University of Illinois Electrical and Computer Engineering

ECE 496/499 Senior Project

> Author: Kelvin Ma

> Advisor: Dr. Zbigniew T Kalbarczyk

Swift *JPEG*: Contributor's Guide

Swift *JPEG* is a cross-platform pure Swift framework which provides a full-featured JPEG encoding and decoding API. The core framework has no external dependencies, including *Foundation*, and should compile and provide consistent behavior on *all* Swift platforms. The framework supports additional features, such as file system support, on Linux and MacOS. Swift *JPEG* is available under the GPL3 open source license.

i. project motivation

> **Summary:** Unlike *UIImage*, Swift *JPEG* is cross-platform and open-source. It provides a rich, idiomatic set of APIs for the Swift language, beyond what wrappers around C frameworks such as *libjpeg* can emulate, and unlike *libjpeg*, guarantees safe and consistent behavior across different platforms and hardware. As a standalone SPM package, it is also significantly easier to install and use in a Swift project.

i.i. problem

Today, almost all Swift users rely on two popular system frameworks for encoding and decoding the JPEG file format. The first of these system frameworks is *UIKit*, which is available on Apple platforms and includes a multi-format image codec, *UIImage*. However, this codec is proprietary and unavailable on Linux platforms, making tools and applications that depend on *UIImage* non-portable.

The second popular system framework is the C library *libjpeg* which comes pre-installed with most Linux distributions. The *libjpeg* codec, which has existed since 1991, has the advantage of having a large user base, and unlike *UIImage*, is free and open source software.

The *libjpeg* codec however, has a number of drawbacks which make it unsuitable for use in Swift projects. Despite Swift's excellent C-interop, installing and importing *libjpeg* into Swift projects can be challenging for all but advanced Swift users.

Owing to vast differences in programming paradigms and preferred design patterns between C and Swift, APIs designed for (and constrained by) the C language can also be extremely awkward, and needlessly verbose when called from Swift code. Swift wrappers around C APIs can mitigate some of these issues, but must still incur necessary overhead to bridge the gap

between a framework designed for a language without dynamic arrays, automatic reference counting, or the concept of memory state, and a calling language which relies on modern data structures and guarantees for safe and efficient operation.

The *libjpeg* codec specifically also suffers from serious technical flaws which preclude its safe inclusion in Swift projects. Error handling in *libjpeg* relies heavily on the `setjmp` family of POSIX functions, which are unsafe to use in Swift (and many other languages as well). The output from *libjpeg* can also vary across different hardware due to differences in platform rounding and SIMD architecture.

i.ii. proposed solution

A major, and in our opinion, beneficial, trend in modern language design, has been to distribute language compilers with package managers that can pull code from the internet to be compiled locally by a developer's compiler (or interpeter) toolchain. The most famous examples might be Node and Python's `pip` tool. In Swift, the equivalent is the Swift Package Manager (SPM). While the Swift Package Manager is capable of linking to system C libraries, this process is generally not automated and entails some complexity on the part of users. A native-Swift framework, on the other hand, can be automatically downloaded, versioned, installed, and imported by the package manager, greatly streamlining its use.

This, and the previously discussed issues with existing system frameworks, motivates the creation of a pure Swift implementation of JPEG. A pure Swift JPEG library can vend a natural, idiomatic API. By default, pure Swift code compiles on *all* Swift platforms, and the lack of undefined/implementation-defined behavior in the language ensures consistent behavior across those platforms. First-class language support for concepts such as SIMD also make native-Swift codecs considerably more portable than their C counterparts, which are often compiled as a patchwork of macro-defined cores and extensions.

i.iii. prior art

Currently, no production-ready JPEG codec exists for the Swift language today.

Many language communities have "experimental" implementations of JPEG and other image formats. Most experimental implementations begin as personal projects, and many are non-compliant, or even not fully functional. However, they sometimes mature into formidable local competitors to *libjpeg* and other system libraries. Experimental JPEG implementations rarely meet the threshold to qualify as a usable framework, but the few that do serve as a proof-of-concept for the idea of commodotizing image processing into something that can be handled by a native-language package, as opposed to relying on system dependencies. While this can imply additional code-size costs, the portability and

usability gains inherent in "demoting" a system dependency into a regular package are significant.

Language communities with strong "hacker" traditions, such as the Rust community, often sport advanced native codec libraries in their package indices. In the Swift world, however, we could only locate a single, unfinished Github project which implements JPEG in native Swift, by Github user `sergeysmagleev`.

Why does Swift have such poor support for JPEG (and other image formats) compared to languages such as Rust which has a comparatively tiny user base? There are in fact, no technical limitations — performance or otherwise — inherent to the Swift language that would preclude a native Swift implementation of JPEG, or make such an implementation inferior to existing C implementations. The only real constraint is the fact that all open source code (in fact, all code) has to be authored by someone, and in the FOSS ecosystem especially, the limiting factor in producing new libraries and frameworks has been the availability and willingness of someone "up to the task" to write that code.

Without funding, interest and technical difficulty are the main determinants of whether a library will arise in a particular language community. This is true for any language community, including the Swift community. For example, because game development is a popular developer hobby, many algorithms and toolkits relevant to the field have been implemented natively in most languages.

In the field of image codecs, this has meant that "easier" formats such as GIF and, to a much lesser extent, PNG, often have high quality native-language implementations, while more technically challenging formats such as JPEG often remain unsupported. However, we forsee that as libraries and frameworks become increasingly decoupled from operating systems, the monopoly of `libjpeg` and proprietary system frameworks will too be broken, in favor of portable, native implementations. As such, developing such a resource contributes to the language community-level goal of expanding the Swift library ecosystem.

ii. project goals

> **Summary:** Swift *JPEG* supports all three popular JPEG coding processes (baseline, extended, and progressive), and comes with built-in support for the JFIF/EXIF subset of the JPEG standard. The framework supports decompressing images to RGB and YCbCr targets. Lower-level APIs allow users to perform lossless operations on the frequency-domain representation of an image, transcode images between different coding processes, edit header fields and tables, and insert or strip metadata. The framework also provides the flexibility for users to extend the JPEG standard to support custom color formats and additional coding processes.

ii.i. the jpeg standard

JPEG images as commonly encountered today are actually governed by three overlapping (and slightly contradictory) standards. The most important is the **ISO/IEC 10918-1** standard (also called the **ITU T.81** standard), which this document will refer to simply as the *JPEG standard*.

The JPEG standard is color format agnostic, meaning it supports any combination of user-defined color components (YCbCr, RGB, RGBA, and anything else). The standard defines no fewer than thirteen different *coding processes*, which are essentially distinct image formats grouped under the umbrella of "JPEG formats". Coding processes can be classified by their *entropy coding*:

```
enum Coding
{
    case huffman
    case arithmetic
}
```

Coding processes can also either be *hierarchical* or *non-hierarchical*. A summary of JPEG coding processes is given below:

	process type	entropy coding	hierarchical
1.	**baseline**	**huffman**	**false**
2.	**extended**	**huffman**	**false**
3.	extended	arithmetic	false
4.	extended	huffman	true
5.	extended	arithmetic	true
6.	**progressive**	**huffman**	**false**
7.	progressive	arithmetic	false
8.	progressive	huffman	true
9.	progressive	arithmetic	true
10.	lossless	huffman	false
11.	lossless	arithmetic	false
12.	lossless	huffman	true
13.	lossless	arithmetic	true

> *Note: processes this project supports are **bolded**.*

Among these formats, only the baseline huffman non-hierarchical process is commonly used today, though the progressive huffman non-hierarchical

process is sometimes also seen. This is in large part due to the other two technical standards relevant to the JPEG format, discussed shortly.

Until very recently, the arithmetic entropy coding method was patented, which resulted in its exclusion from software implementations of the standard. The lossless and hierarchical processes are seldom-used today, and are considered out of scope for this project. However, the extended (huffman, non-hierarchical) process is a relatively straightforward derivation from the baseline process, and sees some usage in applications such as medical imaging, so this project supports this process in addition to processes 1 and 6.

The framework is designed to still parse and recognize the unsupported coding processes, even if it is unable to encode or decode them. As such, it supports, for example, editing and resaving metadata for all conforming JPEG files regardless of the coding process used. In theory, users can use the lexing and parsing components of the framework to implement codec extensions implementing the unsupported processes.

ii.ii. color formats

A *color format* for a JPEG image is a set of *component identifiers* and a defined meaning for each of those components. A component identifier is an integer from 1 to 255, denoted $[c_i]$ in this document, and the identifiers need not be contiguous or in increasing order (or any order at all). An example of a (non-standard) color format for RGBA might be:

```
{
    [5]: red,
    [6]: green,
    [8]: blue,
    [1]: alpha
}
```

JPEG color formats are defined by the two other standards besides the ISO 10918-1, which we will refer to as the JFIF/EXIF standards. The JFIF/EXIF standards are subsets of the JPEG standard which define common color format meanings for JPEG images on the web (primarily JFIF) and from digital cameras (primarily EXIF). They “strongly recommend” use of the baseline coding process only, though they are compatible with the other coding processes as well. The JFIF and EXIF standards are mutually incompatible due to differences in file structure, but most codecs tolerate both.

Both the JFIF and EXIF standards use the YCbCr color model. The JFIF standard allows both full YCbCr triplets, and a Y-only grayscale form. The EXIF standard only allows YCbCr triplets. Both standards share the same identifier–channel mapping, and in addition, the JFIF YCbCr format is compatible with the Y format.

```
{
    [1]: Y  (luminance),
    [2]: Cb (blueness),
    [3]: Cr (redness)
}
```

The framework includes built-in support for the JFIF/EXIF color formats, which we will refer to as the *common format*. However it also provides support through Swift generics for custom user-defined color formats, which may be useful for certain applications.

ii.iii. color targets

Color targets are related to but distinct from color formats. A color format specifies how colors are represented and stored within a JPEG image, while a color target specifies how those colors are presented to users. This framework includes built-in support for both YCbCr and RGB as color targets. The conversion formula from JPEG-native YCbCr colors to RGB is defined by the JFIF/EXIF standards, and given (in matrix form) below:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.00000 & 0.00000 & 1.40200 \\ 1.00000 & -0.34414 & -0.71414 \\ 1.00000 & 1.77200 & 0.00000 \end{bmatrix} \times \begin{bmatrix} Y \\ Cb - 128 \\ Cr - 128 \end{bmatrix}$$

The inverse formula is given below:

$$\begin{bmatrix} Y \\ Cb - 128 \\ Cr - 128 \end{bmatrix} = \begin{bmatrix} 0.2990 & 0.5870 & 0.1140 \\ -0.1687 & -0.3313 & 0.5000 \\ 0.5000 & -0.4187 & -0.0813 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The framework supports rendering to multiple color targets from the same decoded image, without having to redecode the image for each target. As with custom color formats, the framework also supports user-defined color targets, which much also define an associated color format type since the JFIF/EXIF conversion formulas assume a specific YCbCr input format.

ii.iv. levels of abstraction

Rendering to (or saving from) an RGB/YCbCr pixel array is the most common JPEG codec use-case, but it is not the only one. As is well-known, the full JPEG encoding—decoding pipeline is lossy, which results in both image degradation and increased file size each time a JPEG is reencoded.

However, most of the steps in that pipeline are actually reversible, which means many common image operations (ranging from editing metadata to performing crops and rotations, and even color grading) can be done losslessly. Doing so requires a codec which exposes each abstracted stage of the coding pipeline in its API:

1. structural representation
2. spectral representation
3. dequantized representation
4. spatial representation
5. color representation

For example, metadata editing is best performed on the structural representation, while lossless crops, reflections, and rotations can only be performed on the spectral representation. Changing the compression level is performed on the dequantized representation, while changing the subsampling level is best performed on the spatial representation. As such, the framework allows users to interact with JPEG images at all five major levels of abstraction.

iii. concepts

> **Summary:** JPEG is a frequency transform-based compressed image format. Decompressing the file format can be roughly divided into lexing, parsing, and decoding stages. Decoding involves assembling multiple image *scans* into a single image *frame*. A scan may contain one or more color *components* (channels). In a progressive JPEG, a single scan may contain only a specific range of bits for a specific frequency band. JPEG images also use *huffman* and *quantization* tables. Huffman tables are associated with image components at the scan level. Quantization tables are associated with image components at the frame level. Multiple components can reference the same huffman or quantization table. The "compression level" of a JPEG image is almost fully determined by the quantization tables used by the image.

This section is meant to give a concise overview of the JPEG format itself. For the actual format details, consult the ISO 10918-1 standard.

iii.i. jpeg segmented structure

Structurally, JPEG files are sequences of *marker segments* and *entropy-coded segments*. It is possible to segment JPEG files without having to parse the body of each segment. Marker segments have headers, while entropy-coded segments are "naked" byte sequences. Because entropy-coded segments can have zero length, a JPEG file can be conceptualized as a sequence of alternating marker and entropy-coded segments. The terminator for an entropy-coded segment is one or more `0xFF` bytes; an entropy-coded segment together with its terminator is a *prefix*.

```
JPEG                ::= <Marker Segment> (<Prefix> <Marker Segment>) *
Prefix              ::= <Entropy-Coded Segment> (0xFF)+
```

Because the delimiter for an entropy-coded segment is an `0xFF` byte, this means that any `0xFF` bytes in its payload data must be escaped with the escape sequence `0xFF 0x00`.

```
Entropy-Coded Segment ::= <Escape> *
Escape                ::= [0x00-0xFE]
                        |  0xFF 0x00
```

Marker segments consist of a *type*, *length field*, and a *segment body*, in that order. The type is always one byte; the JPEG standard defines which values of this byte correspond to which marker segment types. The length field is a big-endian 16-bit integer. The length includes the length field itself, so the length of the segment body is always two less than the value of the length field. (Because the length of a marker segment is always known, no escaping takes place.)

```
Marker Segment      ::= <Type> <Length> <Body>
Type                ::= [0x01-0xFE]
Length              ::= [0x00-0xFF] [0x00-0xFF]
Body                ::= [0x00-0xFF]{ (Length[0] << 8 | Length[1]) - 2 }
```

There are many different types of marker segments, but the most important are *header segments* and *table segments*.

iii.ii. header segments

There are two types of JPEG header segments: *frame headers* and *scan headers*.

iii.ii.i. frame headers

A frame header is a header segment which describes a rectangular image as a whole. Except when the JPEG file uses a hierarchical coding process, there is only one frame, and therefore, one frame header per image. A frame header contains the following fields:

- Bit depth (integer, usually 8 or 12)
- Image width (integer, greater than zero)
- Image height (integer)
- Resident components (array)

Note that, as a technical detail, the height can be initialized to 0 by the frame header segment, and set later by a separate segment called a *height*

redefinition segment.

The resident components array defines the color components in the image, and includes image-global parameters for each component. A resident component definition contains the following fields:

- Component identifier (c_i)
- Quantization table reference (q_i)
- Horizontal sampling factor (integer, between 1 and 4)
- Vertical sampling factor (integer, between 1 and 4)

The sampling factors determine the chroma subsampling level of the image. All components having a sampling factor of (1, 1) corresponds to a 4:4:4 subsampling scheme. A sampling factor of (2, 2) for the Y channel, and (1, 1) for the Cb and Cr channels corresponds to a 4:2:0 subsampling scheme.

iii.ii.ii. scan headers

A scan header is a header segment which describes data, a *scan*, which makes up a portion of a complete image. There can be one or more scans, and therefore, scan headers, for a single frame. The decomposition of image data into multiple scans is always done spectrally, by bit-index, and by component, never spatially, so each scan contains data for the entire spatial extent of the image. A scan header is always immediately followed by an entropy-coded segment containing the scan data the header describes.

A scan header contains the following fields:

- Band range (integer range, between 0 and 63)
- Bit range (integer range)
- Component reference array

The *band range* is given in terms of discrete frequencies. The lowest frequency, 0, is the DC frequency, all other frequencies, up to a maximum of 63, are AC frequencies.

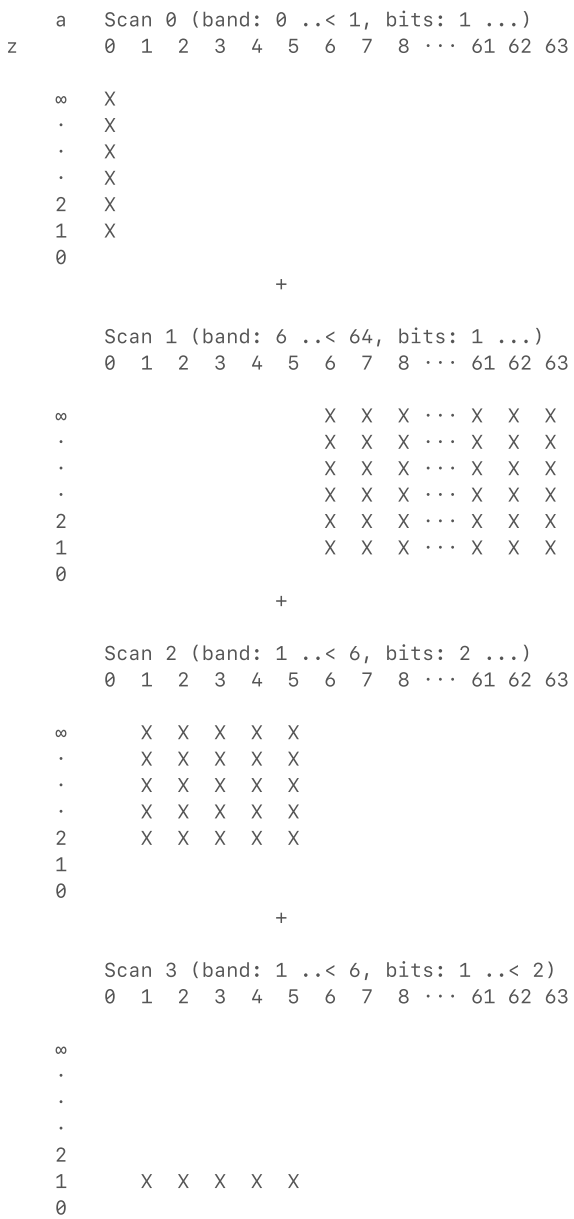
The *bit range* is given in terms of bit indices. The bit range refers to bits in the frequency-domain representation of the image, not its spatial-domain representation, so the bit range is not limited to the bit depth given in the frame header.

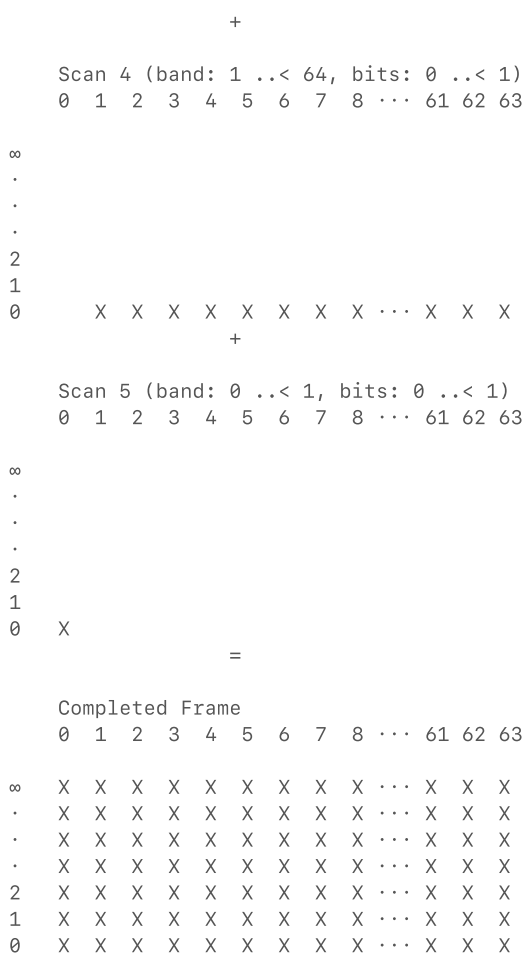
For non-progressive coding processes, the band range is always set to [0, 64). Likewise, the bit range is always set to [0, ∞).

For progressive coding processes, the band range can be anything within the interval [0, 64), as long as the range doesn't mix DC and AC frequencies. This means that [0, 1) and [1, 6) are both valid band ranges, but [0, 6) is not. Furthermore, when there are multiple scans for each component, the [0, 1) scan must come first. This decomposition is called *spectral selection*.

Progressively-coded images can also optionally use a decomposition called *successive approximation*, in which the first scan for each component (called an *initial scan*) has a bit range with an upper limit of infinity, and later scans (called *refining scans*) step down one bit at a time to zero. An example of a valid successive approximation sequence is { [3, ∞), [2, 3), [1, 2), [0, 1) }. The sequence { [3, ∞), [1, 3), [0, 1) } is invalid because the second scan contains a bit range with two bits, while the sequence { [3, ∞), [1, 2), [2, 3), [0, 1) } is invalid because bit 1 is refined before bit 2.

The sequence of scan-specified band ranges and bit ranges for a particular component is called a *scan progression*. The following is a visual example of a possible scan progression for one component of a progressively-coded image:





The component reference array specifies which of the components defined in the frame header is present within the scan. If there is more than one component in a scan, then the scan is *interleaved*, otherwise it is *non-interleaved*. Interleaving is not allowed for progressive scans which define AC coefficients only, though it is allowed for non-progressive scans which define all 64 frequencies, including the AC frequencies.

The ordering of component references within the array (if there are more than one) is meaningful, both because it must follow the ordering of component definitions in the frame header, and also because the ordering specifies the ordering of the interleaved data units in the entropy-coded segment following the scan header. A component reference contains the following fields:

- Component reference (c_i, matching one of the components in the frame header)
- DC huffman table reference
- AC huffman table reference

Note that quantization tables (described in the next section) are associated with components at the frame level, while huffman tables (also described in the next section) are associated with components at the scan

level. It is allowed (and standard practice) for the same component to use a different huffman table in each scan.

iii.iii. table segments

Table segments define resources which are referenced by the header segments. There are two types of table segments — *quantization table definitions*, and *huffman table definitions* — which define three types of resources.

iii.iii.i. quantization tables

A quantization table definition consists of 64 multiplier values, which correspond to the 64 discrete frequencies, and some basic information about the table:

- Quantization table identifier (q_i)
- Table precision (8- or 16-bit)

The table precision is not necessarily the same as the image bit depth (though it is subject to some constraints based on the image bit depth). This field is solely used to specify the (big-endian) integer type the table values are stored as.

Note that, as a technical detail, quantization tables do not actually identify themselves with a q_i identifier, nor do component definitions in a frame header use those identifiers to reference them. However, table identifiers are a useful conceptual model for understanding resource relationships within a JPEG file. This issue will be discussed further in the contextual state section.

iii.iii.ii. huffman tables

Huffman table definitions are somewhat more sophisticated than quantization tables. There are two types of huffman tables — AC and DC — but they are defined by the same type of marker segment, and share the same field format.

Like a quantization table definition, a huffman table definition includes some basic information about the table:

- Huffman table identifier
- Resource type (DC table or AC table)

A huffman table definition does not contain the table values verbatim. (That would be far too space-inefficient.) Rather, it specifies the shape of huffman *tree* used to generate the table, and the symbol values of the (up to 256) leaves in the tree. The algorithm for generating the huffman table from the huffman tree is discussed in more detail in the library architecture section.

Unlike quantization tables, huffman tables have no direct relation to frequency coefficient values themselves. They are only used to

decompress entropy-coded data within a single entropy-coded segment. (It is allowed, but uncommon, for multiple entropy-coded segments to use the same huffman table.) It is for this reason that huffman tables are "locally" associated with scans while quantization tables are "globally" associated with components at the frame level.

iii.iv. blocks, planes, and MCUs

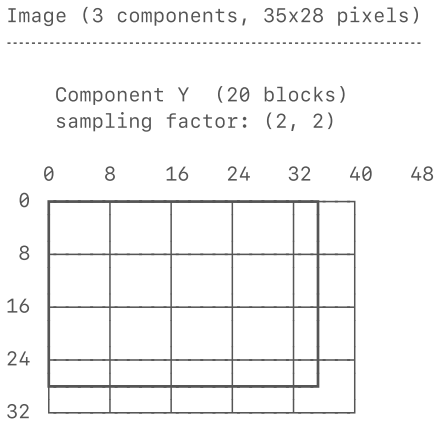
JPEG is a *planar format*, meaning each color channel is represented independently as a monochromatic sub-image. However, *interleaving* is still possible down to the granularity determined by the *minimum-coded unit* (MCU) of the image. (Within a single minimum-coded unit, the format is fully planar.) Minimum-coded units in turn are composed of constant-size *blocks*, sometimes called *data units*, which are the smallest spatial unit of a JPEG.

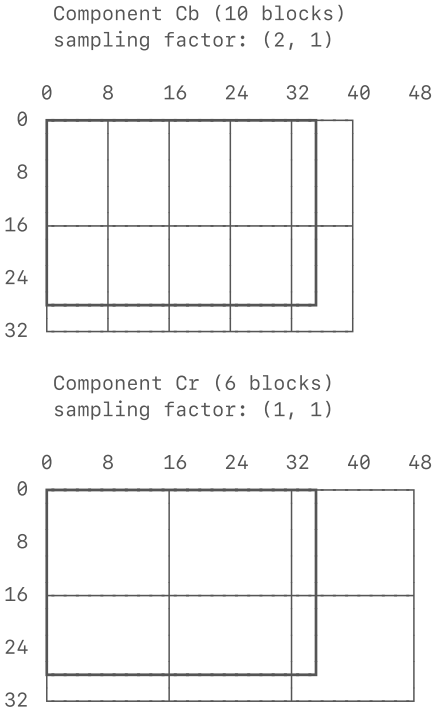
iii.iv.i. blocks

Each JPEG block contains 64 frequency coefficients which correspond to a block of pixels in the visual image. It is often stated that these are 8x8 pixel blocks, but the size actually depends on the component sampling factor. (Subsampled blocks are linearly interpolated to fill in intermediate pixels; the frequency transform is not evaluated per-pixel.)

All blocks for a particular component are the same size, even if the image pixel width and height would indicate fractional blocks along the right and bottom edges of the image. In these cases, the image data is padded (when encoding) to fill an integer number of blocks, and this padding is discarded when decoding. If different components use different sampling factors, the block grid for one component may cover areas that the block grid for another component does not.

The following diagram shows the block decomposition of a 35x28 pixel image using sampling factors (2, 2), (2, 1), and (1, 1). Note that all three block grids cover pixels that are outside the 35x28 pixel bounds (bolded rectangle), and furthermore, the last block grid covers pixels that the other two grids do not:



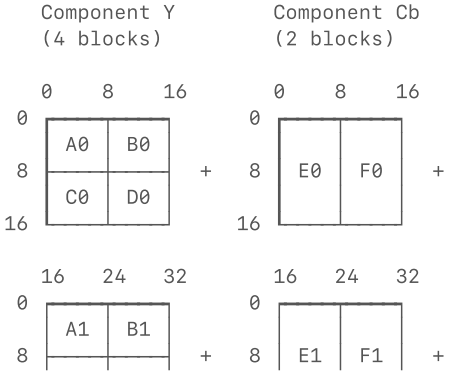


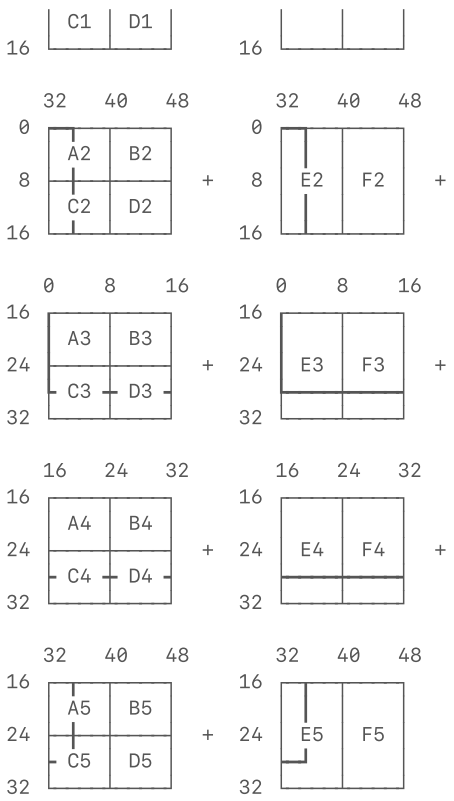
It is important to remember that, even though less densely-sampled blocks are spatially bigger, all blocks contain the same amount of information.

iii.iv.ii. minimum-coded units

If (and only if) a JPEG scan encodes more than one component, then the blocks are organized into minimum-coded units. (Single-component scans do not use the concept of a minimum-coded unit, and simply store their blocks as a row-major rectangular array.)

The spatial size of the minimum coded unit is the size of a block with a component sampling factor of (1, 1), even if the scan contains no such component. The blocks are stored within the minimum-coded unit in the same order they were declared in the scan header. For example, the minimum-coded units from a scan containing the Y and Cb components from the previous example would look like this:






```
   Sequential order:
[
    A0, B0, C0, D0,     E0, F0,
    A1, B1, C1, D1,     E1, F1,
    A2, B2, C2, D2,     E2, F2,
    A3, B3, C3, D3,     E3, F3,
    A4, B4, C4, D4,     E4, F4,
    A5, B5, C5, D5,     E5, F5
]
```

Note that blocks B2, D2, F2, B5, D5, and F5 have been added to complete the minimum-coded units they appear in. They would not appear in a non-interleaved scan.

iii.iv.iii. planes

Planes are a very simple concept — they are simply the collection of all the blocks for a particular component. Even though blocks may be stored in an interleaved arrangement, planes are conceptually independent. Even when interleaved, each plane uses its own huffman and quantization tables, which means that a single entropy-coded segment can actually contain codewords from multiple huffman coding schemes.

Converting planes into a rectangular array of color pixels entails expanding subsampled planes, and then clipping them to the pixel dimensions of the image so that each plane has the same spatial width and height. The planes are then pixel-wise interleaved to form color tuples.

iii.v. contextual state

All of the aforementioned concepts are related by the *contextual state* of a JPEG file. The state is determined by the ordering of marker and entropy-coded segments in the file.

iii.v.i. sections

All JPEG files must start with a *preamble section*, which begins with an *start-of-image* marker segment, followed by JFIF/EXIF metadata segments, and then any number of table segments. While huffman table definitions can live in the preamble, usually it is only quantization table definitions that appear here, since quantization tables are the only JPEG resources that have a whole-frame scope.

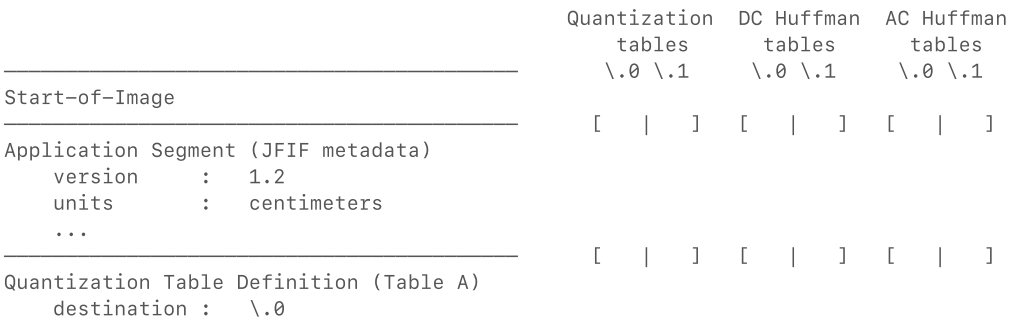
In a non-hierarchical JPEG file, the *body section* comes after the preamble. The body starts with a frame header segment, and then contains any number of scan header + entropy-coded segment pairs and table definitions. It is rare for quantization table definitions to appear in the middle of this section, so most of these table definitions are huffman table definitions. The body section, and the JPEG file as a whole, concludes with an *end-of-image* marker.

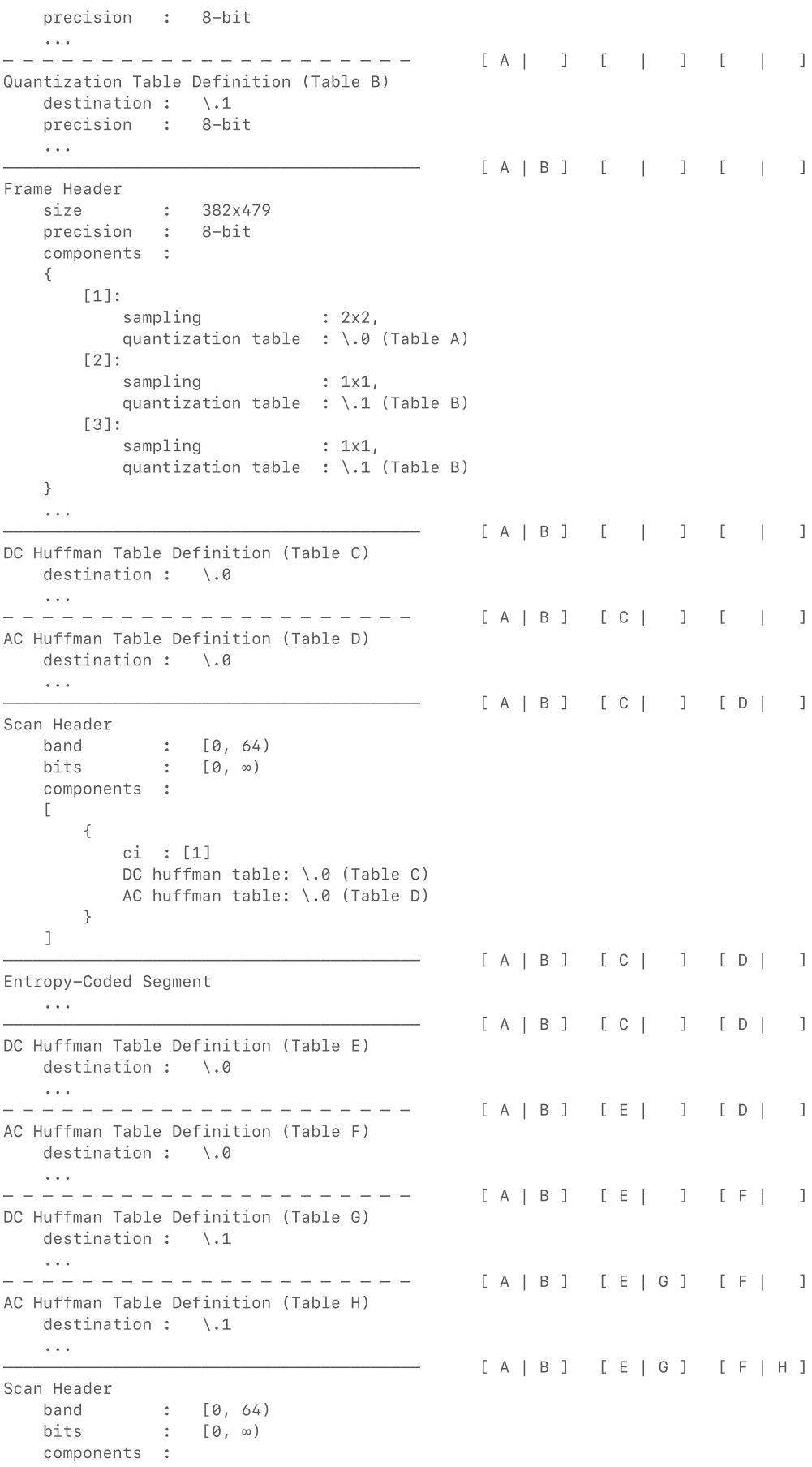
iii.v.ii. table slots

The JPEG format establishes relationships between table resources and reference holders using the concept of *table slots*. Each type of table (there are three: quantization, DC huffman, and AC huffman) has a fixed number of binding points: 2 for the baseline coding process, and 4 for all other processes. In this document, we use the Swift keypath syntax `\.i` to denote a binding point *i*.

Whenever a table definition appears, it specifies a *table destination*, which is the binding point to which the table is attached. Whenever a consumer (such as a component definition in a frame header, which references a quantization table, or a component reference in a scan header, which includes references to a DC and/or AC huffman table) references a resource, it does so by specifying a binding point, which resolves to whatever table is attached to it at the time. Table bindings are stateful, so the same slot can be overwritten multiple times within the same JPEG file.

The following is an example structure of a (sequential) JPEG from start to finish, with the state of the table slots given on the right:




precision : 8-bit
...
[A |] [|] [|]
Quantization Table Definition (Table B)
destination : \.1
precision : 8-bit
...
[A | B] [|] [|]
Frame Header
size : 382x479
precision : 8-bit
components :
{
[1]:
sampling : 2x2,
quantization table : \.0 (Table A)
[2]:
sampling : 1x1,
quantization table : \.1 (Table B)
[3]:
sampling : 1x1,
quantization table : \.1 (Table B)
}
...
[A | B] [|] [|]
DC Huffman Table Definition (Table C)
destination : \.0
...
[A | B] [C |] [|]
AC Huffman Table Definition (Table D)
destination : \.0
...
[A | B] [C |] [D |]
Scan Header
band : [0, 64)
bits : [0, ∞)
components :
[
{
ci : [1]
DC huffman table: \.0 (Table C)
AC huffman table: \.0 (Table D)
}
]
[A | B] [C |] [D |]
Entropy-Coded Segment
...
[A | B] [C |] [D |]
DC Huffman Table Definition (Table E)
destination : \.0
...
[A | B] [E |] [D |]
AC Huffman Table Definition (Table F)
destination : \.0
...
[A | B] [E |] [F |]
DC Huffman Table Definition (Table G)
destination : \.1
...
[A | B] [E | G] [F |]
AC Huffman Table Definition (Table H)
destination : \.1
...
[A | B] [E | G] [F | H]
Scan Header
band : [0, 64)
bits : [0, ∞)
components :

```
    [
        {
            ci  : [2]
            DC huffman table: \.0 (Table E)
            AC huffman table: \.0 (Table F)
        },
        {
            ci  : [3]
            DC huffman table: \.1 (Table G)
            AC huffman table: \.1 (Table H)
        }
    ]
_____________________________________________      [ A | B ]  [ E | G ]  [ F | H ]
Entropy-Coded Segment
    ...
_____________________________________________      [ A | B ]  [ E | G ]  [ F | H ]
End-of-Image
_____________________________________________
```

Note how tables C and D were overwritten partway through the JPEG file.

iv. user model

> **Summary:** The Swift *JPEG* encoder provides unique abstract *component key* and *quantization table key* identifiers. The component keys are equivalent in value to the component idenfiers (c_i) in the JPEG standard, while the quantization table identifiers (q_i) are a library concept, which obviate the need for users to assign and refer to quantization tables by their slot index, as slots may be overwritten and reused within the same JPEG file. Users also specify the *scan progression* by band range, bit range, and component key set. These relationships are combined into a *layout*, a library concept encapsulating relationships between table indices, component indices, scan component references, etc. When initializing a layout, the framework is responsible for mapping the abstract, user-specified relationships into a sequence of JPEG scan headers and table definitions.

> JPEG layout structures also contain a mapping from abstract component and quantization table keys to linear integer indices which point to the actual storage for the respective resources. (The framework notations for these indices are *c* and *q*, respectively.) The linear indices provide fast access to JPEG resources, as using them does not involve resolving hashtable lookups.

> Layout structures are combined with actual quantization table values to construct *image data* structures. All image data structures (except the `Rectangular` type) are planar, and are conceptually `Collection`s of planes corresponding to a single color component. The ordering of the planes is determined by the *image format*, which is generic and can be replaced with a user-defined implementation. The framework vends a default "common format" which corresponds to the 8-bit Y and YCbCr color modes defined by the JFIF standard.

> Plane indices range from 0 to p_{max}, where p_{max} is the number of planes in the image. The library assigns linear component indices such that $c = p$.

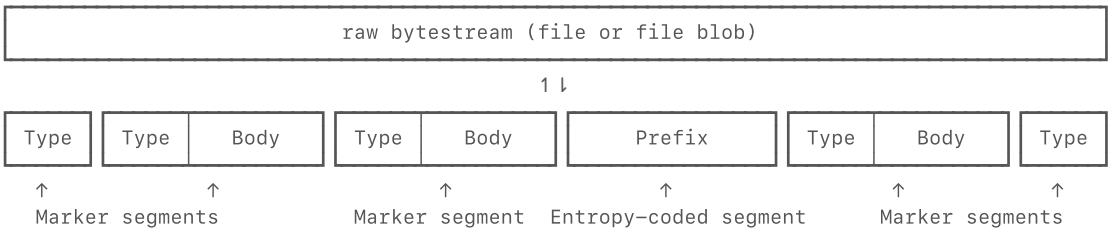
The JPEG format, as previously discussed, contains a great deal of complexity meant to facilitate implementation. However, much of this complexity is unnecessary for users, which is why this framework attempts to abstract away most of the user-irrelevant aspects of the format.

This framework provides two sets of top-level APIs: a *segmentation API* and an *decoding/encoding API*. Both are top-level in that they are capable of interpreting or outputting a JPEG file from start to finish. The segmentation API is essentially a lexer/formatter in that it detects JPEG segment boundaries, and classifies them by type. It does not attempt to interpret the contents of the segments. The decoding/encoding API reads or writes a JPEG file as a whole; its output/input is a complete bitmap image. This API is essentially built atop of the segmentation API.

While the segmentation and decoding/encoding APIs roughly correspond to the lexing/formatting and decoding/encoding stages of JPEG interpretation, there is no such top-level API for the parsing/serializing stage. This is because each lexed or formatted JPEG segment requires a different parser or serializer implementation, and which implementation it requires depends on the type of the segment. As such, a "top-level" parsing/serializing API would not be a useful abstraction, and so this framework does not seek to provide one.

iv.i. segmentation API

As mentioned already, the segmentation API takes a file input (or byte stream), and divides it into its constituent segments. Its inverse API takes raw segment buffers and concatenates them with appropriate segment headers into an output bytestream.



Its operations can be best summarized by the pseudoswift below:

```
var input:Source
while true
{
```

```
    let (prefix, type, body):([UInt8], JPEG.Marker, [UInt8]) = input.segment()
    switch type
    {
        ...
    }
}

var output:Destination
let pairs:[([UInt8], JPEG.Marker, [UInt8])]
for (prefix, type, body):([UInt8], JPEG.Marker, [UInt8]) in pairs
{
    output.format(prefix: prefix)
    output.format(marker: type, tail: body)
    ...
}
```

Note that this is *not* how the segmentation API is actually spelled, as the real API expects the user to know whether to expect an entropy-coded segment to be present, as well as to be aware of error handling.

iv.ii. decoding/encoding API

While the segmentation API only goes so far as to lex or format a JPEG file, the decoding/encoding API does the heavy lifting of actually converting a JPEG to and from its bitmap data. This this is the most common use-case for JPEG, this set of APIs is likely to be the one most commonly used by users.

Internally, this set of APIs handles JPEG state management, abstracting away the confusing system of table slots, plane indices, and binding points, and replacing it with resource identifiers (c_i's and q_i's) which are unique over the lifetime of the JPEG. The purpose of this abstraction is not only to present a simpler mental model for users, but also to make it harder for users to accidentally create an invalid JPEG file (for example, switching out a quantization table while its corresponding component is still being encoded.)

iv.ii.i. keys, indices, and binding points

To users, this framework replaces the concept of resource binding points with *keys* and *indices*. (These terms are used in accordance to Swift convention.) The framework also uses the system of keys and indices to identify components, and by extension, image planes.

Keys are unique identifiers for either a color component or a quantization table. The identifiers $[q_i]$ and $[c_i]$ are keys in this context, and we use the same notation to refer to them. Keys are essentially integer identifiers, and in the case of component keys, they have the same wrapped value as the component identifiers assigned in the image frame header. (Quantization table keys are a framework concept, they do not appear in the JPEG standard itself.) However, the framework uses Swift's strong type system to distinguish them from actual indices to prevent user mixups.

Indices, as the name suggests (according to Swift convention) are shortcuts used for efficient dereferencing of entities that would otherwise have to go through expensive hashtable lookups. Because the storage type is always some kind of `Array`, all indices have the type `Int`. The library API is written to discourage direct use of keys as accessors, rather, it nudges users towards looking up an index from a key once, and then using the index for all subsequent accesses.

The framework uses the notation *c* and *q* for indices, corresponding to the $[q_i]$ and $[c_i]$ notation for keys.

By library convention, the quanta key –1 is assigned to the "default" (all zeroes) quantization table when decoding a JPEG file. This key has index 0, so all file-defined quantization tables have indices counting up from 1. Quanta keys are assigned by the user when encoding a JPEG file.

Component keys are completely data-defined. Component indices start from 0, and are determined by the order that the component identifiers appear in the color format. For the JFIF/EXIF common format, the key-to-index mapping is:

```
{
    [1]: 0,
    [2]: 1,
    [3]: 2
}
```

Component indices are the same as plane indices, which use the notation *p* in the framework. In the above common format, component [1] would be plane $p = 0$, component [2] would be plane $p = 1$, and component [3] would be plane $p = 2$.

While the builtin common format does not do this, custom color formats are allowed to support more *resident components* (components that a frame header can define without causing the library to emit a validation error) than *recognized components* (components that the decoder maintains pixel storage for and includes in its output). In this case, only the recognized components have corresponding planes. An example of a use-case for this kind of component subsetting is a custom RGB color format, which supports an optional alpha channel. In this case, custom RGBA JPEG images can be made compatible with another custom RGB color format using component subsetting.

When a color format defines optional resident components, the recognized components get assigned contiguous indices starting from 0, and the optional components come after them.

The encoder does not allow optional resident components, since it would not make sense to encode an image component for which no plane data has been provided.

iv.ii.ii. layouts and definitions

An image *layout* specifies all the parametric characteristics of the image save for the actual pixel values. It contains:

- The image color format
- The image coding process
- The set of resident components
- The list of recognized components (which is always a subset of the residents)
- The parameters for image planes (an array)
- The sequence of definitions in the image (also an array)

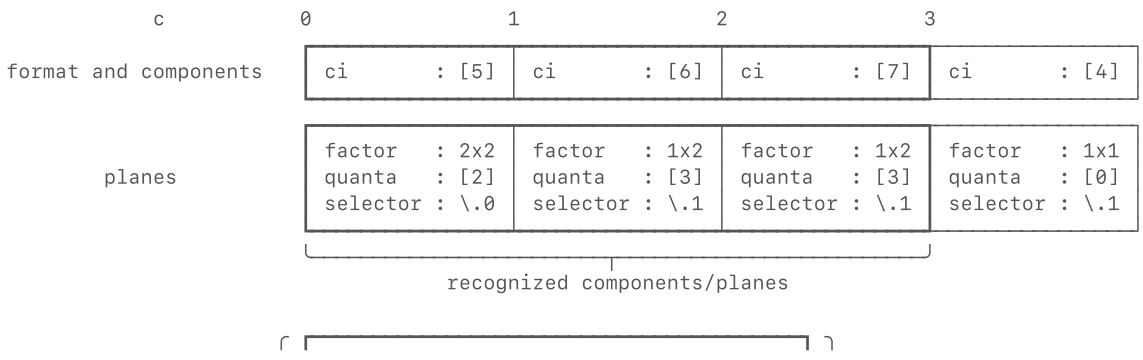
Each plane in the image has its own layout parameters. (The framework, of course, follows the same component/plane indexing scheme for this array.) A plane layout contains:

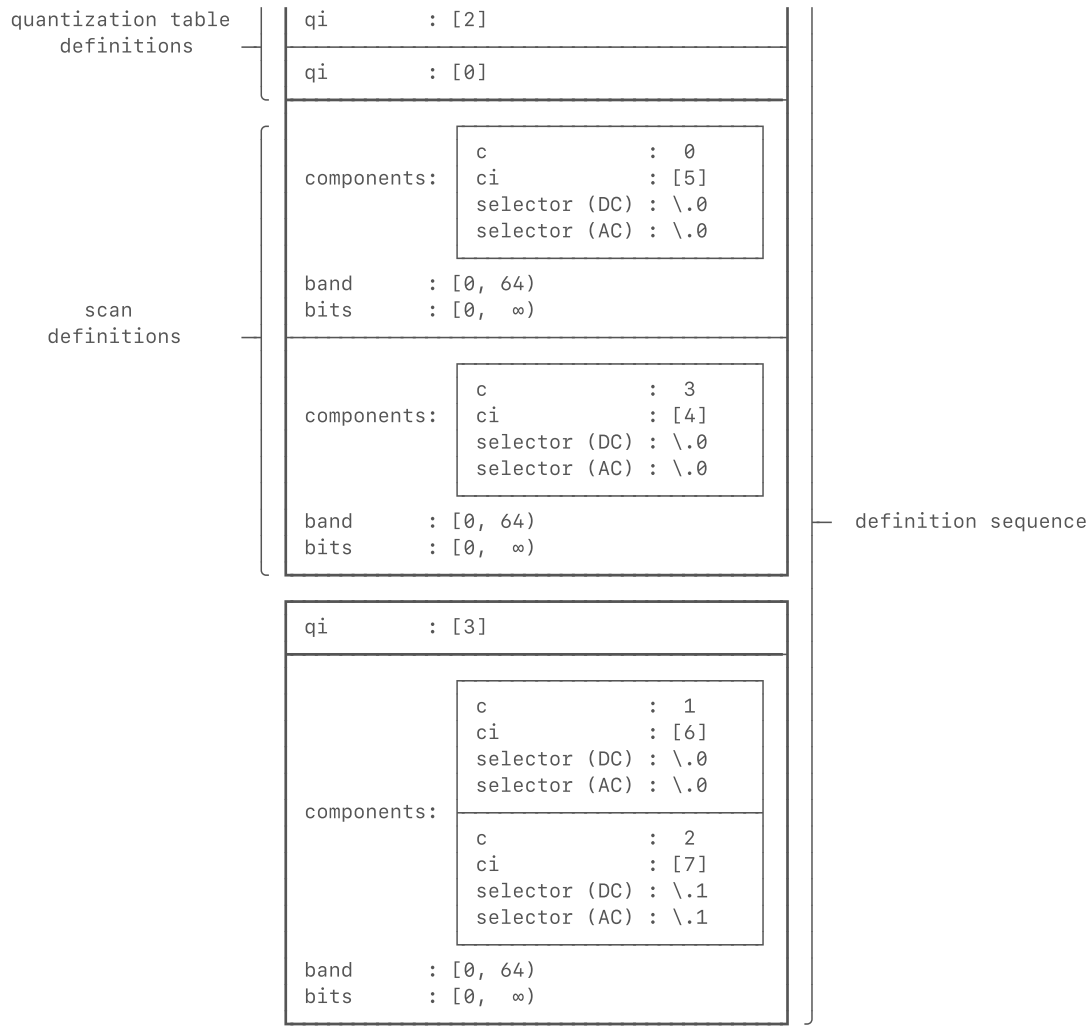
- The component sampling factor
- The component quanta key ($[q_i]$)
- The component quantization table binding point

The quanta key and the table binding point are always related. When a user initializes a layout, the binding points are assigned by the library. (In some cases, it is impossible to assign a large number of overlapping quanta keys to a limited number of binding points, in which case the library throws an error.) When a layout gets read from a JPEG file, the quanta keys get assigned by the library, as discussed in the last section.

The definition sequence is a list of alternating runs of quantization table definitions and scan definitions. The quantization table definitions say nothing about the actual contents of the tables, they only specify that the quantization table for a particular quanta key $[q_i]$ should appear in that position in the sequence.

The following is a block diagram of a layout for an image with a custom color format with four components:





Note that in interleaved scans, the scan components are always in ascending *key* order (not index order).

iv.ii.iii. data representations

The products of the decoder (and the inputs of the encoder) are *data representations*, structures which represent an image as a whole. All data representations contain an image layout. In fact, all data representations contain a common descriptor "core" consisting of:

- Image pixel dimensions
- Image layout
- Metadata (array)

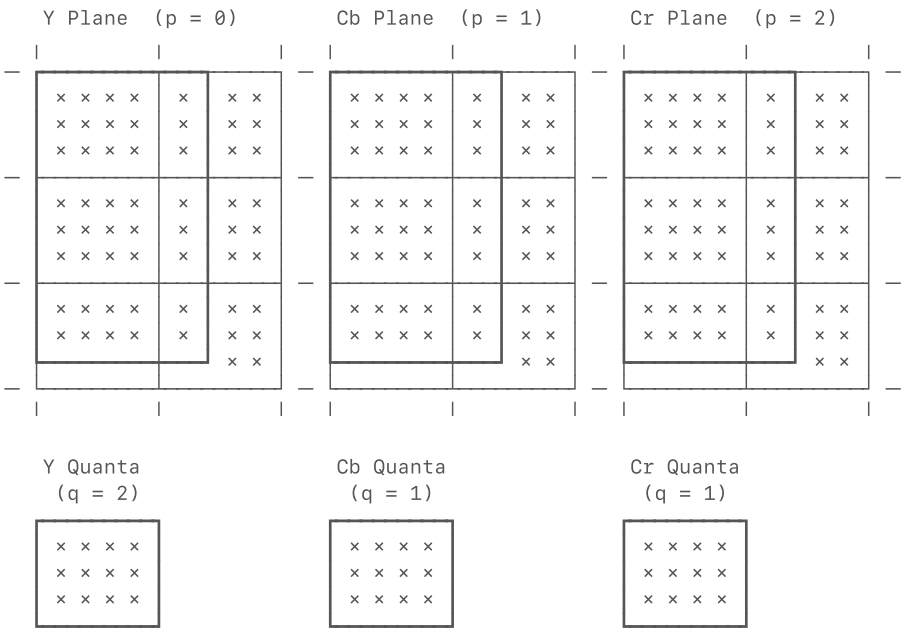
There are three types of data representations, which can be thought of as images in different stages of processing.

1. Spectral data
2. Planar data
3. Rectangular data

Spectral data is the most important type, as it is the native representation of a JPEG image. Spectral data can be decoded and reencoded without information loss (though the bitwise spelling may be slightly different). As the name suggests, it stores an image in its frequency-domain representation, grouped into 8x8-sample blocks. Spectral data is stored in a planar format, so the concept of the minimum-coded unit is not part of its organization. However, this data structure does store the image *scale*, which specifies the number of 8x8-pixel (not sample!) blocks that constitute a minimum-coded unit.

A spectral data structure also stores the quantization table values separately from the quantized frequency coefficients. The dequantized coefficients are obtained by multiplying each stored coefficient with its corresponding quantum.

The addressing scheme for spectral data is first by 2D block index (x, y), and then by zigzag coefficient index z. The z index is always between 0 and 63.

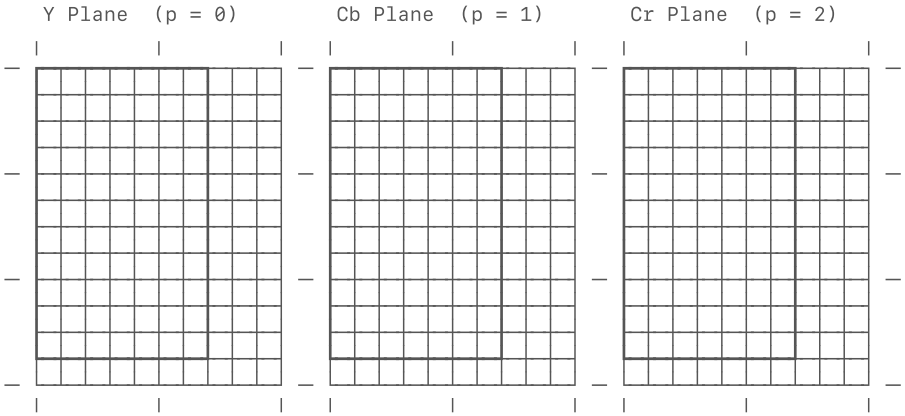


Note how the same quantization table (q = 1) is shared by the Cb and Cr components.

Spectral data is converted into *planar data* through the *inverse discrete cosine transform* (IDCT). This transform converts each spectral block into its spatial-domain representation. Planes in planar data have the same size as their spectral counterparts, but they are indexed by sample (not pixel!) rather than by block and then coefficient index. For example, the coordinate region (x, y, z) = (1, 2, z), $0 \le z < 64$ in the spectral representation is equivalent to the coordinate region ($8 + \Delta x$, $16 + \Delta y$), $0 \le (\Delta x, \Delta y) < 8$ in

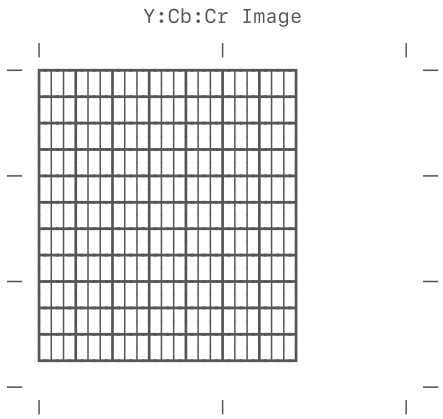
the planar representation. (The offsets Δx and Δy are *not* related to z, since the inverse discrete cosine transform is applied to entire blocks at a time.)

Even though planar data is indexed by sample, and not by block, each plane still contains whole blocks of data. It follows that the sample dimensions are always a multiple of 8.



Finally, planar data can be *interleaved* for obtain *rectangular data*. If the components of the image use different sampling factors, then the subsampled planed are interpolated to fill in missing samples. When going from planar to rectangular representation, plane indices turn into intra-pixel offsets. (However, users should rarely have to access color channels by offset directly; this is a job for the pixel accessor API.)

Because rectangular data is fully interleaved, padding samples are discarded when converting to this representation.



---

iv.ii.iv. pixel accessors

Rectangular data provide their pixel contents through the *pixel accessor* API. The output of this API is an array of pixels in the familiar YCbCr or RGBA (or another color type) form.

Why does the framework not simply store pixel values in the rectangular data itself? This is because JPEG is natively a YCbCr-based image format, therefore, converting to a form such as RGB would cause additional data loss, and make the YCbCr image inaccessible without having to redecode the image. In addition, because the conversion to the rectangular representation often involves additional processing (such as the upsampling operation), it is highly motivating to be able to do this step once, and be able to read that image as multiple color targets without having to recompute the rectangular representation.

v. library architecture

> **Summary:** The library is broadly divided into a decompressor and a compressor. The decompressor is further subdivided into a lexer, parser, and decoder, while the compressor is divided into an encoder, serializer, and formatter. Accordingly, the framework distinguishes between parseme types, returned by the parser and taken by the serializer, and model types, used by the decoder and encoder. For example, the parser returns a scan *header*, which is then "frozen" into a scan *structure*.

> The framework is architected for extensibility. For example, although the decoder and encoder do not support JPEG processes beyond the baseline, extended, and progressive processes, all JPEG processes, including hierarchical and arithmetic processes are recognized by the parser. Similarly, the lexer recognizes JPEG marker types that the parser does not necessarily know how to parse.

This section is meant to be an introductory guide to the organization of the code base, and an overview of the various type relationships the framework establishes. It also overviews the generic customization points the library offers users.

Broadly, the framework is divided into a decompressor and a compressor. As an arbitrary design decision, the decompressor is positioned as somewhat more fundamental than the compression, so many type definitions are associated with the decompressor, with the compressor in some ways written atop of the decompressor. The decompressor can be further divided into a lexer, parser, and decoder; the corresponding components of the compressor are the formatter, serializer, and encoder. The lexer, parser, formatter, and serializer generally work with *parseme types*, which get narrowed and further validated into *model types* used by the encoder and decoder.

At the time of writing, contributors will find five files containing most of the core library code:

- `jpeg/common.swift`
- `jpeg/decode.swift`
- `jpeg/encode.swift`
- `jpeg/debug.swift`
- `jpeg/os.swift`

v.i. `common.swift`

This file contains data structures and language extensions which are used by the framework, but not conceptually related to JPEG. It defines the top-level namespace `Common`, with the following type members:

- `Common`
  - `struct Common.MutableStorage<I>`
  - `struct Common.Storage<I>`
  - `struct Common.Storage2<I>`
  - `struct Common.Heap<Key, Value>`
  - `struct Common.Range2<Bound>`
  - `struct Common.Range2Iterator<Bound>`

It also extends the standard library `Array<UInt8>` and `ArraySlice<UInt8>` types to support the following methods:

- `extension Swift.ArraySlice<UInt8>`
  - `Swift.ArraySlice<UInt8>.load<T, U>(_:)(bigEndian:as:)`
- `extension Swift.Array<UInt8>`
  - `Swift.Array<UInt8>.load<T, U>(_:)(bigEndian:as:at:)`
  - `Swift.Array<UInt8>.store<T, U>(_:asBigEndian:)`

v.i.i. storage types

- `struct Common.MutableStorage<I>`
- `struct Common.Storage<I>`
- `struct Common.Storage2<I>`

These types are Swift property wrappers used to store `Int` values with fewer bits than a normal 64-bit integer. (This is useful because unlike in C/C++, `Int` is the *only* canonical integer type, so a wrapper which provides a way of accessing shorter integer types as a plain `Int` is highly valuable.) The only reason it is currently necessary to have these property wrappers is that current bugs in the compiler (as of version 5.2) place a hard 32 byte size limit on element types that are used with `read` / `modify` subscripts.

The `Storage<I>` type implements a read-only version of `MutableStorage<I>`, while the `Storage2<I>` type implements the same concept for a `(x:Int, y:Int)` tuple, since property wrappers cannot be directly applied to tuple elements.

v.i.ii. heap type

- `struct Common.Heap<Key, Value>`

This type implements a standard heap (priority queue). This heap is a min-heap (which sorts by `Key` type). It is used to assign codewords to symbols when constructing huffman trees.

v.i.iii. 2D range types

- `struct Common.Range2<Bound>`
- `struct Common.Range2Iterator<Bound>`

These types provide support for 2-dimensional index loops. Within the library, they look like this:

```
for (x, y):(Int, Int) in (0, 0) ..< (a, b)
{
    ...
}
```

Which is equivalent to this:

```
for y:Int in 0 ..< b
{
    for x:Int in 0 ..< a
    {
        ...
    }
}
```

To avoid cluttering user scopes, the `..<` operator is non-public, however, 2-dimensional range iterators can still be used through the various `.indices` properties on many framework types.

v.ii. `decode.swift`

The majority of the library code lives in this file. It includes both implementations for the decompressor, and data types common to both the decoder and the encoder. It defines the top-level namespace `JPEG`, with the following type members:

- `JPEG` (color format protocols and color targets)
    - `protocol` `JPEG.Format`
    - `protocol` `JPEG.Color`
        - `associatedtype` `Format`
    - `struct` `JPEG.YCbCr`
    - `struct` `JPEG.RGB`

- `JPEG` (model types)
  - `enum` `JPEG.Metadata`
  - `struct` `JPEG.Component`
    - `struct` `JPEG.Component.Key`
  - `struct` `JPEG.Scan`
    - `struct` `JPEG.Scan.Component`
  - `struct` `JPEG.Layout<Format>`
- `JPEG` (compound types)
  - `enum` `JPEG.Process`
    - `enum` `JPEG.Process.Coding`
  - `enum` `JPEG.Marker`
- `JPEG` (decompression error types)
  - `protocol` `JPEG.Error`
  - `enum` `JPEG.LexingError`
  - `enum` `JPEG.ParsingError`
  - `enum` `JPEG.DecodingError`
- `JPEG` (stream types, and lexer)
  - `protocol` `JPEG.Bytestream.Source`
  - `struct` `JPEG.Bitstream`
- `JPEG` (parseme types, and parser)
  - `protocol` `JPEG.Bitstream.AnySymbol`
  - `enum` `JPEG.Bitstream.Symbol.DC`
  - `enum` `JPEG.Bitstream.Symbol.AC`
  - `struct` `JPEG.JFIF`
    - `enum` `JPEG.JFIF.Version`
    - `enum` `JPEG.JFIF.Unit`
  - `protocol` `JPEG.AnyTable`
    - `associatedtype` `Delegate`
  - `struct` `JPEG.Table.Huffman<Symbol>`
  - `struct` `JPEG.Table.Quantization`
    - `struct` `JPEG.Table.Quantization.Key`
    - `struct` `JPEG.Table.Quantization.Precision`
  - `struct` `JPEG.Header.HeightRedefinition`
  - `struct` `JPEG.Header.Frame`
  - `struct` `JPEG.Header.Scan`
- `JPEG` (huffman decoder)
  - `struct` `JPEG.Table.Huffman<Symbol>.Decoder`
- `JPEG.Data` (data representations)

    - `struct` `JPEG.Data.Spectral<Format>`
        - `struct` `JPEG.Data.Spectral<Format>.Plane`
        - `struct` `JPEG.Data.Spectral<Format>.Quanta`
    - `struct` `JPEG.Data.Planar<Format>`
        - `struct` `JPEG.Data.Planar<Format>.Plane`
    - `struct` `JPEG.Data.Rectangular<Format>`

- `JPEG` (decoder types and decoder)

    - `extension` `JPEG.Bitstream`
    - `extension` `JPEG.Data.Spectral`
    - `extension` `JPEG.Data.Spectral.Plane`
    - `struct` `JPEG.Context<Format>`

- `JPEG.Data` (staged APIs)

    - `extension` `JPEG.Data.Spectral`
        - `extension` `JPEG.Data.Spectral.Plane`
            - `typealias` `JPEG.Data.Spectral<Format>.Plane.Block8x8<T>`
    - `extension` `JPEG.Data.Planar`
        - `extension` `JPEG.Data.Planar.Plane`
    - `extension` `JPEG.Data.Rectangular`

- `JPEG` (built-in color formats and color target conformances)

    - `enum` `JPEG.Common`

v.ii.i. color format protocols and color targets

- `protocol` `JPEG.Format`
- `protocol` `JPEG.Color`
    - `associatedtype` `Format`
- `struct` `JPEG.YCbCr`
- `struct` `JPEG.RGB`

As the names suggest, the protocols `JPEG.Format` and `JPEG.Color` define the requirements for a user-defined color format and color target, respectively:

```
protocol JPEG.Format
{
    static
    func recognize(_ components:Set<JPEG.Component.Key>, precision:Int) -> Self?

    var components:[JPEG.Component.Key]
    {
        get
    }
    var precision:Int
    {
        get
    }
```

```
}

protocol JPEG.Color
{
    associatedtype Format:JPEG.Format

    static
    func pixels(_ interleaved:[UInt16], format:Format) -> [Self]
}
```

All color targets must have a specific associated format type. At first glance, this seems restrictive, since there can only be one color format that can produce each color target, but in practice, any meaningfully distinct color format would have to define its own set of target color types anyway.

This section of the code also declares two built-in 8-bit color targets, `JPEG.YCbCr` and `JPEG.RGB`, but implements no conformances.

v.ii.ii. model types

- `enum` `JPEG.Metadata`
- `struct` `JPEG.Component`
  - `struct` `JPEG.Component.Key`
- `struct` `JPEG.Scan`
  - `struct` `JPEG.Scan.Component`
- `struct` `JPEG.Layout<Format>`

These types are the model types produced by cross-validating the framework's parseme types. In general, they store pre-resolved resource indices. Most of them have already been discussed in the user model section.

The `JPEG.Metadata` enumeration stores typed and untyped metadata records. At present, JFIF segments are the only kind of metadata stored as parsed metadata. All other application segments are stored as untyped, raw byte buffers

v.ii.iii. compound types

- `enum` `JPEG.Process`
  - `enum` `JPEG.Process.Coding`
- `enum` `JPEG.Marker`

These types are effectively lexeme types, though they also have relevance in deeper levels of the library. As the names suggest, `JPEG.Process` cases represent coding processes, while `JPEG.Marker` cases represent marker segment types.

v.ii.iv. decompression error types

- `protocol` `JPEG.Error`
- `enum` `JPEG.LexingError`

- `enum` `JPEG.ParsingError`
- `enum` `JPEG.DecodingError`

These types form the basis of the framework's error handling system. The `JPEG.Error` protocol refines Swift's normal `Swift.Error` errors, to add namespace, message, and detailed-message properties. This allows errors to be printed to the terminal with a common formatting.

```
protocol JPEG.Error:Swift.Error
{
    static
    var namespace:String
    {
        get
    }
    var message:String
    {
        get
    }
    var details:String?
    {
        get
    }
}
```

---

v.ii.v. stream types and lexer implementation

- `protocol` `JPEG.Bytestream.Source`
- `struct` `JPEG.Bitstream`

These types define the data inputs to the decoder. The `JPEG.Bytestream.Source` protocol abstracts a data source, which could be a file handle, in-memory data blob, or anything else. The `JPEG.Bitstream` type provides bit-level access to binary-coded data. Note that the bitstreams, unlike the bytestreams, are random-access.

```
protocol JPEG.Bytestream.Source
{
    mutating
    func read(count:Int) -> [UInt8]?
}
```

The lexer is implemented atop of the `JPEG.Bytestream.Source` protocol as an extension.

v.ii.vi. parseme types and parser implementation

- `protocol` `JPEG.Bitstream.AnySymbol`
- `enum` `JPEG.Bitstream.Symbol.DC`
- `enum` `JPEG.Bitstream.Symbol.AC`
- `struct` `JPEG.JFIF`

    - `enum` `JPEG.JFIF.Version`
    - `enum` `JPEG.JFIF.Unit`
- `protocol` `JPEG.AnyTable`
    - `associatedtype` `Delegate`
- `struct` `JPEG.Table.Huffman<Symbol>`
- `struct` `JPEG.Table.Quantization`
    - `struct` `JPEG.Table.Quantization.Key`
    - `struct` `JPEG.Table.Quantization.Precision`
- `struct` `JPEG.Header.HeightRedefinition`
- `struct` `JPEG.Header.Frame`
- `struct` `JPEG.Header.Scan`

These types are produced by parsing raw segment data produced by the lexer. Some of them are generically grouped under protocols such as `JPEG.AnyTable` and `JPEG.Bitstream.AnySymbol`. The strong typing that distinguishes DC and AC huffman tables provides an additional guard against table mismatch bugs.

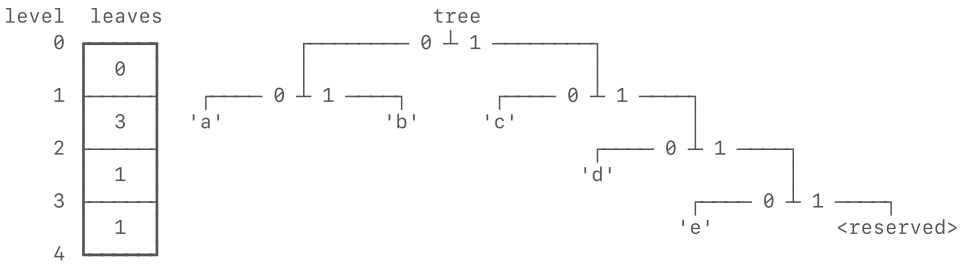
Most parseme types follow a common API pattern — they are constructed from raw data through static `.create(...)` methods, and then converted into cross-validated model types through `.validate(...)` instance methods, to which relevant context is passed.

v.ii.vii. huffman decoder implementation

- `struct` `JPEG.Table.Huffman<Symbol>.Decoder`

This type implements an efficient huffman decoder.

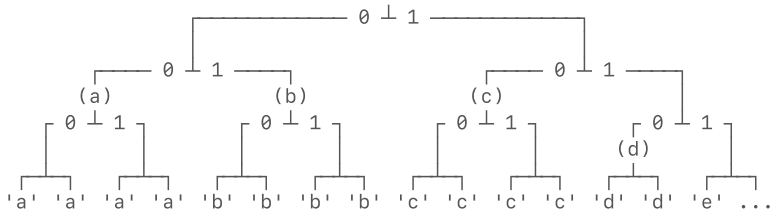
This implementation takes advantage of the fact that JPEG huffman tables are defined gzip style, as sequences of leaf counts and leaf values. The leaf counts indicate the number of leaf nodes at each level of the tree. Combined with a rule that says that leaf nodes always occur on the “leftmost” side of the tree, this uniquely determines a huffman tree.



Note that in a huffman tree, level 0 always contains 0 leaf nodes (why?) so the huffman table omits level 0 in the leaf counts list.

The library *could* build a tree data structure, and traverse it as it reads in the coded bits, but that would be slow and require a shift for every bit.

Instead it extends the huffman tree into a perfect tree, and assigns the new leaf nodes the values of their parents.



This creates a table of huffman codes where all the codes are “padded” to the same length. Note that codewords that occur higher up the tree occur multiple times because they have multiple children. Of course, since the extra bits aren’t actually part of the code, the table stores separately the length of the original code so that the consumer knows how many bits to advance the current bit position by once a match has been looked up.

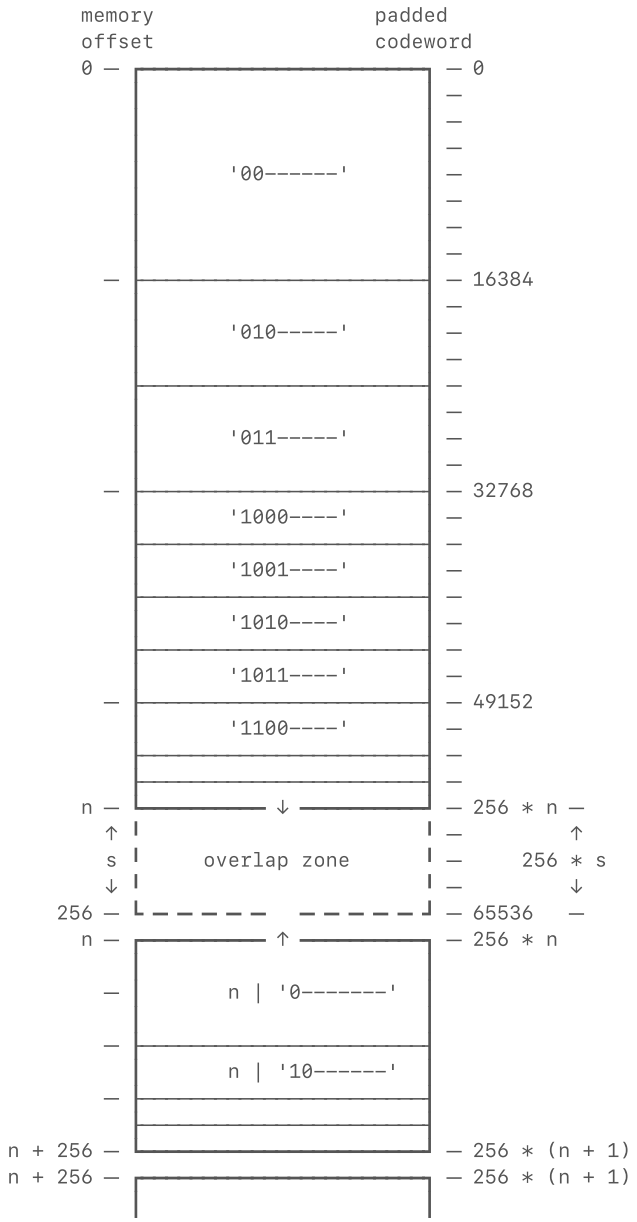
prefix	value	length
`0000`	`'a'`	`2`
`0001`	`'a'`	`2`
`0010`	`'a'`	`2`
`0011`	`'a'`	`2`
`0100`	`'b'`	`2`
`0101`	`'b'`	`2`
`0110`	`'b'`	`2`
`0111`	`'b'`	`2`
`1000`	`'c'`	`2`
`1001`	`'c'`	`2`
`1010`	`'c'`	`2`
`1011`	`'c'`	`2`
`1100`	`'d'`	`3`
`1101`	`'d'`	`3`
`1110`	`'e'`	`4`

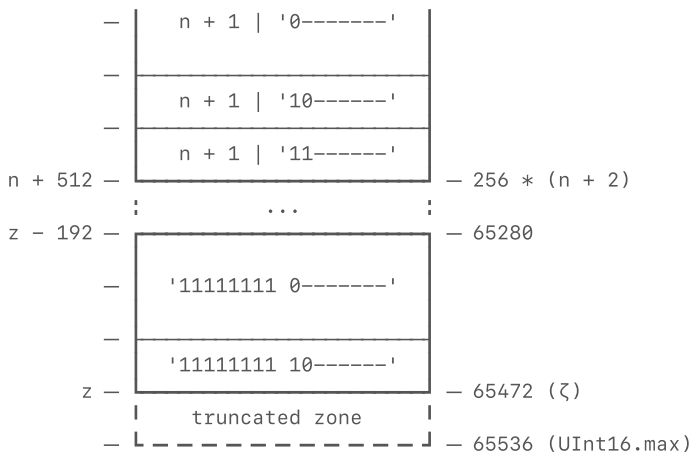
Decoding entropy-coded data then becomes a matter of matching a fixed-length bitstream against the table (the code works as an integer index!) since all possible combinations of trailing “padding” bits are represented in the table.

In JPEG, codewords can be a maximum of 16 bits long. This means in theory a lookup table would have to be 2^{16} entries long. That is a huge table considering there are only 256 actual symbols, and since this is the kind of thing that really needs to be optimized for speed, this needs to be as cache friendly as possible.

We can reduce the table size by splitting the 16-bit table into two 8-bit levels. This means having one 8-bit "root" tree, and *k* 8-bit child trees rooted on the internal nodes at level 8 of the original tree.

So far, we've looked at the huffman tree as a tree. However it actually makes more sense here to look at it as a table, just like its implementation. The tree is right-heavy, so its compacted table will look something like this:





Where *n* is the number of level-0 table entries, *z* is the size of the table in memory, and *ζ* is the logical size of the table (which can be less than or equal to 65,536).

This is convenient because we don't need to store anything in the table entries themselves to know if they are direct entries or indirect entries. If the index of the entry is greater than or equal to *n* (the number of direct entries), it is an indirect entry, and its indirect index is given by the first byte of the codeword with *n* subtracted from it. Level-1 subtables are always 256 entries long since they are leaf tables. This means their positions can be computed formulaically, given *n* (a constant), which is also the position of the first level-1 table.

(For computational ease, we store $s = 256 - n$ instead.
The value *s* can be interpreted as the number of level-1 subtables that trail the level-0 table in the storage buffer.)

How big can *s* be? Remember that there are only 256 different encoded values which means the original tree can only have 256 leaves. Any full binary tree with height at least 1 *must* contain at least 2 leaf nodes. Since the child trees must have a height greater than 0 (otherwise they would be 0-bit trees), every child tree except possibly the rightmost one must have at least 2 leaf nodes. The rightmost child tree is an exception because in JPEG, the all-ones codeword does not represent any value, so the rightmost tree can possibly only contain one "real" leaf node. We can apply the pigeonhole principle to show that we can only have up to $k \le 129$ child trees.

In fact, we can reduce this even further to $k \le 128$ because if the rightmost tree only contains 1 leaf, there has to be at least one other tree with an odd number of leaves to make the total add up to 256, and that number has to be at least 3. In reality, *k* is rarely bigger than 7 or 8, yielding a significant size savings.

Because we don't need to store pointers, each table entry can be just 2 bytes long; 1 byte for the encoded value, and 1 byte to store the length of

the codeword.

A buffer like this will never have size greater than $2 \times 256 \times (128 + 1) = 65{,}792$ bytes, compared with $2 \times 2^{16} = 131{,}072$ bytes for the 16-bit table. In reality the two-layer table is usually on the order of 1–4 kilobytes in size.

Why not compact the child trees further, since not all of them actually have height 8? We could do that, and get some serious worst-case memory savings, but then we couldn't access the child tables at constant offsets from the buffer base. We would need to store whole ≥16-bit pointers to the specific byte offset where the variable-length child table lives, and perform a conditional bit shift to transform the input bits into an appropriate index into the table. This would also require two table lookups, as opposed to one.

v.ii.viii. data representations

- `struct` `JPEG.Data.Spectral<Format>`
  - `struct` `JPEG.Data.Spectral<Format>.Plane`
  - `struct` `JPEG.Data.Spectral<Format>.Quanta`
- `struct` `JPEG.Data.Planar<Format>`
  - `struct` `JPEG.Data.Planar<Format>.Plane`
- `struct` `JPEG.Data.Rectangular<Format>`

These are the data representations discussed in the user model section. The `Spectral` and `Planar` types are `RandomAccessCollection`s of planes. Each collection type (as well as the `Quanta` member of a `Spectral` instance) provides an `index(forKey:)` method used to translate a c_i or q_i key into a p or q integer index. Note that for plane index lookups, this is *not* exactly the same as finding the component index in the image layout, because this method will return `nil` if p is greater than or equal to the number of planes in the data representation (i.e., if the component is a resident component, but not a recognized one). This distinction is, however, irrelevant for the built-in common color format, as it accepts no unrecognized components.

v.ii.ix. decoder types and decoder implementation

- `extension` `JPEG.Bitstream`
- `extension` `JPEG.Data.Spectral`
- `extension` `JPEG.Data.Spectral.Plane`
- `struct` `JPEG.Context<Format>`

The extension methods on `Bitstream` extract composite values from an entropy-coded bitstream. Extension methods on the `Spectral` type then implement the scan-level decoding loops for interleaved scans. For non-interleaved scans, these methods are instead defined on the `Spectral.Plane`. These scan-level decoder functions call the composite-level methods.

In turn, frame-level decoder functions, implemented on the `Context<Format>` type, call the scan-level functions. The `Context` type maintains the state of both a `Spectral` instance, and the state of bound table resources.

v.ii.x. staged conversion APIs (forward)

- `extension` `JPEG.Data.Spectral`
  - `extension` `JPEG.Data.Spectral.Plane`
    - `typealias` `JPEG.Data.Spectral<Format>.Plane.Block8x8<T>`
- `extension` `JPEG.Data.Planar`
  - `extension` `JPEG.Data.Planar.Plane`
- `extension` `JPEG.Data.Rectangular`

These extensions implement the transformations required to convert a spectral image into a spatial-domain planar image, and a planar image to a rectangular image. The inverse discrete cosine transform algorithm is semantically equivalent to the floating-point algorithm used by *libjpeg*, so the framework will replicate the rounding behavior of *libjpeg*.

v.ii.xi. built-in color formats and color target conformances

- `enum` `JPEG.Common`

The `JPEG.Common` enumeration defines the built-in color format for JFIF/EXIF images.

```
enum JPEG.Common
{
    case y8, ycc8
}
```

Note that `y8` only occurs in JFIF images.

This section of the code also conforms the built-in `RGB` and `YCbCr` color targets to the `JPEG.Color` protocol, using `JPEG.Common` as their format types.

v.iii. `encode.swift`

Code and definitions related to to the compressor lives in this file. It extends the top-level namespace `JPEG`, with the following type members:

- `JPEG` (compression error types)
  - `enum` `JPEG.FormattingError`
  - `enum` `JPEG.SerializingError`
  - `enum` `JPEG.EncodingError`

- `JPEG.Data` (staged conversion APIs)
  - `extension` `JPEG.Data.Planar.Plane`
  - `extension` `JPEG.Data.Spectral.Plane`
  - `extension` `JPEG.Data.Planar`
- `JPEG` (parseme encoders)
  - `extension` `JPEG.Data.Spectral`
  - `extension` `JPEG.Layout`
- `JPEG.Table.Huffman` (huffman tree builder)
  - `class` `JPEG.Table.Huffman.Subtree<Element>`
  - `struct` `JPEG.Table.Huffman.Encoder`
    - `struct` `JPEG.Table.Huffman.Encoder.Codeword`
      *
- `JPEG` (encoder implementation)
  - `extension` `JPEG.Bitstream`
    - `extension` `JPEG.Bitstream.Composite.DC`
    - `extension` `JPEG.Bitstream.Composite.AC`
  - `extension` `JPEG.Data.Spectral`
    - `extension` `JPEG.Data.Spectral.Plane`
- `JPEG` (serializer implementation)
  - `extension` `JPEG.JFIF`
    - `extension` `JPEG.JFIF.Version`
    - `extension` `JPEG.JFIF.Unit`
  - `extension` `JPEG.AnyTable`
  - `extension` `JPEG.Table`
    - `extension` `JPEG.Table.Huffman`
    - `extension` `JPEG.Table.Quantization`
  - `extension` `JPEG.Header.Frame`
  - `extension` `JPEG.Header.Scan`
- `JPEG` (stream types, and formatter)
  - `protocol` `JPEG.Bytestream.Destination`
  - `extension` `JPEG.Bytestream.Destination`

v.iii.i. compression error types

- `enum` `JPEG.FormattingError`
- `enum` `JPEG.SerializingError`
- `enum` `JPEG.EncodingError`

These error types are analogous to their counterparts in `decode.swift`. However, because most compressor APIs are designed to fatal-error on failure rather than throw (because the caller is usually responsible for the

data inputs), there are far fewer error cases. In fact, of the three error types, only `FormattingError` has any cases at all; the others are defined as placeholders for future framework expansion.

v.iii.ii. staged conversion APIs

- `extension` `JPEG.Data.Planar.Plane`
- `extension` `JPEG.Data.Spectral.Plane`
- `extension` `JPEG.Data.Planar`

These extensions implement the forward discrete cosine transform, which converts a planar image into a spectral one. (At present, the conversion from rectangular to planar representation is unimplemented.)

v.iii.iii. parseme encoders

- `extension` `JPEG.Data.Spectral`
- `extension` `JPEG.Layout`

These extensions implement methods that encode model types as their parseme forms. Because model types support many more assumptions than parseme types, these APIs do not return optionals nor do they throw errors. In fact, failure can generally only occur due to serious programmer error, for example, a broken custom color `Format` implementation.

v.iii.iv. huffman tree builder

- `class` `JPEG.Table.Huffman.Subtree<Element>`
- `struct` `JPEG.Table.Huffman.Encoder`
  - `struct` `JPEG.Table.Huffman.Encoder.Codeword`

The `Subtree` type is used to construct a (near-) optimal huffman tree given a list of symbols and symbol frequencies. (This functionality is what the `Common.Heap` type is for.)

The huffman `Encoder` table is the inverse of the huffman `Decoder` table; it takes symbols as input (through a subscript interface), and returns `Codeword`s as output. The efficient implementation of this functionality is far more straightforward than for its decoder counterpart — there are only 256 possible symbols, so the symbol-to-codeword mapping can be accomplished with a simple 256-entry lookup table.

v.iii.v. encoder implementation

- `extension` `JPEG.Bitstream`
  - `extension` `JPEG.Bitstream.Composite.DC`
  - `extension` `JPEG.Bitstream.Composite.AC`
- `extension` `JPEG.Data.Spectral`
  - `extension` `JPEG.Data.Spectral.Plane`

These extensions implement the inverse operations to the decoder methods in `decode.swift`. The extensions on `JPEG.Bitstream` handle

the encoding of composite values into entropy-coded bitstreams, while the extensions on `Spectral` and `Spectral.Plane` handle encoding at the scan-level, for interleaved and non-interleaved scans, respectively.

(There is no need for a counterpart to the `Context` handler, since all state-related parameters have already been computed when initializing the image `Layout`.)

v.iii.vi. serializer implementation

- `extension` `JPEG.JFIF`
  - `extension` `JPEG.JFIF.Version`
  - `extension` `JPEG.JFIF.Unit`
- `extension` `JPEG.AnyTable`
- `extension` `JPEG.Table`
  - `extension` `JPEG.Table.Huffman`
  - `extension` `JPEG.Table.Quantization`
- `extension` `JPEG.Header.Frame`
- `extension` `JPEG.Header.Scan`

These extensions implement the serializers for the parseme types defined in `decode.swift`. In most cases, they are defined as instance methods on parseme types, which return untyped marker segment bodies as `[UInt8]` buffers.

v.iii.vii. stream types, and formatter

- `protocol` `JPEG.Bytestream.Destination`
- `extension` `JPEG.Bytestream.Destination`

These types define the data outputs for the encoder. The `JPEG.Bytestream.Destination` protocol abstracts a data destination, which, like the data destinations, could be a file handle, in-memory data blob, or anything else.

```
protocol JPEG.Bytestream.Destination
{
    mutating
    func write(_ bytes:[UInt8]) -> Void?
}
```

The `write(_:)` method should return `Void` on success, and `nil` on failure.

Like the lexer, the formatter is implemented atop of the `JPEG.Bytestream.Destination` protocol as an extension.

v.iv. `debug.swift`

This file is relatively simple; it only implements the various `CustomStringConvertible` conformances that pretty-print JPEG entities. Importantly, it also provides `ExpressibleByIntegerLiteral` conformances for component and quanta key types, making it easier for users to initialize `Layout` and `Spectral` data structures.

- `extension` `JPEG.Component.Key:ExpressibleByIntegerLiteral`
- `extension` `JPEG.Table.Quantization.Key:ExpressibleByIntegerLiteral`

v.v. `os.swift`

This file provides system-dependent features such as file system support. It is only compiled is the operating system is MacOS or Linux, and omitted otherwise, so that other Swift platforms such as Android or iOS can still use the core library features.

It extends the `Common` namespace with the following types, which conform to the respective `JPEG.Bytestream.Source` and `JPEG.Bytestream.Destination` protocols.

- `Common`
  - `enum` `Common.File`
    - `struct` `Common.File.Source`
    - `struct` `Common.File.Destination`

Both system file interfaces are exposed through the static `open` method, which has the following signature:

```
static
func open<Result>(path:String, _ body:(inout Self) throws -> Result)
    rethrows -> Result?
```

This file also extends `JPEG.Data.Spectral`, `JPEG.Data.Planar`, and `JPEG.Data.Rectangular` with staged APIs that take file path names, rather than generic streams as arguments.

- `JPEG.Data`
  - `extension` `JPEG.Data.Spectral`
  - `extension` `JPEG.Data.Planar`
  - `extension` `JPEG.Data.Rectangular`

vi. test architecture

> **Summary:** The Travis Continuous Integration set up for the project repository supports four sets of tests. *Unit tests* verify basic algorithmic components of the library, such as the huffman coders and zigzag index translators. *Integration tests* verify that a sample set of images with different supported coding processes and layouts

> can be decoded and encoded without errors. *Regression tests* run the integration tests and compare them with known outputs. Finally, *fuzz tests* generate randomized test images and compare the output to that output from third-party implementations such as the *libjpeg*-based `imagemagick convert` tool, ensuring inter-library compatibility.

The framework uses Travis to run four sets of automated tests:

- Unit tests
- Integration tests
- Regression tests
- Fuzz tests

All tests are compiled as executable products by the package manager, and are invoked by bash scripts in the `utils/` directory:

- `utils/unit-test`
- `utils/integration-test`
- `utils/regression-test`
- `utils/fuzz-test`

These scripts return `0` on passing.

vi.i. unit tests

These tests validate several important subcomponents of the library, including some `internal` subcomponents, which is why these tests are always compiled in `debug` mode (with `@testable` imports) rather than `release` mode. (Attempting to compile all products at once with the package manager set to `release` mode will fail for this reason.)

Most unit tests consist of a few explicitly written test cases (which serve as the root of trust), coupled with more exhaustive tests that pair certain sets of APIs with their inverses, and attempt to feed the output of one API as the input of the other, and vice-versa. Currently, the unit tests validate the following library components:

- zig-zag coefficient indices
- amplitude coding (translating composite values into numerical values)
- huffman table coding
- huffman table construction

vi.ii. integration tests

These tests attempt to decode and encode various test images without errors. Because the library features extensive internal validation, these tests are highly valuable for enforcing internal logical consistency. They do not attempt to match data outputs, and will succeed as long as no errors occur in the encoding or decoding process.

Integration tests support both `debug` and `release` compilation modes, using the `-c <compilation mode>` command-line option. Currently, the decoding tests run on the following test images:

test image	coding process	components	subsampling
`color-sequential-1.jpg`	baseline	3	4:2:0
`color-sequential-2.jpg`	baseline	3	4:2:0
`color-sequential-3.jpg`	baseline	3	4:4:4
`color-sequential-4.jpg`	baseline	3	4:4:4
`grayscale-sequential-1.jpg`	baseline	1	—
`grayscale-sequential-2.jpg`	baseline	1	—
`color-progressive-1.jpg`	progressive	3	4:2:0
`color-progressive-2.jpg`	progressive	3	4:2:0
`color-progressive-3.jpg`	progressive	3	4:4:4
`color-progressive-4.jpg`	progressive	3	4:4:4
`grayscale-progressive-1.jpg`	progressive	1	—
`grayscale-progressive-2.jpg`	progressive	1	—

The encoding tests take a predefined RGB input image, and encode it as the following test outputs:

output image	coding process	components	subsam
`karlie-kloss-1-color-sequential.jpg`	baseline	3	4:4:4
`karlie-kloss-1-grayscale-sequential.jpg`	baseline	1	—
`karlie-kloss-1-color-progressive.jpg`	progressive	3	4:4:4
`karlie-kloss-1-grayscale-progressive.jpg`	progressive	1	—

vi.iii. regression tests

As the name suggests, the regression tests attempt to decode test images, and compare the output to a set of "golden outputs". The regression tests run on the same JPEG test images as the integration tests, and compare the output of both the RGB and YCbCr built-in color targets.

Like the integration tests, the regression tests support both `debug` and `release` compilation modes, with the same command-line syntax. The `utils/regression-test` tool can be run with the option `-u` (long form `--update`) to regenerate the golden outputs. (The tests will return a failure code for that run.)

vi.iv. fuzz tests

The fuzz tests are the most sophisticated automated tests, because they don't return a pass or fail result, but rather, are used to quantify the difference between the framework output, and the output of a different library, such as *libjpeg*. Unlike other formats, such as PNG, which has a well-defined binary specification, the JPEG standard only specifies the symbolic *mathematical* definition of its discrete cosine transform. This means that different JPEG codecs, different versions of the same JPEG codec (such as *libjpeg*), and even the same version of the same JPEG codec (*libjpeg* as well) on different platforms can produce different output due to discrepancies in floating-point and integer arithmetic spellings.

The core of the fuzz tests is, of course, the fuzzer, which generates randomized 8x8 pixel test images. (This size is chosen because it contains a single JPEG coefficient block.) The fuzzer uses Swift's randomization APIs to generate pixel values, though the ranges are limited to avoid creating YCbCr colors that do not have equivalents in the RGB color space. Out-of-range colors can be problematic for comparing JPEG codecs because while the JPEG standard technically specified that clamping should be used (and the framework conforms to this), in practice, out-of-range color values result in implementation-defined behavior. For example, for performance reasons, *libjpeg* will wrap-around out-of-range color values if they exceed a constant "safety margin".

The number of test images the `utils/fuzz-test` script will generate is set by the `-n <count>` option, by default it is set to 16. The script will use the system Imagemagick `convert` tool to run the reference codec; Imagemagick is powered by *libjpeg*, so this effectively establishes *libjpeg* as the reference implementation. (The Travis CI will install Imagemagick with Homebrew when testing on MacOS platforms.) The script then uses a separate execuable product called `compare` (built by the package manager) to compare the `convert` tool output with the Swift library output, compute statistics, and generate histograms of the output discrepancy.

The framework's discrete cosine transform implementation is written to exactly emulate the floating-point behavior of *libjpeg*, and will match its output exactly so long as no out-of-range pixel values occur. However, since *libjpeg* is not internally consistent with respect to its other arithmetic modes, this means that significant discrepancies (though generally less than 10 gray levels) exist when using *libjpeg*'s "fast" mode or its fixed-point mode.

vii. acknowledgements

The author would like to acknowledge the valuable support and advice of Dr. Zbigniew Kalbarczyk, who oversaw this project from start to completion over the course of approximately one year.