet_transform()
    ###     display_picture()

    # More code per the spec
    return c
```

- When code is added, it should be indicated using either a `## Not in spec` double-hash comment:

```
def example_2(a, b):
    assert b > 0  ## Not in spec

    return a / b
```

Or within a `## Begin not in spec`, `## End not in spec` block:

```
def example_2(a, b):
    ## Begin not in spec
    if b < 0:
        raise Exception("Negative 'b' not allowed")
    elif b == 0:
        raise Exception("Can't divide by 'b' when it is 0")
    ## End not in spec

    return a / b
```

To enable the automated verification that implemented functions match the VC-2 spec (e.g. using *verification.comparators* (page 151)), any amendments to the code must first be undone. The *undo_amendments()* (page 153) function may be used to performs this step.

undo_amendments(*source*)

Given a Python source snippet, undo all amendments marked by special 'amendment comments' (comments starting with `##` and `###`).

The following actions will be taken:

- Disabled code prefixed `### ...` (three hashes and a space) will be uncommented.
- Lines ending with a `## Not in spec` comment will be commented out.
- Blocks of code starting with a `## Begin not in spec` line and ending with a `## End not in spec` line will be commented out.

Returns

(source, indent_corrections) The modified source is returned along with a dictionary mapping line number to an indentation correction. These corrections may be used to map column numbers in the new source to column numbers in the old source.

Raises

tokenize.TokenError[64]

***BadAmendmentCommentError* (page 153)**

***UnmatchedNotInSpecBlockError* (page 154)**

***UnclosedNotInSpecBlockError* (page 154)**

The following exception custom exception types are defined:

[64] https://docs.python.org/3/library/tokenize.html#tokenize.TokenError

exception BadAmendmentCommentError(*comment*, *row*, *col*)
An unrecognised amendment comment (a comment with two or more hashes) was found.

Attributes

comment [str] The contents of the comment

row, col [int] The position in the source where the unrecognised comment starts

exception UnmatchedNotInSpecBlockError(*row*)
An 'End not in spec' amendment comment block was encountered without a corresponding 'Begin not in spec' block.

Attributes

row [int] The line in the source where the 'end' comment was encounterd

exception UnclosedNotInSpecBlockError(*row*)
A 'Begin not in spec' amendment comment block was not closed.

Attributes

row [int] The line in the source where the block was started.

12.3.4 verification.compare: Compare function implementations

This module provides a function, *compare_functions()* (page 154), which uses a specified comparator (*verification.comparators* (page 151)) to determine if a given function matches the reference VC-2 pseudocode.

compare_functions(*ref_func*, *imp_func*, *comparator*)
Compare two Python functions where one is a reference implementation and the other an implementation used in *vc2_conformance* (page 59).

Parameters

ref_func [FunctionType] The reference VC-2 implementation of a function.

imp_func [FunctionType] The implementation of the same function used in *vc2_conformance* (page 59). Will be pre-processed using *verification.amendment_comments.undo_amendments()* (page 153) prior to comparison.

comparator [*verification.node_comparator.NodeComparator* (page 149)] The comparator instance to use to test for equivalence.

Returns

True or *Difference* (page 155) True is returned if the implementations are equal (according to the supplied comparator). Otherwise a *Difference* (page 155) is returned enumerating the differences.

Raises

***ComparisonError* (page 155)**

compare_sources(*ref_source*, *imp_source*, *comparator*)
Compare two Python sources, one containing a reference VC-2 pseudocode function and the other containing an implementation.

Parameters

ref_source [str] The reference VC-2 pseudocode implementation of a function.

imp_source [str] The implementation of the same function used in *vc2_conformance* (page 59). Will be pre-processed using *verification.amendment_comments.undo_amendments()* (page 153) prior to comparison.

comparator [*verification.node_comparator.NodeComparator* (page 149)] The comparator instance to use to test for equivalence.

Returns

True or *Difference* (page 155) True is returned if the implementations are equal (according to the supplied comparator). Otherwise a *Difference* (page 155) is returned enumerating the differences.

Raises

***ComparisonError* (page 155)**

class Difference(*message*, *ref_row=None*, *ref_col=None*, *imp_row=None*, *imp_col=None*, *ref_func=None*, *imp_func=None*)

A description of the difference between a reference function and its implementation.

Use str[65] to turn into a human-readable description (including source code snippets).

Parameters

message [str]

ref_row, ref_col [int or None] The position in the reference code where the difference ocurred. May be None if not related to the reference code.

imp_row, imp_col [int or None] The position in the implementation code where the difference ocurred. May be None if not related to the implementation code.

ref_func, imp_func [FunctionType or None] The Python function objects being compared.

exception ComparisonError(*message*, *ref_row=None*, *ref_col=None*, *imp_row=None*, *imp_col=None*, *ref_func=None*, *imp_func=None*)

An error occurred while attempting to compare two function implementations.

Use str[66] to turn into a human-readable description (including source code snippets).

Parameters

message [str]

ref_row, ref_col [int or None] The position in the reference code where the error ocurred. May be None if not related to the reference code.

imp_row, imp_col [int or None] The position in the implementation code where the error ocurred. May be None if not related to the implementation code.

ref_func, imp_func [FunctionType or None] The Python function objects being compared.

[65] https://docs.python.org/3/library/stdtypes.html#str
[66] https://docs.python.org/3/library/stdtypes.html#str

CHAPTER

THIRTEEN

UTILITY MODULE REFERENCES

13.1 vc2_conformance.fixeddict: Fixed-key dictionaries

The *vc2_conformance.fixeddict* (page 157) module provides the *fixeddict()* (page 159) function for creating new `dict`[67] subclasses which permit only certain keys to be used. These new types may be used like ordinary Python dictionaries but add three main features:

- Explicitness – The VC-2 pseudocode creates and uses many dictionaries (called 'maps' in the specification) in place of struct-like objects. *fixeddicts* (page 157) provide a way to give these clear names.
- Avoidance of typos – Misspelt key names will result in a *FixedDictKeyError* (page 160).
- Better pretty printing – See more below...

13.1.1 Tutorial

Using *fixeddict()* (page 159), dictionary-like types with well defined fields can be described like so:

```
>>> from vc2_conformance.fixeddict import fixeddict

>>> FrameSize = fixeddict(
...     "FrameSize",
...     "custom_dimensions_flag",
...     "frame_width",
...     "frame_height",
... )
```

This produces a 'dict' subclass called `FrameSize` with all of the usual dictionary behaviour but which only allows the specified keys to be used:

```
>>> f = FrameSize()
>>> f["custom_dimensions_flag"] = True
>>> f["frame_width"] = 1920
>>> f["frame_height"] = 1080

>>> f["frame_width"]
1920

>>> f["not_in_fixeddict"] = 123
Traceback (most recent call last):
  ...
FixedDictKeyError: 'not_in_fixeddict'
```

To improve readability when producing string representations of VC-2 data structures, the generated dictionary types have a 'pretty' string representation.

[67] https://docs.python.org/3/library/stdtypes.html#dict

```
>>> print(f)
FrameSize:
  custom_dimensions_flag: True
  frame_width: 1920
  frame_height: 1080
```

To further improve the readability of this output, custom string formatting functions may be provided for each entry in the dictionary. To define these, *Entry* (page 159) instances must be used in place of key name strings like so:

```
>>> from vc2_conformance.string_formatters import Hex
>>> from vc2_data_tables import ParseCodes  # An IntEnum
>>> ParseInfo = fixeddict(
...     "ParseInfo",
...     Entry("parse_info_prefix", formatter=Hex(8)),
...     Entry("parse_code", enum=ParseCodes, formatter=Hex(2)),
...     Entry("next_parse_offset"),
...     Entry("previous_parse_offset"),

>>> pi = ParseInfo(
...     parse_info_prefix=0x42424344,
...     parse_code=0x10,
...     next_parse_offset=0,
...     previous_parse_offset=0,
... )
>>> str(pi)
ParseInfo:
   parse_info_prefix: 0x42424344
   parse_code: end_of_sequence (0x10)
   next_parse_offset: 0
   previous_parse_offset: 0
```

See the *vc2_conformance.string_formatters* (page 160) module for a set of useful string formatting functions.

Finally, documentation can optionally be added in the form of `help` and `help_type` arguments which will combined into the generated type's docstring:

```
>>> ParseInfo = fixeddict(
...     "ParseInfo",
...     Entry("parse_info_prefix", formatter=Hex(8), help_type="int", help="Always␣
↪0x42424344"),
...     Entry("parse_code", enum=ParseCodes, formatter=Hex(2), help_type="int"),
...     Entry("next_parse_offset", help_type="int"),
...     Entry("previous_parse_offset", help_type="int"),
...     help="A deserialised parse info block.",
... )

>>> print(ParseInfo.__doc__)
A deserialised parse info block.

Parameters
==========
parse_info_prefix : int
    Always 0x42424344
parse_code : int
next_parse_offset : int
previous_parse_offset : int
```

13.1.2 API

fixeddict(*name*, **entries*, ***kwargs*)

Create a fixed-entry dictionary.

A fixed-entry dictionary is a `dict`[68] subclass which permits only a preset list of key names.

The first argument is the name of the created class, the remaining arguments may be strings or *Entry* (page 159) instances describing the allowed entries in the dictionary.

Example usage:

```
>>> ExampleDict = fixeddict(
...     "ExampleDict",
...     "attr",
...     Entry("attr_with_default"),
... )
```

Instances of the dictionary can be created like an ordinary dictionary:

```
>>> d = ExampleDict(attr=10, attr_with_default=20)
>>> d["attr"]
10
>>> d["attr_with_default"]
20
```

The string format of generated dictionaries includes certain pretty-printing behaviour (see *Entry* (page 159)) and will also omit any entries whose name is prefixed with an underscore (_).

The class itself will have a static (and read-only) attribute `entry_objs` which is a :py;class:*collections.OrderedDict* mapping from entry name to *Entry* (page 159) object in the dictionary.

The keyword-only argument, 'module' may be provided which overrides the `__module__` value of the returned fixeddict type. (By default the module name is inferred using runtime stack inspection, if possible). This must be set correctly for this type to be picklable.

The keyword-only argument 'help' may be used to set the docstring of the returned class. This will automatically be appended with the list of entries allowed (and their help strings).

class Entry(*name*, ***kwargs*)

Defines advanced properties of of an entry in a *fixeddict()* (page 159) dictionary.

All constructor arguments, except name, are keyword-only.

Parameters

name [str] The name of this entry in the dictionary.

formatter [function(value) -> string] A function which takes a value and returns a string representation to use when printing this value as a string. Defaults to 'str'.

friendly_formatter [function(value) -> string] If provided, when converting this value to a string, this function will be used to generate a 'friendly' name for this value. This will be followed by the actual value in brackets. If this function returns None, only the actual value will be shown (without brackets).

enum [`Enum`[69]] A convenience interface which is equivalent to the following `formatter` argument:

```
def enum_formatter(value):
    try:
        return str(MyEnum(value).value)
    except ValueError:
        return str(value)
```

[68] https://docs.python.org/3/library/stdtypes.html#dict

And the following `friendly_formatter` argument:

```
def friendly_enum_formatter(value):
    try:
        return MyEnum(value).name
    except ValueError:
        return None
```

If `formatter` or `friendly_formatter` are provided in addition to `enum`, they will override the functions implicitly defined by `enum`.

help [str] Optional documentation string.

help_type [str] Optional string describing the type of the entry.

exception FixedDictKeyError(*key*, *fixeddict_class*)

A `KeyError`[70] which also includes information about which fixeddict dictionary it was produced by.

Attributes

key The key which was accessed.

fixeddict_class The *fixeddict* (page 157) type of the dictionary used.

13.2 vc2_conformance.string_formatters: Value-to-string formatting utilities

The *vc2_conformance.string_formatters* (page 160) module contains facilities for formatting (or pretty printing) Python values as strings.

When we say 'string formatter' we mean a function/callable which takes a value and returns a string representation of that value. Instances of the classes in this module act as formatters. For example, the *Hex* (page 160) class may be used as a formatter for n-digit hexadecimal integers:

```
>>> from vc2_conformance.string_formatters import Hex

>>> # Create a formatter for producing 8-digit hex numbers
>>> hex32_formatter = Hex(8)

>>> # Format some values
>>> hex32_formatter(0)
'0x00000000'
>>> hex32_formatter(0x1234)
'0x00001234'
```

class Number(*format_code*, *num_digits=0*, *pad_digit='0'*, *prefix=''*)

A formatter which uses Python's built-in `str.format()`[71] method to apply formatting.

This formatter is quite low level, see *Hex* (page 160), *Dec* (page 161), *Oct* (page 161) and *Bin* (page 161) for ready to use derivatives.

Parameters

format_code [str] A python `str.format()`[72] code, e.g. "b" for binary.

prefix [str] A prefix to add before the formatted number

num_digits [int] The length to pad the number to.

pad_digit [str] The value to use to pad absent digits

[69] https://docs.python.org/3/library/enum.html#enum.Enum
[70] https://docs.python.org/3/library/exceptions.html#KeyError
[71] https://docs.python.org/3/library/stdtypes.html#str.format
[72] https://docs.python.org/3/library/stdtypes.html#str.format

class Hex(*num_digits=0*, *pad_digit='0'*, *prefix='0x'*)
Prints numbers in hexadecimal.

Parameters

num_digits [int] Minimum number of digits to show.

pad_digit [str] The value to use to pad absent digits

prefix [str] Defaults to "0x"

class Dec(*num_digits=0*, *pad_digit='0'*, *prefix=''*)
Prints numbers in decimal.

Parameters

num_digits [int] Minimum number of digits to show.

pad_digit [str] The value to use to pad absent digits

prefix [str] Defaults to ""

class Oct(*num_digits=0*, *pad_digit='0'*, *prefix='0o'*)
Prints numbers in octal.

Parameters

num_digits [int] Minimum number of digits to show.

pad_digit [str] The value to use to pad absent digits

prefix [str] Defaults to "0o"

class Bin(*num_digits=0*, *pad_digit='0'*, *prefix='0b'*)
Prints numbers in binary.

Parameters

num_digits [int] Minimum number of digits to show.

pad_digit [str] The value to use to pad absent digits

prefix [str] Defaults to "0b"

class Bool
A formatter for `bool`[73] (or bool-castable) objects. For the values 0, 1, False and True, just shows 'True' or 'False'. For all other values, shows also the true value in brackets.

For example:

```
>>> bool_formatter = Bool()
>>> bool_formatter(False)
"False"
>>> bool_formatter(True)
"True"
>>> bool_formatter(0)
"False"
>>> bool_formatter(1)
"True"
>>> bool_formatter(123)
"True (123)"
>>> bool_formatter(None)
"True (None)"
```

class Bits(*prefix='0b'*, *context=4*, *min_length=8*, *show_length=16*)
A formatter for `bitarray.bitarray` objects. Shows the value as a string of the form '0b0101', using *`ellipsise()`* (page 163) to shorten very long, values.

Parameters

[73] https://docs.python.org/3/library/functions.html#bool

prefix [str] A prefix to add to the string

context [int]

min_length [int] See *ellipsise()* (page 163).

show_length [int or bool] If an integer, show the length of the bitarray in brackets if above the specified length (in bits). If a bool, force display (or hiding) of the length information.

class Bytes(*prefix='0x'*, *separator='_'*, *context=2*, *min_length=4*, *show_length=8*)
A formatter for bytes[74] strings. Shows the value as a string of the form '0xAB_CD_EF', using *ellipsise()* (page 163) to shorten very long, values.

Parameters

prefix [str] A prefix to add to the string

separator [str] A string to place between each pair of hex digits.

context [int]

min_length [int] See *ellipsise()* (page 163).

show_length [int or bool] If an integer, show the length of the bitarray in brackets if above the specified length (in bytes). If a bool, force display (or hiding) of the length information.

class Object(*prefix='<'*, *suffix='>'*)
A formatter for opaque python Objects. Shows only the object type name.

class List(*min_run_length=3*, *formatter=<class 'str'>*)
A formatter for lists which collapses repeated entries.

Examples:

```
>>> # Use Python-style notation for repeated entries
>>> List()([1, 1, 1, 1])
[1]*4

>>> # Also displays lists with some non-repeated values
>>> List()([1, 2, 3, 0, 0, 0, 0, 0, 4, 5])
[1, 2, 3] + [0]*5 + [4, 5]

>>> # A custom formatter may be supplied for formatting the list
>>> # entries
>>> List(formatter=Hex())([1, 2, 3, 0, 0, 0])
[0x1, 0x2, 0x3] + [0x0]*3

>>> # Equality is based on the string formatted value, not the raw
>>> # value
>>> List(formatter=Object())([1, 2, 3, 0, 0, 0])
[<int>]*6

>>> # The minimum run-length before truncation may be overridden
>>> List(min_run_length=3)([1, 2, 2, 3, 3, 3])
[1, 2, 2] + [3]*3
```

class MultilineList(*heading=None*, *formatter=<class 'str'>*)
A formatter for lists which displays each value on its own line.

Examples:

```
>>> MultilineList()(["one", "two", "three"])
0: one
```

(continues on next page)

[74] https://docs.python.org/3/library/stdtypes.html#bytes

(continued from previous page)

```
1: two
2: three

>>> # A custom formatter may be supplied for formatting the list
>>> # entries
>>> MultilineList(formatter=Hex())([1, 2, 3])
0: 0x1
1: 0x2
2: 0x3

>>> # A heading may be added
>>> MultilineList(heading="MyList")(["one", "two", "three"])
MyList
  0: one
  1: two
  2: three
```

13.3 vc2_conformance.string_utils: String formatting utilities

The *vc2_conformance.string_utils* (page 163) module contains a selection of general purpose string formatting routines.

indent (*text*, *prefix=' '*)

Indent the string 'text' with the prefix string 'prefix'.

Note: This function is provided partly because Python 2.x doesn't include `textwrap.indent()`[75] in its standard library and partly to provide an indent function with sensible defaults (i.e. 2 character indent, and always indent every line).

---

ellipsise (*text*, *context=4*, *min_length=8*)

Given a string which contains very long sequences of the same character (e.g. long mostly constant binary or hex numbers), produce an 'ellipsised' version with some of the repeated characters replaced with '…'.

Exactly one shortening operation will be carried out (on the longest run) meaning that so long as the original string length is known, no ambiguity is introduced in the ellipsised version.

For example:

```
>>> ellipsise("0b10100000000000000000000000000000000000001")
"0b1010000...00001"
```

Parameters

text [str] String to ellipsise.

context [int] The number of repeated characters to retain before and after the ellipses.

min_length [int] The minimum number of characters to bother replacing with '…'. This means that no change will be made until 2*context + min_length character repetitions.

ellipsise_lossy (*text*, *max_length=80*)

Given a string which may not fit within a given line length, trnucate the string by adding ellipses in the middle.

[75] https://docs.python.org/3/library/textwrap.html#textwrap.indent

split_into_line_wrap_blocks(*text*, *wrap_indented_blocks=False*)

Deindent and split a multi-line markdown-style string into blocks of text which can be line-wrapped independently.

For example given a markdown-style string defined like so:

```
'''
    A markdown style title
    ======================

    This is a string with some initial indentation
    and also some hard line-wraps inserted too. This
    paragraph ought to be line-wrapped as an
    independent unit.

    Here's a second paragraph which also ought to be
    line wrapped as its own unit.

    * This is a bulleted list
    * Each bullet point should be line wrapped as an
      individual unit (with the wrapping indented
      as shown here).
    * Notice that bullets don't have a newline
      between them like paragraphs do.

    1. Numbered lists are also supported.
    2. Here long lines will be line wrapped in much
       the same way as a bulleted list.

    Finally:

        An intended block will also remain indented.
        However, if wrap_indented_blocks is False, the
        existing linebreaks will be retained (e.g. for
        markdown-style code blocks). If set to True,
        the indented block will be line-wrapped.
'''
```

This will be split into independently line wrappable segments (as described).

Returns

blocks [[(first_indent, rest_indent, text), ...]] A series of wrappable blocks. In each tuple:

- first_indent contains a string which should be used to indent the first line of the wrapped block.
- rest_indent should be a string which should be used to indent all subsequent lines in the wrapped block. This will be the same length as first_indent.
- text will be an indentation and newline-free string

An empty block (i.e. `("", "", "")`) will be included between each paragraph in the input so that the output maintains the same vertical whitespace profile.

wrap_blocks(*blocks*, *width=None*, *wrap_indented_blocks=False*)

Return a line-wrapped version of a series of text blocks as produced by *split_into_line_wrap_blocks()* (page 163).

Expects a list of (first_line_indent, remaining_line_indent, text) tuples to output.

If 'width' is None, assumes an infinite line width.

If 'wrap_indented_blocks' is False (the default) indented (markdown-style) code blocks will not be line wrapped while other indented blocks (e.g. bullets) will be.

wrap_paragraphs(*text*, *width=None*, *wrap_indented_blocks=False*)
Re-line-wrap a markdown-style string with hard-line-wrapped paragraphs, bullet points, numbered lists and code blocks (see *split_into_line_wrap_blocks()* (page 163)).

If 'width' is None, assumes an infinite line width.

If 'wrap_indented_blocks' is False (the default) indented (markdown-style) code blocks will not be line wrapped while other indented blocks (e.g. bullets) will be.

13.4 vc2_conformance.py2x_compat: Python 3.x backports

The *vc2_conformance.py2x_compat* (page 165) module provides backported implementations of various functions from Python 3 which are used by this software.

zip_longest()
In Python 3.x an alias for `itertools.zip_longest()`[76], in Python 2.x, an alias for `itertools.izip_longest`.

get_terminal_size()
In Python 3.x an alias for `shutil.get_terminal_size()`[77], in Python 2.x, a dummy function always returning `(80, 20)`.

wraps()
In Python 3.x an alias for `functools.wraps()`[78]. In Python 2.x, an alternative implementation of `functools.wraps` which includes the Python 3.x behaviour of setting the `__wrapped__` attribute to allow introspection of wrapped functions (see *unwrap()* (page 165)).

unwrap()
In Python 3.x an alias for `inspect.unwrap()`[79]. In Python 2.x a backported implementation of that function. Relies on the backported *wraps()* (page 165) implementation provided by this module.

quote()
In Python 3.x an alias for `shlex.quote()`[80], in Python 2.x, an alias for `pipes.quote`.

string_types
A tuple enumerating the native string-like types. In Python 3.x, `(str, )`, in Python 2.x, `(str, unicode)`.

gcd()
In Python 3.x an alias for `math.gcd()`[81], in Python 2.x, an alias for `fractions.gcd`.

zip()
In Python 3.x an alias for *zip()* (page 165), in Python 2.x, an alias for `itertools.izip`.

makedirs()
In Python 3.x an alias for `os.makedirs()`[82]. In Python 2.x, a backport of this function which includes the `exist_ok` argument.

FileType()
In Python 3.x an alias for `argparse.FileType`[83]. In Python 2.x, a wrapper around `argparse.FileType`[84] adding support for the 'encoding' keyword argument when opening with mode "r".

[76] https://docs.python.org/3/library/itertools.html#itertools.zip_longest
[77] https://docs.python.org/3/library/shutil.html#shutil.get_terminal_size
[78] https://docs.python.org/3/library/functools.html#functools.wraps
[79] https://docs.python.org/3/library/inspect.html#inspect.unwrap
[80] https://docs.python.org/3/library/shlex.html#shlex.quote
[81] https://docs.python.org/3/library/math.html#math.gcd
[82] https://docs.python.org/3/library/os.html#os.makedirs
[83] https://docs.python.org/3/library/argparse.html#argparse.FileType
[84] https://docs.python.org/3/library/argparse.html#argparse.FileType

BIBLIOGRAPHY

[vc2_data_tables] The vc2_data_tables[85] manual.

[vc2_bit_widths] The vc2_bit_widths[86] manual.

[vc2_conformance_data] The vc2_conformance_data[87] manual.

[85] https://github.com/bbc/vc2_data_tables/
[86] https://github.com/bbc/vc2_bit_widths/
[87] https://github.com/bbc/vc2_conformance_data/

INDEX

Symbols

A

B

C

D

M

N

O

P

Q

R

S

T

U

V

W

X

Z