# A JFIF/JPEG decoder (jfif)

by Simon Southwell
12th July 2010

## Introduction

Presented here is a program to decode JPEG files (including JFIF format jpeg files) and dumping the decoded RGB data to a 24 bit bitmap file. In addition, the decoded image can be displayed in a pop-up window for visual verification.

Source code is included in the download available from github, and the program has been written with a structure that reflects the intention to implement the code as HDL for synthesis in an FPGA or ASIC environment. The core functionality is written in C++ (i.e. the code that is compiled to a library), with ancilliary wrapper functions written in C (such as the top level command line user interface and the graphical output).

The article on JPEG concepts and format (in the same folder as this file) is a good place to start if you haven't done so already, as this article will assume that you have, as some of the implementation concepts are going to be put in to practice. The software itself has been written with speed in mind, but the main purpose of the code is as a reference for the digital hardware implementation to follow, and as a learning aide. As such, the architecture choices and some of the implementation is constructed to be hardware efficient, and allow direct comparisons between the software internal state and the hardware internal state.

## Usage

The usage message for jfif is as follows:

```
Usage: jfif [-h] [-d] [-i ] [-o ]
    -h display help message
    -d display generated bitmap file's image in a window
    -i define input filename (default test.jpg)
    -o define output filename (default test.bmp)
```

JFIF is simple to use. Just typing "jfif" will result in a file test.jpg being decoded (if exists), and a bitmap output test.bmp be written. The -i and -o options are used to alter the default input and output filenames. The resultant bitmap can also be optionally displayed in a popup window, scaled to a maximum display area of 800x600, for validating the conversion by eye, using the -d option. This is generated from the actual bitmap file rather than internal memory to guarantee no additional artifacts in bitmap generation are missed in the display. This delays the display of the file a fraction, but in the interests model integrity.

## Top Level Program Structure

This article is not meant to be a tutorial on the JPEG format (see here), but the source code is available for download as a starting point for those who wish a working example to study. Below is show some outline pseudo-code of the way jfif is constructed, with all the major functions listed to aid navigation around the source code. Later sections will look at these functions in more details, but this should start as a good starting point.

```
main()
   READ infile into buffer

   jpeg_process_jfif()             -- Converts input baseline DCT JFIF buffer to 24 bit bitmap
       jpeg_extract_header()       -- Extracts jpeg header information (scan, frame, quantisation/huffman tables, DRI)
           jpeg_dht()              -- Constructs a Huffman decode structure from huffman table data
       jpeg_bitmap_init()          -- Creates space for appropriately sized bitmap and initialises header data

     LOOP for each MCU:            -- jpeg_process_jfif() process an MCU at a time until EOI (or error)
       jpeg_huff_decode            -- Gets an adjusted Huffman/RLE decoded amplitude value and ZRLs, or marker or end-of-block
           jpeg_dht_lookup()       -- Does huffman lookup on code and extracts amplitude data, or flags a marker
               jpeg_get_bits()     -- Gets top bits from barrel shifter and (optionally) removes. Pulls in extra input if needed.
               jpeg_amp_adjust()   -- Adjusts decoded huffman decoded amplitude to +/- amplitude value
           <dequantise>            -- De-quantisation done in jpeg_huff_decode directly from selected table
       jpeg_idct()                 -- Inverse discrete cosine transform
       jpeg_ycc_to_rgb()           -- Converts YCbCr to RGB on an MCU
       jpeg_bitmap_update()        -- Updates bitmap data buffer with 8x8 RGB values
     ENDLOOP

   WRITE bitmap buffer to outfile

   IF display requested
       jpeg_display_bmp_file()     -- Display bitmap file in a popup window
```

At the coarsest level, the program simply reads a JPEG image into a buffer, calls `jpeg_process_jfif()` to process that buffer, and then writes out the returned bitmap data. An optional display of the written bitmap file can be done afterwards. All the hard work, of course, is done in the `jpeg_process_jfif()` function.

The `jpeg_process_jfif()` breaks down into to distinct phases. Firstly it parses the JPEG header, extracting information required for decode, including quantisation tables, and Huffman table data, which is processes into a useful format for later with `jpeg_dht`. In this phase, the bitmap buffer is initialised. The second phase is a loop to process MCUs until an end-of-image marker (see article on JPEG concepts and format), or some exception encountered. In each iteration, `jpeg_huff_decode()` is called which will decode all the 8x8 arrays within the MCU and return the results. The returned result is usually a pointer to the decoded scan data, but can indicate a marker (e.g. EOI). THe decode function makes use for two other functions: `jpeg_get_bits()` to manage a barrel shifter for extracting the variable length codes, and `jpeg_amp_adjust()` to rescale the returned codes to amplitude values (see article on JPEG concepts and format). Within the decode function is also located the dequantisation step—for no better reason than all the required data is available at this point, and the operation is simple, not requiring another function call.

After decode, the individual 8x8 arrays within the MCU are inverse DCT transformed with one ore more calls to `jpeg_idct()`. The data is updated in place, as no extra space is required since expansion has already occured. Within the call to the iDCT function the normalisation and clipping also occurs, to place the output values in the 0 to 255 range. At this point we now have YCbCr (or just Y) arrays. For colour images `jpeg_ycc_to_rgb()` is called to convert to RGB, and in all cases a call to `jpeg_bitmap_update()` places the data in the bitmap buffer created earlier by `jpeg_bitmap_init()` in the appropriate location and format.

The majority of the code is located in `jfif.cpp`. The two exceptions to this are the inverse DCT function (`jpeg_idct`) found in `jpeg_idct.cpp`, and the GTK+ code (`jpeg_display_bmp_file`) for displaying the bitmap in a window, found in `jfif_gtk.c`. The header files are pretty self explanitory, but the majority of internal definitions are in `jfif_local.h`, whilst the external definitions are in `jfif.h`.

It should be noted that decoding a JPEG file, strictly speaking, only needs to decode to the luminance and (if present) chrominance information. Converting this to RGB in a bitmap format is a `jfif` specific implementation. Thus the `jpeg_bitmap_init()`, `jpeg_ycc_to_rgb()`, `jpeg_bitmap_update()` and `jpeg_display_bmp_file()` functions are not strictly part of the JPEG decode proper. However, it is in the `jpeg_ycc_to_rgb()` function that sub-sampling reconstruction occurs—with averaged chrominance data being used with multiple luminance data, in both the vertical and horizontal directions.

The above description is just an overview of the top level features. More details of the sub-function architectures are provided in the JPEG decoder article on implementation.

## Unsupported Features

The full JPEG standard is broad in it's scope, with alternatives, and enhancing features optionally available. The `jfif` program is intended to be a starting point for a hardware implementation, and support of all the features of the standard is not practical and not really desirable. For instance, the compression technique, post-DCT, can be a Huffman table based algorithm, or an Arithmetic coding algorithm. Also there are extended and Progressive DCT formats, as well as a lossless compression mode. `jfif` has been limited to supporting a sub-set of these options—chosen to be the sub-set with the broadest overlap with real jpeg encoded data, whilst being most likely to be suitable in hardware decoding applications. The following list gives details of the known unsupported features. Anything not on the list has either been ommited by mistake, or is a bug in the application.

- Extended and Progressive DCT
- All differential DCT
- All Arithmetic coding modes
- Lossless Compression
- Image components (Nf) greater than 4
- Define number of Lines (DNL)/multiple scans in a frame
- Horizontal sub-sampling factor (Hi) > 2
- Vertical sub-sampling factor (Vi) > 2

## Download

The `jfif` program and source code is available from github. It includes a precompiled windows executable, the source code and a makefile for building under Cygwin or Linux, as well as files for MSVC Express 2010.

## Useful Links and further reading

The following are some links to the standards and useful online and offline material.

- JPEG standard (ITU-T recommendation T.81)
- JPEG File Interchange Format (JFIF) version 1.02
- JPEGsnoop file decoding utility by Calvin Hass. Great for studying JPEG format.
- JPEG concepts from Steven W. Smith's excellent "The Scientist's and Engineer's Guide to Digital Signal Processing" book (freely available online)
- Wikipedia entry for bitmap format, with C header details.
- "The Data Compression Book", 2nd Ed., Mark Nelson and Jean-loup Gailly. Now out of print, but try Abebooks or Amazon