OGC® DOCUMENT: 25-047

External identifier of this OGC® document: http://www.opengis.net/doc/PER/T21-GIMI-D201

Open Geospatial Consortium

OGC TESTBED-21 DOCUMENTATION OF UPDATES TO LIBHEIF REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2026-05-13
Approval Date: 2026-06-04
Publication Date: 2026-08-12
Editor: Dirk Farin, Sina Taghavikish

Note

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

CONTENTS

I OVERVIEW

This Engineering Report documents the extensions made to the open-source libheif library and to heif-enc, a command-line tool that uses libheif to convert images and image sequences to HEIF, during OGC Testbed-21. The work continues the libheif development carried out during Testbed-20 and aligns the implementation with the requirements of the GIMI standard.

Four main areas of work are described.

First, inter-coded video sequences are now supported in libheif using H.264, H.265, H.266, and AV1. Adding inter-coding required a redesign of the internal image-sequences processing pipeline and of the codec-plugin API to accommodate encoder buffering and frame reordering, including the reordering of the associated Sample Auxiliary Information (SAI). A new x264-based encoder plugin was added, and the existing ffmpeg-based decoder plugin was extended to decode H.264 streams as an alternative to the OpenH264 decoder plugin.

Second, the ISO/IEC 23001-17 ("uncompressed") codec was rewritten around a modular encoder/decoder architecture and extended to cover a wide range of integer and floating-point data types, multi-channel and non-visual imagery, Bayer and other filter-array sensor images, and sensor-related metadata boxes such as `snuc`, `sbpm`, and `cpat`. A benchmark of the deflate and brotli general-purpose compressors applied to uncompressed imagery quantifies the trade-off between file size and encoding time.

Third, heif-enc gained first-class GIMI encoding support: GIMI-compatible brands, UUID-based content identifiers assigned to images, tiles, components, tracks, and sequence frames, metadata tracks, per-frame SAI, and integration with RDF/Turtle metadata. A dedicated GeoTIFF-to-GIMI conversion tool was built on top of heif-enc to handle very large tiled and pyramidal geospatial datasets while preserving their structure and metadata.

Fourth, the tile-viewer demonstration was extended to load HEIF files served over HTTP through a libcurl-based implementation of the libheif reader interface, with trivial and block-level request caches that make interactive access to multi-gigabyte files practical without local download.

Smaller improvements are also described, including updated support for `mif3` files with `mini` minimized headers, support for HEIF items larger than 4 GB, additional options in `heif-enc` and `heif-info`, and a new `heif-gen-bayer` utility used to generate synthetic Bayer-image test content.

II EXECUTIVE SUMMARY

This OGC Testbed-21 Engineering Report documents significant enhancements to the libheif library and to heif-enc, a command-line tool that uses libheif to convert images and image sequences to HEIF, undertaken to advance support for HEIF- and GEOINT Imagery Media for Intelligence, Surveillance, and Reconnaissance (GIMI)-based imagery workflows. The work

responds to growing requirements for efficient storage, exchange, and streaming of large, complex geospatial and sensor-derived imagery, including tiled datasets, multi-resolution pyramids, image sequences, and richly annotated image content.

A primary focus of the libheif work in Testbed-21 was the extension of image sequence and video-like capabilities in libheif. Building on prior Testbed-20 work, the library was enhanced to support inter-coded video sequences using modern codecs (H.264, H.265, H.266/VVC, and AV1), enabling substantially improved compression efficiency for temporal imagery. The sequence encoding and decoding APIs were refactored to support buffered, frame-reordering video pipelines while preserving accurate synchronization of metadata, including Sample Auxiliary Information (SAI) such as timestamps and content identifiers.

A major architectural contribution of Testbed-21 is the comprehensive refactoring and expansion of [ISO_IEC_23001-17] ("uncompressed") codec support. The codec was redesigned using a modular encoder/decoder architecture that improves performance, maintainability, and extensibility. Support was added for a wide range of data types, multi-channel and non-visual imagery, Bayer and other filter-array sensor images, and multiple sensor-related metadata boxes, enabling faithful representation of advanced remote-sensing, scientific, and ISR imagery.

Testbed-21 also introduced first-class GIMI encoding support. The heif-enc tool now supports GIMI-required compatible brands, unified ID namespaces, and systematic assignment of UUID-based content identifiers to images, tiles, components, tracks, and sequence frames. Integration with RDF/Turtle metadata enables direct linkage between HEIF/GIMI content and external or embedded semantic metadata, making HEIF a practical carrier for semantically rich geospatial imagery.

To support operational geospatial workflows, Testbed-21 delivered a GeoTIFF-to-GIMI conversion pipeline capable of handling very large, tiled, and pyramidal datasets while preserving geospatial structure and metadata. In parallel, significant advances were made in network-efficient access to large HEIF and GIMI files through HTTP range-based streaming and intelligent caching, enabling cloud-native visualization and interactive exploration of multi-gigabyte imagery without full file downloads.

Collectively, the Testbed-21 results demonstrate that libheif, HEIF, and the emerging GIMI ecosystem can support scalable, metadata-rich, temporally aware, and cloud-optimized imagery workflows. While several areas for future enhancement remain, the work establishes a strong technical foundation for continued OGC standards development and real-world adoption.

KEYWORDS

The following are keywords to be used by search engines and document catalogues.

HEIF, GIMI, testbed, imagery, libheif

IV FUTURE OUTLOOK

The outcomes of OGC Testbed-21 demonstrate that libheif and HEIF-based formats, particularly in the context of GIMI, are now technically capable of supporting complex, large-scale, and metadata-rich imagery workflows. Future work should focus on consolidating these capabilities, aligning them with evolving standards, and improving performance, usability, and ecosystem adoption.

A first priority is continued alignment with evolving HEIF, ISO/IEC 23001-17, and GIMI specifications. Several features prototyped in Testbed-21, such as advanced tiling methods, unified identifier namespaces, and extended uncompressed image metadata, depend on standards that are still maturing. Future Testbeds and associated Standards Working Groups should track and incorporate finalized HEIF version updates and assess whether Testbed-21 extensions should be proposed formally for standardization or profiles.

Performance and scalability remain key areas for improvement. While Testbed-21 demonstrated robust handling of tiled imagery and network-based access using HTTP range requests, further gains are expected from parallel tile encoding and decoding, improved buffering strategies, and reduced memory consumption when generating very large files. In addition, investigation of hardware-accelerated codecs, particularly for H.265/HEVC and AV1, could significantly reduce decoding latency and energy consumption, especially for mobile and embedded platforms.

With respect to ISO/IEC 23001-17 (“uncompressed”) imagery, Testbed-21 identified opportunities for more efficient lossless compression methods tailored to image data rather than relying solely on general-purpose compressors such as deflate or brotli. Future research may explore new lossless image compression techniques that preserve the flexibility of uncompressed representations while improving storage efficiency and encoding time, potentially leading to new codec extensions.

Metadata handling is another area for continued evolution. The integration of content identifiers, Sample Auxiliary Information, metadata tracks, and RDF/Turtle files in Testbed-21 illustrates a strong foundation for semantically enabled imagery. Future work may focus on improved tooling for metadata validation, stronger alignment with OGC APIs and semantic frameworks, and enhanced support for linking imagery to external knowledge graphs and catalogues in operational environments.

From an ecosystem perspective, broader adoption will benefit from higher-level language bindings and developer tooling. While libheif’s C API is comprehensive, many user communities rely on Python and JavaScript environments. Providing full-fidelity bindings that expose tiling, streaming, sequence handling, and GIMI-specific metadata would lower adoption barriers and encourage wider experimentation and deployment.

Finally, Testbed-21 highlighted that HEIF- and GIMI-based workflows are increasingly relevant for cloud-native geospatial processing and visualization. Future Testbeds may explore tighter integration with OGC API standards, cloud object storage patterns, and web-based clients, including hybrid use of HEIF imagery alongside conventional video and raster services.

In summary, Testbed-21 establishes libheif and HEIF/GIMI as a strong technical foundation for next-generation imagery. Continued collaborative development through OGC Testbeds

and Working Groups will be essential to transition these capabilities from experimental implementations into stable, interoperable, and widely adopted solutions.

V VALUE PROPOSITION

The Testbed-21 work provides OGC, its members, and the broader geospatial and ISR communities with a comprehensive, open-source toolchain for producing and consuming GIMI-compliant HEIF imagery. Because the work extends the existing libheif library and its command-line tools rather than introducing a new software stack, the new capabilities become available through the same interfaces that HEIF-aware applications already rely on, keeping the cost of adoption low for implementers.

The concrete benefits for stakeholders include the following.

- **Open-source path to GIMI compliance.** Practitioners can now generate and consume GIMI-compliant HEIF files end-to-end with open-source software, removing a significant barrier to the practical adoption of the GIMI standard.
- **Geospatial imagery workflows at scale.** The GeoTIFF-to-GIMI conversion tool handles very large tiled and pyramidal geospatial datasets while preserving their structure and metadata, enabling existing GeoTIFF archives to be transitioned into the HEIF/GIMI ecosystem without proprietary tooling.
- **Efficient storage of image sequences.** Inter-coded video support across H.264, H.265, H.266, and AV1 substantially reduces the storage footprint of image time-series, bursts, and sensor recordings compared to the intra-only encoding that was previously available.
- **Rich sensor metadata for scientific and ISR imagery.** The refactored ISO/IEC 23001-17 codec covers a wide range of integer and floating-point data types, multi-channel and non-visual imagery, and named sensor-metadata boxes including sensor non-uniformity correction (`snuc`), sensor bad-pixel maps (`sbpm`), color-filter-array patterns (`cpat`), and polarization patterns (`splz`), making HEIF a practical carrier for scientific and ISR-grade sensor data.
- **Native support for Bayer sensor imagery.** The libheif color-conversion pipeline has been extended to support de-bayering of Bayer-pattern images for any RGB filter arrangement, so raw sensor data can be encoded once and displayed natively as RGB without external tools.
- **Cloud-native access to large imagery.** HTTP range-based streaming with block-level caching allows interactive exploration of multi-gigabyte HEIF files without local download, supporting cloud-hosted imagery services and web-based clients.
- **Generation of semantically annotated HEIF/GIMI files.** The libheif tools can produce HEIF and GIMI files, for both still images and image sequences, with UUID-based content identifiers, per-frame Sample Auxiliary Information (SAI), metadata tracks, and embedded

RDF/Turtle metadata, supporting provenance tracking and semantic annotation of the encoded imagery.

For the standards community, the Testbed-21 work also serves as a reference implementation that OGC, the Standards Working Groups, and MPEG can use to validate and refine the still-evolving HEIF, ISO/IEC 23001-17, and GIMI specifications.

1 INTRODUCTION

1 INTRODUCTION

The main goal of this Report is to document additional feature implementations for the libheif library carried out in Testbed-21.

A key part of this work was related to the heif encoder (heif-enc) command line tool. The heif-enc command line tool is a reference application for converting images and image collections into HEIF-based formats. It supports input formats such as JPEG, PNG, TIFF, and Y4M, and can generate multiple HEIF variants including HEIC (HEVC), AVIF (AV1), VVC, JPEG, JPEG-2000, HT-JPEG-2000, and ISO 23001-17 uncompressed content. The tool selects the encoding format either through explicit command-line switches or automatically based on the output file suffix. Although termed “uncompressed,” the ISO 23001-17 mode may still use lossless compression algorithms such as deflate, zlib, or brotli, depending on library configuration.

The heif-enc tool provides advanced support for high-resolution and structured imagery. It can encode tiled images by automatically assembling multiple tile files into a single image, detecting tile numbering patterns and layout, and supporting several tiling modes that balance compatibility, efficiency, and scalability. It also supports multi-resolution pyramids, allowing multiple image layers of different resolutions to be combined into a single HEIF file for efficient multiscale visualization and analysis. Both tiled and non-tiled images can be used within these pyramid structures.

heif-enc can encode image sequences as HEIF image-sequence files, enabling video-like content with configurable frame rates, GOP structures, alpha-channel handling, looping behavior through edit lists, and support for transparency. It further supports rich metadata integration through timed metadata tracks and Sample Auxiliary Information (SAI), including timestamps and content identifiers. Metadata can be provided in textual or binary form and precisely synchronized with frames, enabling detailed temporal annotation and provenance tracking within HEIF image sequences.

2 TOPICS

2 TOPICS

2.1. Inter-coded video implementation

In Testbed-20, support for intra-coded video sequences was added to libheif. Intra-coded means that each frame is coded independently like a still image. For video sequences, the change between successive images is usually small such that a higher compression can be obtained by leveraging the similarities between successive images to achieve higher compression ratios. A compression format where the redundancies between images are exploited to improve compression ratios is denoted as inter-coding. In Testbed-21, support for inter-coded video sequences was added to libheif.

Inter-coded video is only possible for compression standards that support this. Inter-coding is supported by the typical video compression codecs H.264, H.265, H.266, and AV1. Support for inter-coded video was implemented in libheif for the mentioned four video codecs. In contrast, JPEG-2000, JPEG, and uncompressed (ISO 23001-17) video do not support inter-coding.

2.1.1. Prediction structures

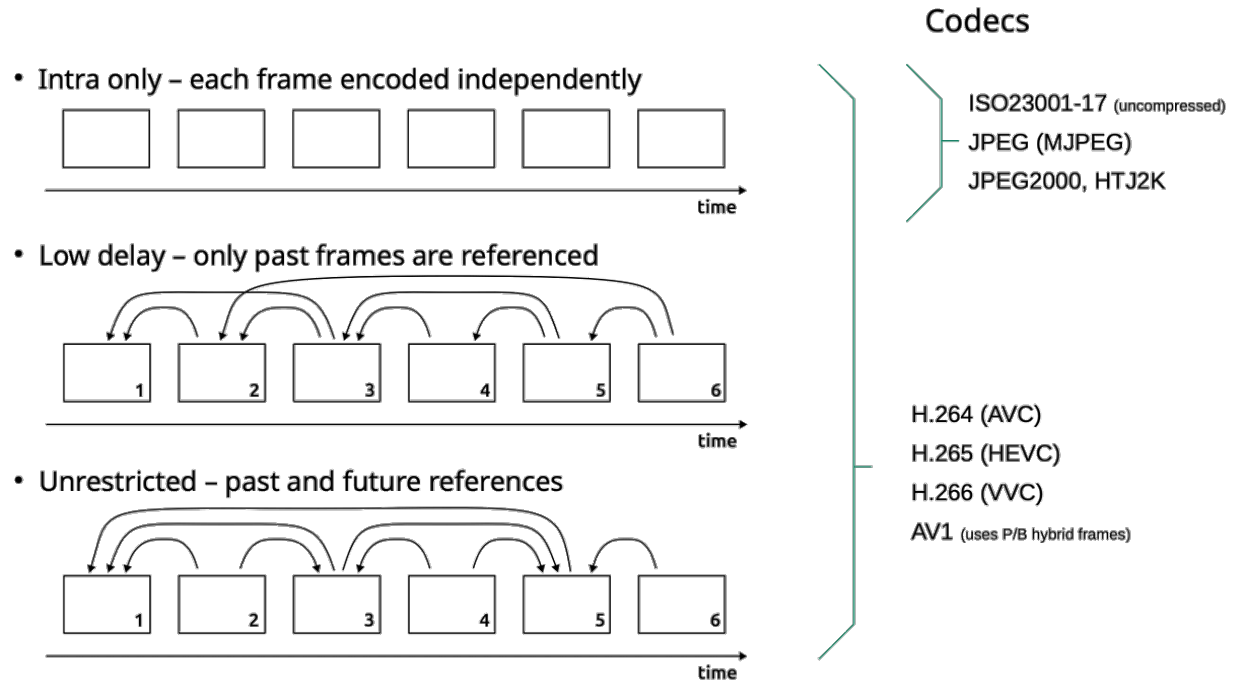


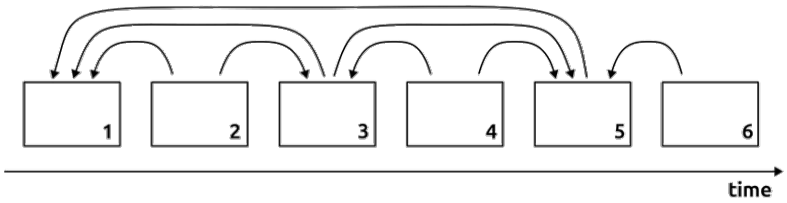
Figure 1 — Prediction Structure in Video Coding

When coding video sequences, three different encoding modes can be coarsely differentiated:

- intra only, referring to intra-coded video;
- low-delay, where images are coded using only earlier images in the same video sequence as reference; and
- unrestricted, where images may be coded using earlier and future images as reference.

While intra only and low-delay sequences can be decoded in the natural frame order, unrestricted video sequences may reorder the original images. Thus, the input video has to be buffered for a few frames such that the reordering can take place. This introduces delay on both the encoder side and the decoder side. Images in video without reordering can be encoded and decoded immediately, hence the name “low-delay”.

See Figures Figure 1 and Figure 2.



- Frame presentation order:
  1, 2, 3, 4, 5, 6
- Compression order:
  1, 5, 3, 2, 4, 6

- Pro
  - Higher compression ratio
- Contra
  - decoding delay
  - frames have to be buffered for reordering

Figure 2 — Frame Ordering in Video Coding

The implementation in libheif provides the option to choose between these three coding modes. If a mode is not supported by the selected video codec, the library will fall back to a supported mode (for example “intra only” for JPEG-2000). More specifically, it depends on the exact video encoder whether a mode is supported. There can be different encoders for the same compression format that differ in which modes they support.

2.1.2. Example: per-frame sizes for the three prediction structures

To illustrate the effect of the prediction structure on compression efficiency, a 100-frame video sequence was encoded three times with H.264 (x264), once for each prediction mode, while keeping all other encoder parameters identical. The resulting per-frame sizes are shown in Figures Figure 3, Figure 4, and Figure 5.

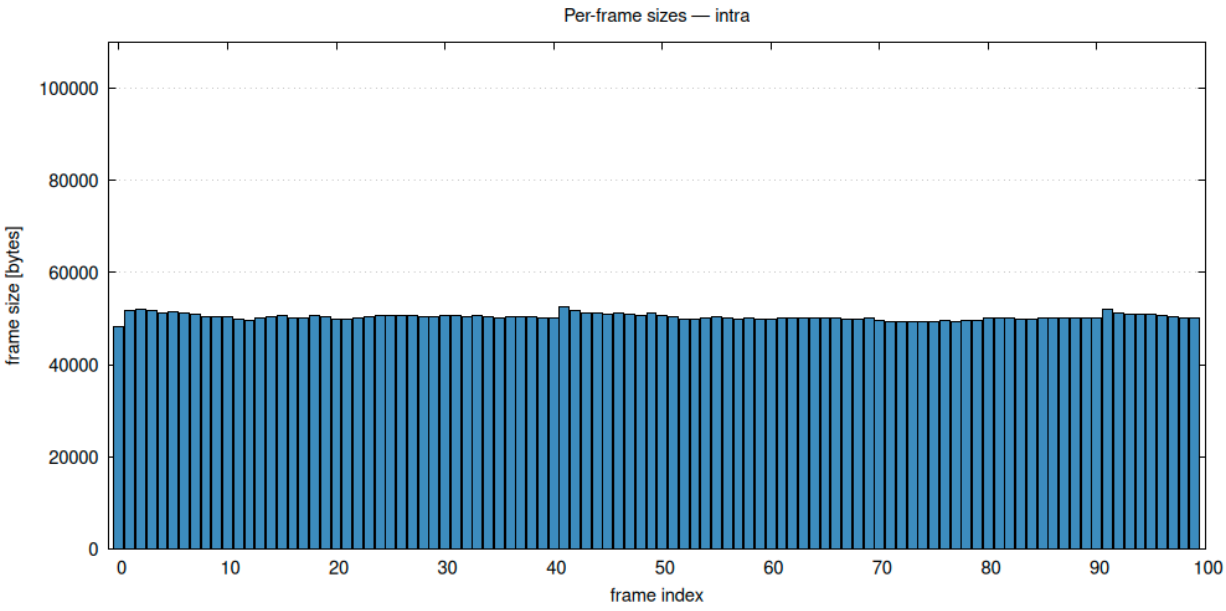


Figure 3 — Per-frame sizes for the "intra-only" mode. Total file size: 6,299,508 bytes.

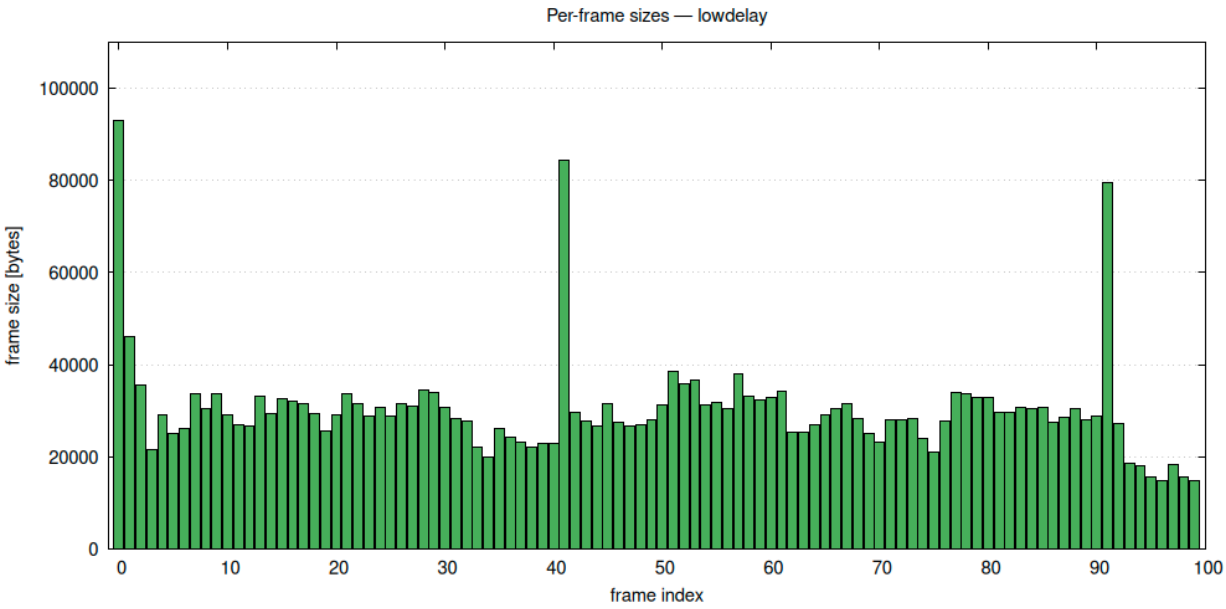


Figure 4 — Per-frame sizes for the "low-delay" mode. Total file size: 2,842,387 bytes.

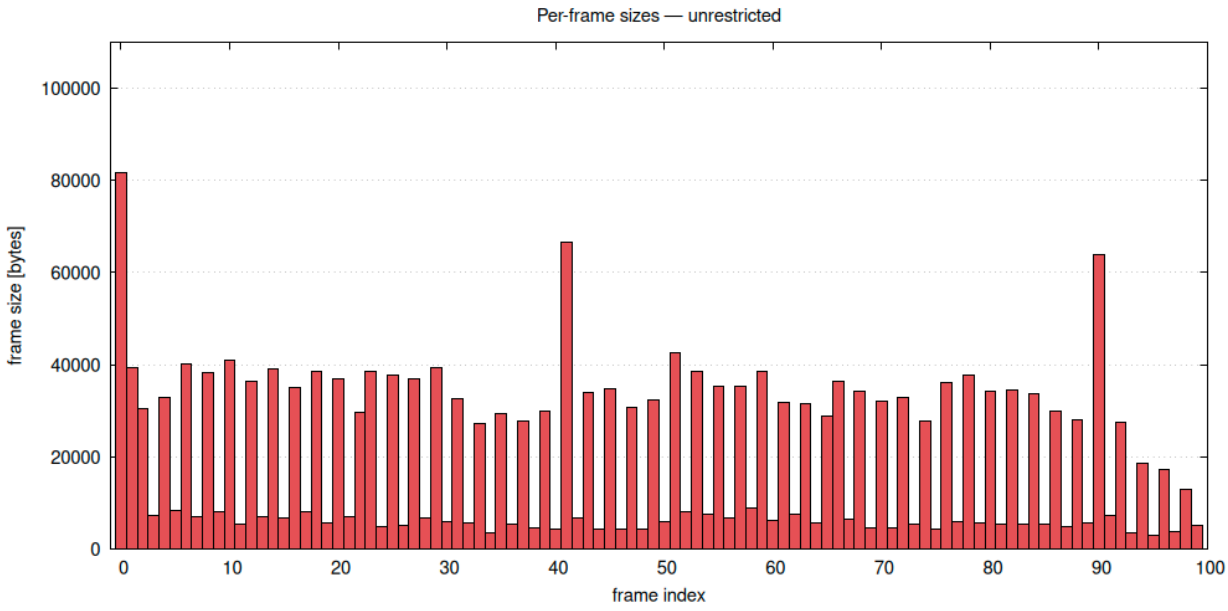


Figure 5 — Per-frame sizes for the "unrestricted" mode. Total file size: 2,193,068 bytes.

In the “intra-only” mode, every frame is coded independently, so the per-frame sizes are nearly equal and uniformly large. This mode produces the largest file because no inter-frame redundancy is exploited. On the other hand, since each frame can be decoded independently of all others, this mode allows random access to any frame without first decoding preceding frames.

In the “low-delay” mode, periodic keyframes (visible as the tall bars at the start and at frame indices 40 and 90) are coded as intra frames, while the frames in between are predicted from preceding frames and are therefore much smaller. The total file size drops to about 45% of the intra-only file.

In the “unrestricted” mode, the encoder uses a more complex prediction pattern with both forward and backward references. This produces a characteristic alternation between larger reference frames and very small frames that are predicted from both neighbours, resulting in roughly half of the frames being extremely small. The total file size is the smallest of the three modes, at about 35% of the intra-only file.

Note that the exact file-size differences between these modes are highly content-dependent. Sequences with little change over time and slow motion benefit much more from inter-frame prediction than sequences with rapid scene changes or strong motion, where prediction is less effective and the relative gain over intra-only coding shrinks.

2.2. MP4 video vs. HEIF image sequences

Image sequences as defined in HEIF have many similarities with MP4 video, as both are based on the ISO 14496-12 ([ISO-IEC_14496-12-2022]), ISO Base Media File Format (ISOBMFF) standard. The main difference lies in the handler type where MP4 video uses 'vide' while HEIF image sequence uses 'pict'. Semantically, MP4 video plays like a video/movie while HEIF image sequence efficiently stores image bursts, time series, exposure bracketing, live photos, and short animations (GIFs). Initially, HEIF image sequences did not allow image reordering, but this restriction has been recently lifted in the HEIF standard.

Interpretation of timing (frames per second) in HEIF image sequences depends on the application. HEIF sequences have an additional 'CodingConstraints' box and optionally a 'refs' box to signal the reference frames in the video. These two boxes make it easier for a decoder to access a specific image in the sequence without decoding all preceding images.

This allows for special image prediction structures as shown in Figure 6, where there is one master image and all other images are using predictive coding from this master image. However, currently available open-source encoders are only targeting video applications and thus do not explicitly support this prediction structure. Such image sequences can be decoded, but there is currently no easy way to encode such structures.

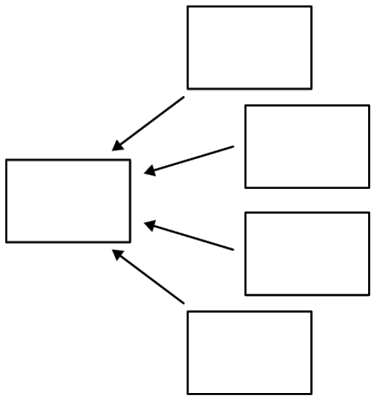


Figure 6 — Example: master image with image collection predicted from master.

2.3. Libheif codec plugins

A list of plugins is provided in Table 1. The libheif library processes each compression format through dedicated codec plugins. Prior to Testbed-21, H.264 sequences (and images) could only be decoded by libheif. Encoding support was added in Testbed-21 by implementing a new encoding plugin based on the x264 encoder.

Since libheif supports a large number of formats and even multiple encoders or decoders for each format, the number of software library dependencies would be large if every option is

compiled into it. Unused codecs can be disabled at compile time, but since it is not known beforehand which of them are required when precompiled binary packages are compiled, it is difficult to make a selection. However, including many codecs would result in a large library binary and difficult packaging because of the large number of dependencies, as well as slower startup times. For this reason, libheif has a plugin system, where codecs can be compiled into independent plugin files. Users can decide which codecs they need and they may decide to install only those plugins.

An exception is the ISO/IEC 23001-17 (uncompressed) codec. This is built into libheif itself and because of its extensive use of ISOBMFF boxes, it is not easily possible to extract this as a plugin. Compared with other video codecs, the implementation of ISO/IEC 23001-17 is relatively small, so that this should not pose a problem.

Another exception is an H.265 decoder that employs the JavaScript WebCodecs API provided by web browsers. This decoder can only be used when libheif is compiled to JavaScript or WASM.

2.3.1. New plugins

During the OGC Testbed-21 initiative, a new plugin based on the x264 library was written to encode H.264 sequences. Decoding of H.264 sequences was previously done with a plugin based on the OpenH264 library. Additionally, the existing H.265 decoding plugin based on the ffmpeg library was extended to also decode H.264 streams.

Table 1 — Heif plugins

FORMAT	DECODERS	ENCODERS
HEIC	libde265, ffmpeg, WebCodecs	x265, kvazaar
AVIF	AOM, dav1d	AOM, rav1e, svt-av1
VVC	vvdec	vvenc, uvg266
AVC	openh264, ffmpeg	x264
JPEG	libjpeg(-turbo)	libjpeg(-turbo)
JPEG2000	OpenJPEG	OpenJPEG
HTJ2K	OpenJPEG	OpenJPH
uncompressed	built-in	built-in

2.4. Libheif sequences encoding API changes

The library API for inter sequences encoding is mostly identical to the API introduced in Testbed-20 for intra sequences encoding. The encoding options `heif_sequence_encoding_options` have been extended with options to set the encoding mode as described above and to set a range for the distance of keyframes that the encoder should use. If keyframes are closer together, seek operations in the encoded video are faster on average, since there are more keyframes at which the decoding can start. However, if keyframes are further apart, the compression efficiency increases, resulting in smaller files. Ultimately, most encoders will decide dynamically when to place keyframes.

A new function was also added that can be optionally called during sequence encoding at the end of the sequence to immediately flush the encoder buffers. As the previous implementation only supported intra-coding, each image was encoded independently, and there were no encoding buffers as there were also no reference pictures. However, this had to change for inter-coding. The encoder object is now retained for the whole sequence encoding. If the encoder does not know when the sequence end has been reached, the picture buffers have to be retained. These buffers are ultimately released automatically when the HEIF is written, but it is preferable to signal the end of the sequence to the library so that the memory is not allocated longer than needed.

```
enum heif_sequence_gop_structure
{
  // Only independently decodable keyframes.
  heif_sequence_gop_structure_intra_only,

  // No frame reordering, usually an IPPPP structure.
  heif_sequence_gop_structure_lowdelay,

  // All frame types are allowed, including frame reordering, to achieve
  // the best compression ratio.
  heif_sequence_gop_structure_unrestricted
};


typedef struct heif_sequence_encoding_options
{
  .....

  // version 2 options

  enum heif_sequence_gop_structure gop_structure;
  int keyframe_distance_min; // 0 - undefined
  int keyframe_distance_max; // 0 - undefined
} heif_sequence_encoding_options;
```

Figure 7 — code extract for heif sequence encoding

```
/**
 * When all sequence frames have been sent, you have to call this function to let the
 * library know that no more frames will follow. It will then finalize writing the track.
 */
LIBHEIF_API
heif_error heif_track_encode_end_of_sequence(heif_track*,
                                             heif_encoder* encoder);
```

Figure 8 — sequence encoding function signature

API changes were also necessary for sequence decoding.

Sequence encoding API documentation: https://github.com/strukturag/libheif/wiki/Reading-and-Writing-Sequences

2.5. Changes in internal libheif sequences architecture

Previously, plugins were designed around a simple single-frame encoding workflow. When a plugin was given one frame, it would:

1. initialize the codec and configure encoding parameters;
2. encode the image;
3. flush the encoder input queue;
4. retrieve the compressed image data along with bitstream headers (for example VPS and SPS); and then
5. release the encoder.

This model was initially introduced for still-image encoding, but it also worked for the intra-frame sequences implementation. For inter-sequence encoding, this simple approach could not be maintained as-is, because an encoder instance has to be kept alive for the whole sequence encoding duration. To this end, the five steps that were previously combined into one call had to be broken up into separate calls. Every encoding plugin had to be refactored to this new interface.

Also, the way the encoded video stream is processed when returned from the plugin had to change. While it was previously guaranteed that all returned data belongs to a specific stream, the situation is more complicated now. Parameter packets like H.265 SPS (sequence parameter set) or PPS (picture parameter set) need to be extracted from the video bitstream and placed into the configuration header boxes (e.g., `hvcC`). Some encoders repeat these packets in the bitstream. These duplicates can be sorted out and removed.

Because of the buffering and image reordering that may take place in the encoder, there is now no longer a 1:1 correspondence between submitting frames and receiving encoded data. It may be that for some frames passed into the encoder, nothing is returned, because the encoder still fills its reordering-buffer. On the other hand, it may be that many encoded images are returned at once. Since MP4/HEIF requires that pointers to the beginning and the size of each frame is noted in the metadata, the bitstream has to be parsed to find the frame beginnings. It is also required to extract the information whether a frame is a keyframe, since this information is also stored in the MP4 metadata. Some encoders provide this information, but not all of them. Thus, libheif has to find this information by itself from observing the bitstream. This bitstream processing is different for each compression format (H.264, H.265, H.266, AV1). Intra-only codecs (JPEG-2000, JPEG, uncompressed) are handled as before.

Finally, since the encoder may reorder the video frames, it is necessary to know exactly in which order the encoded video frames are received because SAI (sample auxiliary information) packets, such as content IDs or TAI timestamps, may need to be assigned to them. Previously this was a simple 1:1 direct assignment, but due to the frame reordering and buffering, the input SAI packets also have to be buffered until the encoded frames leave the video encoder. In order to know the correspondence between an input frame and an SAI packet, continuously increasing frame numbers are assigned to the input video frames and passed into the video encoder

plugins together with the video frames. The plugins pass the frame numbers through the codecs so that the reordered frame numbers can be observed in the output. When encoded frames are received, the correct SAI packets can be assigned based on the frame number.

The changed architecture required some new non-backwards compatible plugin APIs. The plugins are already versioned such that libheif will only load plugins with a minimum version number. However, that does not allow to remove functions from the plugin interface as this will lead to failure when using these plugins together with older libheif versions that expect these functions to exist. The solution was to introduce a 'minimum libheif version' property in each plugin that defines which minimum libheif version is needed for this plugin. That way, non-backwards compatible changes can be introduced. Unfortunately, since old libheif versions do not check this property, there will be a transition period where both APIs have to be kept in the plugin interface. The plan is to support both plugin APIs in parallel for a few years until these old libheif versions have been phased out and then remove the old API.

2.6. Changes in command-line tools

Even though libheif is primarily a programming library to be used in other software, it also comes with a number of command-line tools.

- `heif-enc` encodes HEIF images and image sequences.
- `heif-dec` decodes HEIF images as full images, separate tiles, and image sequences.
- `heif-view` displays HEIF image sequences and MP4 video in real-time.
- `heif-info` lists the content of a HEIF file either as a coarse textual overview, or (with the `--dump/-d` option) gives a detailed dump of the file's box structure.

During Testbed-21, a new tool `heif-gen-bayer` was implemented to help in the generation of simulated Bayer-filtered images.

The changes will be described below, grouped by topic.

2.6.1. Sidecar items

Arbitrary data can be embedded into a HEIF file as non-image items. Functions for adding such custom items to a HEIF file and to extract them from existing files were already added to libheif in Testbed-20. During Testbed-21, options were added to `heif-enc` to easily add a sidecar file with a MIME type and optional name string:

```
--add-mime-item TYPE       add a mime item of the specified content type
(experimental)
--mime-item-file FILE      use the specified FILE as the data to put into the
mime item (experimental)
--mime-item-name NAME      assign the name to the embedded item (experimental)
```

Libheif and `heif-enc` will not do any processing on this data (except optional compression), it will be included verbatim into the file. The optional compression method can be specified by

```
--enable-metadata-compression ALGO  enable metadata item compression
(experimental)
                                    Choose algorithm from {off,deflate,zlib,
brotli}.
```

There also exists a new option for `heif-dec` to extract such a sidecar file from the HEIF file by specifying the MIME type of the item that should be extracted:

```
--extract-mime-item TYPE   extract the MIME item with the given content type
into a file (mime-item.data)
```

2.7. GIMI encoding support

When encoding GIMI files, a few extra requirements on the generated files apply and several new options were added to support the generation of GIMI-compliant files. First, conformance to GIMI is signaled in the file by an additional `geo1` brand. The new option `--add-compatible-brand` was added to `heif-enc` to include any custom compatible brand, in this case `geo1`.

Another requirement for GIMI files is that all item, track, and entity-group IDs used in the file are drawn from a global namespace. These IDs are integer numbers, and in general HEIF files, it is allowed to reuse the same ID number for different types. For example, sequence tracks may have the same ID number as an image item without any semantic consequence as both are considered different namespaces. However, a HEIF file can also be generated such that all ID numbers are unique over all different types. If this is the case, the file is said to be having a unified ID namespace and the compatible brand `unif` may be added. Support for encoding with a unified ID namespace was implemented and can be enabled with the option `--unif`. This is a requirement of GIMI, but note that using this option slightly reduces the number of `grid` tiles that can be stored in the file because some of the valid ID numbers will now be blocked for other use.

```
--add-compatible-brand BRAND  add a compatible brand to the output file (4
characters)
--unif                        use unified ID namespace (adds 'unif' compatible
brand)
```

A central concept of GIMI files is that most objects in a HEIF file (images, tiles, tracks, sequence frames, image color components) get assigned a content ID, which is a UUID string. This content ID is used to build the correspondences to an RDF/Turtle file which employs the same content IDs and provides further metadata for the objects in the HEIF file. The actual RDF/Turtle file may be either embedded into the HEIF file or distributed as a separate sidecar file. This does not influence the way the content IDs are processed and interpreted. For encoding a HEIF file, this provides the challenge that content IDs have to be injected into the encoding process such that they match the content IDs used in the RDF/Turtle file.

```
--turtle FILENAME          read Turtle (RDF) file and assign GIMI content IDs to
tiles (experimental)
```

```
--embed-turtle              also embed the Turtle file as metadata item in the
HEIF (experimental)
```

2.8. Image sequences with SAI metadata

Support for sample auxiliary information (SAI) metadata was added to libheif during Testbed-20 in the C API, but there was no way yet to pass this metadata to `heif-enc`. This functionality was added in Testbed-21. SAI metadata are small data packets that are attached to the sequence frames. Typical use cases in GIMI are sequence frame content IDs and TAI timestamps.

The new option for `heif-enc` is:

```
--sai-data-file FILE        use the specified FILE as input data for the video
frames SAI data
```

This option requires a data file that contains the SAI packet data for each frame.

The data file starts with a header that defines which SAI types should be included in the encoded file. Each line in the header should contain the SAI type, optionally followed by global track parameters. The current options are:

- `suid` — content ID. No track parameters required.
- `stai` — TAI timestamps. The header line shall contain four numeric parameters that specify the `tai_clock_info` values (time uncertainty, clock resolution, clock drift rate, and clock type). Refer to the GIMI standard for further information on these values.

The header is ended by a line containing a triple dash (`---`).

After the header, the SAI packet content follows for each sequence frame, with one line per SAI packet. For each header line, there should be a data line. If a frame should not receive an SAI data packet, the line can be kept empty. The data format depends on the SAI type:

- `suid` — string that is used verbatim as content ID.
- `stai` — one to four values defining the `tai timestamp packet` (tai timestamp, synchronization state, timestamp generation failure, timestamp is modified). Only the timestamp value is mandatory, the 3 flags are optional and a default value `false` is used when they are not specified.

An example data file is depicted in Figure 9:

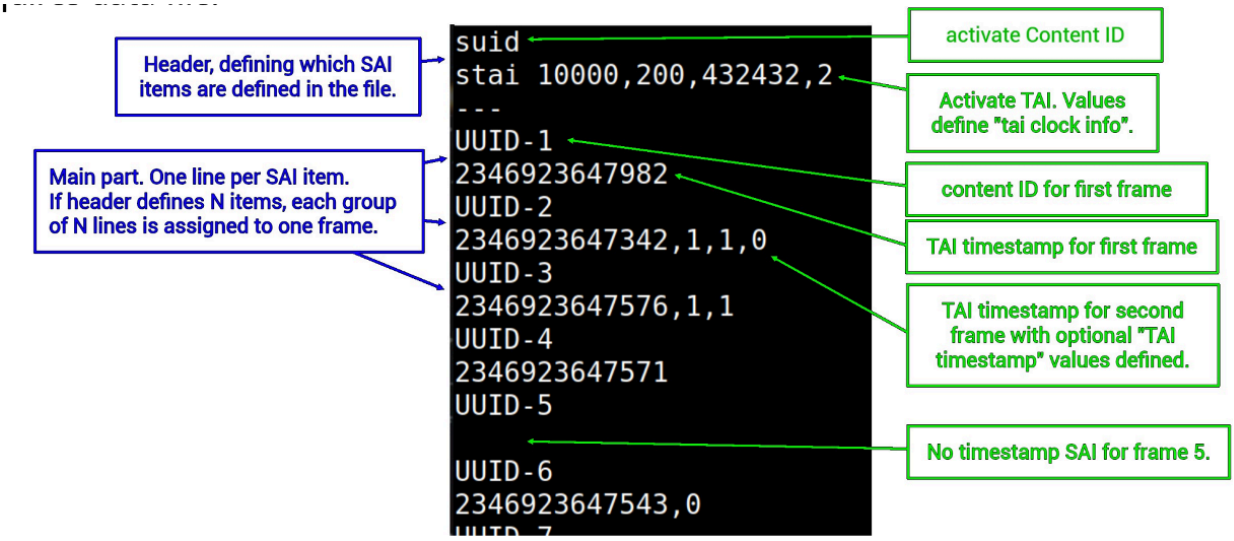


Figure 9 — SAI Datafile

The SAI metadata in an encoded file can be viewed by displaying the image sequence or video with `heif-view` and using the following options:

```
--show-sai                show sample auxiliary information
--show-track-metadata     show metadata attached to the track (e.g., TAI config)
```

These will output the SAI packets on the terminal while the video is decoding.

2.9. Metadata tracks

Metadata tracks are tracks with only metadata information packets that are timed independently from the main video track. These were also implemented in libheif during Testbed-20, but could only be generated through the C API. In Testbed-21, an option was added to create metadata tracks for image sequences with `heif-enc`.

The new options in `heif-enc` for this function are:

```
--vmt-metadata FILE       encode metadata track from VMT file (experimental)
--binary-metadata-track   parses VMT data as hex values that are written as raw
binary (experimental)
--metadata-track-uri URI  uses the URI identifier for the metadata track
(experimental)
```

The metadata is passed to `heif-enc` in a separate file in a file format inspired by WebVMT. Each metadata frame in the data file begins with a line containing a timestamp followed by an `-->`. The succeeding lines up until the next blank line are copied verbatim into the metadata track. An example file could be:

```
00:00:00.000 -->
This is a first metadata packet at the start of the video.

00:00:00.100 -->
```

```
This second metadata packet is timed
at 100ms after the start of the video.

00:00:05.000 -->
Five seconds have passed.
```

If the metadata frame should not contain ASCII data, binary data can be specified as hex values:

```
00:00:00.000 -->
a4 2b

00:00:00.100 -->
2c 14 7e ff c2

00:00:00.300 -->
5ff78ebca3d98bc276e1
ff5e768acdd217f2b3
```

Any characters except the hexadecimal digits are ignored. Whitespace is optional. The binary interpretation of the metadata file is enabled with the option `--binary-metadata-track`.

Note that "AwayTeam" has contributed an enhancement that parses the input, following more closely the WebVMT standard. It is described in the OGC Testbed-21 Advancement of the GIMI Standard Report (OGC 26-012).

A metadata track also needs a URI to uniquely describe its content. This is specified with `--metadata-track-uri URI`.

2.10. Other GIMI-specific options

There are two more new options in `heif-enc` to add GIMI content IDs. The first enables setting (random) content IDs for ISO 23001-17 image components:

```
--component-content-ids    assign random content IDs to the components of an ISO
23001-17 image.
```

This results in a property box assigned to the `unci` image item as follows:

```
| | | index: 2
| | | Box: cmpd -----
| | | size: 18   (header size: 8)
| | | component_type: Y
| | | component_type: Cb
| | | component_type: Cr
...
| | | index: 7
| | | Box: 9db9dd6e-373c-5a4e-8110-21fc83a911fd ----- (GIMI Component Content
IDs)
| | | size: 166   (header size: 24)
| | | [0] content ID: urn:uuid:64d2aacf-09ae-4a6a-9ded-ba2c5896b3fa
| | | [1] content ID: urn:uuid:79766bcc-c5b1-463b-b257-b8568c65a17a
| | | [2] content ID: urn:uuid:ecae342f-6ae7-4235-a827-c2610c20e2be
```

These content IDs assigned to the components are also shown in the `heif-info` output as follows:

```
...
properties:
  components:
    [0] type: Y (1)   content ID: urn:uuid:64d2aacf-09ae-4a6a-9ded-ba2c5896b3fa
    [1] type: Cb (2)  content ID: urn:uuid:79766bcc-c5b1-463b-b257-b8568c65a17a
    [2] type: Cr (3)  content ID: urn:uuid:ecae342f-6ae7-4235-a827-c2610c20e2be
...
```

Finally, there is a new `heif-enc` option to set the content ID for the visual image sequence track:

```
--set-gimi-track-id ID     set the GIMI track ID for the visual track
(experimental)
```

2.11. Changes in heif-info

The purpose of the `heif-info` tool is to list the content of a HEIF file either as a coarse textual overview, or (with the `--dump/-d` option) a detailed dump of the file's box structure. The overview output has been extended to also show the images' and the sequence tracks' content IDs. It also lists the embedded sidecar files together with their MIME type. No special option is needed to get this information.

2.12. GeoTIFF to GIMI conversion tool

During Testbed-21, it turned out that a common issue for many groups was to convert existing GeoTIFF images to corresponding HEIF/GIMI files, especially when the tiling structure and the overview image hierarchy of the GeoTIFF should be reflected in the GIMI file. Thus, even though it was not planned, the functionality to load GeoTIFF images was added to `heif-enc`, generating GIMI files that closely resemble the GeoTIFF input. A pair of Python scripts was also added that convert the coordinates stored in the GeoTIFF to a prototypical RDF/Turtle file that can either be embedded directly into the output GIMI or kept as a sidecar file. The scripts also automatically add content IDs into the RDF/Turtle file and the GIMI file such that corresponding tiles or images are linked through the content ID.

2.12.1. TIFF loader

Reading the TIFF input is implemented in `heif-enc` as an additional loader, just like the existing JPEG or PNG loaders. Reading TIFF turned out to be much more complex since there is a wide variety of TIFF files that all need special code. The following input formats and TIFF file organizations are supported:

Table 2 — Sample Format / Bit Depth / Colorspace

Sample Format	Bits/Sample	Samples/Pixel	Colorspace	Notes
UINT	8	1	Monochrome	
UINT	9-16	1	Monochrome	downshifted to output_bit_depth
UINT	8	3	RGB	
UINT	9-16	3	RGB	downshifted to output_bit_depth
UINT	8	4 (alpha)	RGBA	alpha via EXTRASAMPLE_ ASSOCALPHA/UNASSALPHA
UINT	9-16	4 (alpha)	RGBA	
UINT	8	4 (no alpha)	RGB	4th channel silently dropped
UINT	9-16	4 (no alpha)	RGB	4th channel silently dropped
UINT (YCbCr)	8	3	YCbCr 4:4:4 / 4:2:2 / 4:2:0	PHOTOMETRIC_YCBCR, subsampling-aware
Signed INT	8	1	Monochrome (nonvisual)	requires WITH_UNCOMPRESSED_ CODEC
Signed INT	9-16	1	Monochrome (nonvisual)	requires WITH_UNCOMPRESSED_ CODEC
Float (IEEE)	32	1	Monochrome (nonvisual)	requires WITH_UNCOMPRESSED_ CODEC

1. Organization / Layout

Table 3

Layout	Notes
Stripped, PLANARCONFIG_CONTIG (pixel-interleaved)	TIFFReadScanline path
Stripped, PLANARCONFIG_SEPARATE (band-interleaved)	reads scanlines per-band, interleaves into output
Tiled, PLANARCONFIG_CONTIG	TIFFReadEncodedTile path
Tiled, PLANARCONFIG_SEPARATE	reads tiles per-band, interleaves into output

Tiled (via TiledTiffReader)	tile-at-a-time API for large images; detects overview directories (reduced-resolution sub-IFDs)

The `TiledTiffReader` does not read the full image at once, but provides an interface to iterate through all overview images and image tiles such that the TIFF image organization can be mirrored in the GIMI output, with each tile being encoded independently. This allows to convert images much larger than the available memory.

The following TIFF features are not supported yet:

- palette (no support in libheif yet),
- floating point with bits-per-pixel other than 32,
- multi-channel float / signed integer,
- integer (unsigned and signed) with bits-per-pixel larger than 16, and
- samples per pixel other than 1, 3, or 4.

2.12.2. JPEG loader changes

The JPEG loader was extended to support CMYK colorspace in the input. This is converted to RGB since libheif has no CMYK encoding support yet.

2.12.3. RAW image loader

Since some Testbed participants wanted to convert raw data, especially floating point data, a reader for raw data files was also implemented. This is also a loader module for `heif-enc`, but the user has to specify the image dimensions since there is no header in these files:

```
--raw                       force raw pixel data input (for files without .raw/.
img suffix)
--raw-width #               width of raw input image (computed from file size if
omitted)
--raw-height #              height of raw input image (computed from file size if
omitted)
--raw-type TYPE             pixel data type: uint8, sint8, uint16, sint16,
uint32, sint32, float32, float64
--raw-endian ENDIAN         byte order of input data: little (default), big
```

This is currently the only method to input 32 bit signed/unsigned integers and 64 bit floats into `heif-enc`. The `--raw` option can be omitted if the input filename ends with a `.raw` or `.img` suffix. It is also sufficient to specify either a width or height. The other dimension is computed automatically from the file size and the known dimension.

2.13. Conversion tool usage

To simplify the conversion of existing GeoTIFF imagery to GIMI, a set of scripts is provided in the `gimi-tools` repository (https://github.com/farindk/gimi-tools). The conversion is a two-step pipeline, illustrated in Figure 10:

1. A Python script (`geotiff_metadata_to_turtle.py`) reads the geospatial metadata from the GeoTIFF and generates an RDF/Turtle file containing the geographic coordinates as GeoSPARQL geometries. This python script is a prototype implementation and is separate from the `geotiff_to_gimi.sh` script so that it can be easily replaced with a different, more elaborate metadata generation script.

2. The shell wrapper (`geotiff_to_gimi.sh`) invokes this Python script and then calls `heif-enc` to encode the GeoTIFF as a GIMI file, passing the generated Turtle metadata along with the `--unif` and `--add-compatible-brand geo1` flags required for GIMI conformance.

By default, the Turtle metadata is written to a temporary file and embedded into the GIMI output via `--embed-turtle`. Alternatively, the `--ttl` option saves the Turtle file to a user-specified path for use as a sidecar, in which case `--embed-turtle` is not added.

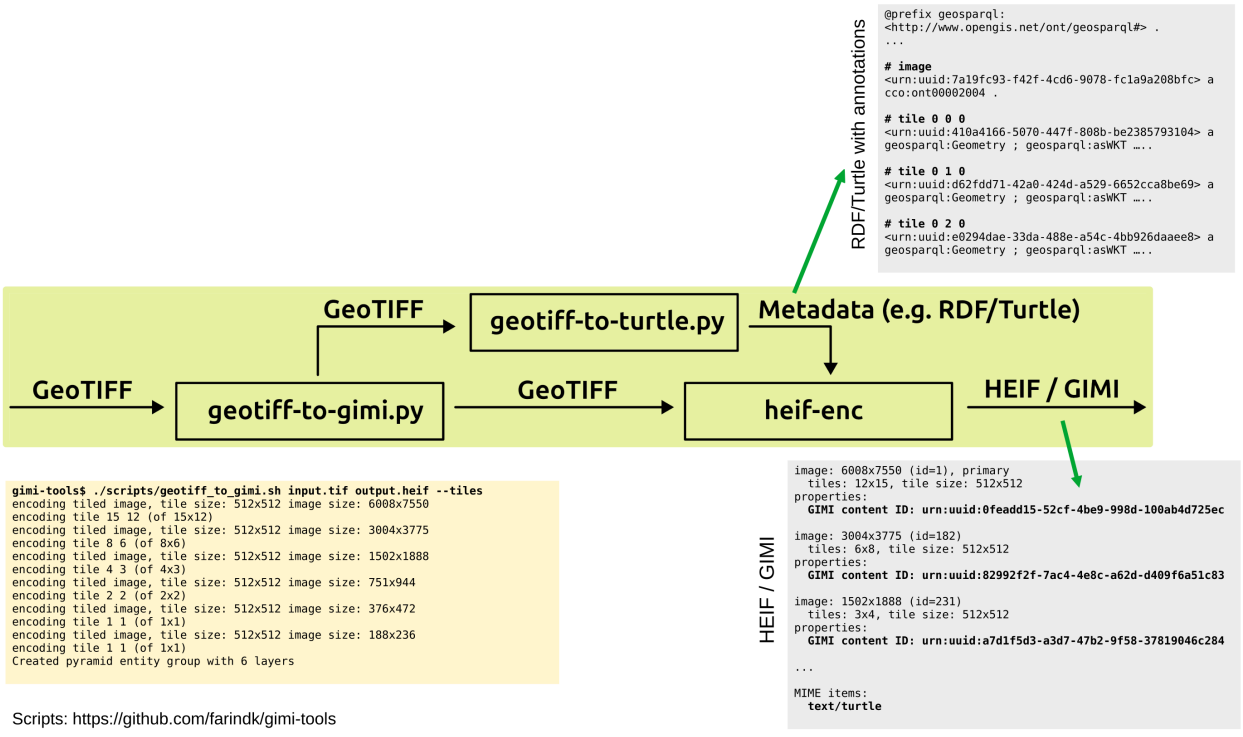


Figure 10

2.13.1. Shell wrapper usage

The basic syntax of the wrapper script is:

```
geotiff_to_gimi.sh <input.tif> [options...] -o output.heif
```

Options handled by the script itself are:

```
--tiles                    generate per-tile geometry metadata (forwarded to the
Python script)
--ttl NAME                 save the Turtle metadata to NAME instead of embedding
it
--metadata-script FILE     use FILE instead of the default `geotiff_metadata_to_
turtle.py` script
```

All other options are forwarded verbatim to `heif-enc`. This means that any `heif-enc` option, such as the compression format, quality, or tiling mode, can be appended directly.

```
# Simple conversion with default settings (H.265, embedded Turtle)
geotiff_to_gimi.sh image.tif -o output.heif

# Tiled conversion with per-tile metadata and quality 50
geotiff_to_gimi.sh image.tif --tiles -q 50 -o output.heif

# Keep the Turtle file as a sidecar
geotiff_to_gimi.sh image.tif --tiles --ttl metadata.ttl -o output.heif
```

Listing — Examples

2.13.2. Metadata generation

The Python script `geotiff_metadata_to_turtle.py` uses the `rasterio` library to read the GeoTIFF and emits RDF/Turtle to standard output. The script requires `rasterio` (`pip install rasterio`).

The script can be used stand-alone:

```
python3 geotiff_metadata_to_turtle.py image.tif            # full-image bounds
only
python3 geotiff_metadata_to_turtle.py image.tif --tiles    # add per-tile bounds
```

The generated Turtle consists of:

- an **image resource** typed as `cco:ont00002004`, identified by a randomly generated UUID; and
- one or more **geometry resources** of type `geosparql:Geometry`, each carrying a WKT polygon in EPSG:4326 coordinates (`geosparql:asWKT`) and linked to the image resource via `cco:ont00001808`.

If the GeoTIFF's coordinate reference system is not EPSG:4326, the bounds are automatically reprojected to WGS 84.

An example of the generated Turtle for a single tile:

```
@prefix cco: <https://www.commoncoreontologies.org/> .
@prefix geosparql: <http://www.opengis.net/ont/geosparql#> .

# image
<urn:uuid:7d29f93-f42f-4cd8-9078-fc1a9a208bfc> a cco:ont00002004 .

# tile 0 0 0
<urn:uuid:430a4166-5079-4477-888b-bc23857931b4> a geosparql:Geometry ;
    geosparql:asWKT "POLYGON((-64.83 18.37,-64.83 18.33,...))"^^geosparql:
wktLiteral ;
    cco:ont00001808 <urn:uuid:7d29f93-f42f-4cd8-9078-fc1a9a208bfc> .
```

When `heif-enc` reads this Turtle file via `--turtle`, it matches the geometry UUIDs to the corresponding GIMI content IDs assigned to the tiles in the HEIF file, establishing the link between the geospatial metadata and the image tiles.

2.13.3. Tiling and overview pyramids

Without the `--tiles` flag (or when the input GeoTIFF is not internally tiled), the script emits a single geometry covering the full image bounds.

With `--tiles`, the script iterates over the GeoTIFF's internal tile grid and emits one geometry per tile. It also processes all overview (pyramid) levels stored in the GeoTIFF. For each overview level, the pixel-to-coordinate mapping is scaled by the overview's downsample factor, and the reduced-resolution tile grid is iterated in the same manner. This produces a complete set of geospatial geometries for every tile at every resolution level, which `heif-enc` can then use to populate the GIMI content IDs across the full image pyramid.

When converting a tiled GeoTIFF, `heif-enc` reads the tiles individually through its `TiledTiffReader`, allowing conversion of images much larger than available memory. The resulting GIMI file mirrors the GeoTIFF's tiling structure and overview hierarchy as a pyramid entity group.

2.14. ISO 23001-17 uncompressed codec

The ISO 23001-17 uncompressed codec is built directly into libheif rather than being a separate plugin, because it relies extensively on ISOBMFF box parsing and writing that are integral to the library. During Testbed-21, the uncompressed codec underwent a major refactoring and was extended with support for new data types, multi-channel images, Bayer/filter-array images, and several ISO 23001-17 metadata boxes.

2.14.1. Encoder/decoder architecture

The previous implementation of the ISO 23001-17 codec already had separate code paths for each interleave type, but these were selected by conditional branches within a single

function and shared much of their surrounding logic. As the number of supported configurations grew in interleave types, bit depths, padding modes, endianness, and data types, this became increasingly difficult to maintain and extend. The complex, very general implementation also made the decoding unnecessarily slow. During Testbed-21, both the decoder and encoder were refactored into a modular architecture where a subset of configuration variants is implemented in its own class. A factory class selects the appropriate module at runtime based on the full set of `uncC` (uncompressed configuration) box parameters, not just the interleave type. This means that, for example, the same interleave mode can have multiple decoder implementations: one optimized for byte-aligned components with arbitrary data types, another for sub-byte bit depths with block packing, and so on. This results in an easier implementation and much faster runtime for the commonly used formats.

An important design principle is that the `heif_image` representation used by libheif and the application does not expose the full combinatorial space of `uncC` configurations. Instead, libheif maintains a small set of choices for pixel formats in `heif_image`:

- independent planar components, each with varying bits per pixel and varying data type; and
- interleaved RGB(A) pixels with 1-16 bits per color component.

All the many possible `uncC` data layouts are decoded into one of these formats, and conversely, these formats are encoded into the appropriate `uncC` configuration by the encoder factory. This keeps the API simple for users, who only need to work with a small, well-defined set of image formats. Furthermore, libheif provides automatic conversion between all supported pixel formats, so an application can request any output format regardless of how the data is stored in the file.

This modular architecture improves maintainability: adding support for a new `uncC` configuration requires only adding a new codec module without touching the existing decoders or encoders.

Decoder architecture.

When decoding an ISO 23001-17 image, a factory class inspects the `uncC` box parameters and selects the first decoder module whose requirements match the configuration. This enables having a set of simple but fast decoders that are preferably selected when they are capable of decoding the input image, and only switches to more complex implementations when required. Seven concrete decoder modules are currently implemented, each in a separate decoder class:

Table 4 — Decoder Modules

DECODER MODULE NAME	INTERLE MODE	BPP	COMPOI FORMAT	SAMPLIN	ROW_ ALIGN	TILE_ ALIGN	PIXEL_ SIZE	BLOCK_ SIZE	ENDIANNESS
component	compone tile_ compone	1-16	UINT	none, 4:2:2, 4:2:0	yes	yes	no	no	
pixel	pixel	1-16	UINT	N/A	yes	yes	optional	no	big only

DECODER MODULE NAME	INTERLE MODE	BPP	COMPOI FORMAT	SAMPLIN	ROW_ ALIGN	TILE_ ALIGN	PIXEL_ SIZE	BLOCK_ SIZE	ENDIANNESS
mixed	mixed	1-16	UINT	4:2:2, 4:2:0	yes	yes	no	no	
row	row	1-16	UINT	N/A	yes	yes	no	no	
block_pixel	pixel	1-16	UINT	N/A	yes	yes	required	0 or pixel_ size	controlled by block flags
block_component	compone	1-16	UINT	N/A	yes	yes	no	1-8 bytes	controlled by block flags
bytealign_ component	compone	[8, 16, 32, 64, 128]	all	N/A	yes	yes	no	no	little + big

All decoder modules support any values for `row_align_size` (row padding) and `tile_align_size` (tile padding). The decoder modules are generally independent of the image colorspace. They operate on the data layout described in the `uncC` box and do not need to know whether the components represent RGB, YCbCr, or arbitrary sensor channels. The exception is the mixed-interleave decoder, which is specific to YCbCr images with chroma subsampling.

The `bytealign_component` decoder is the only module that supports data types beyond unsigned integers. It handles signed integers, IEEE 754 floating-point, and complex numbers at bit depths of 8, 16, 32, 64, and 128 bits (128-bit for complex numbers only, representing two 64-bit floats). This module also supports both little-endian and big-endian component byte order.

The `component` decoder also handles the tile_component interleave mode (mode 4), where tile data for each component is stored separately in the file rather than being concatenated into a single block per tile.

Encoder architecture.

The encoder follows a symmetric factory class design. The factory class selects the appropriate encoder based on the image's colorspace, chroma format, data type, and encoding options. The selected encoder decides which 'uncC' parameters are best for encoding the given image. The encoder's primary goal is to write the data as compactly as possible.

Four concrete encoder modules are implemented:

Table 5 — Encoder Modules

ENCODER MODULE	INPUT COLORSPACE	INPUT CHROMA	DATATY	BPP	OUTPU INTERLI	ALPHA	UNCC OUTPUT PARAMETERS
rgb_pixel	RGB	interleaved RGB/RGBA	UINT	1-8	pixel	yes	profile `rgb3/rgba` for 8-bit; no block, padding for <8 bit

ENCODER MODULE	INPUT COLORSPACE	INPUT CHROMA	DATATY	BPP	OUTPU INTERLI	ALPHA	UNCC OUTPUT PARAMETERS
rgb_block_pixel	RGB	interleaved RRGGBB_LE/BE	UINT	9-13	pixel	no	components packed into blocks; block_little_endian = true
rgb_bytealign_pixel	RGB	interleaved RRGGBB[AA]_LE/BE	UINT	9-16	pixel	yes	pixel_size set; components always big-endian
component	all	all non-interleaved	all	1-32, 64, 128, 256	compon	yes	native endian for byte-aligned data; bit-packing big-endian otherwise; sampling_mode for YCbCr 4:2:2/4:2:0

The native libheif image formats are either separate pixel planes for each component or interleaved RGB. Three of the encoder modules are specific for the RGB interleaved case with different bit depths. The `component` encoder is the universal fallback that accepts any colorspace and data type. It is the only encoder that supports signed integers, floating-point, complex numbers, and YCbCr subsampling. The three RGB-specific encoders handle pixel-interleaved output in various packed formats.

The following figures illustrate the bit-level data layout produced by the `rgb_pixel` and `rgb_block_pixel` encoders for a single RGB pixel at different bit depths. The colored blocks represent the bits of the red, green, and blue components. Bit numbers within each byte are shown at the top.

Generally, for bit-depths smaller than 8 bits per component, the values are padded to 8 bit. For bit-depths 9 to 13, the RGB values are packed densely into 4 or 5 byte blocks and then padded to the next byte boundary. The rationale for this is that the data should be arranged for a good compressibility with the “deflate” or “brotli” algorithms. These algorithms work on a per-byte basis, detecting runs and repetitions of byte sequences, and their efficiency would be reduced if pixel values are not aligned to byte boundaries. For values with 9 to 13 bits per component, on the other hand, padding would inflate the amount of data so much that it probably cannot be compensated by a better compression. Thus, the RGB pixel values are packed into 4 or 5 bytes. This allows to still efficiently compress runs of similar RGB pixels.

How much can be gained by this arrangement depends on the input data content. A thorough evaluation is still open for research.

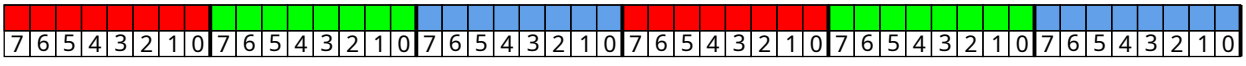


Figure 11 — Pixel-interleaved RGB at 8 bits per component. Each component occupies exactly one byte, resulting in 3 bytes per pixel.

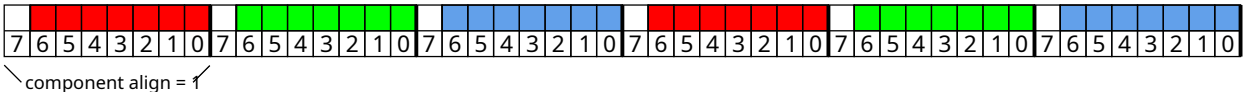


Figure 12 — Pixel-interleaved RGB at 7 bits per component with `component_align_size = 1`. Each 7-bit component is padded to a full byte, resulting in 3 bytes per pixel.

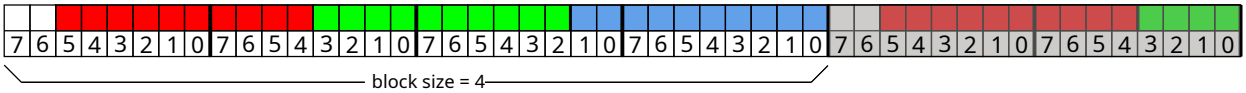


Figure 13 — Block-packed RGB at 10 bits per component with `block_size = 4`. Three 10-bit components (30 bits total) are packed into a 4-byte (32-bit) block.

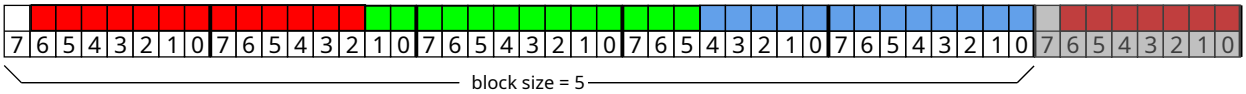


Figure 14 — Block-packed RGB at 13 bits per component with `block_size = 5`. Three 13-bit components (39 bits total) are packed into a 5-byte (40-bit) block.

2.14.2. Data types and multi-channel images

Extended data type support. Prior to Testbed-21, the uncompressed codec only supported unsigned integer components at 8 to 16 bits per sample. During Testbed-21, the codec was extended to support a wider range of data types as defined in ISO 23001-17:

Table 6 — Supported Data Types

DATA TYPE	SUPPORTED BIT DEPTHS	C API ACCESSOR TYPE
Unsigned integer	8, 16, 32, 64	`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
Signed integer	8, 16, 32, 64	`int8_t`, `int16_t`, `int32_t`, `int64_t`
IEEE 754 floating-point	32, 64	`float`, `double`
Complex number	2×32, 2×64	`heif_complex32`, `heif_complex64`

These data types are expressed in the ISO 23001-17 `uncC` box via the `component_format` field, with values `component_format_unsigned` (0), `component_format_signed` (3), `component_format_float` (1), and `component_format_complex` (2). On the C API side, the new `heif_channel_datatype` enum represents these four data types. Each data type has a corresponding pair of pixel accessor functions (read-only and read-write) in `heif_uncompressed.h`, all indexed by component index:

```
const float*  heif_image_get_component_float32_readonly(image, component_id,
&stride);
float*        heif_image_get_component_float32(image, component_id, &stride);
```

```
// similarly for uint8, uint16, uint32, uint64, int8, int16, int32, int64,
//   float64, complex32, complex64
```

Multi-channel image model.

Alongside the data type extensions, the internal image representation was changed from a fixed map of `heif_channel` enum values (such as `heif_channel_Y`, `heif_channel_Cb`, `heif_channel_R`) to a list of components identified by integer index. This new model directly reflects the ISO 23001-17 component definition box (`cmpd`), where each component has an index and a type.

The key API additions for the multi-channel model are:

```
// Add a component with arbitrary type, data type, and bit depth
heif_image_add_component(image, width, height, component_type, datatype, bit_
depth, &out_idx);

// Query the number of components with pixel data and the total cmpd components
heif_image_get_number_of_used_components(image);
heif_image_get_total_number_of_cmpd_components(image);

// Get the valid component ids (which may be non-contiguous)
heif_image_get_used_component_ids(image, out_ids);

// Query per-component properties
heif_image_get_component_width(image, component_id);
heif_image_get_component_height(image, component_id);
heif_image_get_component_bits_per_pixel(image, component_id);
heif_image_get_component_type(image, component_id);
```

Not all components declared in the `cmpd` box need to have pixel data planes. For example, a Bayer-pattern image may have a single `filter_array` component with pixel data, but the `cmpd` box also declares reference components (red, green, blue) that are used by the color pattern box (`cpat`) to describe which color each filter array pixel represents. These reference components have `cmpd` entries but no pixel planes.

The set of supported component types was extended to cover the full ISO 23001-17 Table 1:

```
monochrome, Y, Cb, Cr, red, green, blue, alpha,
depth, disparity, palette, filter_array, padded,
cyan, magenta, yellow, key_black
```

This enables images with arbitrary component configurations, such as CMYK images (cyan/magenta/yellow/key_black), depth maps, or disparity images, alongside the traditional RGB and YCbCr colorspaces.

Since backwards compatibility has to be maintained, images can be constructed and read using either the new component types or with the established `heif_colorspace` / `heif_channel` combination. When reading an ISO 23001-17 image, libheif analyzes what components have been defined. If these match a common pattern that can be expressed with `heif_colorspace` and `heif_channel`, otherwise, it assigns the new `heif_colorspace_nonvisual` to it. For the opposite direction, when the user generates an image based on a `heif_colorspace` preset, libheif creates the corresponding ISO 23001-17 components internally.

2.14.3. Bayer-image decoding

A color filter array (CFA) sensor captures a single color per pixel in a repeating pattern. The most common arrangement is the 2×2 Bayer pattern (RGGB), but other patterns such as RGBW (with a panchromatic/white channel) or Quad Bayer Coding (QBC) with 4×4 patterns are also used. ISO 23001-17 represents such images using a single `filter_array` component for the pixel data and a color pattern box (`cpat`) that describes the repeating pattern layout.

Encoding Bayer images.

To encode a Bayer image, the application creates an image with the new `heif_colorspace_filter_array` colorspace, which uses a single channel for the filter array pixel data. The Bayer pattern is attached to the image via:

```
typedef struct heif_bayer_pattern_pixel
{
  uint32_t component_id;
  float component_gain;
} heif_bayer_pattern_pixel;

heif_error heif_image_add_bayer_component(heif_image*,
                                          uint16_t component_type,
                                          uint32_t* out_component_id);

heif_error heif_image_set_bayer_pattern(heif_image*,
                                        uint32_t bayer_component_id,
                                        uint16_t pattern_width,
                                        uint16_t pattern_height,
                                        const heif_bayer_pattern_pixel*
patternPixels);
```

where each `heif_bayer_pattern_pixel` specifies a component ID and an optional gain value. To get the component IDs, a Bayer component first has to be added with `heif_image_add_bayer_component()`, which maps the component type to a new component ID. These can then be used in the `heif_bayer_pattern_pixel`.

Note that `heif_image_set_bayer_pattern()` requires the `bayer_component_id` to which filter-array component this pattern belongs, but ISO 23001-17 only allows one filter-array component with a `cpat` per image. Since this might be an oversight in the ISO 23001-17 standard, the API was designed to be future-proof for potential updates of the standard.

Decoding Bayer images.

On the decoder side, the `cpat` box is read and its component indices are validated against the `cmpd` table. The Bayer pattern is then attached to the decoded `heif_image`. An application can access the raw filter-array pixel data directly, or it can request automatic demosaicing during the color conversion step. A bilinear interpolation demosaicing algorithm was implemented that converts the filter-array image to interleaved RGB, supporting both 8-bit and 9-16 bit images. The demosaicing algorithm is integrated into the color-conversion pipeline such that it operates transparently if the application specifies a specific output colorspace that libheif should convert the image to.

The following functions have been added to the API to read the Bayer pattern information. The function `heif_image_get_bayer_pattern()` returns the pattern with the component IDs. These can be resolved to component types, with `heif_image_get_component_type()`.

```
int heif_image_get_bayer_pattern_size(const heif_image*,
                                      uint32_t bayer_component_id,
                                      uint16_t* out_pattern_width,
                                      uint16_t* out_pattern_height);

// Get the Bayer / filter array pattern pixels.
// The caller must provide an array large enough for pattern_width * pattern_
height entries
// (use heif_image_get_bayer_pattern_size() to query the dimensions first).
// Returns heif_error_Ok on success, or an error if no pattern is set.
heif_error heif_image_get_bayer_pattern(const heif_image*,
                                        uint32_t bayer_component_id,
                                        heif_bayer_pattern_pixel* out_
patternPixels);

uint16_t heif_image_get_component_type(const heif_image*, uint32_t component_id);
```

The `heif-gen-bayer` tool.

To facilitate testing and demonstration of Bayer-pattern image support, a new command-line tool `heif-gen-bayer` was implemented. This tool takes an RGB PNG image as input, simulates a Bayer-filtered sensor capture by sampling the appropriate color channel at each pixel according to the selected pattern, and writes the result as an ISO 23001-17 uncompressed image with the `cpat` box.

The tool supports several built-in filter array patterns:

- **RGGB** (2×2): the standard Bayer pattern;
- **RGBW** (4×4): a pattern including panchromatic (white/unfiltered) pixels; and
- **QBC** (4×4): Quad Bayer Coding pattern, where each color occupies a 2×2 block.

Custom patterns can also be specified as a string of R/G/B characters (e.g., `-p BGGR`).

```
heif-gen-bayer [options] <input.png> <output.heif>
heif-gen-bayer -S [options] <frame_NNN.png> <output.mp4>

Options:
  -b, --bit-depth #       output bit depth (default: 8, range: 8-16)
  -p, --pattern <name>    filter array pattern (default: rggb)
  -S, --sequence          sequence mode (expand numbered PNGs)
  -V, --video             use video track handler (vide) instead of pict
      --fps <N>           frames per second (default: 30)
```

In sequence mode (`-S`), the tool encodes multiple numbered PNG frames as an uncompressed image sequence or video track with the Bayer metadata attached to each frame.

Table 7 shows the result of encoding an RGB image through `heif-gen-bayer` with the RGGB pattern and then decoding it back with `heif-dec`, which applies bilinear demosaicing.

Table 7 — Bayer encoding/decoding example. Left: original RGB image. Right: image after Bayer-pattern encoding with `heif-gen-bayer` and decoding/demosaicing with `heif-dec`.

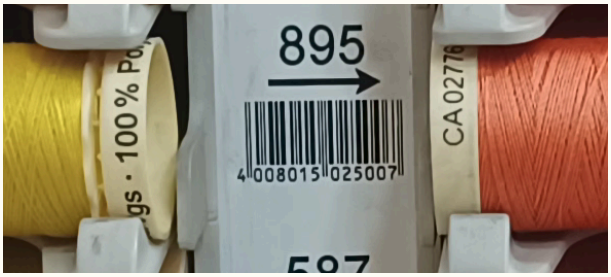


Figure 15

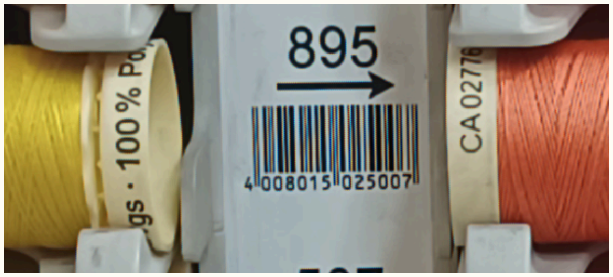


Figure 16

2.14.4. ISO 23001-17 image metadata

ISO 23001-17 defines several optional metadata boxes that can be associated with uncompressed image items. During Testbed-21, support was added for five of these metadata types, covering sensor characteristics and imaging parameters. All metadata types survive the encode/decode roundtrip: the encoder reads the metadata from the `HeifPixelImage` and writes the corresponding ISO boxes, while the decoder reads the boxes and attaches the data to the decoded image.

Table 8 — Supported ISO 23001-17 Metadata Boxes

BOX	ISO SECTION	PURPOSE	API PATTERN
`cpat`	6.1.3	Color filter array pattern (Bayer)	`heif_image_set/get_bayer_pattern`
`cloc`	6.1.4	Chroma sample location	`heif_image_set/has/get_chroma_location`
`splz`	6.1.5	Polarization filter pattern	`heif_image_add/get_polarization_pattern`
`snuc`	6.1.6	Sensor non-uniformity correction	`heif_image_add/get_sensor_nuc`
`sbpm`	6.1.7	Sensor bad pixels map	`heif_image_add/get_sensor_bad_pixels_map`

Color pattern box (`cpat`). This box describes a repeating color filter array pattern as used in Bayer-type sensors. Each position in the pattern references a component index in the `cmpd` table and optionally specifies a gain value. The encoding and decoding of this box is described in detail in the Bayer-Image Decoding section above.

Chroma sample location (`cloc`). This box specifies the spatial position of chroma samples relative to luma samples in 4:2:0 subsampled images. It uses integer values 0 through 5 from

ISO 23091-2 / ITU-T H.273, plus an additional value 6 defined by ISO 23001-17 for a two-row specification of chroma positions. The API provides a straightforward set/has/get interface:

```
heif_image_set_chroma_location(image, location);   // location: 0-6
heif_image_has_chroma_location(image);              // returns non-zero if set
heif_image_get_chroma_location(image);              // returns location value
```

Polarization pattern (`splz`). This box describes a polarization filter array over the sensor pixels. Each position in the pattern specifies a polarization angle in degrees (0.0 to 360.0), or a special IEEE 754 NaN value (bit pattern `0xFFFFFFFF`) to indicate "no polarization filter" at that position. Multiple polarization patterns can be added, each associated with a different group of components. The API supports querying which pattern applies to a given component index:

```
heif_image_add_polarization_pattern(image, num_component_ids, component_ids,
                                    pattern_width, pattern_height, polarization_
angles);
heif_image_get_number_of_polarization_patterns(image);
heif_image_get_polarization_pattern_info(image, pattern_index, ...);
heif_image_get_polarization_pattern_data(image, pattern_index, ...);
heif_image_get_polarization_pattern_index_for_component(image, component_id);
```

Sensor non-uniformity correction (`snuc`). This box stores per-pixel gain and offset correction tables. The correction equation is `y = gain * x + offset`, where `x` is the raw sensor value and `y` is the corrected value. The gain and offset arrays are full image-sized arrays of `float` values. A flag indicates whether the correction has already been applied to the pixel data. Multiple NUC tables can be added, each for a different group of components:

```
heif_image_add_sensor_nuc(image, num_component_ids, component_ids,
                          nuc_is_applied, image_width, image_height,
                          nuc_gains, nuc_offsets);
heif_image_get_number_of_sensor_nucs(image);
heif_image_get_sensor_nuc_info(image, nuc_index, ...);
heif_image_get_sensor_nuc_data(image, nuc_index, ...);
```

Sensor bad pixels map (`sbpm`). This box records known defective sensor positions: entire bad rows, entire bad columns, and individual bad pixels identified by (row, column) pairs. Like the NUC box, a flag indicates whether the bad-pixel correction has already been applied. Multiple maps can be added for different component groups:

```
heif_image_add_sensor_bad_pixels_map(image, num_component_ids, component_ids,
                                     correction_applied,
                                     num_bad_rows, bad_rows,
                                     num_bad_columns, bad_columns,
                                     num_bad_pixels, bad_pixels);
heif_image_get_number_of_sensor_bad_pixels_maps(image);
heif_image_get_sensor_bad_pixels_map_info(image, map_index, ...);
heif_image_get_sensor_bad_pixels_map_data(image, map_index, ...);
```

All of these API functions are declared in the public header `heif_uncompressed.h`, with supporting type definitions in `heif_uncompressed_types.h`.

2.14.5. Mapping of component IDs to `cmpd` and `uncC` indices

A challenge during implementing multi-spectral or Bayer-encoded images according to the ISO 23001-17 standard was that there is no consistent way to identify the components. Furthermore, the implementation has to be backwards compatible to the existing libheif API. The existing libheif implementation was targeting either monochrome, RGB, or YCBCr images

with optional auxiliary channels like alpha, depth, or disparity. The pixel data was organized in planes that directly mapped to the colorspace, e.g., R, G, B, Y, Cb, Cr as planar pixel planes, indexed by the color channel. As a special case, there was also an interleaved RGB(A) format as a single plane, since many applications exchange image data in this organization.

For a complete ISO 23001-17 implementation, this scheme does not work anymore since an ISO 23001-17 image may contain several monochrome components, or even user-defined components with no pre-defined type. On the other hand, there is no direct correspondence to an "interleaved RGB" component in ISO 23001-17. Another difficulty was that ISO 23001-17 may contain Bayer images, which define components with no directly coded image data, but metadata can still be assigned to these non-coded components.

When choosing a suitable ID for identifying components in the API, it was noticed that

- `cmpd` indices, as are often used in ISO 23001-17 for assigning metadata to components, does not work in the API because several coded components can share the same `cmpd` index,
- `uncC` indices does not work because Bayer images define additional components without any corresponding index in `uncC`,
- the existing `heif_channel` enumeration type does not work because of its limitation to RGB, YCbCr, and a few metadata channels, but not several of the same type.

Thus, a new "component ID" had to be introduced that is used in the library API to uniquely identify the components, working both for coded image components and Bayer pattern components. There is no direct, easy correspondence between the "component ID" and a `cmpd` or `uncC` index. Instead, libheif will use the component ID exclusively in the API and generate suitable `cmpd` and `uncC` entries when encoding. Trying to combine similar entries to save space.

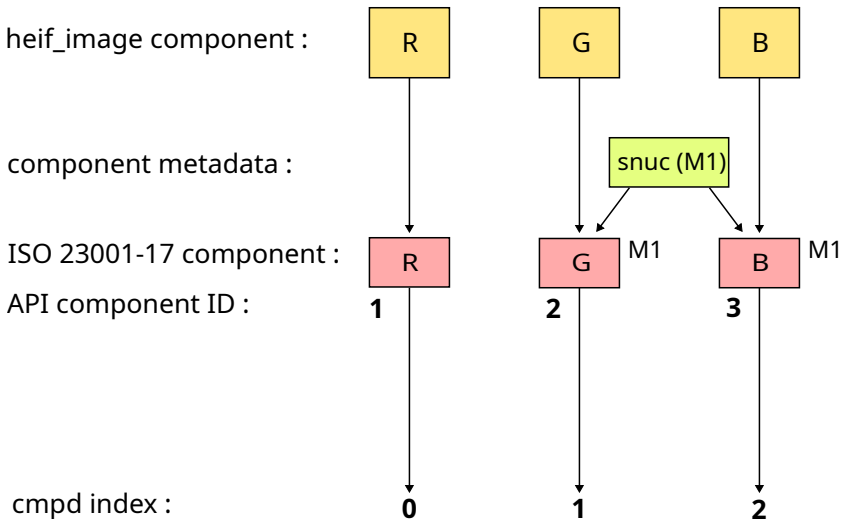


Figure 17 — Simple case of mapping `cmpd` indices to component IDs in a RGB image.

A very simple case, without any of the aforementioned difficulties, is shown in Figure 17. It shows a `heif_image` three image planes containing the red, green, and blue components (yellow boxes). Each of these `heif_channel` components maps directly to an ISO 23001-17 component (red boxes) and receives its unique component ID. A `snuc` metadata box (Sample Non-Uniform

Correction) is assigned to the green and blue component. Since all three components have a different type, they receive their own cmpd entry and the snuc metadata box is stored with references to cmpd indices 1 and 2.

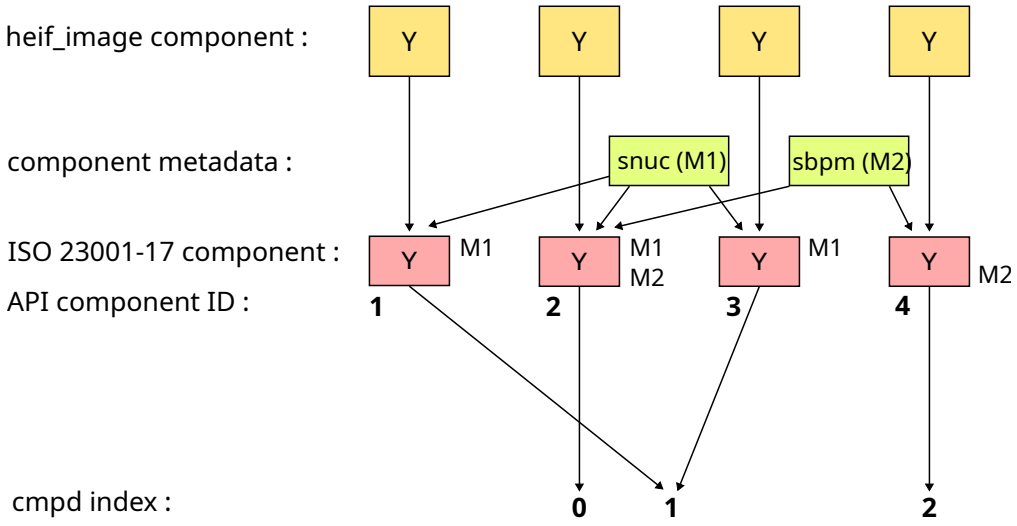


Figure 18 — Mapping component IDs to cmpd indices for a multi-spectral image.

A more complex case is shown in Figure 18. It shows a multi-spectral image with four monochrome channels. A snuc metadata box is assigned to the components with IDs 1,2,3 and a sbpm metadata box is assigned to the components with IDs 2 and 4. Now, even though all components have the same type, they cannot be mapped to the same cmpd entry because the ISO 23001-17 standard says that metadata boxes are assigned to the components indirectly through the cmpd index. Thus, libheif examines which components have the same type and the same metadata assigned to it. In this example, this is the case only for components 1 and 3. These two component IDs can be both mapped to the cmpd index 1. The snuc metadata box will finally reference cmpd entry 0 and 1, while sbpm will reference 0 and 2.

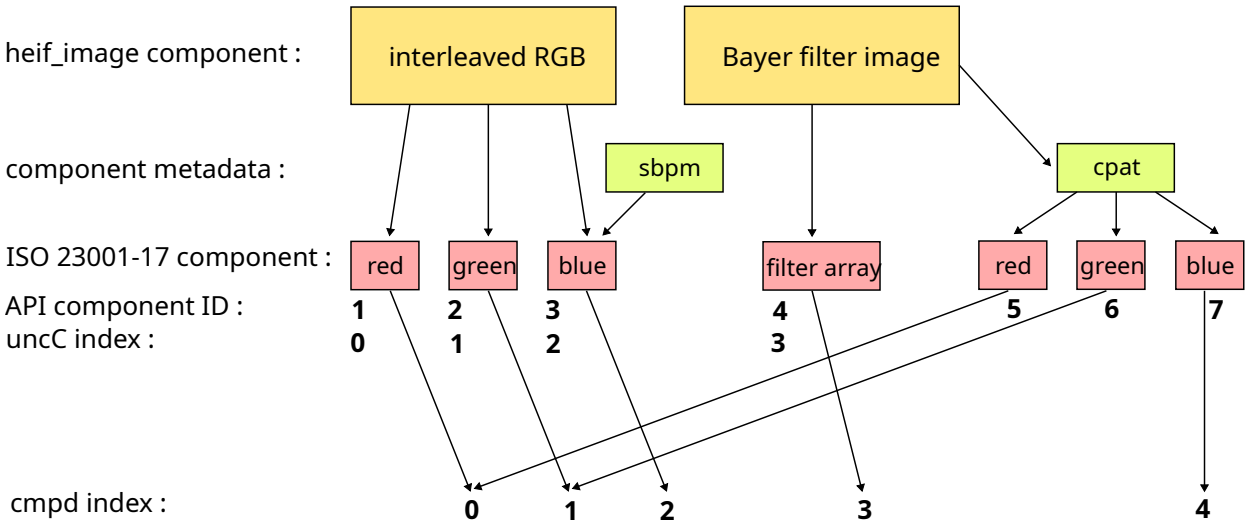


Figure 19 — Mapping component IDs to cmpd indices for an image with interleaved RGB and a Bayer image.

The example in Figure 19 depicts an image with an interleaved RGB plane and an independent Bayer filter image. The interleaved RGB plane is stored in a single `heif_image` plane inside libheif, but since there is no direct correspondence to that in ISO 23001-17, this interleaved RGB plane, maps to three ISO 23001-17 components (1-3). The Bayer filter image only has one coded component (4) for the filter array, but the `cpat` box in this example signals that the filter array should decode to three R,G,B components (not to be confused with the independent interleaved image). These Bayer RGB components have no directly coded pixel data, but they receive their own component IDs (5-7) such that it is possible to assign metadata to them. In this example, libheif recognizes that the R,G,B components have the same type, but since the blue component in the interleaved image has a `sbpm` metadata box assigned to it, a separate `cmpd` entry is required. In the figure of this example, a line with the `uncC` indices has been added to show that there is a second mapping inside libheif that maps the component IDs to `uncC` indices.

When decoding a HEIF file with an ISO 23001-17 image, the whole process runs in the opposite direction. The `cmpd` indices are replaced with unique component IDs that are then used in the API. Note that the component IDs are volatile and may be different at the encoding and decoding side.

2.14.6. “unci” compression benchmark

During image encoding experiments, it was observed that saving “unci” images with `brotli` compression was much slower than with `deflate` compression. Both standard implementations of these encoding algorithms come with a selectable compression level setting, with 0-11 for `brotli` and 1-9 for `deflate` (excluding `deflate` level 0, which is uncompressed). The current implementation in libheif always uses the default levels, which is 11 (highest compression) for `brotli` and 6 (balanced) for `deflate`.

To assess the trade-off between file size and encoding time offered by the compression options of ISO 23001-17 (“unci”), a benchmark was run on a large test image (approximately 6.2 GB uncompressed) using both the `deflate` and `brotli` general-purpose compressors at all supported compression levels. Measurements were taken on an AMD Ryzen 9 9950X CPU. Both compression algorithms utilize only a single CPU core.

Figure 20 shows file size versus encoding time for the relevant range of compression levels. The uncompressed case (about 6.2 GB, 11 s) and the two slowest `brotli` settings (level 10: 16 min; level 11: 35 min) are omitted from the plot so that the interesting region, where `brotli` and `deflate` can be directly compared, is visible. The numbers next to each data point indicate the compression level.

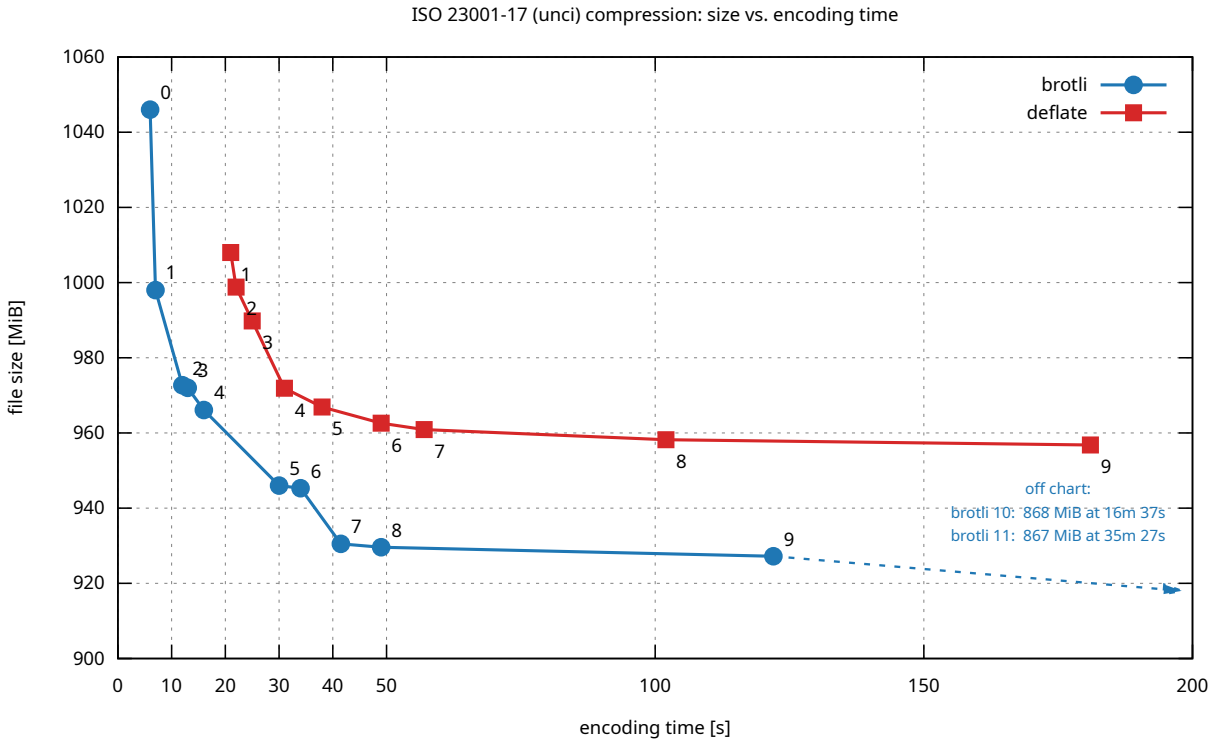


Figure 20 — File size versus encoding time for "unci" compression with brotli and deflate. Labels indicate the compression level.

For this image, `brotli` clearly dominates `deflate`: across the entire range, the `brotli` curve lies below and to the left of the `deflate` curve, meaning `brotli` produces smaller files in less time. Already at lowest settings (`brotli` level 1 to 4), `brotli` compresses faster than any `deflate` level while producing a file comparable in size to `deflate` at moderate levels. At higher quality settings, `brotli` reaches file sizes that `deflate` cannot match at any level.

It is also worth noting that `brotli` level 0 and 1 (6 s and 7 s) is faster than storing the uncompressed data (11 s): the compression effort is more than offset by the time saved writing a roughly six times smaller file to the SSD. In other words, using light `brotli` compression is not only smaller but also faster than writing “unci” data without any compression.

These results confirm that for this type of imagery, `brotli` is the preferred choice among the general-purpose compressors offered by ISO 23001-17. It should be reconsidered, however, whether the default level 11 for Brotli is the best choice given that its computation time is significantly larger, but also producing smaller file sizes. We keep this setting for the moment since encoding an image is a one-time cost. Whether there are any differences in decoding speed is yet to be explored.

2.15. Image streaming over the network

Large tiled HEIF and GIMI files, such as satellite or aerial imagery, can reach sizes of several gigabytes. Downloading such a file in its entirety before viewing the image is impractical, especially when only a small region at a particular resolution level is of interest. Because HEIF stores each tile as an independently decodable unit with its byte offset and size recorded in the file metadata, it is possible to fetch and decode individual tiles on demand using HTTP range requests. This requires that the HEIF library can request arbitrary byte ranges from the data source rather than reading from a local file.

2.15.1. The libheif reader interface

In Testbed-20, the `heif_reader` callback interface in libheif was extended to version 2, adding functions specifically designed for on-demand network streaming. The version 1 interface already provided basic callbacks for sequential reading (`read`, `seek`, `get_position`, `wait_for_file_size`), but these are inefficient over a network because libheif issues many small reads during parsing. Each of these would translate into a separate HTTP request with significant round-trip latency.

The version 2 interface addresses this by adding the following callbacks.

- `request_range(start_pos, end_pos)` — called by libheif before it accesses a file region. The reader implementation can fetch the entire range in a single HTTP request. Subsequent small `read()` calls within this range are then served from a local cache without network access.
- `preload_range_hint(start_pos, end_pos)` — a non-blocking hint that a range may be needed in the future, allowing the reader to prefetch data in a background thread.
- `release_file_range(start_pos, end_pos)` — signals that libheif no longer needs a previously requested range, allowing the reader to free the corresponding cache memory.
- `release_error_msg(msg)` — releases dynamically allocated error messages returned by the reader.

The typical interaction between an application, the `heif_reader` implementation, and libheif is illustrated in Figure 21. When the application opens a remote HEIF file, libheif first requests the file header and the metadata box (`meta`) through `request_range()` calls. From the metadata, libheif learns the byte offsets of all tiles. When the application later requests decoding of a specific tile, libheif issues a `request_range()` for that tile's data range, and the reader implementation translates this into an HTTP range request.

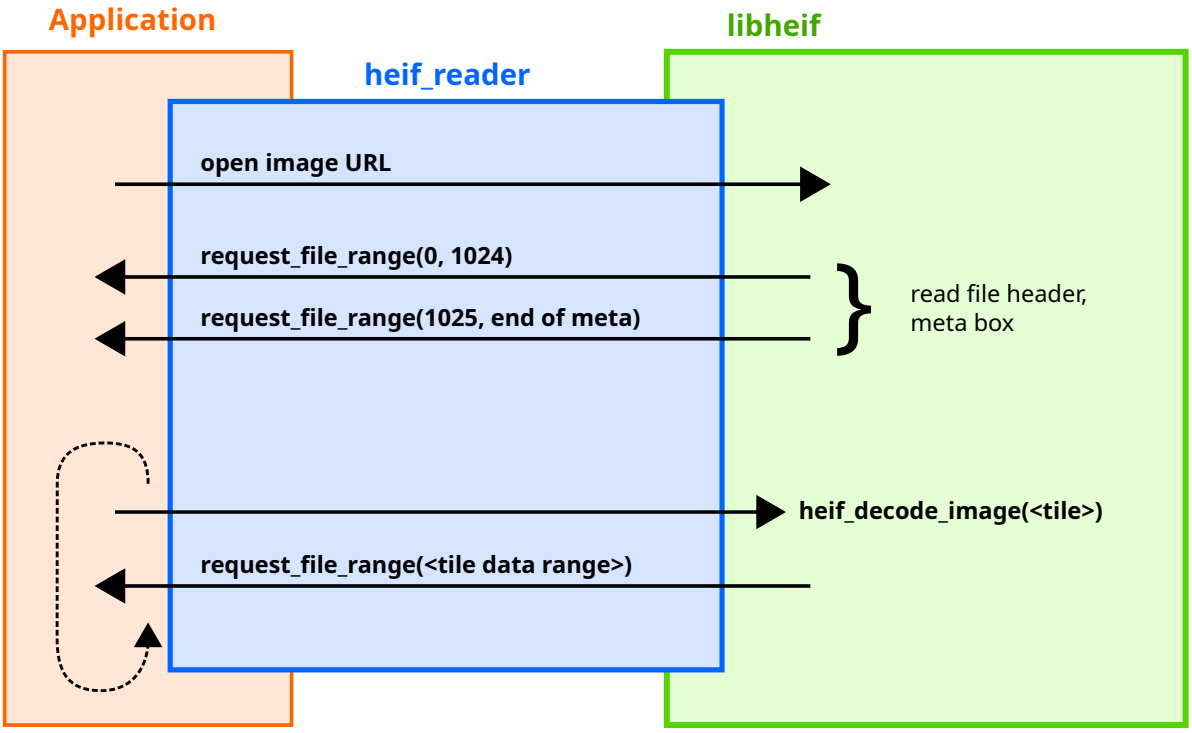


Figure 21 — Program flow through the `heif_reader` interface for network streaming. Libheif requests byte ranges from the reader, which translates them into HTTP range requests. Tile decoding triggers additional range requests as needed.

2.15.2. Connecting libheif to curl

The tiled-image-viewer demonstration application, which previously only supported loading HEIF files from the local filesystem, was extended during the first of the Testbed-21 Code Sprints to also load and display HEIF files served over HTTP. The `heif_reader` version 2 callbacks are implemented in C++ classes that use libcurl to perform the actual HTTP communication.

When the viewer is started with a URL, the reader performs the following initialization.

1. A `curl_easy_init()` call creates a curl session handle.
2. An HTTP HEAD request is issued to determine the file size via the `Content-Length` header.
3. The response is checked to verify that the server supports HTTP range requests.

After initialization, the reader is passed to libheif via `heif_context_read_from_reader()`. From this point on, libheif drives all data access through the callback functions. Each `request_range()`

call translates to an HTTP range request using curl's `CURLOPT_RANGE` option, and the fetched data is stored in a local cache for subsequent `read()` calls.

Two caching strategies were implemented, described below.

2.15.3. Trivial cache

The first and simpler implementation stores each fetched byte range as a variable-length entry in a list. When libheif calls `request_range()`, the reader fetches the requested range from the server and appends it to the cache. Subsequent `read()` calls search the cache linearly for an entry that covers the requested position.

The `release_file_range()` callback removes cache entries that fall within the released range, freeing memory for data that libheif no longer needs.

This implementation is straightforward and was developed first as a proof of concept. However, it can be inefficient in practice because some compressed tiles are very small, resulting in many small HTTP requests with high per-request overhead.

2.15.4. Block cache

The second, more advanced implementation divides the file into fixed-size blocks (64 KB by default, configurable via command-line option). At initialization, an empty block vector is pre-allocated with one entry per block covering the entire file.

When libheif calls `request_range()`, the reader:

1. extends the requested range to align with block boundaries;
2. skips blocks that are already present in the cache;
3. fetches all missing blocks in the contiguous range with a single HTTP range request; and
4. distributes the received data into the corresponding cache blocks.

Because the blocks have a fixed size, looking up whether a particular file position is already cached is an O(1) index computation rather than a linear search. More importantly, fetching data in large, fixed-size blocks means that a single HTTP request often covers several small compressed tiles at once, significantly reducing the total number of network round-trips compared to the trivial cache.

Figure 22 illustrates how this caching scheme works in practice. The top row represents the compressed tile data as it is laid out in the file, with each segment corresponding to one tile (A through W). The bottom row shows the fixed-size cache blocks (1 through 8) into which the file is divided. Note that tile sizes vary considerably: highly detailed tiles may occupy a large portion of a cache block, while smooth or uniform tiles compress to only a few bytes.

In the first example, suppose the application needs to decode tile E. Libheif issues a `request_range()` for tile E's byte range, indicated by the two green arrows. The reader extends this range to the enclosing cache block boundaries and fetches cache blocks 2 and 3 in a single HTTP request. Because tiles C through H are all small and located close together in the file, this single request also brings in the data for tiles C, D, F, G, and H (shown in the green highlighted region). When any of these neighboring tiles are requested later, their data is already present in the cache and no additional network request is needed. Tiles that are spatially close in the image tend to be stored close together in the file, so this read-ahead effect is particularly beneficial when panning across a region of the image.

In the second example, tile P is requested. Its data (indicated by the two red arrows) falls entirely within cache block 6, which is fetched in a single HTTP request. This also loads part of tile Q's data. When tile Q is subsequently requested, the reader finds that part of its data is already cached in block 6. Only the remaining portion needs to be fetched (yellow arrows), which triggers a request for cache block 7. This request, in turn, also brings in the data for tiles R and S (yellow highlighted region), again providing read-ahead for future tile requests in that area.

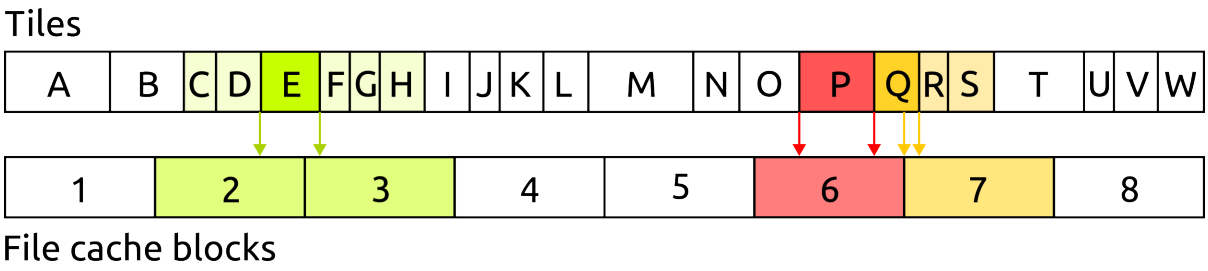


Figure 22 — Example of the block-based caching scheme. The top row shows compressed tile data (A-W) in the file. The bottom row shows fixed-size cache blocks (1-8). Green arrows and highlighting: requesting tile E fetches cache blocks 2 and 3, which also cover tiles C-H. Red arrows and highlighting: requesting tile P fetches cache block 6. Yellow arrows and highlighting: the subsequent request for tile Q only needs cache block 7, since part of Q's data was already loaded with block 6.

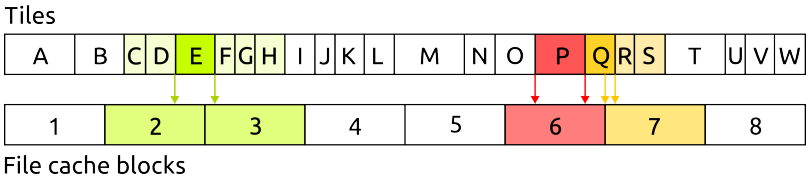


Figure 23

In both implementations, tile loading and network transfer run on a background worker thread so that the user interface remains responsive while data is being fetched.

2.15.5. Download progress visualization

When operating in URL mode, the viewer displays a progress bar at the top of the window indicating which parts of the file have been fetched. The bar shows unfetched byte ranges in red and cached ranges in green, giving the user immediate visual feedback on how much of the file has been downloaded.

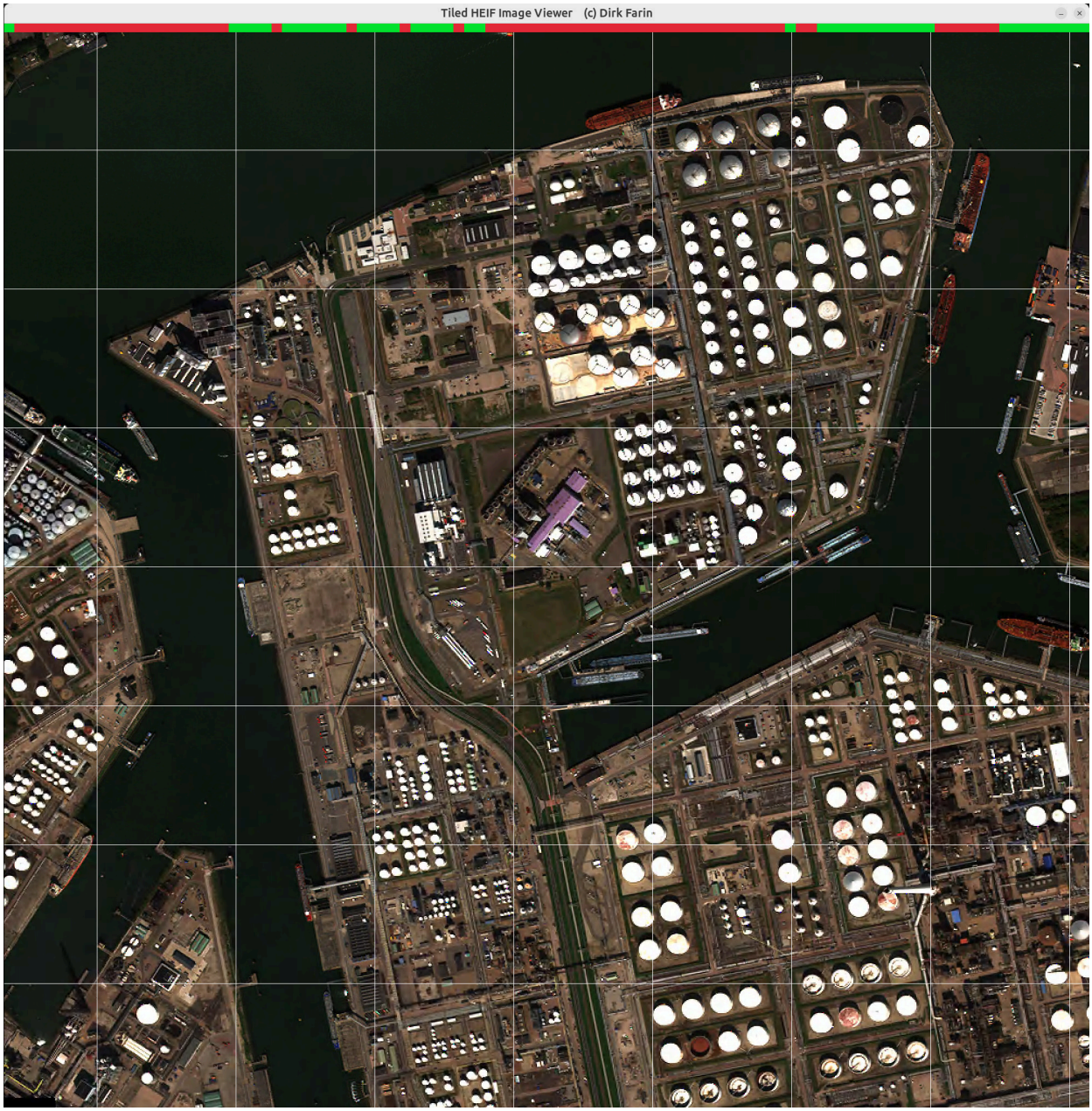


Figure 24 — The tiled-image-viewer displaying a remote HEIF file. The progress bar at the top shows fetched byte ranges in green and unfetched ranges in red.

2.15.6. Command-line options

The following command-line options were added to the tiled-image-viewer for network streaming support:

```
-u, --url              treat input as HTTP/HTTPS URL instead of local file
-t, --trivial-reader   use trivial cache reader instead of block cache (URL mode
only)
-b, --block-size <kB>  block size in kB for block cache reader (default: 64)
```

```
-p, --primary          start with primary image (default: start at overview
image)
```

Example usage:

```
# View a remote HEIF file with default block cache (64 KB blocks)
./tiled_image_viewer -u https://example.com/large_image.heif

# Use larger blocks for high-latency connections
./tiled_image_viewer -u -b 256 https://example.com/large_image.heif

# Use the trivial cache for comparison
./tiled_image_viewer -u -t https://example.com/large_image.heif
```

2.16. Other improvements

The following are other improvements made to libheif during Testbed-21:

1. Updated reading support for `mif3` HEIF files with minimized headers `mini` to the latest HEIF version 4 draft. Support for writing images with `mini` header has been added.

2. Added support to write HEIF items with a size of more than 4 GB (32 bit limit).

3

OUTLOOK

3 OUTLOOK

This section presents an outlook of potential future work and enhancements to how libheif handles the current and future versions of GIMI.

3.1. Tiling

1. During Testbed-20, the “tili” image tiling method was developed and proposed to MPEG for inclusion in the HEIF standard. As of April 2026, the proposal has reached working draft status in HEIF version 4, and the final standard text can be expected soon. The HEIF standard will include our proposal with some changes in the file format syntax that are not yet reflected in our prototype implementation. The final HEIF standard including “tili” came too late for inclusion in the official libheif release during Testbed-21, and a change of the file format while people are actively working with such HEIF files was not practical. An update to the official specification should be carried out at the start of a Testbed so that other participants can work with a stable file format.

2. The “tili” and the “unci” tiling methods do not yet offer a way to assign GIMI content IDs or other metadata to specific tiles. Extensions to specify such metadata should be explored, implemented as a prototype and maybe proposed to MPEG as an amendment for “tili” and “unci”.

3. While implementing and experimenting with the tile-based HEIF reader (see Section …), it was observed that although the method of downloading single tiles with HTTP range requests can be improved with, for example, the implemented block-based caching, the limitation that each tile is requested independently remains. On the other hand, when panning through a large image, it is often the case that a complete row or column of tiles will be needed at the same time. Thus, it would be advantageous to be able to request several tiles at once from the libheif API. These can then be combined internally into a single HTTP request, and decoding can be done in parallel.

4. In addition to parallel tile decoding, it would also significantly increase efficiency to be able to encode several tiles in parallel. Currently, tiles have to be passed into libheif sequentially. The actual image encoder (e.g., H.265) may use parallel threads, but since tiles are usually small, this is inefficient, and parallelization at the tile level would be more efficient. Parallel tile encoding would also be particularly useful for compressed “unci” tiles because their compression algorithms are single-threaded and particularly slow.

5. Another advantage of requesting or encoding several tiles at once may be that the tile images can be processed by the video codecs as a video sequence. This might eliminate the setup times for starting a new codec instance for every tile.

6. In the current implementation, H.265 is decoded in software. With slower CPUs or embedded systems, this may be too slow. It also requires much more energy than a hardware codec implementation, which is especially important for embedded and mobile devices running on battery. A hardware codec could be naturally integrated as a codec plugin in libheif. It will be interesting to see how large the efficiency gains are when using a hardware codec instead of software.

3.2. Format support

1. Support for palettized images in libheif is still an open point. This requires a new image type (`heif_colorspace_palettized`) to be able to process these images correctly in the colorspace-conversion pipeline. Moreover, it requires implementing the corresponding boxes of the ISO 23001-17 standard.

2. Support for multispectral images and other component types in JPEG-2000 was requested (https://github.com/strukturag/libheif/issues/1651). This could build upon the extension of `heif_image` components implemented for ISO 23001-17 in this Testbed.

3. In this Testbed, inefficiencies were described for the `snuc` (Sample non-uniformity correction) box (version 0) in ISO 23001-17. A more efficient box (version 1) could be designed that stores the correction factors in image components that can be transparently compressed with the existing methods (e.g., deflate).

3.3. API interfaces

Libheif is a software library with a C-language interface to make integration in other languages easy. It was recognized that many applications want to use a JavaScript or Python library. For this reason, libheif is often not used directly, but indirectly through “pillow-heif” (Python) or “libheif-js” (JavaScript). However, these wrappers do not give access to the full API functionality of libheif. Both were designed for simple photo applications, not for the special requirements of GIMI. Important functionality like tile-based access, cloud streaming, image sequences, and access to metadata such as content IDs or ISO 23001-17 metadata is not available. The pixel type support is also limited to 8-bit unsigned integers.

In order to facilitate use in applications written in JavaScript or Python, it would be advantageous to provide a fully-featured language interface for these languages.

3.4. Better “unci” compression

The ISO 23001-17 format supports compression through standard general-purpose algorithms (deflate, brotli). These algorithms were originally developed for text compression. Thus, they are strictly byte-oriented and not very efficient for image data. Both deflate and brotli try to find repeating sequences in the byte string, but since the algorithms were developed for text (or program code binaries) compression, they do not exploit typical redundancies in 2D images, like vertical repetitions.

Concerning the compression speed (Issue#100 in the private Testbed-21 GIMI repository), the parallel tile encoding mentioned above might provide some improvement, but for a better compression ratio, dedicated lossless image compression algorithms should be developed. There is recent research on lossless image compression that has not yet been implemented in widely used image formats. This could be developed and proposed as an extension for ISO 23001-17.

3.5. Miscellaneous

1. While generating large files with sizes of several GB during the Testbed, it was observed that writing such large images currently requires large amounts of RAM, since the output image is built in memory before being written to disk. This was originally designed this way because file-position pointers have to be patched after writing the main image data, and the data-field size may need to be switched from 4 bytes to 8 bytes for large files. This is not easy because the total file size is not known beforehand, and switching the field size after writing the data is not possible, since bytes cannot be inserted into an existing file. This requires a change in the writer architecture.
2. Support for the `altr` entity group was proposed to increase compatibility with web browsers that can decode AVIF, but not H.265 or ISO 23001-17 (Issue#50 in the private Testbed-21 GIMI repository).
3. Support for sound tracks was proposed in Issue#59 in the private Testbed-21 GIMI repository. While modern audio codecs like AAC might be out of scope for libheif, an implementation of uncompressed audio tracks seems simple and small enough to be included.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Nicholas J. Car, Open Geospatial Consortium: OGC 22-047r1, *OGC GeoSPARQL – A Geographic Query Language for RDF Data*. Open Geospatial Consortium (2024). http://www.opengis.net/doc/IS/geosparql/1.1.0.

[2] Emmanuel Devys, Ted Habermann, Chuck Heazel, Roger Lott, Even Rouault, Open Geospatial Consortium: OGC 19-008r4, *OGC GeoTIFF Standard*. Open Geospatial Consortium (2019). http://www.opengis.net/doc/IS/GeoTIFF/1.1.0.

[3] Berners-Lee T, Fielding R, Masinter L: IETF RFC 3986, *Uniform Resource Identifier (URI): Generic Syntax*. RFC Publisher (2005). https://www.rfc-editor.org/info/rfc3986.

[4] Freed N, Klensin J, Hansen T, Internet Engineering Task Force: IETF RFC 6838, *Media Type Specifications and Registration Procedures*. RFC Publisher (2013). https://www.rfc-editor.org/info/rfc6838.

[5] Internet Engineering Task Force (committee): IETF RFC 9110, *HTTP Semantics*. RFC Publisher (2022). https://www.rfc-editor.org/info/rfc9110.

[6] Davis K, Peabody B, Leach P, Internet Engineering Task Force: IETF RFC 9562, *Universally Unique IDentifiers (UUIDs)*. RFC Publisher (2024). https://www.rfc-editor.org/info/rfc9562.

[7] International Organization for Standardization (committee): ISO/IEC 14496-12:2022, *Information technology — Coding of audio-visual objects — Part 12: ISO base media file format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2022). https://www.iso.org/standard/83102.html.

[8] International Organization for Standardization (committee): ISO/IEC 15948:2004, *Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2004). https://www.iso.org/standard/29581.html.

[9] International Organization for Standardization (committee): ISO/IEC 23001-17:2024, *Information technology — MPEG systems technologies — Part 17: Carriage of uncompressed video and images in ISO base media file format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2024). https://www.iso.org/standard/82528.html.

[10] International Organization for Standardization (committee): ISO/IEC 23008-12:2022, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 12: Image File Format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2022). https://www.iso.org/standard/83650.html.

[11] [NO INFORMATION AVAILABLE]

[12] [NO INFORMATION AVAILABLE]

[13] [NO INFORMATION AVAILABLE]

[14] [NO INFORMATION AVAILABLE]

[15] [NO INFORMATION AVAILABLE]

[16] [NO INFORMATION AVAILABLE]

[17] National Geospatial-Intelligence Agency: *NGA.STND.0076-01 v1.0.0 — GEOINT Imagery Media for ISR (GIMI)*, 2024.

A

ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

A ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

API	Application Programming Interface
AVIF	AV1 Image File Format
CPU	Central Processing Unit
GEOINT	Geospatial Intelligence
GIMI	GEOINT Imagery Media for ISR
GOP	Group of Pictures
GPU	Graphics Processing Unit
HEIF	High Efficiency Image File Format
HEVC	High Efficiency Video Coding
HTTP	Hypertext Transfer Protocol
ISOBMFF	ISO Base Media File Format
ISR	Intelligence, Surveillance, and Reconnaissance
MPEG	Moving Picture Experts Group
OGC	Open Geospatial Consortium
PNG	Portable Network Graphics
RAM	Random-Access Memory
RDF	Resource Description Framework
RGB	Red, Green, Blue
SAI	Sample Auxiliary Information
TAI	International Atomic Time

TIFF	Tagged Image File Format
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VVC	Versatile Video Coding
WASM	WebAssembly