Index

Scope and purpose of this document

This document is intended to be a stand-alone specification of what is henceforth called "TIFF/A".

TIFF/A is a subset of TIFF. The subset is chosen such that it meets specific goals. The most important ones are...

- It needs to enjoy wide decoder support. Most, of not nearly all mainstream TIFF decoders and viewers need to be able to decode and render any TIFF/A file correctly.
- It needs to circumvent anything that is ambiguous in the existing TIFF specifications, as well as real-life practice. We need a subset that contains only what anyone can unambiguously agree upon, now and in the foreseeable future.
- It needs to be free of patents and license restrictions. Whilst in our opinion compression support is an absolute requirement, the subset needs to be chosen such that any implied technology is free to use, and all required specifications and possibly reference implementations are freely available and widely adopted.
- It needs to cover the needs of the target user base. For instance, many digitization and archival projects struggle with a strict budget, and computer equipment is often times limited in many ways. The TIFF/A subset needs to be chosen such that it does not involve unnecessarily large processing. Often times, the target user base maintains huge archives, and limitations in budget and hardware implies file size is as important as the computational weight of encoding and decoding. Files need to be stored on limited size storage solutions, and need to travel through limited bandwidth networks.

Whilst TIFF/A is a subset of TIFF, this document is meant to evolve into a stand-alone specification, for several reasons.

- This will enable us to identify and eliminate any troublesome licensing issues that surround other specifications.
- Secondly, centralization of all required specification into a single document will ease implementation and conformance checking.
- Last, but certainly not least, this document aims to specify more then just a particular TIFF subset. It also contains guidelines for codec implementation, detecting compliance with the TIFF/A specification, converting to TIFF/A, etc. We believe that in general there is some tendency in the industry to move away from purely theoretical data format specifications, towards more practical specifications of dealing with the data. Aside from obvious benefits, this often results in a more uniform and predictable robustness in applications dealing with that flavor of data, and it leads to a better user experience at the end of the road. Equally importantly, it allows us to formalize and share our knowledge on dealing with the many potential problems in TIFF, from a practical codec and application development point of view. When a purely theoretical data format specification would often restrict us to merely stating something is correct or incorrect, and more algorithm and implementation oriented approach gives us the means to describe ways to actually deal with it.

This TIFF/A document, aside from actually specifying TIFF/A, thus often mentions particular target applications. These applications include encoding and decoding, but also “conversion”. This “converting” application covers a lot of functionality as it aims to convert TIFF into legit TIFF/A.

- For starters, the TIFF decoding part of such a conversion, in as far as actual decoding is required, can be made to be robust and compensate for a lot of common mistakes in TIFF, therefore “repairing” its input as it translates incorrect TIFF as well as correct TIFF into a particular correct TIFF subset, namely TIFF/A.

- Taking merely the decoding part, ignoring the output end of the “conversion”, and adding some level of verbosity, the “converter” functionally turns into a compliance checker.
- Last but not least, conversion is a vital functionality for the TIFF/A target audience all by itself. As TIFF/A gains adoption as a long-term storage and archival file format, the need will arise to convert existing archives of TIFF into TIFF/A. Even more importantly in the long run, image processing chains that output the files that need to be archived, can be updated to directly output guaranteed safe and compliant TIFF/A by simply appending this converter to the existing chain.

This document therefore often zooms in on conversion as a target TIFF/A application that complements decoding and encoding. Only when necessary does it elaborate on particular derived functionality like automated repair of illegitimate files with common mistakes against TIFF, or compliance checking, when it doesn't this derived functionality is implied.

When this document needs to explicitly refer to the Preforma project and project goals, the text layout changes, as it does right here. Our goal is to end up with a TIFF/A specification on the one hand, that can be “extracted” by simply removing or ignoring text blocks layout out this way, whilst on the other hand accurately and completely responding to the Preforma context in particular.

In the context of the Preforma project, there is a distinction between two separate levels of this conversion functionality, and functionality derived thereof. These two distinct levels propagate to two distinct solutions LIBIS and AWare Systems aim to build. More precisely, Preforma demands “some” level of functionality in converting either incorrect TIFF, or correct TIFF that does not fit the TIFF/A subset, to “auto-corrected” legit TIFF/A.

In this document, with this Preforma specific text layout clearly marking the section as no part of the TIFF/A specification, we aim to describe exactly what level of converting functionality we propose to include in our open source solution. TIFF being an extremely wide file format, with plenty of options lots of which hardly any codec on this planet supports up to this day, this open source response to the Preforma goals that we propose, is necessarily limited in some fashion. This will always result in some possible input files being rejected with proper problem document, rather than being converted into legitimate TIFF/A.

Building on LIBIS and AWare Systems expertise in dealing with TIFF, it is our intention to evolve past this point, in a pro version. AWare Systems has a proprietary TIFF codec that exposes a level of functionality far beyond the open source LibTiff, and we believe that building on this codec and the AWare Systems expertise, our pro solution should be able to deal with just about any obscure input TIFF output there, converting very nearly anything to proper TIFF/A.

We view this pro solution as something to add to our open source Preforma proposal, never as something to be subtracted from it. In other words, our primary goal is a functional open source converter able to deal with anything it can be reasonably expected to deal with, covering a large percentage of practical needs that will arise in the TIFF/A target audience. Our pro solution adds to it, ensuring TIFF/A users have a plan B and a viable option left when they find themselves having to deal with just about any really obscure or even plain wrong files, thus turning TIFF/A into an attainable goal in just about any situation.

Most importantly in this stage, in this document we aim to specify the exact level of functionality of our proposed open source response to the Preforma need for “some” level of functionality in converting either incorrect TIFF, or correct TIFF that does not fit the TIFF/A subset, to “auto-corrected” legit TIFF/A.

Building upon a long time existing file format, specifying a subset thereof, a stand-alone

specification is certainly one way to go, for above mentioned reasons. However, this approach can be lacking in that it may be somewhat elaborate, repeating some things already detailed in the said existing file format specification. Furthermore, it can become hard to quickly conclude how TIFF/A relates to TIFF, TIFF/IT, TIFF/EP. With lots of professionals already very familiar with TIFF, TIFF/IT and/or TIFF/EP, a more direct comparison between these existing formats and TIFF/A may be more practical in some situations. Thus, appendices A, B, and C, provide such a direct comparison between TIFF, TIFF/EP, and TIFF/IT on the one hand, respectively, and TIFF/A on the other.

1. General principles

1.1. Burden on the decoder

TIFF contains a lot of options when it comes to file organization. TIFF/A, being a TIFF subset, imposes some limit, but still there's many different encoding possibilities in TIFF/A.

As a general principle, TIFF and TIFF/A encoders are not required to support all possible options. In fact, if a TIFF or TIFF/A encoder can even produce one single flavor of the file format, it is said to be a compliant encoder. The burden of dealing with all these options is shifted to the decoder. A TIFF decoder is compliant if it is able to deal with all possible combinations of all options.

This might not seem intuitive. After all, in general one should probably expect multiple decoding sessions, for every single encoding session, so from this point of view it would make sense to expect the encoder to go the extra mile, and the decoder to have an easy time. The reason for this seemingly counterproductive burden shift to the decoder, is that, at least historically, there's a wide range of devices that are low in memory and processing power, encoding and exporting images, whilst decoding these files for conversion or rendering or whatever is usually delegated to more powerful machines.

1.2. Error attribution and recovery

When building a batch converter from one simple image file format A to another simple image file format B, there's often a problem in error attribution and recovery. Suppose the batch program runs overnight, without human supervision, converting many thousands of files. However, the moment the last person leaves the room and goes home for the night, the application encounters an error. How should it respond?

Suppose it's a system-wide error, “Out of memory”. The batch program may be leaking some memory, and we've run out. It's appropriate, in this application, to actually stop the converting process and visually display an error.

But what if the error is related to the file in format A that just got scanned? The error says “Decoding file 0123.A: Incorrect signature value, this is probably not a valid A”. You certainly don't want the batch program to stop working, for whomever comes in first the next morning to discover pretty much nothing happened, for no good reason. Likely, the desired response may be to make a log entry somewhere, and move on to the next file, instead.

Your batch program is in need of proper error attribution. At the very least, when an error occurs, it requires a way to find out whether or not the error is related to the decoding process of a specific file. If it's not, it's likely OK for the program to assume the error is system-wide.

The situation with TIFF and TIFF/A both, is a lot more complicated, because TIFF nor TIFF/A is a simple file format. Errors can originate on the file level, for instance when a file header is so totally off that it doesn't make any sense to go on. But an error can equally likely originate from decoding the raster contained in the fifth IFD, and it's quite possible all IFDs up to that point can be decoded just fine and images can be extracted. Or, the pointer to the value of the “Software” tag in that fifth IFD could be totally corrupted and point to a location beyond EOF, which would imply that Software tag cannot be read, but the image can still be retrieved.

For every possible application to respond appropriately in every different error situation, a TIFF decoder will be required to provide a means to check the origin of the error. In an object oriented decoder design, it's likely there are different objects representing the file, each IFD, and even each tag. Error attribution can be fully supported by simply having an error flag inside each of these objects, with the decoder setting the appropriate flag when the error occurs, and the calling application checking those flags as needed in order to respond meaningfully.

Proper error response can be quite complicated and something to keep in mind during application or system design. For example, TIFF as well as TIFF/A may contain downsampled versions of an image, enabling an application to avoid the overhead of the full-size image in some circumstances. However, it's quite possible an error occurs when decoding such a downsampled image, and the error may be attributed to the downsampled image IFD. The tricky part is, this does not imply the actual image cannot be read. It may be that the full-size version IFD of this image can be decoded just fine. Meaning, an application that decides to use a downsample, must be designed such that retrieving the next bigger downsampled version or the full-size image remains a valid plan B.

In the remainder of this document, whenever an error situation is discussed, it's clearly stated to which object or level the error should be attributed.

The different types or levels of error defined, are...

- File level error. This can be a consequence of encountering EOF during header decoding, or header values making no sense at all. Pretty much nothing can be accessed.
- IFD level error. The error is local to this IFD. The caller cannot even access a tag count. However, other IFDs may be fine.
- IFD-image level error. The IFD itself is fine. It's just that it doesn't make sense as an image. For example, maybe there is no ImageWidth tag, or maybe the PhotometricInterpretation tag that ought to be an integer specification of colorspace family, is an ASCII string instead. The caller can access the tags and their value. But it cannot get a decoded image stream.

2. TIFF/A file header

2.1. First part

Offset in file	Datatype	Value
0	Unsigned 16bit integer	ASCII "II" or "MM"
2	Unsigned 16bit integer	Version number

The first two bytes in TIFF/A are required to have the same value. This value is either ASCII "I" (decimal 73, hex 49) or ASCII "M" (decimal 77, hex 4D).

These first two bytes are the byte order indicator. "M" is historically short for "Motorola", "I" is short for "Intel". "M" indicates network byte order, higher order bytes come first. "I" indicates reversed byte order, higher order bytes come last.

This byte order indicator applies to any multibyte value in the file, from the subsequent version number, other members of the header specified below, all data members of the IFD structure, all tag values, up to and including even the actual image pixel color data.

The file format itself does not contain any multibyte data with a byte count that is not a power of 2. Therefore, there is at this level no ambiguity. However, image data in TIFF may contain values that have a byte count exceeding 1 and not equal to a power of 2. The original specification does not explicitly mention whether the byte order indicator applies to such odd byte count data. TIFF experts do not agree on this issue, some feel that for example a 3 byte value is always to be specified in network byte order because the notion of doing it any differently is very uncommon. Others feel byte order applies to this data as well, and 3 byte values should have reversed order in "II" TIFF files.

We eliminate this ambiguity in TIFF/A by limiting image data to 8bit or 16bit integer, and 32bit or 64bit floating point.

As is usually the case in TIFF, encoders are never required to be able to generate every possible output. It is very common for encoders to limit support to generation of TIFF files that reflect the byte order that is native to the machine on which they run. Decoders, on the other hand, are required to support both byte order flavors if they intend to claim support for the full TIFF specification. TIFF/A allows both byte order flavors, thus any complete TIFF/A decoder needs to support both. See also “1.1. Burden on the decoder”.

The unsigned 16bit version number immediately follows the byte order indicator. Its value is either 42, for "ClassicTIFF", or 43, for "BigTIFF". The difference between these flavors of the file format is essentially that any offset or length indication in ClassicTIFF is at most 4 bytes long, whilst any offset or length indication in BigTIFF can be up to 8 bytes in length. ClassicTIFF is widely adopted, BigTIFF is relatively new and not included in the TIFF 6.0 specifications. BigTIFF is however included in TIFF/A for obvious reasons. TIFF/A is being targeted at users that commonly deal with very large images, whilst the TIFF/A standard itself is aimed to be long-lived and thus needs to be future-proof.

As usual, TIFF or TIFF/A encoders may chose to not support either ClassicTIFF, or BigTIFF, or they may chose to support both. Decoders, however, need to support both.

The second and final part of the header depends on this TIFF version.

2.2. Second part in ClassTIFF/A

Offset in file	Datatype	Value

4	Unsigned 32bit integer	File offset of first IFD

This second and final part of the ClassicTIFF/A header specifies a 4 byte unsigned integer value that is the file offset of the first IFD. IFD structures are explained below in “4.2. The IFD structure”.

Any value of this field is to be taken as is. It might be clearly incorrect. For example, it might point to EOF or a position beyond EOF, or it might point to a position inside the file header. Still, at this point this is not to be regarded an error, for reasons explained below, in the IFD structure specification section.

The ClassicTIFF/A file header is thus 8 bytes long.

2.3. Second part in BigTIFF/A

Offset in file	Datatype	Value
4	Unsigned 16bit integer	8
6	Unsigned 16bit integer	0
8	Unsigned 64bit integer	File offset of first IFD

This second and final part of the BigTIFF/A header starts of with 2 unsigned 16bit integer members, with fixed values, respectively 8 and 0.

Immediately following these two members, is an 8 byte unsigned integer value that is the file offset of the first IFD. IFD structures are explained below in “4.2. The IFD structure”.

Any value of this field is to be taken as is. It might be clearly incorrect. For example, it might point to EOF or a position beyond EOF, or it might point to a position inside the file header. Still, at this point this is not to be regarded an error, for reasons explained below, in the IFD structure specification section.

The BigTIFF/A file header is thus 16 bytes long.

2.4. Fault tolerance in decoding

In real-life TIFF, it is not uncommon to see lowercase "ii" or lowercase "mm" for the first member of the header structure. A robust, fault-tolerant decoder is required to interpret these incorrect values as "II" or "MM" respectively.

Any other values in that first member will need to be regarded an error on the TIFF file level.

In real-life, it is not uncommon to encounter otherwise perfectly well formed TIFF files that have a version number different from 42 and 43. This is most often generated when software designers don't feel up to the task of inventing a new file format and implementing a proprietary codec, but wish to eliminate data interchange nonetheless. Fortunately, this practice mostly dates back to the pre-BigTIFF era, and it's expected BigTIFF-aware manufacturers abandon this practice. So it's reasonable to assume all such files are ClassicTIFF.

For this reason, a fault tolerant decoder is recommended to interpret anything other then 43, as if it were 42.

A fault-tolerant decoder may want to ignore the first two members of the second part of the header, in BigTIFF/A. To this day they have fixed values that don't convey any real meaning.

If a premature EOF is encountered decoding the header, a decoder can only meaningfully respond with a fatal error on the TIFF file level.

The open source “converter” solution proposed by LIBIS and AWare Systems will include the fault tolerance specified above in the decoding part, meaning, TIFF files with any of these mistakes will get converted to perfectly legitimate TIFF/A.

The pro version will try and go beyond said fault tolerance. If the first member is neither “II”, “MM”, “ii”, nor “mm”, it'll check the version number, and beyond, up to and including the offset to the first IFD and this first IFD itself, to try and derive the proper byte order. This is added for completeness sake only, in nearly 20 years dealing with TIFF we've never seen a file that can be meaningfully decoded with this feature and would not make sense without it. Thus, we don't regard this as a real limitation of the open source solution, but rather an “above and beyond” in the pro version.

2.5. Test files

- **Header\Classic Intel.tif**: Perfectly well formed ClassicTIFF/A, Intel byte order, 16 bits per sample in the actual image data enabling testing byte order correctness up to image data decoding.
- **Header\Classic Motorola.tif**: Perfectly well formed ClassicTIFF,/A Motorola byte order, 16 bits per sample in the actual image data enabling testing byte order correctness up to image data decoding.
- **Header\Big Intel.tif**: Perfectly well formed BigTIFF/A, Intel byte order, 16 bits per sample in the actual image data enabling testing byte order correctness up to image data decoding.
- **Header\Big Motorola.tif**: Perfectly well formed BigTIFF,/A Motorola byte order, 16 bits per sample in the actual image data enabling testing byte order correctness up to image data decoding.
- **Header\Lowercase byteorder E.tif**: ClassicTIFF/A, Intel byte order, 16 bits per sample, but the first member of the header is “ii” where it should actually be “II”.
- **Header\Nonsense byteorder E.tif**: ClassicTIFF/A, Intel byte order, 16 bits per sample, but the first member of the header is “xx” where it should actually be “II”.
- **Header\Incorrect version E.tif**: ClassicTIFF/A, Intel byte order, 16 bits per sample, but the version number in the header is 78 where it should actually be 42.
- **Header\Incorrect bigconst E.tif**: BigTIFF/A, Intel byte order, but the first two member of the second part of the header are totally off.

Please note that the final four test files included for this section are not actually legitimate TIFF nor legitimate TIFF/A. They are included for testing fault tolerance and/or error behavior, only.

3. Data block order, location, and organization

3.1. TIFF and TIFF/A

Any data block in TIFF is either the file header specified in previous section “2. TIFF/A file header”, or...

- a TIFF IFD
- a tag value
- an image strip or tile

A TIFF IFD more or less corresponds to a single raster image. It's a collection of tags, followed by a pointer to the next IFD in the same chain.

A tag is more or less a data member in the IFD. Its tag type, identified by an unsigned 16bit integer, indicates what exact data is given. For example, the tag with ID 256 is used to convey the width of an image. An IFD contains a collection of tags freely chosen by the encoder, with the single restriction that it needs to make sense. For example, the encoder may or may not choose to include a tag that specifies a model and name of the encoding software, but if it is to encode a raster then it absolutely needs to include tags that indicate raster width and height. Inside the IFD structure, for each included tag, there's limited storage space for the actual tag value. If the value fits that space, it's encoded right there. If it does not fit, the value is encoded in a separate data block, and the file offset to that data block is encoded inside the IFD.

The raster pixel color data is divided into rectangular parts, either strips or tiles. Every such strip or tile ends up in a separate data block. These data block file offsets and lengths are specified in dedicated tags.

It follows that a TIFF file is in essence a header followed by a series of data blocks. These data blocks have no fixed order or position. The ways in which data blocks can point to other blocks, are limited, and fully specified. With the exception of these fully specified pointers, each data block needs to be self-contained and relocatable, meaning it cannot point to any outside position, and any pointer to an inside position must be relative to the block offset.

Additionally, any data block is supposed to be linked from only one location. For example, if multiple pages in a TIFF file include the same encoding software name, and the name does not fit the tag value space inside the IFD, they are required to each link to their own instance of that name.

This serves multiple purposes...

- Encoders can flush data in any order they see fit. They never need huge buffers. This is important because encoders can be low-memory devices like faxes and scanners.
- Due to the fact that pointers from one data block to another are limited and fully specified on the lowest levels, algorithms can deal with these blocks while remaining ignorant of their high-level meanings. For example, it's relatively easy to “transplant” one IFD (raster image) from one TIFF file to another, or concatenate TIFF files, or remove (unlink) one raster image in a TIFF file. At the same time, though, TIFF 6.0 remains extendable, new tags with new meanings can be defined, even for private purposes, and still these low-level file operations work just fine.

The specific context of TIFF/A changes this picture slightly.

In TIFF/A, we need to guarantee that all data contained is fully and completely specified by this specification. Thus, no extendability. TIFF allows for new tag types to be defined for private purposes, or even new tag types to be defined and registered in a more global context. TIFF/A does not. This doesn't fundamentally change things, though. TIFF/A remains a subset of TIFF, so even if

some of the above rules might not be strictly necessary in TIFF/A, they remain unchanged.

More importantly, TIFF allows for parts of the addressing space inside a file to be unused. This actually naturally happens as a consequence of page deletion, for example, as this is most commonly implemented as a mere IFD unlinking.

TIFF/A cannot allow such unused space. The TIFF/A target user often aims to build vast archives, and unused space inside files is unwanted. Furthermore, these unused spaces can contribute to a security issue, in the sense that they remain hidden to most, and possibly not all... In real-life TIFF files, it's not entirely uncommon to find unused space actually being used for some private purpose. This is not desirable in an archival file format. Building archives requires the archived files to not contain any wasted unused space, nor can they contain anything with unclear or private meaning.

Therefore, a final step in a TIFF/A compliance check, should map all addressing space usage. This enables detection of...

- partly overlapping data blocks
- multiple pointers to a single data block
- unused spaces

These first two problems imply the file is invalid TIFF, all three problems imply the file is invalid TIFF/A.

None of these problems can normally be readily detected by a mere decoding process. However, all of these problems should be solved as a natural consequence of conversion to TIFF/A, as defined in "Scope and purpose of this document".

The open source "converter" solution proposed by LIBIS and AWare Systems will include resolution of these problems, and ensure TIFF/A compliance, as a natural consequence of the conversion. The open source compliance checker will include addressing space mapping and detection of these problems. There's nothing left for the pro version to add, here, the open source solution covers everything.

3.2. Alignment

In ClassicTIFF, all data blocks are required to be aligned on a 2 byte boundary. Some known implementations go beyond this, and ensure an alignment on a 4 byte boundary. If there is to be an alignment, that is probably sensible, seeing structures like the IFD structure contain 32bit values and all of these are sitting on 4 byte boundary offsets from the beginning of that structure. In BigTIFF, the required alignment is 8 bytes.

This historical requirement as a whole does not make a lot of sense, and is often ignored. Many people feel alignment is an issue reserved for data structures that naturally sit in working memory, a file format that defines structures that naturally sit in files should aim for dense packing instead.

Still, TIFF requires this alignment, and TIFF/A is a subset of TIFF, so it cannot be ignored.

Note that this alignment will often result in tiny unused spaces that are at most 3 bytes long in ClassicTIFF and at most 7 bytes long in BigTIFF. Spaces this small, correctly ensuring proper alignment, should be ignored and not count as actual unused space in the above mentioned TIFF/A compliance check.

The open source "converter" solution proposed by LIBIS and AWare Systems will include

restoration of proper alignment as a natural consequence of the conversion.

3.3. Test files

- **Block\Overlap E.tif**: ClassicTIFF/A, Intel byte order, 8 bits per sample. Contains two IFDs. First one has a Software tag defining the name of the encoding software “Overlap test encoding software”. As this tag value does not fit inside the IFD, it's contained in a separate data block. The second IFD also has a Software tag. This tag points to the same data block offset, plus 8, and has the same length, minus 8. Its value therefore ends up being “test encoding software”.
- **Block\Block reuse E.tif**: ClassicTIFF/A, Intel byte order, 8 bits per sample. Contains two IFDs. Both have a Software tag defining the name of the encoding software, both point to the same data block containing this tag's value “This tag value data block pointed to from two locations”.
- **Block\Unused space E.tif**: ClassicTIFF/A, Intel byte order, 8 bits per sample, single IFD. This files contains an unused space of 128 bytes immediately before the single IFD data block.
- **Block\Bad alignment Classic E.tif**: ClassicTIFF/A, Intel byte order, 8 bits per sample, single IFD. Some of the data blocks in this file don't sit on 4 byte boundary offsets.
- **Block\Bad alignment Big E.tif**: BigTIFF/A, Intel byte order, 8 bits per sample, single IFD. Some of the data blocks in this file don't sit on 8 byte boundary offsets, even though all blocks sit on 4 byte boundary offsets.

Please note that none of these files are actually valid TIFF/A, and, except for “Block\Unused space E.tif”, none of these files are actually valid TIFF. They are included for testing fault tolerance, error behavior and/or conversion and compliance checking, only.

4. The IFD tree

4.1. Historical context

In TIFF, the original meaning of the acronym "IFD" was "Image File Directory". It corresponded more or less to a single raster structure, a single two-dimensional array of pixel color values with appropriate metadata like array dimensions, resolution, specification of used color space, etc.

In the early days, TIFF was mostly used as a container for a single such raster.

This scheme evolved in different ways. Since the top level was actually designed to be a single-linked list of IFD structures, with the first IFD being pointed to in the file header and every IFD including a pointer to the next IFD in the list, TIFF quite naturally became a multiraster format.

There was however no good way to encode and convey all possible meanings of this multiraster list. Were these different planes of the same image? Different layers? Different pages in a scan? A base image and multiple possible masks? A base image and some convenient premade more manageable downsampled versions thereof? Some attempts were made to come up with a standard to specify these intended relations between the different rasters, and some of these standards still live on in real-life practical TIFF usage today.

It became clear, however, that standards could not foresee every possible multiraster relation in every possible application and encoding context.

On top of that, there was a problem with compatibility between software with different levels of support for different versions of these standards. Suppose, for example, an old-time page deletion algorithm gets fed an old-time multiraster TIFF. Both the algorithm and the file predate or ignore these standards to convey multiraster relations. Say the TIFF file contains 4 pages of a single fax. And let's agree the algorithm is used to delete the second page. This is fine, all goes as planned from the user's point of view.

Now let's apply the same algorithm with the same parameters and user intention to a newer TIFF file. This file, too, has 4 rasters. However, whilst the first and second are marked a full-size new image, the third and fourth are marked as downsampled versions of the previous full-size image. The user may view this TIFF with a software that conforms to these new standards, and conclude there are two pages in this file. He may next use the said page deletion algorithm to delete the second one. Suddenly, all goes haywire, we end up with the first full-size new image, followed by IFDs that are said to be downsamples of this full-size image, whilst they are actually downsamples of the deleted raster. When the user next views the result, what he sees may actually depend on his viewing scale and zooming level, as his newfangled viewing software may decide to use these downsamples or use the full-scale image depending on the size of the intended rendering.

In other words, these standards fell short, and introduced the potential for data corruption.

It became clear that a different strategy was needed. Modern practical usage of TIFF regards every IFD in this top-level linked list as a single independent page of a multipage document. No raster relation is to be specified at this level, ensuring wide compatibility. The oldest software that predates all raster relation specification standards, will just assume every raster on this level is a separate image, and it is correct in doing so. The various raster relation specification standards are such that in the absence of any raster relationship tag, again the IFDs in this list are assumed to be separate images, so that's OK too. And thirdly, it fits fine with modern TIFF handling software too, of course.

Instead of including every encoded raster in the same top-level linked list, modern TIFF usage allows for secondary lists to be started from within any IFD, with specific tags to indicate the meaning of these secondary lists, thus turning the single list into a full IFD tree. For example, raster IFDs can include the SubIFDs tag. This tag, if included, points to the first IFD in a list of

downsampled versions of the IFD that includes the tag.

Building on the previous example, a modern TIFF file with four rasters, the third and fourth being a downsampled version of the second, should have two IFDs in the top-level list. The second IFD should contain the SubIFDs tag, and this tag should point to the third raster, which in turn links to the fourth. The old-time page deletion algorithm has no problem correctly deleting the second page by unlinking it. It doesn't understand the SubIFDs tag, but it doesn't need to either because unlinking the second IFD, the third and fourth disappear too. If instead the page deletion algorithm is built to comply with one of the old-time raster relationship specifications, it'll note no raster relationship tags are present in the two IFDs it's seeing and, again, merely unlink the second IFD, as was the user's intention.

The modern way to turn TIFF into an IFD tree rather than a single list, has some important additional advantages. For starters, decoders do not need to scan ahead in order to understand an IFD. It used to be a subsequent IFD could potentially be a mask that needed to be applied to the previous IFD, but now instead that previous IFD contains an in-place indication of everything related to it. Another advantage lies in the fact that the IFD structure inherits the extendability scheme inherent to tags. There is no standard “LayerIFDs” tag, but there could be, and you could be the one to register it or you could define a private “LayerIFDs” tag if your workflow does not require good interchange of this particular data.

Furthermore, it's now possible to encode anything existing of any number of rasters with any possible relationship, and still ensure perfect data interchange of at the very least the composed image. For example, this hypothetical “LayerIFDs” tag should be used in a top-level IFD that contains the composed image resulting from appropriately blending all layers. The tag should point to a list of rasters representing each layer, including the base layer. Software that doesn't understand the “LayerIFDs” tag, will of course not be able to access and make sense of these individual layers, nor will it be able to compose anything the way that is defined in this hypothetical “LayerIFDs” tag specification, but it will decode and render the composed image just fine.

All this is important in the context of TIFF/A because it motivates the choice to limit the TIFF/A subset to the modern practical and unambiguous multipage scheme. The target user of TIFF/A often deals with big images and is likely to decode on machines with limited processing power, so inclusion of the SubIFDs tag standard in the TIFF/A subset is vital. Also, any TIFF decoder, including the one on the input end of the TIFF-to-TIFF/A converter, should understand old-time raster relationship specification sufficiently to at least skip old-time downsamples and such in the top-level IFD list and not mistake them for independent pages.

The next subsection below specifies the IFD structure. Next, we build on all this and specify the TIFF/A IFD tree. The final subsection includes recommendations on how to skip old-time downsamples and other such “slave images”.

4.2. The IFD structure

4.2.1. Format and primary decoder considerations

Whilst the data members of the IFD structure, and their meaning, remain the same in ClassicTIFF and BigTIFF alike, their size and datatype does not.

ClassicTIFF IFD structure is...

Offset in IFD structure	**Datatype**	**Value**
0	Unsigned 16bit integer	Number of tags in IFD
2+(Tag index)*12	Tag structure	Tag
2+(Tag count)*12	Unsigned 32bit integer	File offset of next IFD, or 0 if

		there is no next IFD

ClassicTIFF tag structure used inside this IFD structure is...

Offset in tag structure	**Datatype**	**Value**
0	Unsigned 16bit integer	Tag identification code
2	Unsigned 16bit integer	Datatype of tag data
4	Unsigned 32bit integer	Number of values in tag data
8	Tag datatype or Unsigned 32bit integer	Tag data or File offset of tag data

BigTIFF IFD structure is...

Offset in IFD structure	**Datatype**	**Value**
0	Unsigned 64bit integer	Number of tags in IFD
8+(Tag index)*20	Tag structure	Tag
8+(Tag count)*20	Unsigned 64bit integer	File offset of next IFD, or 0 if there is no next IFD

BigTIFF tag structure used inside this IFD structure is...

Offset in tag structure	**Datatype**	**Value**
0	Unsigned 16bit integer	Tag identification code
2	Unsigned 16bit integer	Datatype of tag data
4	Unsigned 64bit integer	Number of values in tag data
12	Tag datatype or Unsigned 64bit integer	Tag data or File offset of tag data

Reading, decoding and interpreting an IFD, is a process that starts with a given file offset for this IFD. This file offset can be incorrect. It can point to a location inside the file header, or it can point to EOF or beyond. Partial downloads often result in such situations.

It is important to the user to be able to access the parts of the data that are present in a file, even when other parts are corrupted or absent. For example, it is entirely possible that the offset to the fifth IFD may point beyond EOF due to file truncation, but still the first four pages in the file may be partly or completely specified and a user may want to retrieve whatever he can.

This is why an incorrect IFD offset should result in an error attributed to that IFD, not to the structure the offset was actually read from. An incorrect IFD offset, is an IFD-level error, attributed to that IFD.

Any TIFF handling code needs to also check for circular IFD reference at this point. The most clearly understood circular IFD reference situation, is when the file header points to a particular IFD A as the first IFD. This IFD A in turn points to IFD B as next IFD, and IFD B points to IFD A as the next IFD. Any code scanning through IFDs for whatever purpose may end up in an infinite loop in this situation. To protect itself from this, a TIFF decoder needs to maintain a table of IFD offsets for

the file. Before starting the process of reading and decoding an IFD, the offset needs to be matched with offsets in this table, and if a match is found then this IFD offset is considered incorrect similar to how it would be viewed if it pointed to a location inside the file header or beyond EOF. If no match is found, the offset is added to the table, and reading and decoding can proceed.

The first member of the IFD structure is the tag count. The number of tags inside an IFD is usually quite limited, in the order of a couple of dozen at most. Every tag identification code can be present at most once inside a particular IFD, and a tag identification code is an unsigned 16bit integer, so a theoretical limit to the number of tags would be 65536. However, it makes more sense to impose a smaller limit. We suggest any tag count exceeding 4096 to be regarded clearly corrupt, as this is at least an order of magnitude bigger than any sensible IFD we've ever seen. This more restrictive sanity check enables the assumption that reading and decoding of the IFD structure is a fast operation with near immediate return, and that does tend to simplify the decoder design.

A tag count that fails the sanity check and exceeds the recommended maximum of 4096, should result in an IFD-level error.

Next up are the tag structures. A tag is identified by the first member in this structure, the tag ID code. For example, this can be 256, called ImageWidth, which would mean this tag specifies the width of the image that is encoded in this IFD. The lists of tags and their possible data types and values that are allowed in TIFF/A are specified below in "5. The TIFF/A image IFD".

The second member of the tag structure, is a datatype indication. The possible values and their meaning are specified in the table below.

Datatype value	**Name**	**Meaning**
1	Byte	Unsigned 8bit integer data. In real-life TIFF usage, this datatype is often used where the datatype 7 (Undefined) should have been used instead, so, depending on the exact tag type, datatype conversion may not make sense.
2	ASCII	The tag specifies an ASCII string. Byte values should range from 32 up to and including 126, as there exists no standard meaningful ASCII interpretation for any other value. TIFF specifies that a 0 string terminator should be included. In real-life TIFF, decoders should be prepared to see values outside this range, 0 terminators inside the string, and strings without 0 terminator.
3	Short	Unsigned 16bit integer
4	Long	Unsigned 32bit integer
5	Rational	Structure composed of 2 unsigned 32bit integers. The floating point value is obtained

		by dividing the first integer by the second.
6	SByte	Signed 8bit integer
7	Undefined	This value indicates the tag value is bytesized "raw" data, no byteswapping or datatype conversion operation should ever be attempted.
8	SShort	Signed 16bit integer
9	SLong	Signed 32bit integer
10	SRational	Structure composed of a signed 32bit integer followed by an unsigned 32bit integer. The floating point value is obtained by dividing the first integer by the second.
11	Float	32bit. This datatype is not defined in TIFF/A.
12	Double	64bit. This datatype is not defined in TIFF/A.
13	IFD	Unsigned 32bit integer file offset of child IFD.
14	Unicode	This datatype is not defined in TIFF/A.
15	Complex	This datatype is not defined in TIFF/A.
16	Long8	Unsigned 64bit integer. This datatype is defined in BigTIFF(/A) only.
17	SLong8	Signed 64bit integer. This datatype is defined in BigTIFF(/A) only.
18	IFD8	Unsigned 64bit integer file offset of child IFD. This datatype is defined in BigTIFF(/A) only.

Some datatypes are not actually valid for any of the TIFF/A tags. They are thus marked "not defined in TIFF/A".

Following the tag datatype specifier, is the integer specification of the number of values in the tag data. Every tag value is regarded an array of a particular datatype. If it's a single element, it's simply an array of length 1.

The number of values in the tag data, multiplied by the byte length indicated by the tag datatype, is the byte length of the total tag data array. If this does not exceed the byte length of a file offset (i.e. 4 in ClassicTIFF, 8 in BigTIFF), then this data is directly encoded in the final member of the tag

structure. If it's smaller than the byte length of a file offset, it's encoded in the lower portion of this final member. Meaning, the bytes in this member that have the highest file offsets, are not used.

If the byte length of the total tag data array does exceed the byte length of a file offset, the tag data array is stored externally to this IFD structure, in its own dedicated data block in the file. The array's file offset is given in this final member of the tag structure.

The TIFF specification states that tag identification codes in an IFD should be unique, and written in ascending order. It's easily understood that tag identification codes ought to be unique, it simply does not make sense to have multiple ImageWidth specifications, and, tag values being arrays, in cases where multiple values do make sense this should normally be covered with a single tag having multiple values. The ascending order, may not conceptually be required, but it's part of the TIFF specification, so it's part of TIFF/A.

A decoder should be prepared to deal with multiple occurrences of the same tag, and incorrect sort order of tags. In the interest of predictable uniform response to unexpected situations, it is required that a decoder that sees multiple instances of the same tag inside a single IFD, holds on to the first instance and ignores all subsequent instances. It may issue a warning, depending on the desired level of verbosity.

The final member of the IFD structure, following the tag structure array, is the file offset to the next IFD in the single-linked list. 0 is a valid value for this offset, this simply means there is no next IFD. The offset can be different from 0 and clearly incorrect, if it points to a location inside the file header, or if it points to EOF or beyond. At this stage, though, even if the offset is clearly incorrect, the value is to be taken as is. No error or warnings should be issued, as the error will be attributed to this next IFD when it's decoding process starts. This is detailed near the top of this section "4.2.1. Format and primary decoder considerations".

4.2.2. Datatype tolerance

When a decoder expects a particular integer datatype, and another integer datatype is actually specified in the tag structure, it should attempt conversion. For example, if a decoder reads the ImageWidth tag, it expects to find either datatype 3 (Short, Unsigned 16bit integer) or datatype 4 (Long, Unsigned 32bit integer). If the tag specifies datatype 1 (Byte, Unsigned 8bit integer) instead, it should read the Byte data, convert as necessary, and interpret this as a valid ImageWidth specification. In a verbose context, a message can be emitted alerting to the situation, but this should not be regarded an error on any level.

Integer conversions should also be attempted if the specified type is not a subset of the expected type. For example, if the ImageWidth tag specifies datatype 9 (SLong, Signed 32bit integer), and its value is 30.000, this should still be converted and taken as a valid ImageWidth specification. Conversion is not always meaningful or possible if the specified type is not a subset of the expected type. For example, if the ImageWidth tag specifies datatype 9 (SLong, Signed 32bit integer), the value may very well be negative. In this case, reading the tag value should result in an error. The attribution of the error, is up to the calling code. For instance, in the case of ImageWidth, it may be the code that makes sense of the image configuration that attempted to read the value, noticed that it cannot due to conversion issues, and put this down as an IFD-image-level error. Sometimes conversion is simply not defined. The ImageWidth tag could specify datatype 2 (ASCII) or datatype 5 (Rational). No conversion attempt should be made from anything non-integer (including IFD and IFD8), to anything integer. Again, actual error an error attribution should be up to the calling code that's actually trying to access the value because that code has a meaningful context where this particular data is needed, and the exact error attribution will be a consequence of this context. Likely, this will be the code that builds some insights on the image configuration and required decoding processes and such, and the end-result should probably be an IFD-image-level error.

When a decoder expects the datatype 7 (Undefined), it should also accept datatype 1 (Byte),

because many encoders mistakenly use this. Depending on the desired level of verbosity, again, a warning message may be emitted. No other conversion to datatype 7 (Undefined) should ever be attempted, because it is illogical and because data could easily be corrupted due to byte order issues.

When a decoder expects the datatype 5 (Rational), it should attempt to also convert from datatype 10 (SRational), and the other way around.

There is a particularly nasty issue surrounding the datatype 13 (IFD), and, in BigTIFF only, datatype 18 (IFD8). At one point in the history of TIFF, there used to be no such datatype, and datatype 4 (Long) was used instead. There's no real issue when a known tag is accessed. If it's know to have the IFD datatype, and it specifies the Long datatype, this Long data needs to be reinterpreted as IFD data.

However, TIFF (not TIFF/A) is freely extendable, custom tags can be registered and private tags can be used, so applications can encounter tags that are unknown to them. If any such unknown tag has the Long datatype, it's unclear whether it's actually meant to be true integer data, or points to auxiliary IFDs. This problem has the potential to invalidate the functionality detailed in section "3. Data block order, location, and organization". In other words, it's not really a problem for a decoder, an unknown tag should not be of interest as it should not impact retrieval of the data the decoder does know about. It is however a problem for any application that aims to relocate parts of the IFD tree, for example when merging multiple TIFF files into a single multipage file, or the other way around.

BigTIFF is relatively new, but it's expected the Long/IFD issue will probably persist and we'll likely find there will be a similar Long8/IFD8 issue in the near future.

In TIFF, this problem can only be solved in a heuristic manner. In a first pass, the decoder needs to map the used ranges in the file, assuming every unknown tag with Long data is meant to contain actual integers. Next, it needs to examine said Long data. If the values point inside the file header, or if they point to EOF or beyond EOF, or inside a used range as indicated by the previously build usage map, it's very likely this tag's data is intended to be integer. If not, the possibility that this Long is actually meant to be an IFD pointer should be examined. If the data located at that position makes for a sensible IFD structure, it probably is.

In valid TIFF/A, this problem does not exist, because TIFF/A is not extendable. It contains only the tags specified below in section "5. The TIFF/A image IFD", so all tags are known. Furthermore, in valid TIFF/A, the datatypes 13 (IFD) and 18 (IFD8) are the only datatypes allowed for tags that point to auxiliary IFDs.

Whilst the decoder ought to be very tolerant when it comes to tag datatypes, as specified above, a TIFF/A encoder should aim to get it exactly right. It's recommended but not required an encoder should always use the smallest possible allowed datatype. For example, the ImageWidth tag has valid datatypes 3 (Short) and 4 (Long). If the actual image width value can be encoded in the shorter of these, then so it should. Meaning, if the image width value does not exceed 65535, it ought to get datatype 3 (Short).

4.2.3. Premature EOF

Usually, encountering a premature EOF in the midst of a structure, is a reasonably unambiguous and intuitive thing to handle. For example, when the decoder tries and makes sense of the first part of the TIFF file header, but instead of 4 bytes it can read only 3 because this file is exactly 3 bytes long, these three byte values hardly even matter. This can only result in a TIFF-level error, and if you need a meaningful error description it'll probably be something along the lines of "This is not a valid TIFF file".

Such a premature EOF during IFD reading is however a bit more tricky. We start with an IFD offset that turns out to be potentially valid and it's not already present in the table used to avoid circular

reference. So we go ahead and read the tag count.

Suppose it turns out there are 13 tags in this ClassicTIFF IFD. However, we've got 147 bytes left before EOF. That's 12 tag structures, and 3 stray bytes. It's quite possible we can make perfect sense of the image encoded in this IFD building from just those 12 tags. In fact, it's reasonably likely. Remember that tags are supposed to be sorted in ascending order. The most vital tags for image encoding, were defined in the earliest TIFF versions, and have the lowest ID numbers. Meaning, the more you advance in this IFD structure, the more likely it is you're dealing with tags that have little to do with basic image interpretation. Furthermore, many encoders will write out actual strip or tile data blocks and auxiliary tag value data blocks before they append the IFD block, so chances are everything we really need in order to make sense of the image is right there.

It goes without saying a decoder is highly recommended to at least emit a warning message, if it's verbose at all. Compliance checkers and the like, will need to flag this issue. At the same time, though, in the interest of providing best possible access to whatever is accessible, a premature EOF during IFD structure reading should not be treated as a fatal error on any level. Decoders should adjust the internal representation of the tag count to accurately reflect the number of tag structures that are fully read, assume the offset to the next IFD to equal zero, and move on.

It is not uncommon to find TIFF files with a single IFD, IFD data block at the end of the file, with all IFD tag structures intact but premature EOF right where the offset to the next IFD is expected to be. This has been observed with a frequency that indicates there is or at least was at one point some reasonably wide spread encoder producing these files. The above detailed handling of premature EOF should enable decoders to handle these files perfectly fine.

4.2.4. IFD arrays vs IFD lists

With all tag values being arrays, and IFDs having a pointer to the next IFD therefore forming a linked list, there are actually two different ways in which a tag with datatype 13 (IFD) or 18 (IFD8) (or Long/Long8, see “4.2.2. Datatype tolerance”) can point to multiple IFDs.

Both are in use. In fact, it's not uncommon to see both used at the same time, within a single tag. Sometimes both are used at the same time, pointing in two different ways to the same IFDs.

A decoder must be prepared to deal with that situation. In the interest of uniform decoding behavior, and in particular uniform order of child IFDs, the following algorithm or equivalent thereof is strongly recommended.

A decoder starts of with the tag's value, being an array of IFD offsets. This may be an array with a single element, it may be more. The decoder is to iterate through all these values, and add each unique value to an internal value array. Meaning, if even at this stage multiple instances of the same offset value are detected, all subsequent instances are simply ignored.

Scanning through the list of IFDs is next implemented by starting with the first offset value in the internal array, reading the IFD structure at that position, and ending up with a pointer to a next IFD. This new offset value, again, is checked against the internal array of values, and appended to the array if and only if it's unique. If, on the other hand, the offset value is found in the array, it's simply ignored.

The scanning process continues with the next element in the internal array, until that array is exhausted.

In TIFF/A, tags with datatype 13 (IFD) or 18 (IFD8) (not Long/Long8, these are not valid datatypes for IFD offsets in TIFF/A), are required to have an element count equal to 1. In other words, TIFF/A unambiguously opts for the list, not the array. Whilst TIFF (and robust TIFF/A) decoders have to be prepared to deal with both situations and any possible combination thereof, TIFF/A encoders are required to output IFD lists only.

Naturally, decoders may opt to emit a warning message when detecting IFD arrays rather than lists, but they may want to be careful pointing out that this is perfectly valid TIFF and merely not part of the valid TIFF/A subset. Additionally, TIFF/A compliance checkers should point out this situation. The TIFF-to-TIFF/A converter will remedy the problem as a natural consequence of the conversion process.

All of the problems included in this section 4.2. are handled by the open source TIFF-to-TIFF/A converter LIBIS and AWare Systems aim to provide, as a natural consequence of the conversion process. Our open source decode will also explicitly check for these problems, and be able to report them. There's nothing left for the LIBIS and AWare Systems pro version solution to add, here.

4.3. The TIFF/A IFD tree

Every IFD in a TIFF/A file is required to be an image IFD, as specified in section "5. TIFF/A image IFD".

Every IFD in the main single-linked list in a TIFF/A file, the list starting with the file offset to the first IFD read from a member of the TIFF file header, is a separate page in the file. The page order, if conceptually meaningful, is determined by the order in the list, not by a page number tag in the IFD.

An IFD in the main single-linked list can point to child IFDs using the SubIFDs tag. Every child IFD contains the same image data, but image dimensions (meaning, horizontal and vertical pixel count) and/or colorspace specification should be different.

In TIFF/A, it is highly recommended, though not absolutely required, to include a full list of SubIFDs for every image, starting with a same-size 8 bit per channel sRGB SubIFD if and only if the main page image exceeds this level of color precision. Every subsequent SubIFD is highly recommended to have one fourth the horizontal and vertical pixel count of the previous SubIFD, or in the case of the first SubIFD of an 8 bit per channel sRGB main image, one fourth the horizontal and vertical pixel count of the main image, rounded down, resulting in one sixteenth the number of pixels, or slightly less due to this rounding. All subsequent SubIFDs, too, should contain 8 bit per channel sRGB pixel data. It is highly recommended to continue this list of SubIFDs up to and including the point where both image width and image height are at most 400, and no further.

The following are examples of this SubIFD lists complying to this recommendation.

Example 1.

Main list IFD: 4000x6000 pixels, 16 bits per channel sRGB

SubIFD 0: 4000x6000 pixels, 8 bits per channel sRGB

SubIFD 1: 1000x1500 pixels, 8 bits per channel sRGB

SubIFD 2: 250x375 pixels, 8 bits per channel sRGB

Example 2

Main list IFD: 4000x6000 pixels, 8 bits per channel sRGB

SubIFD 0: 1000x1500 pixels, 8 bits per channel sRGB

SubIFD 1: 250x375 pixels, 8 bits per channel sRGB

Example 3:

Main list IFD: 400x290 pixels, 8 bits per channel sRGB

Many naive TIFF decoders, converters and renderers support only the main list IFDs. They see only

the correct full-sized and full-precision page IFDs in any TIFF/A page and ignore the SubIFD list. Meaning, no support for any old-style raster relation specification nor for the new-style SubIFD tag nor for the TIFF/A recommendation is required from TIFF decoders, converters and renderers in order to at the very least arrive at correct results when decoding a TIFF/A file.

A naive TIFF decoder that attempts to support any of the old-style raster relation specifications of TIFF, also arrives at correct conclusions regarding the relationship between main list IFDs, because in the absence of any old-style raster relation specification in these IFDs a simple separate page relationship is to be assumed according to each of these various old-style raster relation specifications.

Any TIFF decoder that does support the SubIFDs tag, should not merely arrive at correct results this way, but also see the downsamples in size and possibly colorspace, and make use of these as it sees fit.

In a rendering context, it's recommended a TIFF/A decoder works from the smallest possible raster, main page IFD or SubIFD, that is at least as wide and as high as the target output, delivering the fastest possible result with the least possible IO and processing needs, with only limited negative impact on rendering quality. In an asynchronous progressive rendering context, it's recommended a TIFF/A decoder and renderer works from the full-sized and full-precision main list page IFD in a subsequent pass, delivering the highest output quality attainable.

This being a recommendation, rather than a requirement, there are several options open to encoders, TIFF/A compliance checkers, and last but not least, the TIFF-to-TIFF/A converter.

Unless targeted at a specific application or applied in a specific context that limits the options, it is recommended a TIFF-to-TIFF/A converter exposes these options to the calling context, or user. At the very least, a converter should probably offer the option to ensure either...

- the smallest possible output. If the calling context or user chooses this option, no SubIFDs are to be included in the target output.
- exact transcoding. In this case, the exact SubIFDs (or downsamples specified in an old-school manner, see “4.4. Old-school IFD relations a decoder should be aware of”), with the exact same dimensions, are taken from TIFF input and included in the TIFF/A output as proper SubIFDs.
- compliance with the TIFF/A recommendations. Input SubIFDs and other downsamples are completely ignored. Output is produced by (re)downsampling the main page image, resulting in the exact SubIFDs list recommended above.

> The open source solution proposed by LIBIS and AWare Systems will largely ignore this issue. The open source TIFF-to-TIFF/A converter will skip over any form of input downsample, and produce no downsample in the TIFF/A output. That is in essence the first option mentioned above.
>
> The pro version will add support for the other two options.

4.4. Old-school IFD relations a decoder should be aware of

A complete history lesson on the origins and practical usage of all old-style IFD relation specifications is clearly beyond the scope of this TIFF/A specification. In valid TIFF/A, every IFD in the top-level list is its own independent page, every downsample is encoded as a SubIFD linked in the list starting with the SubIFDs tag in the main page IFD it belongs to.

Nonetheless, a TIFF decoder, including the one on the input side of the TIFF-to-TIFF/A converter, will need additional measures in order to make sure it never mistakes such an old-school downsample in the top-level list for a new independent page.

This can be achieved by checking for the following conditions.

- The IFD is not the first IFD in the top-level list, includes tag 255 (SubfileType) with datatype 3 (Short, unsigned 16bit integer), count 1 and single value 2 (reduced-resolution image data).
- The IFD is not the first IFD in the top-level list, includes tag 254 (NewSubfileType) with datatype 4 (Long, unsigned 32bit integer), count 1 and single value 1 (reduced-resolution version of another image in this TIFF file).

If either of the these two conditions is fully met, the decoder needs to completely ignore this IFD. Even though this specification does not provide more detail on these tag types or the ways they were used and the specifications that include them, and even though the code footprint of this additional measure is equally limited, this should result in an appropriate response to the vast majority of practical real-life TIFF files that attempt to use any of the old-style schemes in a reasonably unambiguous manner.

The open source solution proposed by LIBIS and AWare Systems will comply with this requirement and correctly ignore old-school downsamples this way. The converter will therefore eliminate these problematic downsamples as a natural consequence of the conversion process.

4.5. Test files

- **IFD struct\Beyond EOF E.tif**: Perfectly well formed first IFD in the top-level list, pointing to a position beyond EOF as the next IFD.
- **IFD struct\Circular E.tif**: Perfectly well formed first two IFD in the top-level list. First one points to the second as the next IFD. Second one, points to the first, as the next IFD.
- **IFD struct\Circular short E.tif**: Perfectly well formed first IFD in the top-level list, pointing to itself as the next IFD.
- **IFD struct\Insane tag count E.tif**: Two IFDs in the top-level list. First one is OK. Second one has a corrupt tag count of 65000.
- **IFD struct\Unsorted tags E.tif**: Single IFD, tags are not properly sorted.
- **IFD struct\Duplicate tag E.tif**: Single IFD containing two instances of the ImageWidth tag.
- **IFD struct\Premature EOF E.tif**: Single IFD, EOF in the midst of the IFD structure, but remaining IFD tag structures and file suffice to make sense of the image.
- **IFD struct\SubIFDs list.tif**: Single top-level IFD, four SubIFDs in a single-linked list.
- **IFD struct\SubIFDs array E.tif**: Single top-level IFD, four SubIFDs, each directly pointed to by an element in the SubIFDs tag value array.
- **IFD struct\SubIFDs list array mixed E.tif**: Single top-level IFD, four SubIFDs, bizarre mixture of list and array with the four SubIFDs linked to form a list, but both the first and second in the list pointed to in the tag value array. A decoder should still see all four SubIFDs, sorted from bigger to smaller, if it complies with section "4.2.4. IFD arrays vs IFD lists".

Please note that with the exception of the SubIFDs test series, none of these are perfectly valid TIFF. With the exception of "SubIFDs list.tif" only, none of these are perfectly valid TIFF/A. They are included for testing error handling and recovery, robust decoding, and conversion.

- **IFD tree\Recommended list.tif**: Single IFD in the main list, encoding a rather big raster.

The IFD includes the SubIFDs tag, pointing to a list of downsamples, with sizes and colorspaces as per TIFF/A recommendation.

- **IFD tree\Old school E.tif**: The first IFD in the main list is marked full size with an old-school tag 254 (NewSubfileType). It's followed by two other IFDs, in the same top-level main list, marked downsample, that are effectively downsamples of this first IFD.

Both these files are valid TIFF. However, the second one is not valid TIFF/A.

5. The TIFF/A image IFD

5.1. General guidelines

It is strongly recommended TIFF and TIFF/A decoders make a firm distinction between tag structures and values on the one hand, and higher level image properties on the other.

For example, on the one hand, there's tag structures inside an IFD structure, and one of these is likely to be the ImageWidth tag, probably with a value encoded in-place, because if one builds from the assumption that it makes any sense at all this value likely at most 4 bytes long. Then again, decoders should not be build on any assumption other than the need for proper handling of the unexpected. An ImageWidth tag in a TIFF file of unknown origin could have a value that is an array of 100 floating points.

On the other hand, there's the image width, a different concept entirely.

This distinction has several advantages.

- A decoder that is build to be robust and do some tolerant auto-correction of tag values, can be made to have a single centralized location for this auto-correcting code. If every location internal to the codec, as well as all possible calling layer code, that needs access to the image width, is required to query a literal tag value instead of having easy access to an integer image property, there is considerable overhead, and many locations that consistently require the same auto-correction.
- A decoder that makes the firm distinction between literal tags and tag values on the one hand, and interpreted image properties on the other, is able to provide access to both. Simply put, it has no problem supporting TIFF tag viewing and TIFF image viewing applications alike.

For less obvious reasons, the same distinction is recommended in a TIFF or TIFF/A encoder. The advantages are essentially the same, even though the applications that actually require these advantages are less numerous.

Many of the exotic, or sometimes even plain incorrect test files accompanying this specification have been built with an encoder making this firm distinction between image properties and tag values. Often times, this encoder was used to build an encoded image and its IFD in a perfectly ordinary manner, and the particular TIFF-level oddities were next achieved by some fiddling with the resulting tags. This illustrates how an encoder that makes this distinction can be a powerful tool providing exactly the output that is required, in any situation, even when it lacks high-level support for some of the more exotic or incorrect requirements.

The following sections aim to specify the relationship between tag values and higher level image properties, and how to convert one into the other. Doing so, it specifies exactly what tags are valid in TIFF/A, with what datatypes and counts, and how they ought to be used and interpreted.

Each of the following sections describes a particular set of tags and image properties. A decoder can thus be build to lazily interpret the property section that is required when it's required for the first time, if ever. However, robust and fault tolerant decoding of one property section, may require some values derived in another section. For example, the algorithm deriving the horizontal tile count as described in “5.4. Raster organization” uses the horizontal pixel count, as derived in “5.2. Image dimensions”. In order to easily and unambiguously reference all these properties used throughout the different sections, this specification names them. Unlike actual tag names, however, these property names are provided for easy reference only, they are not strictly speaking part of the TIFF/A specification.

The tag list defined in the following sections is exhaustive. TIFF/A does *not* support the

extendability inherent to TIFF because this conflicts with the needs of an archival file format.

5.2. Image dimensions

5.2.1. Tags and codec operation

ImageWidth

Tag identification code	256
Type	3 (Short, unsigned 16bit integer)
	4 (Long, unsigned 32bit integer)
Count	1

ImageLength

Tag identification code	257
Type	3 (Short, unsigned 16bit integer)
	4 (Long, unsigned 32bit integer)
Count	1

The ImageWidth and ImageLength tags are mandatory in any TIFF/A IFD. They encode the horizontal and vertical pixel count, not to be confused with any sort of physical rendering dimensions.

If either of these tags is absent, or has a datatype that does not convert or a value that does not make sense (zero, or any negative value when converting from a signed integer datatype), a decoder usually has no other option then responding with an IFD-image level error.

The values read from the ImageWidth and ImageLength tags are henceforth referred to as *ImageWidthValue* and *ImageLengthValue*.

Encoders should emit both tags, as always picking from the allowed datatypes the shortest that can accommodate the correct value.

XResolution

Tag identification code	282
Type	5 (Rational, structure of 2 unsigned 32bit integers)
Count	1

YResolution

Tag identification code	283
Type	5 (Rational, structure of 2 unsigned 32bit integers)
Count	1

ResolutionUnit

Tag identification code	296
Type	3 (Short, unsigned 16bit integer)
Count	1
Default	2 (Pixels per inch)

The XResolution, YResolution, and ResolutionUnit tags are optional. They can be used to encode a resolution, or pixel aspect ratio. This resolution, taken together with the ImageWidth and

ImageLength values, can be an indication of the physical size of the original document, if this document is scanned. It may also be some guideline for intended physical printing size.

If the XResolution tag or the YResolution tag is absent, or either of these has a datatype that does not convert or a value that does not make sense (zero, or any negative value when converting from the signed rational datatype), an aspect ratio of 1.0 is to be assumed and no further information on physical size is present. Aspect ratio of 1.0, also known as “square aspect ratio”, means that the width of a single pixel is equal to its height.

If sensible values for XResolution and YResolution can be read, but the ResolutionUnit tag is absent or it has a nonconvertible datatype or value that does not make sense, the unit of resolution is assumed to be pixels per inch.

Allowed sensible values for the ResolutionUnit tag are...

1 No absolute unit of measurement. XResolution and YResolution, taken together, merely encode a pixel aspect ratio.

2 The unit of the XResolution and YResolution is pixels per inch.

3 The unit of the XResolution and YResolution is pixels per centimeter.

A TIFF/A decoder is required to support this resolution scheme, as is a compliant TIFF decoder. This can be challenging, because TIFF and TIFF/A support non-square pixel aspect ratios, and many imaging libraries do not.

Whilst the above rules make for tolerant and robust decoding, and encoder should attempt to be consistent and predictable.

If no actual resolution information is present in a given image, an encoder should never revert to making up some default and including the tags anyway. Such a fictional default value does not truly contribute information, in fact, it obscures the only real information being that no actual resolution is known. In this situation an encoder should, instead, simply not include any of the XResolution, YResolution, or ResolutionUnit tags.

If a pixel aspect ratio equal to 1.0 can be assumed in an image that is to be encoded, but no additional resolution information is given, that amounts to exactly the same thing. Don't include any of the tags, and a decoder should assume the square pixel aspect ratio.

In any other scenario, an encoder is recommended to add all three tags with correct values.

A full compliance and sanity check can be achieved by reading the information as would a decoder, next from deriving tags and tag values from this information as would an encoder, and comparing the results with the original input. If these don't match, a compliance or sanity checker may want to emit or log some warning message.

The open source solutions proposed by LIBIS and AWare Systems do exactly as specified above. The pro solution adds compensation for some of the bugs in files collected over many years, without contradicting above specified functionality or meaning. The pro converter solution also adds the option to convert IFDs with non-square pixels aspect ratios to a square pixel aspect ratio, appropriately resizing the encoded image in the process.

5.2.2. Test files

- **Dimensions\Nonsquare aspect.tif**: Perfectly well formed single IFD. The challenge is in the fact that pixels are twice as wide as they are high. A proper decoder glued correctly to a

good imaging library, should render this image fine. Others will render it distorted, thin and long.

5.3. Image colorspace

5.3.1. Tags

BitsPerSample

Tag identification code	258
Type	3 (Short, unsigned 16bit integer)
Count	SampleCount (see below)

PhotometricInterpretation

Tag identification code	262
Type	3 (Short, unsigned 16bit integer)
Count	1

SamplesPerPixel

Tag identification code	277
Type	3 (Short, unsigned 16bit integer)
Count	1

ExtraSamples

Tag identification code	338
Type	3 (Short, unsigned 16bit integer)
Count	Detailed below

SampleFormat

Tag identification code	339
Type	3 (Short, unsigned 16bit integer)
Count	SampleCount (see below)

The BitsPerSample tag encodes the number of bits, separately, for each image sample. Its count, therefore, should equal the number of samples per pixel. In TIFF, this tag is not mandatory and has a default value of 1. However, this default cannot be valid in the TIFF/A subset. It follows that this tag is mandatory in TIFF/A. In the TIFF/A subset, valid values for a single sample are 8, 16, 32, and 64, and all samples should have the same value.

The PhotometricInterpretation tag is mandatory in TIFF and TIFF/A. It provides an indication of “colorspace family”. In TIFF/A, the single only valid value is 2, meaning the colorspace is sRGB.

The SamplesPerPixel tag encodes the number of samples in the image, i.e. the number of values that fully define a pixel. In TIFF, the SamplesPerPixel tag is not mandatory and has a default value of 1. However, this default cannot be valid in the TIFF/A subset. It follows that this tag is mandatory in TIFF/A.

The ExtraSamples tag encodes the meaning and function of any image sample beyond the base color samples. For example, in a sRGB IFD that includes two masks, arriving at a total of 5 samples per pixel, this tag should be present with count equal to 2, both values indicating the samples following the R, G, and B samples are to be interpreted as noncomputed alpha (also known as “unassociated alpha”). In TIFF/A, valid values are...

0 unspecified, application dependent data

2 noncomputed alpha

The ExtraSamples tag is not mandatory, in the sense that it should not be present if the IFD includes only the base color samples. It should however be present if the IFD includes any additional samples, and have exactly one value per additional sample.

Typical usage scenario's for unspecified, application dependent samples are elevation data in geospatial imagery, x-ray data in scans of paintings, etc...

The SampleFormat tag encodes the datatype, separately, for each image sample. Its count, therefore, should equal the number of samples per pixel. In TIFF, this tag is not mandatory and has a default value of 1 (unsigned integer data). This default is also valid and default in the TIFF/A subset, and therefore, this tag is not mandatory in TIFF/A either.

In TIFF/A, valid values for SampleFormat are...

1 unsigned integer data

3 IEEE floating point data

TIFF/A furthermore imposes restrictions on the combination of BitsPerSample and SampleFormat. A BitsPerSample value of 8 or 16, can only be combined with a SampleFormat of 1 (unsigned integer data), whilst a BitsPerSample value of 32 or 64 can only be combined with a SampleFormat of 3 (IEEE floating point data).

5.3.2. Fault tolerant decoding

A decoder should first investigate the PhotometricInterpretation tag. If this tag is absent, has a nonconvertible datatype or count different from 1, a TIFF or TIFF/A decoder usually has no other option then giving up on this IFD with an IFD-image level error.

In TIFF/A, the only valid value for this tag is 2, meaning the colorspace is sRGB. TIFF defines a number of additional options. A TIFF decoder may support any number of these options, but a TIFF/A decoder is allowed to throw an IFD-image level error seeing any value other then 2.

A TIFF or TIFF/A decoder should next derive a value for what is henceforth referred to as *BaseSampleCount*. This is the number of samples per pixel that are implied by the image colorspace. This value is a direct consequence of the colorspace family, a TIFF decoder should have some functional equivalent of an array of *BaseSampleCount* constants, one for each supported colorspace family, from which it picks the value that is applicable in this IFD. In TIFF/A, this colorspace family is always sRGB, so *BaseSampleCount* always equals 3.

The TIFF/A decoder should next attempt to read the SamplesPerPixel tag. If this tag is absent, has a nonconvertible datatype or count different from 1, the TIFF/A decoder is allowed to back out with an IFD-image level error. The value read from this tag is henceforth referred to as *SampleCount*.

If *BaseSampleCount* is greater than *SampleCount*, this is an IFD-image error situation.

Next up, the TIFF/A decoder takes a peek at the ExtraSamples tag and derives a property henceforth called *ExtraSampleCount*. At this point, the decoder is not yet interested in this tag's value, but merely its presence and count. If the tag is not present, or has a nonconvertible datatype, *ExtraSampleCount* should be made to equal 0. If the other hand the tag is present and has a datatype that can make sense, *ExtraSampleCount* should be made to equal this tag's count.

At this point, *BaseSampleCount*+*ExtraSampleCount* should equal *SampleCount*. If it does not, the IFD contains inconsistent and contradictory information. This is an IFD-image error situation.

If the decoder survives up to this point without throwing an IFD-image error, it has a basic understanding of the color image data encoded in this IFD, and it's ready to try and derive a more

thorough insight into each sample's datatype and bitdepth, and each extra sample's meaning.

The decoder needs to next investigate the BitsPerSample tag. If this tag is not present, or if it has a nonconvertible datatype, this implies an IFD-image level error.

TIFF allows different image samples to have different bitcounts. For this reason, the BitsPerSample tag count should always equal *SampleCount* in a valid TIFF IFD. However, in real life, codecs supporting samples with different bitcounts are few and far between. Some buggy encoders don't actually bother to provide a BitsPerSample value for every sample and encode a single value instead. A fault-tolerant TIFF or TIFF/A decoder that sees a BitsPerSample count equal to 1, should therefore proceed as if the BitsPerSample tag had a count equal to *SampleCount*, with all values equal to the single value that is actually given.

If the BitsPerSample tag has a count different from the correct count *SampleCount*, and different from the tolerated count 1, this is an IFD-image error.

In the TIFF/A subset, valid values for a single sample are 8, 16, 32, and 64, and all samples should have the same value. A TIFF/A decoder is allowed to revert to an IFD-image error if any other BitsPerSample tag value is detected. The decoder thus arrives at an image property value that is henceforth referred to as *BitsPerSampleValue*.

Reading and interpreting the SampleFormat tag next, is very similar, except that there is a sensible default for this tag.

Meaning, if the SampleFormat tag is not present or has a nonconvertible datatype, a decoder should proceed as if it were present, had the correct datatype, a count equal to *SampleCount*, and all values equal to 1 (unsigned integer data).

If the SampleFormat tag is present, has a correct or convertible datatype, and a count equal to 1, a decoder should proceed as if it had a count equal to *SampleCount* with all values equal to the single value that is actually given.

If the SampleFormat tag is present, has a correct or convertible datatype, and a count different from the correct count *SampleCount*, and different from the tolerated count 1, this is an IFD-image error.

In the TIFF/A subset, if *BitsPerSampleValue* is either 8 or 16, all SampleFormat values should be 1 (unsigned integer data). In combination with any other *BitsPerSampleValue*, 32 or 64, all SampleFormat values should be 3 (IEEE floating point data). If this is not the case, the IFD is not valid TIFF/A, and the decoder is allowed to back out with an IFD-image level error.

If *ExtraSampleCount* is 0, then, at this point, the decoder has a complete understanding of the image colorspace. If it's not, the decoder needs to take a second peek at the ExtraSamples tag.

We've already checked the presence of the ExtraSamples tag, and its count. If *ExtraSampleCount* is not 0, and we arrive at this point without error, the ExtraSamples tag is present and its count is guaranteed to be correct and equal to *ExtraSampleCount*. The ExtraSamples tag contains an indication of meaning or intent, per image channel beyond those that are implied by the colorspace. In TIFF/A, valid values are 0, for unspecified, application dependent data, and 2 for noncomputed alpha. A TIFF/A decoder is allowed to respond to any other value with an IFD-image level error.

A full compliance and sanity check can yet again be achieved by reading the information as would a decoder, next from deriving tags and tag values from this information as would an encoder, and comparing the results with the original input. If these don't match, a compliance or sanity checker may want to emit or log some warning message.

5.3.3. Sample data ranges

With unsigned integer sample formats, sRGB is encoded as expected, meaning RGB(0,0,0) is black and RGB(x,x,x) with x representing the maximum integer value that fits the bitdepth (either 255 or 65535), is white.

With floating point sample formats, the range is 0.0 to 1.0 inclusive, meaning RGB(0.0,0.0,0.0) is black and RGB(1.0,1.0,1.0) is white. It is permissible to go beyond this range for data that is outside of the sRGB gamut. A decoder dealing with floating point data, is expected to crop the range, if it needs data to remain inside that sRGB gamut.

Uncomputed alpha is required to be encoded with the same range. Meaning, integer alpha 0 and floating point alpha 0.0 is fully transparent. Integer alpha x with x representing the maximum integer value that fits the bitdepth (either 255 or 65535), and floating point alpha 1.0 is fully opaque.

In unspecified application-specific extra samples, the range may depend on the particular application. Still, it is highly recommended to make sure the range makes some sense when naive viewers isolate the application-specific channel and render as if it were a single linear brightness channel with above specified ranges.

For example, when encoding a floating point elevation data channel in a geospatial application, one may be mistakenly tempted to equate 0.0 to sea level and 1.0 to one unit of elevation like for example 1 meter. This would however result in 99.9% of usual elevation channel data being out of the range expected by a naive viewer that is unaware of your application and convention, and you'd be unable to use TIFF analysis tools to quickly view and visually inspect the channel, or extract the channel into a linear brightness image and do some quick imaging operations with general imaging tools that are unaware of your particular application, etc. This being a mere application-specific convention anyway, it is usually much more sensible to try and map the foreseeable “normally expected” minimum and maximum elevation to the above mentioned minimum and maximum for color and alpha channels. In floating point application-specific data, one is still able to go outside this range for the less expected data, similar to how sRGB is permitted to go outside this range to encode out of gamut data.

5.3.4. Precomputed alpha

In TIFF, other possibilities for the ExtraSamples tag value include 1 for precomputed alpha, also known as associated alpha or premultiplied alpha. This tag value effectively means that the values given for each of the base color samples in every pixel is already “multiplied” with the alpha value for that pixel, or, more precisely, “rescaled” to a new range limited by the alpha value.

This is problematic because it leads to loss of color information. For example, suppose all channels contain 8bit integer data. Thus, the effective range is normally 0 to 255. With noncomputed alpha, a pixel could end up with an R value of 200, and an alpha value of 2. This quite red pixel is almost completely transparent. Precomputing this alpha, the effective range of R, given this alpha value of 2, is limited to 0 to 2. The precomputed R value is round(200*2/255) = 2. Actually, given this small alpha value 2, any noncomputed R value in the range 192 to 255 corresponds to the same precomputed R value 2. Meaning, quite a lot of color precision is lost. The smaller the alpha value, the bigger the loss, culminating in total color loss when alpha equals 0. For this reason, it's not recommended to convert precomputed alpha pixels back to noncomputed alpha. It would obscure the loss of color precision and likely lead to artifacts further down the road.

On top of that, these days imaging engines don't usually do any alpha calculations in colorspaces like sRGB, for the simple reasons that these spaces don't linearly match anything that is intrinsic to light, color, or human perception. Precomputed alpha in a colorspace like sRGB is more or less considered “not a thing” these days, and for good reason.

This whole issue is moat in TIFF/A, because precomputed alpha is not a part of the valid TIFF/A subset anyway. However, it is relevant when decoding the input in the context of conversion from TIFF to TIFF/A. Even in this context though, it is strongly recommended to not support precomputed alpha, for the above reasons.

5.3.5. Extra samples and default rendering

Multiple alpha samples are possible. For example, these alpha samples can implement masks, each selecting some meaningful parts of the image. Multiple unspecified samples are also possible. Alpha samples and unspecified samples may be combined in any order.

Without application-specific context, a naive TIFF/A renderer is recommended to take the first alpha sample as a true transparency mask, if indeed there is at least one alpha sample. A naive renderer would be correct to completely ignore any samples with unspecified data, as well as any alpha samples beyond the first one.

5.3.6. Test files

- **Colorspace\I8.tif**: Perfectly well formed single IFD, 8bit per sample integer sRGB, no extra samples.
- **Colorspace\I16.tif**: Perfectly well formed single IFD, 16bit per sample integer sRGB, no extra samples.
- **Colorspace\F32.tif**: Perfectly well formed single IFD, 32bit per sample IEEE floating point sRGB, no extra samples.
- **Colorspace\F64.tif**: Perfectly well formed single IFD, 64bit per sample IEEE floating point sRGB, no extra samples.
- **Colorspace\I16 motorola alpha.tif**: Perfectly well formed single IFD in a Motorola byte order TIFF, 16bit per sample integer, single extra sample adding noncomputed alpha.
- **Colorspace\F32 app alpha.tif**: Perfectly well formed single IFD, 32bit per sample IEEE floating point, two extra samples. First extra sample is unspecified and application defined. Second extra sample is noncomputed alpha.
- **Colorspace\I8 app alpha app app.tif**: Perfectly well formed single IFD, 8bit per sample integer, 4 extra samples, one of which is noncomputed alpha.
- **Colorspace\I8 bad BitsPerSample count E.tif**: Single IFD, 8bit per sample integer sRGB, no extra samples. The BitsPerSample tag incorrectly has a count equal to 1 and requires above specified fault-tolerance in decoding.

Please note that the last file is not actually valid TIFF/A or even valid TIFF. It's included for testing decoder fault tolerance.

Application defined samples in these testfiles, are renderings of the text “Undefined” followed by an index, and the range is chosen such that it makes sense when interpreted as some form of brightness. If a default rendering of a testfile with an application defined channel somehow displays this text, this proves the decoder makes incorrect conclusions about the nature of this channel.

The open source solutions proposed by LIBIS and AWare Systems do exactly as specified above. Its decoding part, even in the context of TIFF to TIFF/A conversion, is limited to sRGB support. It does not support any other colorspace in TIFF that is not valid in TIFF/A anyway.

The pro solution has a very wide range of TIFF colorspace support in the input decoding part of the TIFF to TIFF/A converter. The pro solution also adds compensation for some of the bugs in files collected over many years, without contradicting above specified functionality or meaning. Additionally, the pro converter solution also adds the option to eliminate any or all extra samples during the conversion process.

5.4. Raster organization

5.4.1. Tags

Orientation

Tag identification code	274
Type	3 (Short, unsigned 16bit integer)
Count	1

PlanarConfiguration

Tag identification code	284
Type	3 (Short, unsigned 16bit integer)
Count	1

RowsPerStrip

Tag identification code	278
Type	3 (Short, unsigned 16bit integer) 4 (Long, unsigned 32bit integer)
Count	1

StripByteCounts

Tag identification code	279
Type in ClassicTIFF/A	3 (Short, unsigned 16bit integer) 4 (Long, unsigned 32bit integer)
Additional Type in BigTIFF/A	16 (Long8, unsigned 64bit integer)
Count	StripCount

StripOffsets

Tag identification code	273
Type in ClassicTIFF/A	3 (Short, unsigned 16bit integer) 4 (Long, unsigned 32bit integer)
Additional Type in BigTIFF/A	16 (Long8, unsigned 64bit integer)
Count	StripCount

TileWidth

Tag identification code	322
Type	3 (Short, unsigned 16bit integer) 4 (Long, unsigned 32bit integer)
Count	1

TileLength

Tag identification code	323
Type	3 (Short, unsigned 16bit integer) 4 (Long, unsigned 32bit integer)

Count	1

TileByteCounts

Tag identification code	325
Type in ClassicTIFF/A	3 (Short, unsigned 16bit integer)
	4 (Long, unsigned 32bit integer)
Additional Type in BigTIFF/A	16 (Long8, unsigned 64bit integer)
Count	TileCount

TileOffsets

Tag identification code	324
Type in ClassicTIFF/A	3 (Short, unsigned 16bit integer)
	4 (Long, unsigned 32bit integer)
Additional Type in BigTIFF/A	16 (Long8, unsigned 64bit integer)
Count	TileCount

The Orientation tag is not mandatory. If it's not present, a default value of 1 is assumed. This means the encoded top row is to be rendered as the visual top row in a default rendering, and the encoded left column is to be rendered as the visual left column. Valid values for the Orientation tag are...

Value	Encoded top row	Encoded left column
1	Visual top row	Visual left column
2	Visual top row	Visual right column
3	Visual bottom row	Visual right column
4	Visual bottom row	Visual left column
5	Visual left column	Visual top row
6	Visual right column	Visual top row
7	Visual right column	Visual bottom row
8	Visual left column	Visual bottom row

The PlanarConfiguration tag is not mandatory, if it is not present the default value 1 (Chunky) is assumed. If it is present, the following are valid values.

1 Chunky

2 Planar

A PlanarConfiguration value of 1 (Chunky, the default value), indicates that sample data values for each pixel are stored contiguously. In other words, in the context of an sRGB TIFF/A IFD, the R value for pixel 0 is followed by the G and B values for the same pixel, the R value for pixel 1, etc. A PlanarConfiguration value of 2 on the other hand, indicates an organization in separate sample planes. Meaning, the R value for pixel 0 is followed by the R value for pixel 1. Following the plane with the R values for all pixels, is a plane with G values for all pixels, etc.

A single plane is divided up, either in strips or in tiles.

Historically speaking, the strip organization is the older. In this scheme, a single plane raster is divided into full-width, limited height strips, sorted from top to bottom. The height of a single strip is encoded in the RowsPerStrip tag. The bottom strip contains the remainder of the plane raster, its

height is at most RowsPerStrip.

The tile organization is newer. In this scheme, a single plane raster is divided into limited width and limited height rectangles, sorted from left to right and top to bottom. The width and height of a single tile is encoded in the TileWidth and TileLength tags. The rightmost and bottommost tiles, do contain TileLength lines of TileWidth pixels, same as all other tiles. To this end the actual plane data is padded as necessary by the encoder. And conversely, a decoder should crop the data in the rightmost and bottommost tiles so as to fit the plane raster dimensions.

If an IFD is organized in strips, the StripByteCounts and StripOffsets tags are mandatory. They encode, respectively, the compressed size and offsets of each individual strip data block. The RowsPerStrip tag is not mandatory. If it's not present, or its value equals or exceeds the number of lines in the plane, the plane is encoded as a single strip. No tile tags (TileWidth, TileLength, TileByteCounts, TileOffsets) should be present in the IFD.

If an IFD is organized in tiles, the TileByteCounts and TileOffsets tags are mandatory. Analogous to StripByteCounts and StripOffsets, these tags encode, respectively, the compressed size and offsets of each individual tile data block. The TileWidth and TileLength tags are mandatory. Their values can actually exceed the width and length of a plane, even though that is probably not very useful. TIFF requires TileWidth to be a multiple of 16, and while an encoder certainly should live up to that requirement, a decoder should be prepared to deal with any TileWidth value. No strip tags (RowsPerStrip, StripByteCounts, StripOffsets) should be present in the IFD.

5.4.2. Fault tolerant decoding

A decoder is to interpret image dimensions and colorspace before it can attempt to derive raster organization. See sections “5.2. Image dimensions” and “5.3. Image colorspace”. The process described below builds on image properties determined in those sections.

A decoder reads the Orientation tag first. If the tag is not present, its datatype is nonconvertible, or its value is invalid, *OrientationValue* is set to equal the default 1. Otherwise, *OrientationValue* is to equal this tag's value.

The width and height of the raster as it is stored can now be derived. For reasons that will become apparent below, we will henceforth call this *PlaneWidth* and *PlaneLength*. If *OrientationValue* is smaller than 5, *PlaneWidth* is to equal *ImageWidthValue*, and *PlaneLength* is to equal *ImageLengthValue*. If *OrientationValue* is 5 or is greater than 5, *PlaneWidth* is to equal *ImageLengthValue*, and *PlaneLength* is to equal *ImageWidthValue*.

Next up, the decoder investigates the PlanarConfiguration tag. If either the tag is not present, or it is but its datatype is not convertible, or its value is not any of the above specified valid values, the decoder should proceed as if PlanarConfiguration has the default value 1 (Chunky).

At this point, the decoder has a value for the PlanarConfiguration tag, which we will henceforth call *PlanarConfigurationValue*. It's now ready to derive the number of planes, in *PlaneCount*, and the number of bits per pixel in a single plane in *BitsPerPlanePixel*. If *PlanarConfigurationValue* is 1, *PlaneCount* should be set to equal 1, and *BitsPerPlanePixel* needs to equal *SampleCount***BitsPerSampleValue*. Otherwise, if *PlanarConfigurationValue* is 2, *PlaneCount* should be set to equal *SampleCount*, and *BitsPerPlanePixel* should be made to equal *BitsPerSampleValue*.

Next up, the decoder should try and determine whether the IFD is striped or tiled.

If a decoder sees either the StripByteCounts or StripOffsets tag, it is to assumed the IFD is strip organized. It should next check that both StripByteCounts and StripOffsets are present, with correct or convertible datatype, and that none of TileWidth, TileLength, TileByteCounts, or TileOffsets is present. If that check fails, this is an IFD-image error.

In decoder design, it's often convenient to view strips as image-wide tiles. This leads to mostly a single code path handling both cases. For convenience, this specification will refer to this concept of “strip or tile” using the word “strile”. The bottommost strip is not vertically padded, whilst tiles in the bottommost tile row are. The single code path dealing with striles therefore will need to support both options, and to this end we'll introduce a boolean image property *StrileBottomPadding*.

If the IFD is strip organized, an image property henceforth called *StrileWidth* is set to equal *PlaneWidth*. If a RowsPerStrip tag is present and has a correct or convertible datatype, *StrileLength* is set to the smaller of the RowsPerStrip value and *PlaneLength,* otherwise *StrileLength* should be made to equal *PlaneLength*. *StrileBottomPadding* is a boolean image property set to false. Finally, *StrileByteCountsTag* and *StrileOffsetsTag* are image properties that are some functional equivalent of tag pointers, and should be made to refer to the StripByteCounts and StripOffsets tags respectively.

If a decoder does not see either of the StripByteCounts and StripOffsets tags, it is assumed the IFD is tile organized. It should next check that the TileWidth, TileLength, TileByteCounts and TileOffsets tags are present and have a correct or convertible datatype, and that the RowsPerStrip is not present. If that check fails, this is an IFD-image error.

StrileWidth is set to the value of TileWidth, *StrileLength* is set to the value of TileLength. *StrileBottomPadding* is set to true. Tag pointing properties *StrileByteCountsTag* and *StrileOffsetsTag* are set to point to the TileByteCounts and TileOffsets tags, respectively.

The decoder is now able to arrive at some conclusions as to the number of striles per plane. *StrileCountHorizontal* is set to *PlaneWidth* divided by *StrileWidth*, rounded up to the nearest integer. *StrileCountVertical* is set to *PlaneLength* divided by *StrileLength*, rounded up to the nearest integer. *StrileCount* is next computed by multiplying *PlaneCount*, *StrileCountHorizontal* and *StrileCountVertical*.

If the count of either the *StrileByteCountsTag* tag or the *StrileOffsetsTag* tag does not equal *StrileCount*, this is an IFD-image error.

At this point, the decoder is able to read strile sizes and offsets from the *StrileByteCountsTag* and *StrileOffsetsTag* tags. These values could be clearly wrong. For example, a size value might be 0, or an offset value might point to a position inside the file header or beyond EOF. Still, these values are to be taken as is, at this point, no validation should occur.

5.4.3. Encoder recommendations

Many mainstream TIFF applications do not properly support the Orientation tag. Therefore, encoders are recommended to not use it routinely. If the internal model of a raster image as is probably maintained by some imaging library that interfaces with the TIFF or TIFF/A encoder, is organized in a manner that does not correspond to the default orientation in TIFF, it's recommended the encoder rotates the internal data when feeding it to the encoder, encoding default oriented TIFF IFDs nonetheless, rather than use the Orientation tag for each and every image written.

On low-memory devices, or when potentially huge rasters are involved, having a complete buffer and rotating the raster internally may be problematic. In such cases, using the Orientation tag is recommended as the lesser of two evils.

When rotating a pre-existing TIFF IFD, it is also recommended to use this tag, avoiding the need for reading, decoding, buffering, rotating, encoding, and finally rewriting the IFD raster data.

An encoder is recommended to opt for planar storage if and only if there are several extra samples, be it alpha samples or application-specific data samples, and its obvious decoder contexts will often need to ignore all or most of these extra samples. A decoder can easily eliminate considerable IO and processing requirements in this situation, as the planar storage ensures the superfluous samples do not need to be read and decoded. In any other situation, chunky storage is recommended instead.

But, again, on encoding low-memory devices, or when streaming potentially huge rasters, the inability to buffer means the availability of data rather than decoder considerations may dictate raster organization choices.

There are multiple advantages in dividing planes in separate strips or tiles, as opposed to encoding a complete plane raster as a single chunk of data. The most important of which used to be that even encoders and decoders that buffer the complete chunks in between the different steps in encoding or decoding, can more easily deal with the image and are less likely to run into memory or even addressing space limitations. Decoders being able more easily access part of a raster when they're not interested in the complete image can be equally important in some applications.

As hardware capabilities have grown over the last couple of decades, so have the minimum dimensions where these advantages start to outweigh the organizational overhead involved. But, there is a new and important consideration to compensate for this. In these modern times, making proper use of multiple CPUs per machine, multiple cores per CPU, and multiple threads per core, is a new and vital holy grail, the importance of which is only expected to grow exponentially in the decades to come. Individual strips and tiles having completely separate encoding and decoding paths, it's relatively easy for a TIFF codec to properly support balancing of encoding and decoding load over multiple threads.

The TIFF 6.0 specification, designed in 1992 when available working memory was measured in kilobytes rather than gigabytes, recommended to choose individual strip and tile dimensions such that uncompressed data size more or less boiled down to 8 kilobyte. Regrettably, some manufacturers to this day try and interpret this recommendation literally, sometimes resulting in every single line of an image getting encoded as a single strip, whilst others are more aware of the overhead involved and resort to the opposite extreme, sometimes encoding huge rasters as a single strip. It is thus deemed important to redefine the recommendation. The present- and future-proof most important reasons to divide plane rasters in strips or tiles is good decoding access to partial rasters, memory management, streaming without too much required buffering, and above all, multithreaded encoding and decoding, so that's what we want to keep in mind. We've also observed that while it is difficult to quantify the exact grow rate of involved hardware, in the past couple of decades the totally unrelated average display size seemed to grow at more or less the same pace. Even though it seems bizarre and farfetched to draw a relation between display size and recommended TIFF tile size, history does prove display size to be quite a good indicator of hardware capability, and we expect it to be quite useful in a future-proof recommendation.

Thus, we recommend the new size threshold replacing the 8 kilobyte threshold in the TIFF 6.0 specification to be loosely equal to about half the number of pixels in the average, lower to middle-end consumer display. At the time of writing, the pixel dimensions of such a display is probably about 1920x1080, so the current threshold recommendation is somewhere in the range of 1.0 megabyte.

Divide this number by the number of bytes per pixel in a single plane, so as to convert from a memory size target, to a pixel count target. Next, take the square root, and round up to the nearest integer, so as to convert from a pixel count to a tile width or height target.

If the resulting number equals or exceeds the image width, divide in strips and use this number as a guideline for the strip height.

If not, we recommend using tiles instead. However, as a division in tiles involves padding as well as, likely, additional memory copies in encoding and decoding both, we recommend some extra care in choosing tile dimensions. Divide the image width by the previously calculated number, rounding up to the nearest integer, so as to arrive at a horizontal tile count. Next, divide the image width by the horizontal tile count, rounding up to the nearest integer again, arriving at a new and improved recommended tile width that should minimize horizontal padding needs. Next, round up this new tile width to the nearest multiple of 16. This final adjusted tile width should help ensure

more optimal memory copy speeds and possibly other size and alignment optimizations in many encoding and decoding scenario's on the one hand, whilst still, on the other hand, probably not adding too much significant overhead in the padding department.

Dividing the original size threshold recommendation of currently 1.0 megabyte, by the number of bytes per tile line, and rounding up to the nearest integer, yields a first approximation of the recommended tile height. Next, divide the image height by this tile height, and round up to the nearest integer, to get at the vertical tile count. Finally, divide the image height by this vertical tile count, rounding up to the nearest integer yet again, so to arrive at a final new and improved tile height that minimizes padding needs.

For example, working from a size threshold recommendation of 1.0 megabyte, and aiming to encode a 5426x10528 8bits per sample sRGB image in a single plane...

- Size threshold 1.0 mega = 1048576
- The number of bytes per pixel in a single plane is 3. Dividing by 3 yields a pixel count target of 349525.3.
- Taking the square root and rounding up to the nearest integer, yields a first approximation of tile width or height target of 592.
- 592 is smaller than the image width of 5426, so we will be tiling.
- Dividing the image width 5426 by 592 and rounding up to the nearest integer, yields a horizontal tile count of 10.
- Dividing the image width 5426 by 10 and rounding up to the nearest integer, yields an improved padding minimizing tile width of 543.
- Rounding up to the nearest multiple of 16, yields the final memory copy size and alignment improving tile width of 544. (Horizontal padding in the rightmost tiles is indeed still very limited at 14 pixel rows.)
- Dividing size threshold 1048576, by the number of byte per tile line 544*3, and rounding up to the nearest integer, yields a first approximation of tile height of 643.
- Dividing the image height 10528 by 643 and rounding up to the nearest integer, yields a vertical tile count of 17.
- Dividing the image height 10528 by 17 and rounding up to the nearest integer, yields an improved padding minimizing tile height of 620. (Vertical padding in the bottommost tiles is indeed very limited at 12 pixel lines. The total uncompressed size of a tile is 620*544*3 bytes, is 0.965 megabyte, which is indeed pretty close to our original size target even if we do expect some deviation here as a consequence of minimizing the padding requirements.)

The above detailed raster organization in terms of exact strip height, or tile width and height, is a strong recommendation, not an absolute requirement. The only strict requirement in TIFF/A, inherited from TIFF, is that TileWidth must be a multiple of 16. In real-life, plenty of TIFF files don't comply with either requirement, though. A TIFF/A decoder is required to support any strip and tile dimensions.

5.4.4. Test files

All test files included in this section have a single IFD, 8bit per sample sRGB.

- **Organization\Chunky singlestrip.tif**: PlanarConfiguration is Chunky, single plane is a single strip
- **Organization\Chunky multistrip.tif**: PlanarConfiguration is Chunky, single plane is divided in multiple strips

- **Organization\Chunky tile.tif**: PlanarConfiguration is Chunky, single plane is divided in tiles
- **Organization\Planar singlestrip.tif**: PlanarConfiguration is Planar, each plane is a single strip
- **Organization\Planar multistrip.tif**: PlanarConfiguration is Planar, planes are divided in multiple strips
- **Organization\Planar tile.tif**: PlanarConfiguration is Planar, planes are divided in tiles

The open source solutions proposed by LIBIS and AWare Systems do exactly as specified above. They support all raster organization options. The open source TIFF to TIFF/A converter organizes the TIFF/A output identically to the TIFF input. The conversion of the actual image pixel data is implemented as a strile-to-strile operation.

The pro solution adds compensation for some of the bugs in files collected over many years, without contradicting above specified functionality or meaning. Additionally, the pro converter solution adds the option to re-organize during the conversion process, taking any raster organization in the TIFF input and producing optimally organized TIFF/A output that conforms to above encoder recommendations.

5.5. Compression

5.5.1. Tags

Compression

Tag identification code	259
Type	3 (Short, unsigned 16bit integer)
Count	1

Predictor

Tag identification code	317
Type	3 (Short, unsigned 16bit integer)
Count	1

TIFF supports a vast array of compression options. TIFF/A, in the interest of interoperability, simplicity, licensing and patent issues, as well as for ensuring the subset is future-proof, limits the compression options to uncompressed or flate compressed.

The valid values for the Compression tag in TIFF/A are...

1 No compression

8 Flate compression

The Compression tag is not mandatory. If it's not present, a default value of 1 (no compression) is assumed.

In principle, the Predictor scheme is unrelated to the Compression scheme. In TIFF and TIFF/A, valid values for the Predictor tag are...

1 No prediction

2 Horizontal prediction

The Predictor tag is not mandatory. If it's not present, a default value of 1 (no prediction) is assumed.

The general idea behind prediction is to subtract previous pixel information from current pixel information. It's likely neighboring pixels resemble each other, and thus, statistically speaking, the prediction operation greatly reduces the amount of information in each pixel, limiting that information to what is “new” compared to the previous pixel. This can impact compression ratio greatly.

Inside the encoder, horizontal prediction, if any, is applied to the image data before compression. The first pixel on every line in a “strile”, remains unchanged. In a strile encoding a single channel, this operation can be visualized as follows.

Before applying prediction in the encoder:

R0	R1	R2	R3	R4	...

After applying prediction, before applying compression

R0	R1-R0	R2-R1	R3-R2	R4-R3	...

Note that the operation applies to pixels, meaning, if the plane contains multiple samples, the value that is subtracted from each sample value, is to be taken from the corresponding sample in the previous pixel.

Before applying prediction in the encoder:

R0	G0	B0	R1	G1	…

After applying prediction, before applying compression

R0	G0	B0	R1-R0	G1-G0	…

Conversely, in the decoder, removal of horizontal prediction, if any, is to be applied after decompression.

Horizontal prediction is only defined in combination with integer raster pixel data, i.e. when SampleFormat is 1 (unsigned integer data). For more information on SampleFormat, see section “5.3. Image colorspace”.

Adobe built a specification supplement defining a third Predictor value, specifically concerning prediction for floating point data, but to this date it did not get published. For this reason, it's not included in TIFF/A, even though it's undeniably useful.

5.5.2. Fault tolerant decoding

As the Compression and Predictor tags are at least conceptually unrelated, fault tolerant decoding is not very involved.

A fault tolerant TIFF/A decoder attempts to read the Compression tag. If the tag is not present, or if it has a nonconvertible datatype or count different from 1, the decoder is required to proceed on the assumption Compression has a default value of 1 (no compression).

If the Compression tag is present, with correct or convertible datatype and a single value that is valid in TIFF/A, its value is apparent. If the Compression tag is present, with correct or convertible datatype and a single value, but that value is neither of the valid TIFF/A values, the TIFF/A decoder should not assume the default because the value is likely to have a quite different meaning in a TIFF context. Either the TIFF/A decoder robustly supports decoding the corresponding compression scheme even though it is invalid in TIFF/A, or it should back out with an IFD-image level error.

Handling the Prediction tag is very similar. If the tag is not present, or if it has a nonconvertible datatype or count different from 1, the decoder is required to proceed on the assumption Prediction

has a default value of 1 (no prediction).

If the Prediction tag is present, with correct or convertible datatype and a single value that is valid in TIFF and TIFF/A, its value is apparent. If the Prediction tag is present, with correct or convertible datatype and a single value, but that value is neither of the valid TIFF and TIFF/A values, the TIFF/A decoder should not assume the default. In other words, this specification anticipates that sooner or later the specification supplement for the third option for prediction of floating point data will get published, and other options for prediction may follow. A TIFF or TIFF/A decoder that reads an unknown and unsupported value from the Prediction tag, should thus back out with an IFD-image level error.

Seeing a Prediction value equal to 2 (horizontal prediction), the decoder should verify that SampleFormat equals 1 (unsigned integer data). A fault tolerant TIFF/A decoder that complies with the decoding schemes proposed in this specification, can do so by checking that *BitsPerSampleValue* is either 8 or 16. If it's not, this is an IFD-image level error.

5.5.3. Encoder recommendations

Even though any combination of Prediction and Compression is valid, not all combinations are deemed useful. Combining horizontal prediction with no compression, adds computation overhead to encoder and decoder both, without any gain whatsoever. Combining no prediction, with flate compression, hurts compression ratio greatly, without significant benefit as both the code footprint and computation need required in the prediction step is not very significant compared to code footprint and computation need required in flate compression or decompression.

Note that both horizontal prediction and the flate compression scheme are lossless. It might not always make a lot of sense to output uncompressed TIFF/A files.

5.5.4. Prediction, Compression, and byteorder

TIFF/A supports 16 bits per channel integer sample data. Meaning, byteorder does interfere.

In normal encoder design, the raster data being fed to the encoder very likely has the byteorder that is native to the machine the encoder is running on. The prediction encoding should operate on this data, with samples in machine byteorder, for the subtraction operation to make correct sense.

Compression, however, as a final encoding step, no longer cares about samples. It's an operation on bytes, instead, and these bytes are expected to be in TIFF byteorder. Therefore, the byteswapping stage, if required, is to be positioned in between the prediction and compression stages.

Conversely, in the decoder, if byteswapping is necessary it is to be positioned in between decompression and deprediction stages.

5.5.5. Test files

- **Compression\Intel nopred nocomp.tif**: Well formed, single IFD, 16 bits per sample sRGB, intel byteorder, no prediction, no compression
- **Compression\Intel nopred comp.tif**: Well formed, single IFD, 16 bits per sample sRGB, intel byteorder, no prediction, flate compression
- **Compression\Intel pred nocomp.tif**: Well formed, single IFD, 16 bits per sample sRGB, intel byteorder, horizontal prediction, no compression
- **Compression\Intel pred comp.tif**: Well formed, single IFD, 16 bits per sample sRGB, intel byteorder, horizontal prediction, flate compression
- **Compression\Motorola nopred nocomp.tif**: Well formed, single IFD, 16 bits per sample sRGB, motorola byteorder, no prediction, no compression

- **Compression\Motorola nopred comp.tif**: Well formed, single IFD, 16 bits per sample sRGB, motorola byteorder, no prediction, flate compression
- **Compression\Motorola pred nocomp.tif**: Well formed, single IFD, 16 bits per sample sRGB, motorola byteorder, horizontal prediction, no compression
- **Compression\Motorola pred comp.tif**: Well formed, single IFD, 16 bits per sample sRGB, motorola byteorder, horizontal prediction, flate compression

The open source solutions proposed by LIBIS and AWare Systems do exactly as specified above. Its decoding part, even in the context of TIFF to TIFF/A conversion, is limited to support for no compression and flate compression. It does not support any other compression scheme in TIFF that is not valid in TIFF/A anyway. The open source converter can take TIFF input with any combination of prediction and the said supported compression schemes, and output TIFF/A with any other combination, and can therefore be used to convert uncompressed into predicted and flate compressed.

The pro solution has a very wide range of TIFF compression support in the input decoding part of the TIFF to TIFF/A converter. The pro solution also adds compensation for some of the bugs in files collected over many years, without contradicting above specified functionality or meaning.

5.6. IFD organization

SubIFDs

Tag identification code	330
Type in ClassicTIFF/A	13 (IFD)
Additional Type in BigTIFF/A	18 (IFD8)
Count	1

Usage of the SubIFDs tag is specified in section "4.3. The TIFF/A IFD tree". Note also section "4.2.4. IFD arrays vs IFD lists" that clarifies above specified count of 1.

5.7. Optional additional information

ImageDescription

Tag identification code	270
Type	2 (ASCII)
Count	Character count, including terminating 0 character

Make

Tag identification code	271
Type	2 (ASCII)
Count	Character count, including terminating 0 character

Model

Tag identification code	272
Type	2 (ASCII)
Count	Character count, including terminating 0 character

Software

Tag identification code	305

Type	2 (ASCII)
Count	Character count, including terminating 0 character

DateTime

Tag identification code	306
Type	2 (ASCII)
Count	Character count, including terminating 0 character. Given the fixed format (see below), this should always be 20.

Artist

Tag identification code	315
Type	2 (ASCII)
Count	Character count, including terminating 0 character

Copyright

Tag identification code	33432
Type	2 (ASCII)
Count	Character count, including terminating 0 character

DocumentName

Tag identification code	269
Type	2 (ASCII)
Count	Character count, including terminating 0 character

PageName

Tag identification code	285
Type	2 (ASCII)
Count	Character count, including terminating 0 character

PageNumber

Tag identification code	297
Type	3 (Short, unsigned 16bit integer)
Count	2

None of these tags are mandatory, none have a default value. None can have any impact on actual image rendering, be it a default rendering or application-specific rendering in a closed system.

The ImageDescription tag describes the subject of the image.

The Make tag value is the manufacturer of the equipment (scanner, video digitizer, ...) used in generating the image. The Model tag is used to communicate the equipment model name or number.

The Software tag contains the name and version number of the software package used to create the image.

DateTime is the date and time of image creation. The format is: "YYYY:MM:DD HH:MM:SS", with hours like those on a 24-hour clock, and one space character between the date and the time.

The Artist tag contains the name of the person who created the image.

The Copyright tag contains a copyright notice. The complete copyright statement should be listed in this field including any dates and statements of claims. For example, "Copyright, John Smith, 1999. All rights reserved."

DocumentName is the name of the document from which this image was scanned.

PageName is the name of the page from which this image was scanned.

The PageNumber tag contains two unsigned 16bit integers. The first is the zero based page index number, the second is the total number of pages in the document.

Note the subtle difference between PageName and PageNumber. If a document has some sort of prelude or foreword, it's not uncommon to have a separate page numbering scheme for that section. If two pages of prelude are followed by three pages of actual core document, the PageNames might be “i”, ”ii”, “1”, “2”, and “3”. The PageNumbers however, should be 0/5, 1/5, 2/5, 3/5, and 4/5.

Note also that the order of the separate pages in a multipage TIFF file is not related to either PageName or PageNumber tags, and is fully and only determined by the order of the IFDs in the top-level linked list. If a TIFF file contains two such top-level linked IFDs, the first IFD has PageNumber 1/2 and the second has a PageNumber tag with value 0/2, the first IFD is still to be regarded the first page in any default rendering context.

6. The TIFF/A codec pipeline

This section is purely informative. A TIFF/A codec can choose to ignore this section completely, and still be a valid and compliant TIFF/A codec, and even output the exact same files from the encoder part.

6.1. The strile pipeline

As mentioned in section "5.4. Raster organization", one major advantage of TIFF 6.0 and TIFF/A support for cutting up the image raster into separate planes, possibly, and strips or tiles, is that each resulting section can be individually encoded or decoded, without dependence on data in any other section. If used with appropriate granularity (which may not always be the case in a file to be decoded, but can be guaranteed in an encoding context), multiple threads can efficiently be deployed, each working on their own independent section. On machines with multiple CPUs, multiple cores per CPU, and/or multiple threads per core, this can speed things up dramatically.

This implies our attentions shifts from a raster pipeline, to a strile pipeline. We'll no longer need the former, we'll need multiple instances of the latter. Ideally, the input end of the encoder strile pipeline can interface directly with some raster object probably defined by an imaging library. Ideally, it can easily retrieve exactly the required sample, if PlanarConfiguration is Planar, and exactly the required rectangular region that is the particular strile. Conversely, a decoding strile pipeline is hopefully able to dump the decoded strile data output into the raster object defined by the imaging library. Hopefully, this raster object supports access such that the decoding strile pipeline is able to write exactly the sample or samples and rectangular region that is the strile, without unnecessary memory copies or precomposition of individual striles.

6.2. The streaming pipeline

Also mentioned in section "5.4. Raster organization", is the fact that a decoder should be able to deal with any strile size. A strile in a file to be decoded, might be 10 samples per pixel 64bit per sample floating point data. It might be a single strip encompassing a huge raster.

If the pipeline is designed to work from a single complete input buffer, and output another complete buffer, memory constraints or even addressing space constraints can easily become an issue. There's a huge burden on the memory manager, allocating large chunks of memory in a single go, which is never a good idea in any multithreading scheme. Swapfile activity may occur and slow things down dramatically. Last but not least, the scheme could never be made to meet some of the requirements imposed in some advanced image handling contexts. For example, this does not sit well with an imaging library that supports asynchronous progressive rendering.

It's highly recommended to implement the encoding and decoding pipelines both as an array or linked list of individual steps that either pull small chunks of data from the previous step, as needed, or push small chunks of data to the next step, as they become available. Such a pipeline can be 'executed' by either continuously pulling data from the final step, if it's a pulling pipeline, or continuously pushing data to the first step, in a pushing design. Execution ends when data is exhausted.

Most steps deal with chunks of raster, and a suitable small chunk of data might be single strile line. The TIFF/A pipeline does not contain any processing step where data from one strile line interferes with data from another strile line, so pushing or pulling strile line sized blocks of data between steps seems very suited.

The output of the compression step, if the IFD image data is indeed flate compressed, is the only output in the encoder pipeline that cannot be strile line sized, as lines of pixels are no longer even a concept in the flate compressed data. But still it's quite feasible to stick with pulling or pushing of small chunks of data. The compression step can be made to output blocks of 8 or up to 32 kilobyte.

Either it'll push such a block whenever it's ready to do so, in a pushing design, or it'll pull whatever number of lines it requires in order to arrive at a compressed data block this size, in a pulling design.

Conversely, the input of the decompression step, if the IFD image data is indeed flate compressed, is the only input in the decoder pipeline that cannot be strile line sized, as lines of pixels are not even a concept yet as long as we're dealing with flate compressed data. But, again, it's quite feasible to stick with pulling or pushing small chunks of data.

Suddenly, there's the option to append encoder strile streams to decoder strile streams. Possibly inserting a step of pixel processing in between. This opens up unforeseen possibilities. One could be transcoding multigiga rasters on quite restrictive hardware, with an extremely low memory footprint and absolutely no fear of involving swapfile activity.

Additionally, there are advantages in codec implementation. Implementing every processing step individually, this way, with all steps sitting in a single encompassing pipeline design, implies all of them can be freely combined. When the prediction encoding step is implemented, and so is the compression encoding step, you have the option to build a pipeline that predicts but does not compress, a pipeline that predicts and compresses, a pipeline that does not predict and not compress, and finally you have to option to build a pipeline that does not predict but does compress. There's no longer any worries about the multitude of possible combinations of features. No more worries about what exact routine to call from the compression step, as, whatever may be the prediction situation, in the pipeline design any steps simply pulls data from the previous step or pushes data to the next step. In other words, the pipeline design is what supports the ability to combine, the only worry left is each feature individually.

This also translates into a performance advantage. In this design, one is easily able to eliminate a lot of branching from the critical path. Instead of having a single byteswapping routine that continuously re-decides if either byteswapping is unnecessary, or byteswapping 16bit samples is required, or byteswapping 32bit samples is required, or if byteswapping 64bit samples is required, one ends up with three implementations for a byteswapping step. The branch is moved to the pipeline creation stage. Maybe no byteswapping is needed, but if it is, the step operating on the correct sample sized is added to the pipeline. There is next no more branching inside the actual byteswapping critical path.

This also translates into a maintenance advantage. If at some point one decides byteswapping 64bit samples could benefit from a new SIMD implementation, there's just one new step to implement and one need not touch any other step's code. In the pipeline building stage, instead of adding the old non-SIMD 64bit byteswapping step, the updated version checks if the machine supports the required SIMD technology, and either adds the old non-SIMD or the newer SIMD step. Updating code, in this design, be it for technological or feature enhancement or whatever, one never has to hunt down all direct or indirect calls to a routine that are potentially everywhere in any of the conceptually possible previous or next routines and make a multitude of error-prone changes. The single centralized point that needs updates in order to change the chain that is encoding or decoding, is where the chain gets created, whatever the update may be or wherever it sits relative to other operations in the chain.

6.3. The pipeline steps

6.3.1. The encoder pipeline

- The input step retrieves image data from some form of raster object.
- Next may be a color conversion or sample bitdepth or datatype conversion step, if the encoder supports input data in colorspaces and/or sample bitdepths or datatypes different from what is to be inside the TIFF IFD.

- Next up is a prediction step, if prediction is required. See “5.5. Compression”. For the sake of performance, apart from a general purpose prediction step for 8bit integer samples, and a general purpose prediction step for 16bit samples, the encoder might have some specialized versions with hardcoded most common sample counts.
- Next is the byteswapping step, if byteswapping is required. See “2. TIFF/A file header” and “5.5. Compression”. The encoder will require at least three flavors of this step, one for each of the 16bit, 32bit, and 64bit bits per sample options.
- Depending on implementation details, some horizontal and/or vertical padding step might be required. See “5.4. Raster organization”.
- Next is the flate compression step, if flate compression is required. See “5.5. Compression”.
- The final step in the encoder strile pipeline, will likely dump to some temporary buffer. The final result of this asynchronous strile encoding task, will be this buffer, finalized and ready to be appended to the file.

6.3.2. The decoder pipeline

The decoder pipeline almost exactly mirrors the encoder pipeline, except for some steps that help ensure a particular output format.

- The input step retrieves compressed strile data from some file object or functional equivalent.
- Next is the flate decompression step, if flate decompression is indeed required. See “5.5. Compression”.
- Depending on implementation details, some horizontal and/or vertical cropping step might be required. See “5.4. Raster organization”.
- If PlanarConfiguration is Chunky, the strile may contains extra samples. It's possible the calling code might not require some of these extra samples. This is the correct place to insert a step that drops the superfluous samples.
- Next is the byteswapping step, if byteswapping is required. See “2. TIFF/A file header” and “5.5. Compression”. The decoder will require at least three flavors of this step, one for each of the 16bit, 32bit, and 64bit bits per sample options. Due to the nature of byteswapping, there's no different requirements in encoder and decoder here, so the same step implementations can be used in either pipeline.
- Next up is the deprediction step, if deprediction is required. See “5.5. Compression”. For the sake of performance, apart from a general purpose deprediction step for 8bit integer samples, and a general purpose deprediction step for 16bit samples, the decoder might have some specialized versions with hardcoded most common sample counts.
- Most decoders will likely support color conversion or at least sample bitdepth and datatype conversion. The need to support 4 different combinations of sample formats and bitdepths in a TIFF file, conflicts with the fact that many application level or imaging library level code is designed to deal with a more limited range of pixel data. For example, TIFF/A supports 64bits per floating point sample sRGB, and decoders are required to make sense of this. But the imaging library may only support 8bits per unsigned integer sample sRGB. Or it may support 64bits per floating point sample sRGB, but may not be ready to deal with data that is outside the [0.0,1.0] range whilst TIFF/A does allow out of gamut data to be encoded outside of this range. In all such cases where TIFF/A supports options that an interfacing imaging library does not, decoders will require conversion steps inserted at this point in the pipeline.
- The final step in the decoder strile pipeline, will likely dump image data into some form of

raster object.

Appendix A: Differences between TIFF 6.0 and TIFF/A

1. Specification

One significant difference in approach between said specifications, is the fact that TIFF 6.0 limits itself to specifying the file format as it should be. This TIFF/A specification exceeds this with guidelines for implementation in many cases. That includes fault tolerant decoding algorithms based on experience with buggy real-life files, and encoder recommendations that ought to result in files that are not just unambiguously valid, but also optimal for storage and decoding both.

Another major difference lies in the fact that TIFF 6.0 tries and divides the valid range of TIFF into separate classes. TIFF 6.0 also makes a distinction between baseline and non baseline.

Many people feel this division is arbitrary and deceptive. Certainly, widespread TIFF usage has evolved since TIFF 6.0 was published in 1992, and much of the basic TIFF usage these days mixes elements from baseline and non baseline both, and does not fully fit any of the original TIFF classes.

TIFF/A focuses on largely independent individual TIFF features, instead of enumerating particular feature combinations into classes. The TIFF/A approach fits the nature of TIFF that is a quite logical consequence of the format's properties, and everyday contemporary usage of the file format, both. Additionally, its approach allows the specification to fully cover the format, as opposed to TIFF 6.0 which these days is regarded an enumeration of usage examples from which the format is to be interpolated.

2. Format

Comparing TIFF/A on the one hand, with TIFF 6.0 as interpolated from the said enumeration of usage examples in the original specification, combined with specification supplements published at later dates and the newer BigTIFF extension, on the other hand...

- **Colorspace** in TIFF/A is limited to sRGB, be it 8bit or 16bit per sample integer sRGB or 32bit or 64bit per sample floating point sRGB. TIFF 6.0 allows for a much wider range of colorspaces and bitdepths. Some of the colorspace features allowed in TIFF 6.0, hardly got any adoption or support, including for example the ability to have different bitdepths and samplesformats for different samples in the same image data. TIFF/A limits the valid colorspace range to a subset that can be universally agreed upon, is very well documented, unencumbered by restrictive licenses, and is very well supported in the vast majority of existing code bases. At the same time, though, by allowing floating point data, TIFF/A enables support for color that is outside of the sRGB gamut which should help ensure TIFF/A remains future-proof.
- TIFF 6.0 and TIFF/A are largely the same when it comes to raster **organization**. Both allow chunky or planar storage, both allow division in strips or tiles. Both require the decoder to support any organization choices. TIFF 6.0 is somewhat deceptive in that it classifies tiled storage as an extension rather than baseline, whilst it is in fact so widespread and adopted that any TIFF decoder that lacks support for tiled storage would be completely useless in real life. TIFF/A does not suffer from this shortcoming, as it does attempt to divide the format into baseline and non baseline.
- TIFF/A restricts **compression** options to no compression or flate compression. TIFF 6.0 is weighed down with a much wider range of compression options. Some are considered obscure and depreciated, including packbits and LZW. Some may or may not come with troublesome licensing issues, depending on one's definition of troublesome licensing issues which may be troublesome in itself, including said LZW and maybe even JPEG and JBIG. Some are lossy and thus conflicting with archiving needs. In all of these categories, some

are considered baseline by the original TIFF 6.0 specification, some are considered an extension, even though some that are considered extension are actually very widely adopted and hard to replace. Some were never actually specified at all. Some weren't even invented yet in 1992, but are nonetheless considered vital in basic TIFF usage today. It's difficult to come up with an exhaustive list of compression schemes a TIFF codec needs to support in order to be relevant and practically useful. The rigorous restriction in TIFF/A, aims to maximize clarity and support for archiving requirements in particular, whilst at the same time minimizing demands made on the TIFF/A codec, and completely eliminating any licensing issues.

- TIFF 6.0 is extremely **extendable**. In closed systems, one can define new colorspaces, new compression schemes, new private tags and even new private IFD types. Additionally, new tags and IFD types can be registered and gain a global definition and more widespread support. This conflicts with the needs of an archival file format like TIFF/A. Minimizing chances for digital dark age issues implies the need for a single central definition covering all that is possible in the file format. TIFF/A therefore does not allow extension.
- TIFF 6.0 specification supplements include a definition for **SubIFDs**. This is essentially identical to the SubIFDs scheme in TIFF/A. However, the original TIFF 6.0 specification includes references to the older scheme based on the SubfileType tag, defining its status as “depreciated”, as well as including a full definition of another old but newer scheme based on the NewSubfileType tag. Depending on ones definition of what exactly is TIFF 6.0, whether or not the word “depreciated” implies or excludes validity, and whether or not practical usage and widespread adoption has any impact on what is valid in a format whose most recent specification is several decades old, any of these schemes may be regarded part of TIFF 6.0. Only the newest SubIFDs scheme is valid TIFF/A.
- TIFF/A, in order to completely eliminate the possibility of hidden data having undefined meaning, excludes **unused addressing space** in a file. TIFF does not.

Appendix B: Differences between TIFF/EP and TIFF/A

1. Overview

The difference between TIFF/EP and TIFF/A could hardly be bigger, due to the fact that both standards target a completely different application and audience. TIFF/EP is a camera file format. TIFF/A is an archival file format. TIFF/EP aims to support specification of all possible camera settings, and ease the workflow in newspaper photography and the like. TIFF/A aims to minimize digital dark age issues and focuses on simplicity and readability twenty or indeed two hundred years from now.

In this sense, a comparison between these vastly different formats resulting from vastly different goals, may seem farfetched. But both have a common parent file format in TIFF 6.0, and TIFF/EP is well known and widely adopted. A comparison therefore may serve as a shortcut highlighting what exactly is TIFF/A for anyone familiar with TIFF/EP. This section may also serve as a guideline for adding TIFF/A support to an existing TIFF/EP codebase, but it is not normative and does not aim to be exhaustive.

2. Format

- **BigTIFF** support is not included in TIFF/EP. It might not make a lot of sense in TIFF/EP, few camera's or photography processing workflows will see the need to support files growing beyond 4 gigabyte in size at this point in time. TIFF/A needs to support scans that often are orders of magnitude bigger than single photographs. Contemporary high-end scanners are often implemented as motorized digital camera's, resulting in scans that are vast stitched arrays of photographs. A target TIFF/A user might also see the need to encode scans of all the pages in one of Leonardo Da Vinci's sketchbooks in a single file. And TIFF/A by its very nature should be made to be future proof and anticipate an age when today's resolutions may seem ridiculously low. Therefore, TIFF/A does include BigTIFF support.
- TIFF/EP does not include the **tag datatype 13 (IFD),** nor, of course, its BigTIFF equivalent 18 (IFD8). It therefore enforces the problematically ambiguous use of the datatype 4 (Long, unsigned 32bit integer). TIFF/A focuses on transparency. It excludes the validity of unused addressing space and requires a validating process to map said addressing space. It therefore makes sense for TIFF/A to distinguish between unsigned 32bit integer data that is meant to communicate an actual integer value on the one hand, and a pointer to an auxiliary IFD on the other hand. (See section “4.2.2. Datatype tolerance” in this specification, section “4.1.2.6 Types” in the TIFF/EP specification.)
- TIFF/EP supports no **compression** and JPEG compression, which, again, is a natural consequence of its target application. TIFF/A requires a similar limitation in compression options to achieve the same level of clarity and transparency and rid itself of the historical accumulation of compression modes in TIFF 6.0. Its different goals however imply a lossless compression scheme requirement instead. Thus, TIFF/A supports no compression and flate compression. (See section “5.5. Compression” in this specification, section “4.2 Image data” in the TIFF/EP specification.)
- TIFF/EP supports the greyscale and RGB **colorspaces**. Its support for JPEG compression de facto necessitates inclusion of the YCbCr colorspace, and its ties to camera technology in particular brings forth the inclusion of CFA. TIFF/A does not require YCbCr, as it does not require JPEG compression, and has no meaningful use for CFA schemes. TIFF/A does not support a flavor of greyscale, because this is a subset of RGB at best, and comes with its own specification problems at worst, depending on ones views on the subject. Whilst in 1992 when TIFF 6.0 got published, RGB was, well... RGB, the industry has since

encountered problems with the fact that one manufacturer's RGB does not quite equal another manufacturer's interpretation of the colorspace. The industry's response to this problem was the sRGB colorspace specification. TIFF/A acknowledges this and all mention of RGB is replaced with sRGB in this specification. (See section "5.3. Image colorspace" in this specification, section "4.2 Image data" in the TIFF/EP specification.)

- Support for JPEG compression and YCbCr in TIFF/EP, requires that standard to deal with **YCbCrSubsampling**. In real life, many code bases have serious trouble with the subject, mainly as a consequence of the redundant encoding of these subsampling values in both the TIFF IFD and the actual JPEG compressed data streams on the one hand, which opens up the possibility for redundant copies of these subsampling values to be different, and the fact that most open source solutions require the use of two different libraries and difficult glue code to pull this off, on the other hand, introducing a lot of potential to get it wrong. TIFF/A, eliminating the need for YCbCr as JPEG compression contradicts the lossless needs inherent to archival workflows, does not suffer these real life practical problems surrounding subsampling that aren't immediately apparent from a theoretical point of view. (See section "4.2 Image data" in the TIFF/EP specification.)
- TIFF/EP supports **organization** in strips and tiles both, as does TIFF/A, as does the parent file format TIFF 6.0. However, TIFF/EP dictates bottom strips ought to be padded (section "4.2 Image data" in the TIFF/EP specification includes the sentence "If necessary, the final strip can be 'padded' with zeros."). This contradicts the TIFF 6.0 specification (in various places, the most clear probably being the RowsPerStrip description on page 19 of the TIFF 6.0 specification where it says "The data in the last strip is not padded..."). TIFF/A opts to comply with TIFF 6.0, and thus does not allow padding of the bottom strip. (See section "5.4. Raster organization" in this specification.)
- The recommended **target uncompressed size** of individual strips or tiles, 8 kilobyte in TIFF 6.0, is redefined to 64 kilobyte in TIFF/EP. This recommendation in TIFF/EP is likely at least partly inspired by implementation details in an era where 16bit processing was still relevant, even if surmountable and temporary implementation detail and solid specification design should probably not easily mix. TIFF/A being targeted at long term usage with potentially vastly bigger imagery, cannot afford a recommended size this small, or indeed a recommended size that is hardly future-proof. TIFF/A instead provides a guideline that is designed to grow at the same pace as does hardware capability, and practical needs. (See section "5.4.3. Encoder recommendations" in this specification, and section "4.2 Image data" in the TIFF/EP specification.)
- TIFF/EP's **IFD tree** is, again, very much targeted at its specific application. TIFF/EP includes mention of a single thumbnail to accompany a full-size image. Dealing with typical photograph sizes, that makes perfect sense. TIFF/A deals with potentially much larger rasters, and is best served with a full 'tile pyramid' scheme. It recommends steps in between thumbnails and very large images, where each step has a pixel count of roughly one sixteenth of the previous bigger image, so as to not take up to much storage space nonetheless. TIFF/EP locates the thumbnail in the top-level linked list of IFDs, and the full-size image in the SubIFD. As this full-size image in TIFF/EP potentially has CFA data that does not make for good interoperability with the majority of TIFF codecs, it's quite logical to try and tuck it away in a SubIFD and have the thumbnail in the main linked list. TIFF/A is unencumbered by CFA interchange issues, and it's deemed more sensible to have the full-size image to be the most apparent one, and likely the only one actually noticed by old software that doesn't support the SubIFDs tag. A third and final difference, is TIFF/EP's attempt to mix the old NewSubfileType tag based IFD relationship scheme into the scheme build on the new SubIFDs tag. This leads to redundancy and opens up the possibility to have conflicting information in a single file. In real life, a TIFF/EP decoder is likely to simply ignore the NewSubfileType tag, as all information it actually needs is readily derived from

SubIFD based data. TIFF/A unambiguously opts for the SubIFDs scheme, NewSubfileType or the even older SubfileType is not valid in TIFF/A. (See section "4.3. The TIFF/A IFD tree" in this specification, and section "4.3 Thumbnail images" in the TIFF/EP specification.)

- TIFF/EP reinterprets the top-level linked list of IFDs to mean "motion sequence burst of images". Yet again, perfectly sensible for a camera file format. Yet again, there's not even an overlap with anything meaningful in the context of an archival file format. TIFF/A sticks with the standard **multipage** interpretation. (See section "4. The IFD tree", as a whole, in this specification, and section "4.4 Burst sequences using chaining" in the TIFF/EP specification.)
- TIFF/EP supports **embedded color profiles**. In a camera file format, this makes perfect sense. And indeed, many people feel modern image processing ought to be based on strict color management standards. In TIFF/A, though, this conflicts with the need of simplicity and brevity of the specification, as well as transparency of all directly and indirectly involved licenses, specification licenses, and patents. Furthermore, color management is not as widespread as maybe it ought to be. The TIFF/A scheme based on sRGB without support for embedded profiles, is compatible with older software that does not support color management (or the correct color management specification flavor and version). But it's also compatible with software that is color management based, as these usually have sRGB profile resources to apply in these circumstances. Whilst TIFF/A with support for embedded profiles would still be compatible with software that includes correct color management, it opens up the possibility to encode data such that it makes no sense for software that doesn't. TIFF/A explicitly allows floating point sRGB data to be out of the normal range, thus not imposing a gamut limitation. sRGB used this way ought to be future-proof. For all these reasons, TIFF/A does not include support for embedded color profiles. (See section "4.5 Camera colour space information" in the TIFF/EP specification.)
- TIFF/EP includes a **versioning** scheme as tag data. This conflicts with the spirit and practical usage of TIFF, where individual IFDs can be unlinked, relinked, and transplanted from one file to another. A version number that applies to the file, cannot correctly be included in any individual IFD. The only correct place is the TIFF file header. This, however, would involve breaking the TIFF file format standard. The single only remaining solution, is to not require version numbers in the first place. If the purpose of the version number is "to allow a TIFF/EP compliant file to identify itself to a TIFF/EP aware reader", and such identification is actually required for real and practical reasons, the software that suffers this requirement should simply directly check for TIFF/EP compliance, instead. TIFF/A does not include any version numbers, neither on the file nor on the IFD level.
- TIFF/EP limits the valid values for the **Orientation** tag. More importantly, it adds a value that is not valid in TIFF 6.0, for communicating "unknown" orientation. Seeing any subsequent image processing based on data retrieved from a source with "unknown" orientation, either wouldn't care about actual orientation, or either is forced to deal with this data as if it had an orientation and thus likely pretend this data has default orientation, it's unclear for what practical real-life purposes unknown orientation differs from default orientation. The implication is though that TIFF/EP cannot be said to be a valid subset of TIFF 6.0. (See section "5.4. Raster organization" in this specification, and section "5.2.12 Orientation" in the TIFF/EP specification.)
- TIFF/EP excludes alpha and application-specific **extra samples**, if section "5.2.19 SamplesPerPixel" is to be taken literally and is meant to be exhaustive. A camera file format likely has little use for these extra samples. A TIFF/A archived file containing data taken from a famous painting, may however want to include x-ray data used in some restoration, or the rasterized results of chemical tests. Infrared data or road map drawings may

accompany aerial photography in map archives, etc. TIFF/A does inherit support for extra samples from TIFF 6.0. (See section "5.3. Image colorspace" in this specification, section "5.2.19 SamplesPerPixel" in the TIFF/EP specification.)

- TIFF/EP supports **extension** with new compression methods, even though it does impose restrictions on the validity of the existing TIFF 6.0 compression methods. TIFF/A is an archival file format, it cannot and does not allow any extensions as it cannot afford potential inclusion of insufficiently specified data.
- TIFF/EP refers to a number of **external technologies and specifications**. Support for color profiles and JPEG compression has been discussed above. It also refers to the IPTC/NAA standard (section "5.2.43 IPTC/NAA"). At the end of the day, TIFF/EP is forced to include in its foreword "Attention is drawn to the possibility that some of the elements of this... may be subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.". This may be an ISO policy, possibly included in every specification published by ISO, but the point is that in the case if TIFF/EP it is actually necessary to include this disclaimer. TIFF/A cannot possibly do that. The whole point of building an archival file format is evading digital dark age issues and aiming to guarantee usability and interoperability today and continued readability in the future. TIFF/A cannot claim to meet these goals if there's too much usage of external technologies, that come with a number of licensing and patent issues all by itself and in turn may add their own references. TIFF/A has one unavoidable such reference, namely to flate compression. We're confident that's a safe reference. TIFF/A thus excludes the possibility that some of the elements in the TIFF/A specification may be subject of patent rights. As to licensing, TIFF/A can safely claim its own specification license and flate compression license are the only ones relevant to this file format.

Appendix C: Differences between TIFF/IT and TIFF/A

1. Overview

It is hardly even possible to compare these file formats, due to the fact that both standards target a completely different application and audience. TIFF/IT is a prepress file format. TIFF/A is an archival file format. TIFF/IT aims to support the specific needs of professional printing technology. It's quite elaborate and complicated, and divides up into different classes, specifically reflecting different printing processes. TIFF/A aims to minimize digital dark age issues and focuses on simplicity and readability twenty or indeed two hundred years from now.

TIFF/IT breaks TIFF 6.0 in many ways. TIFF/IT is certainly inspired by TIFF, but it is not a valid subset of TIFF 6.0 by far. It's best regarded a different file format altogether. This fact is hardly obscured, it is explicitly stated in the TIFF/IT specification itself in various sections...

- 7.4.1 TIFF/IT-LW, TIFF/IT-LW/P1 and TIFF/IT-LW/P2
- 7.5.1 TIFF/IT-HC, TIFF/IT-HC/P1 and TIFF/IT-HC/P2
- 7.6.1 TIFF/IT-MP, TIFF/IT-MP/P1 and TIFF/IT-MP/P2
- 7.7.1 TIFF/IT-BP, TIFF/IT-BP/P1 and TIFF/IT-BP/P2
- 7.8.1 TIFF/IT-BL and TIFF/IT-BL/P1
- 7.9.1 TIFF/IT-SD and TIFF/IT-SD/P2
- 7.10.2.2 FP file structure

In an ideal world, a file format that is not TIFF, does not reuse the name “TIFF”, and above all it does not reuse the TIFF signature values in the file header. This being a less than ideal world, at least some of the time, and TIFF/IT being regarded a distant relative in the TIFF family of formats by some as a consequence, an attempt is made to compare these formats in some of the few areas that have some common name or meaning.

This comparison cannot claim to be complete. It's highly recommended to regard any comparison between these formats as impossible and irrelevant, and build from the fact that TIFF/IT and TIFF/A don't have any common goal, any common content, not even any common ancestor.

2. Format

- TIFF/IT does not support **BigTIFF**, though the notion is absurd as TIFF/IT is not TIFF. TIFF/A does.
- TIFF/IT does not include the **tag datatype 13 (IFD),** nor, of course, its BigTIFF equivalent 18 (IFD8). It therefore enforces the problematically ambiguous use of the datatype 4 (Long, unsigned 32bit integer). TIFF/A focuses on transparency. It excludes the validity of unused addressing space and requires a validating process to map said addressing space. It therefore makes sense for TIFF/A to distinguish between unsigned 32bit integer data that is meant to communicate an actual integer value on the one hand, and a pointer to an auxiliary IFD on the other hand. (See section “4.2.2. Datatype tolerance” in this specification, section “7.1.5 IFD entry” in the TIFF/IT specification.)
- TIFF/IT makes external references and includes technologies external to the specification. This includes but is possibly not limited to references for the IT8Header tag (section “7.2.3 System identification”), ICC profiles (section “7.2.8.4 Colour values”), and JPEG compression.
- **Compression** options in TIFF/IT are numerous. It supports CCIITT G4, JPEG, flate, and

also 4 compression options that are not part of TIFF 6.0 and contribute to the fact that TIFF/IT is not TIFF. TIFF/A needs clarity, simplicity, and also transparency when it comes to licensing and patent issues surrounding the included technologies. TIFF/A supports only no compression and flate compression.

- **PlanarConfiguration** in TIFF/IT includes an option that is not valid in TIFF 6.0 and contributes to the fact that TIFF/IT is not TIFF. TIFF/A opts for TIFF 6.0 PlanarConfiguration options.
- **RasterPadding** tag in TIFF/IT completely breaks TIFF 6.0 interpretation of TIFF/IT files. Unfortunately, it does this in a way that most decoders are probably unlikely to notice. In situations where RasterPadding is actually used, a default TIFF 6.0 rendering is likely to result in random distortion, rather than a detectable error.
- In the **colorspaces** department, TIFF/IT supports greyscale, RGB, YCbCr, Lab, and a form ressembling what is called “Separated” in TIFF 6.0. The latter is problematic from a TIFF 6.0 point of view, all by itself, but the actual implementation in TIFF/IT is such that no TIFF 6.0 decoder can likely make sense of it in most situations. (See section “7.2.8 Colour specification”, and the remark on TIFF/IT-CT/P2 CMYKEquivalent tag in section “7.31 TIFF/IT-CT, TIFF/IT-CT/P1 and TIFF/IT-CT/P2”)
- Even though some valid TIFF 6.0 **PhotometricInterpretation** values are used in TIFF/IT, their actual rendering meaning can be modified with TIFF/IT tags such as “ImageColorIndicator” and “BackgroundColorIndicator”, actually breaking the specification of these PhotometricInterpretation values from a TIFF point of view, also likely resulting in incorrect rendering when interpreted by correct TIFF 6.0 decoders.
- TIFF/IT supports ICC **color profiles**. TIFF/A does not. (See section “7.2.8.4 Colour values” in the TIFF/IT specification.)
- TIFF/IT supports the NewSubfileType tag, though it does not seem to have much purpose for it. TIFF/A supports proper **IFD trees** on the basis of the newer SubIFDs scheme.

Note that nothing here is actually valid criticism on the TIFF/IT file format, nor is it intended as such, on the condition that TIFF/IT is not mistaken for a flavor of TIFF, or an interchange or archival file format. If TIFF/IT is regarded as its own format, serving prepress needs perfectly fine, as it is intended, none of the above is anything other than a useful feature. Except for the badly chosen name, and the impractical and confusing reuse of another format's header and signature value.