EasyJPEG

Mikrokontrolerowa biblioteka dekodera JPEG

Niniejszy artykuł ma za zadanie przybliżenie Czytelnikowi zagadnień związanych z kompresją obrazów w popularnym standardzie JPEG. Prezentuje punkt widzenia programisty, bez zbytniego zagłębiania się w matematyczne meandry. Przedstawiony zostanie także projekt bezpłatnej, uniwersalnej biblioteki dekodującej, stworzonej przez autora w języku C. Dzięki jej małym wymaganiom sprzętowym i prostocie implementacji przez użytkownika końcowego, biblioteka idealnie nadaje się do wykorzystania z popularnymi mikrokontrolerami 32-bitowymi oraz kolorowymi wyświetlaczami TFT.

Dodatkowe materiały na CD i FTP:
ftp://ep.com.pl, user: 15257, pass: 1ajsf046

Dodatkowe informacje:
http://www.impulseadventure.com/photo/
http://www.ijg.org/
http://en.wikipedia.org/wiki/Jpeg

Pliki zakodowane w formacie JPEG (Joint Photography Experts Group) są w dzisiejszych czasach najpopularniejszym i najczęściej spotykanym formatem używanym do przechowywania cyfrowych obrazów w postaci skompresowanej. Kompresja pokrótce opiera się na właściwościach ludzkiego oka, którego wrażliwość spada dla coraz wyższych częstotliwości przestrzennych występujących w obrazie. Innymi słowy – człowiek słabiej rozróżnia drobne szczegóły obrazu w obecności dużych detali obrazu, co jest dość oczywiste. Wiedząc o tym fakcie, jesteśmy w stanie przekształcić obraz z reprezentacji przestrzennej do dwuwymiarowej reprezentacji częstotliwościowej (należy pamiętać, że mówimy cały czas o obrazach stałych, a częstotliwość jest w tym przypadku wartością występującą na fizycznej osi w przestrzeni, a nie w czasie!) i usunąć wyższe składowe, znacznie redukując ilość zachowywanej informacji. Efekt końcowy jest praktycznie niezauważalny dla ludzkiego oka, lecz oczywiście część informacji jest bezpowrotnie tracona. Taki rodzaj kompresji określany jest jako stratna (lossy). Pliki JPEG zapewniają wysoki stopień upakowania danych, często ponad 1:100 w stosunku do surowej bitmapy z 8-bitową reprezentacją RGB.

Dostępne rozwiązania

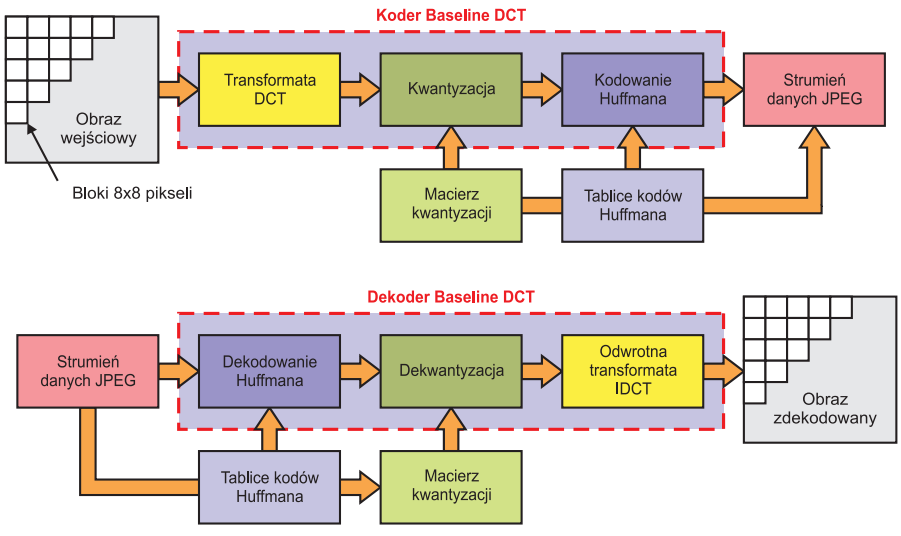
Jednym z podstawowych założeń projektu było stworzenie otwartego, bezpłatnego kodu z możliwością wykorzystania go do dowolnych celów, także komercyjnych. Większość rozwiązań programowych dekoderów JPEG jest częścią systemów operacyjnych lub innych projektów, które nie mają otwartych źródeł albo te udostępnione są na restrykcyjnych licencjach (GPL). Jedyną, znaną autorowi biblioteką niemającą tych ograniczeń, jest *Independent JPEG Group Library*, lecz jej rozmiar i stopień skomplikowania (obejmujący ponad kilkadziesiąt plików źródłowych) może stanowić poważną barierę jej zastosowania w małych projektach, szczególnie na platformach pozbawionych systemu operacyjnego i procedur dynamicznego alokowania pamięci. Z tego powodu zapadła decyzja o napisaniu biblioteki od podstaw, z ewentualnym wykorzystaniem fragmentów biblioteki IJG.

Dokumentacja standardu

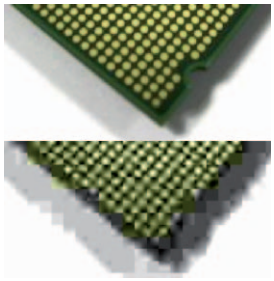
Pierwsze idee i implementacje kompresji obrazu wykorzystujące transformaty kosinusowe pojawiły się już na przełomie lat 70./80. ubiegłego wieku, lecz dopiero w latach 1992–1994 opracowano oficjalne standardy ITU-T T.81 oraz ISO/IEC 10918.

Standard ISO jest płatny, natomiast w drugim przypadku sytuacja nie jest jasna. Według ITU standard również nie jest darmowy, lecz różne poważne organizacje, takie jak *Word Wide Web Consortium* publikują go na swoich stronach WWW nieodpłatnie. Ważnym elementem jest specyfikacja JFIF (JPEG File Interchange Format) stworzona przez IJG, która uzupełniła standardy ISO/ITU w taki sposób, aby postał przenośny i uniwersalny format pliku *.jpg. Obraz w czystym standardzie JPEG jest tylko strumieniem danych, które mogą być enkapsulowane w dowolnym medium, formacie pliku itp. Powszechnie z kompresji JPEG korzystają np. pliki w formacie TIFF.

Jedyną książką w całości poświęconą standardowi jest *JPEG Still Image Data Compression Standard* napisana przez W. B. Pennebakera i J. L. Mitchella w 1993 roku. Fragmenty książki są dostępne poprzez witrynę *Google Books*. Istnieje duża liczba publikacji i artykułów poświęconych ogólnie przetwarzaniu sygnałów, w których znajdują się pobieżne opisy działania kompresji JPEG, lecz jest to materiał niewystarczający do zaprojektowania dekodera lub enkodera.



Rys. 1. Schemat kodera/dekodera JPEG

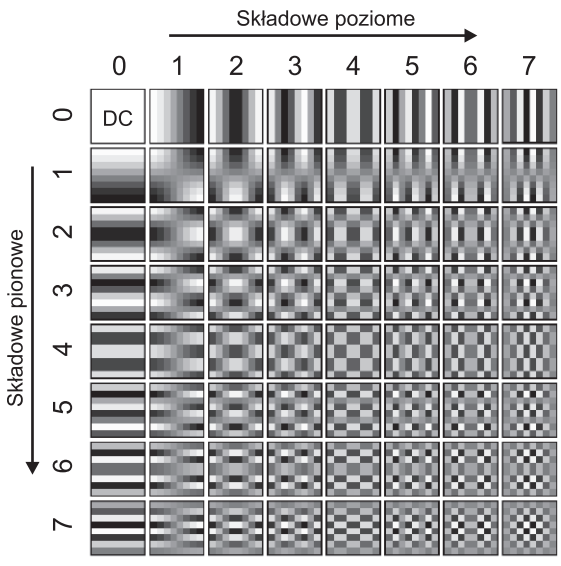


Rys. 2. Różne wielkości pikseli zależnie od stopnia kompresji

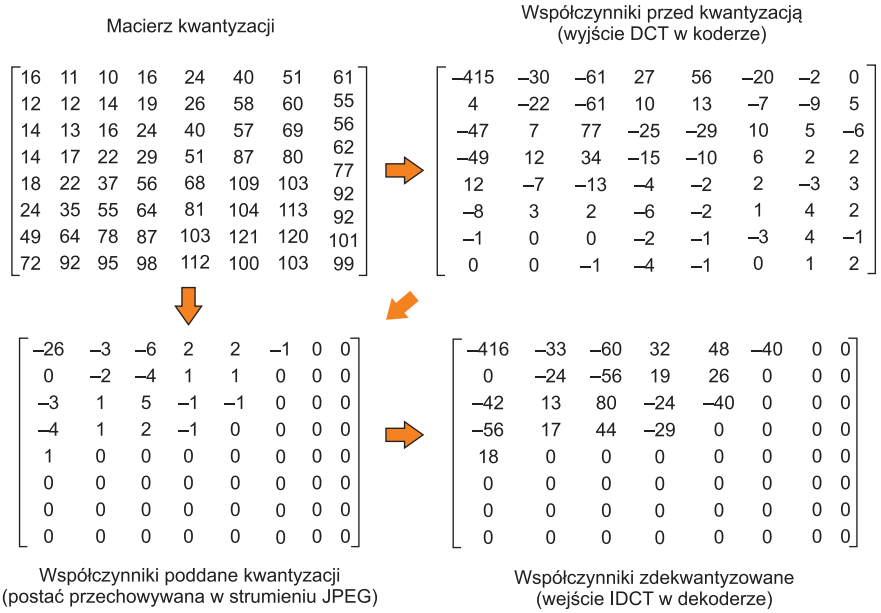
Pomocnym źródłem informacji są artykuły umieszczone na stronie *http://www.impulseadventure.com/photo/*. Ich autor stworzył również specjalny program o nazwie JPEGsnoop, pozwalający na dokładne analizowanie zawartości plików JPEG.

Podstawy działania

Na **rys. 1** przedstawiono ogólny schemat kodera oraz dekodera JPEG. Ich najistotniejszymi elementami są: dyskretna transformata kosinusowa typu II (DCT-II) oraz jej odwrotność IDCT (DCT-III). Są one podobne do dyskretnej transformaty Fouriera w tym sensie, że przekształcają sygnał z domeny przestrzennej do domeny częstotliwości i na odwrót, lecz operują tylko na liczbach rzeczywistych. JPEG używa transformat dwuwymiarowych na blokach-fragmentach obrazu 8×8 pikseli, co jest rozsądnie niską wartością pod względem złożoności obliczeniowej i zapotrzebowania na pamięć operacyjną. Jednak przy tak małym rozmiarze i wysokim stopniu kompresji często na obrazie widoczne są ostre przejścia barw i jasności pomiędzy sąsiednimi blokami. Efekt ten można zaobserwować na **rys. 2**. Transformata kosinusowa, jak większość tego typu transformat, wykorzystuje przedstawienie sygnału jako złożenie wielu funkcji trygonometrycznych o różnych częstotliwościach (w tym przypadku kosinusoid). Na **rys. 3** pokazano wizualne przedstawienie dwuwymiarowych funkcji wynikowych dla każdego z 64 współczynników transformaty 8×8. Współczynnik z lewego górnego rogu (0,0) nazywany jest współczynnikiem stałym DC, a pozostałe 63 – współczynnikami zmiennymi AC. Każdy blok na rysunku reprezentuje obraz 8×8 powstały z odwrotnej transformaty macierzy, w której tylko odpowiadający mu współczynnik nie był zerowy. Ostatecznie, sumując ze sobą takie bloki o „nasileniu" zależnym od wartości współczynników, uzyskujemy rekonstrukcję obrazu. Transformata DCT ma właściwość agregowania najistotniejszych (z psychowizualnego punktu widzenia) informacji obrazu w lewej górnej ćwiartce macierzy wynikowej.



Rys. 3. Dwuwymiarowe funkcje wynikowe

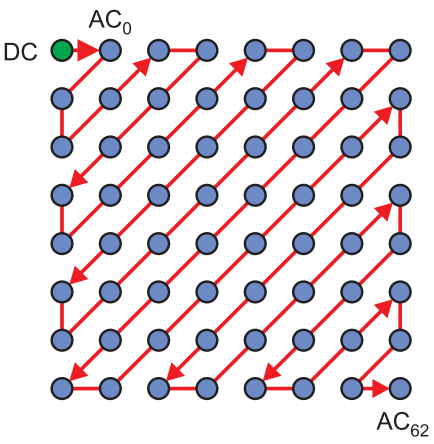


Rys. 4. Operacje na macierzach kwantyzacji

Następnym elementem procesu kompresji jest tzw. kwantyzacja. Polega ona na zredukowaniu w kontrolowany i regulowany sposób liczby składowych wynikowych DCT, które będą przechowywane w pliku. Odbywa się to w wyniku podzielenia współczynników wynikowych DCT przez wartości z macierzy kwantyzacji. Ponieważ używana jest arytmetyka stałoprzecinkowa o zerowej liczby bitów po przecinku, część współczynników zostaje wyzerowana. W dekoderze wszystkie wartości zostają z powrotem pomnożone przez tę samą macierz (przechowywaną w nagłówku strumienia), lecz ilość danych wejściowych jest już znacznie mniejsza. Na **rys. 4** pokazano przykład tego procesu. Macierze kwantyzacji opracowane na podstawie badań sposobu widzenia człowieka zostały zaproponowane w standardzie, lecz kodery mogą używać praktycznie dowolnych wartości. Regulacja stopnia kompresji polega na skalowaniu całej macierzy kwantyzacji poprzez odpowiedni współczynnik.

Ostatnim etapem tworzenia pliku JPEG jest kodowanie kodem Huffmana danych wynikowych z transformat dla wszystkich bloków 8×8 w obrazie. Dane szeregowane są wstępnie w tzw. zygzaku widocznym na **rys. 5**. Taki układ został wybrany eksperymentalnie i zapewnia względnie największe skupienie danych niezerowych na początku tablicy współczynników AC, podczas gdy wyższe składowe częstotliwościowe, zazwyczaj redukowane w procesie kwantyzacji, trafiają na koniec.

Następnie dla każdego kolejnego, niezerowego współczynnika w zygzaku są tworzone dwie 4-bitowe wartości RRRR oraz SSSS poprzedzające go w strumieniu danych. Za pomocą RRRR deklarowana jest liczba współczynników zerowych występujących przed nim, a SSSS deklaruje przedział jego wartości, czyli liczbę bitów następujących po symbolu RRRRSSSS. Wartość współczynników nie jest zapisywana za pomocą typowego kodu U2. Najstarszy bit definiuje, czy wartość jest dodatnia (1), czy ujemna (0),



Rys. 5.

Tab. 1. Zależność parametrów AC i DC od długości bitowej SSSS

SSSS	różnica DC	współczynnik AC
0	0	–
1	–1,1	–1,1
2	–3:–2,2:3	–3:–2,2:3
3	–7:–4,4:7	–7:–4,4:7
4	–15:–8,8:15	–15:–8,8:15
5	–31:–16,16:31	–31:–16,16:31
6	–63:–32,32:63	–63:–32,32:63
7	–127:–64,64:127	–127:–64,64:127
8	–255:–128,128:255	–255:–128,128:255
9	–511:–256,256:511	–511:–256,256:511
10	–1023:–512,512:1023	–1023:–512,512:1023
11	–2047:–1024,1024:2047	–

a kolejne bity ustalają wartość w przedziale odpowiednim dla danej długości. Istnieje odpowiednia kombinacja, tj. RRRR=0 oraz SSSS=0, sygnalizująca dekoderowi, iż był to ostatni niezerowy współczynnik w tym bloku (zakłada się, że wszystkie współczynniki zostały wstępnie wyzerowane). Drugą kombinacją specjalną jest RRRR=15 oraz SSSS=0. Oznacza ona przeskoczenie pełnych 16 zer.

Współczynnik DC jest kodowany w trochę inny sposób. Po pierwsze, jego wartość jest poprzedzona tylko jej długością bitową SSSS. Po drugie, w kolejnych blokach zapisywana jest tylko różnica wartości w stosunku do ostatniego przetwarzanego bloku. Taka technika pozwala na znaczne zredukowanie liczby danych w przypadku obrazów mających duże płaszczyzny w jednolitym kolorze. W **tab. 1** pokazano zależność przedziału wartości od jej długości bitowej SSSS dla współczynników AC i różnic DC.

Po tych operacjach dość efektywnie skompresowane zostały zerowe współczynniki, lecz narzut danych wywołany poprzez obecność symboli RRRRSSSS o stałej, 8-bitowej długości byłby nie do zaakceptowania. Z pomocą przychodzi kodowanie Huffmana (kodowane są nim tylko symbole RRRRSSSS, a nie same wartości współczynników!). Ogólna zasada polega na stworzeniu słownika kodów o zmiennej długości (od 1 do 16 bitów) i przypisaniu im wszystkich możliwych symboli RRRRSSSS (lub SSSS w przypadku DC) dla konkretnego, przetwarzanego obrazu. Do kodów krótkich przypisywane są wartości występujące najczęściej, a do kodów najdłuższych – najrzadsze. Kody Huffmana generowane są w taki sposób, aby dekoder wczytujący kolejne bity ze strumienia był w stanie rozróżnić, czy wczytane *n* bitów jest już poprawnym kodem, czy należy wczytać jeszcze jeden. Przykładową tablicę kodów dla współczynników DC można zobaczyć w **tab. 2**. Tablice liczby symboli dla danej długości kodu oraz listy ich wartości (symboli, nie kodów – kody musimy wygenerować samodzielnie w dekoderze!), podobnie jak tablice kwantyzacji są zawarte w nagłówku strumienia danych. Warto zauważyć, że kodek JPEG nie może działać „w locie" i w celu wykonania statystyki częstości występowanie symboli cały obraz musi zostać przetworzony do postaci pośredniej.

Tab. 2. Tablica kodów współczynnika DC

Długość	Symbol	Kod
2	0x00	00
3	0x01	010
3	0x02	011
3	0x03	100
3	0x04	101
3	0x05	110
4	0x06	1110
5	0x07	11110
6	0x08	111110
7	0x09	1111110
8	0x0A	11111110
9	0x0B	111111110

Odmiany JPEG

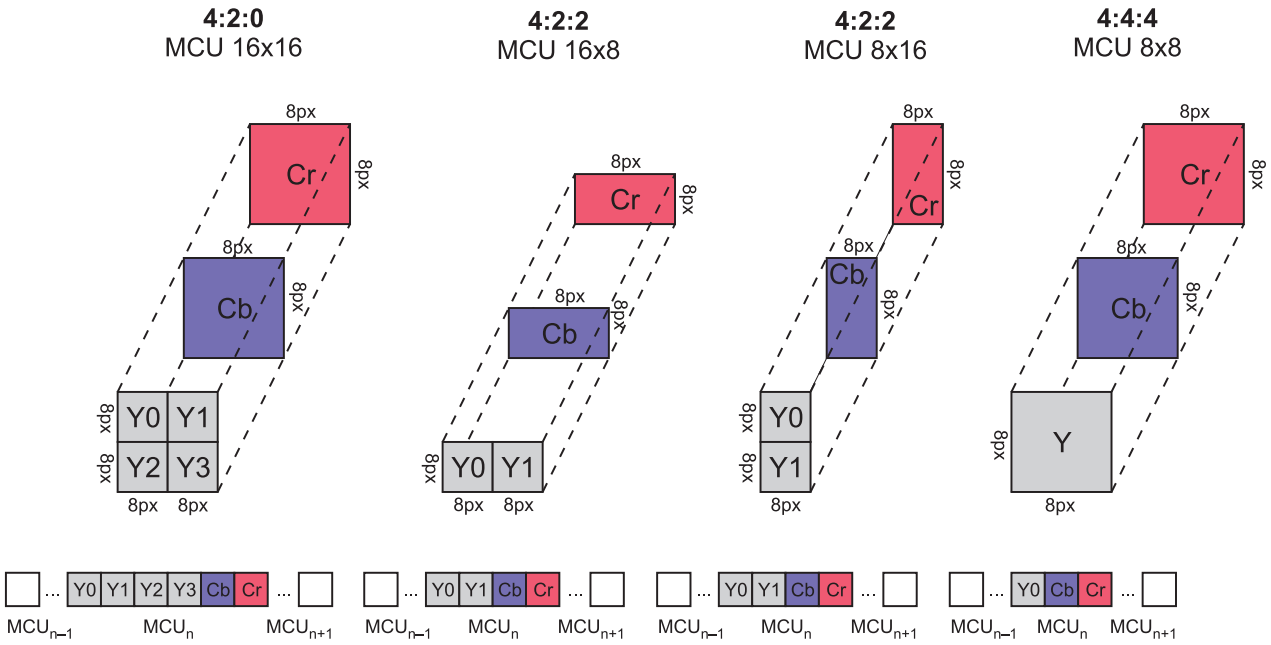
Wyżej opisane ogólne podstawy dotyczą tylko jednego z możliwych wariantów tworzenia plików JPEG – tzw. Baseline DCT. Standard definiuje trzy możliwe sposoby kompresji danych: *Sequential DCT*, *Progressive DCT* oraz *Lossless*.

Jak nazwa wskazuje, trzeci rodzaj jest kompresją bezstratną, która pomija transformatę DCT oraz kwantyzację. Jej efektywność jest niewielka i rzadko stosowana w praktyce. *Progressive DCT* jest rodzajem kompresji, w którym przesyłane są naraz komplety współczynników dla wszystkich bloków w obrazie. Pozwala to na wyświetlenie od razu całej klatki, a następnie wyostrzanie jej i dodawanie kolejnych szczegółów, wraz z napływającymi współczynnikami dla coraz wyższych częstotliwości. Format ten miał rację bytu w czasach bardzo wolnych łączy teleinformatycznych, ponieważ pozwalał na natychmiastowe zaprezentowanie ogólnej zawartości obrazu, podczas gdy dane były ciągle transmitowane. Dziś tę odmianę wykorzystuje się tylko w bardzo specyficznych aplikacjach. *Sequential DCT* przesyła kompletne zestawy współczynników do kolejnych bloków, w kierunku od lewej do prawej, a następnie od góry w dół. Typ ten ma dwie odmiany: *Extended DCT*, który operuje na 12-bitowych próbkach obrazu oraz *Baseline DCT* używający próbek 8-bitowych. *Extended DCT* ma zastosowanie tylko i wyłącznie w aplikacjach specjalnych, naukowych, przemysłowych – jest praktycznie niespotykany w zastosowaniach konsumenckich. Otaczające nas z każdej strony obrazy JPEG są praktycznie w 99 przypadkach na 100 skompresowane w Baseline DCT – generują je wszystkie popularne narzędzia komputerowe, edytory graficzne, przeglądarki plików, konwertery formatów oraz popularne aparaty cyfrowe, telefony komórkowe itp.

Poza powyższym istnieją w standardzie jeszcze dwa sposoby podziału kompresji. Zamiast kodowania Huffmana może zostać użyte kodowanie arytmetyczne. Mimo lepszych o ok. 10% wyników nie jest to rozwiązanie spotykane w praktyce. Powodem jest większe zapotrzebowanie na moc obliczeniową oraz dotyczące go obwarowania patentowe. Jeszcze inny podział pozwala na zapisywanie współczynników AC w sposób różnicowy, tak jak w przypadku DC. Jest to rozwiązanie całkowicie martwe, ze względu na swoją znikomą efektywność. Z powyższych powodów projekt biblioteki skupia się tylko i wyłącznie na podstawowej odmianie JPEG: *Baseline DCT* z kodowaniem Huffmana i bez zapisu różnicowego.

Przestrzeń kolorów

Oryginalny standard JPEG nie narzuca sposobu zapisu obrazów kolorowych. Pozwala on jedynie na zdefiniowanie i przesyłanie dowolnej liczby składowych. Rozwiązanie to jest oczywiście bardzo elastyczne i pozwala na używanie dowolnej przestrzeni barw czy też wygodne zastosowanie w aplikacjach specjalistycznych – jesteśmy np. w stanie wyobrazić sobie specjalną kamerę mającą cztery zmienne filtry na określone długości fali, z których chcemy uzyskać, skompresować i przesłać 4 składowe. W tym przypadku zastosowanie jednego ze standardowych profili kolorów spowoduje utratę lub przekłamanie części informacji. Istniała jednak potrzeba stworzenia formatu pliku – pojemnika dla standardu JPEG, który byłby traktowany w ten sam sposób przez wszystkie kodery oraz dekodery na wszystkich platformach sprzętowych i programowych. Standaryzację wprowadziła propozycja IJG – format JFIF. Poza dodatkowymi nagłówkami zawierającymi takie informacje, jak rozdzielczość DPI czy miniaturka obrazu zapisana w postaci bitmapy, format ten wprowadza rzecz najistotniejszą – ustandaryzowaną przestrzeń barw oraz zdefiniowaną kolejność składowych. Przestrzenią tą jest YCbCr, używana także powszechnie w analogowym standardzie telewizyjnym PAL. Warto wspomnieć, że istnieją profesjonalne aplikacje, w których niezwykle istotne jest zachowanie obrazu w niezmienionej formie RGB. Jednak



Rys. 6. Sposób kompozycji barw w formacie JPEG

standardowe dekodery nie potrafią poprawnie wyświetlić tak stworzonych plików. Zastosowanie przestrzeni YCbCr jest korzystne z punktu widzenia właściwości ludzkiego oka. Ma ono większą zdolność rozdzielczą pod względem różnic jasności sąsiednich punktów niż różnic w ich kolorach. Wyodrębniając osobne płaszczyzny obrazu dla Y – luminancji, oraz Cb i Cr – chrominancji, możemy zmniejszyć rozdzielczość tych drugich nawet 2...4-krotnie przed wykonaniem kompresji. Po rozciągnięciu w dekoderze składowych Cr i Cb z powrotem do docelowego rozmiaru obrazu, różnica dla ludzkiego oka jest praktycznie niezauważalna, przy dużej redukcji ilości danych.

W przypadku JPEG proces ten jest potocznie zwany subsamplingiem koloru. Na **rys. 6** pokazano trzy najczęściej występujące odmiany oraz sytuację bez subsamplingu. Rysunek przedstawia też sposób uszeregowania skompresowany bloków 8×8 dla każdej ze składowych w strumieniu danych. Komplet bloków Y oraz Cr i Cb stanowi tzw. MCU (*Minimal Coded Unit*). Jest to minimalna porcja danych, która pozwala na odtworzenie fragmentu obrazu z uwzględnieniem rekonstrukcji koloru. Na poziomie pojedynczych MCU odbywa się wcześniej wspomniane rozciągnie składowych Cr i Cb. Przykładowo, dla subsamplingu 4:2:0 efektem wynikowym jest blok 16×16 pikseli, gdzie każde 4 tworzące kwadrat piksele mogą mieć różną wartość luminancji, lecz ich chrominancja pozostaje taka sama. Pliki pozbawione subsamplingu generowane są głównie przez narzędzia do edycji i konwersji grafiki wysokiej jakości. Subsampling 16×8 jest najczęściej używany w koderach JPEG aparatów fotograficznych, a 16×16 spotyka się zazwyczaj w plikach o niższej jakości i małym rozmiarze, np. na stronach internetowych. Plik z subsamplinegiem 8×16 może powstać w wyniku operacji bezstratnego obracania obrazu o 90° (bez pośredniej kompresji i dekompresji). Warto zauważyć, że często użytkownik urządzenia lub programu nie ma wpływu na ten parametr.

Projekt biblioteki przewiduje poprawną obsługę wszystkich wyżej wymienionych odmian. Istnieją, co prawda, wersje o współczynnikach proporcji Y:C nawet 4:1, lecz są praktycznie niespotykane na co dzień, dlatego ich implementacja wydaje się zbyteczna. Podobnie wygląda sytuacja w przypadku odmiany mającej tylko składową Y do przechowywania obrazów w skali odcieni szarości. Specyfikacja JFIF poza zdefiniowaniem przestrzeni barw podaje także wzory pozwalające na ujednoliconą konwersję YCbCr <–> RGB, będące prostymi transformacjami liniowymi.

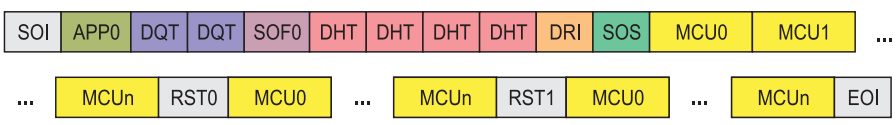
Należy jeszcze wspomnieć o sytuacji, w której wyzmiary obrazu poddanego kompresji nie są wielokrotnością rozmiaru pojedynczego MCU. W takim przypadku obraz musi zostać rozszerzony do właściwych wymiarów, a piksele znajdujące się poza jego prawą i dolną krawędzią zostają wypełnione wartościami z widocznej krawędzi obrazu. Taka technika zapobiega powstawaniu widocznych artefaktów, które nieodłącznie towarzyszą ostrym przejściom kolorów w obrazach JPEG. Dekoder musi obciąć nadmiarowe dane ze zdekodowanych MCU, tak aby rozmiar obrazu odpowiadał zdeklarowanemu w nagłówku strumienia.

Nagłówki danych

Format JPEG pierwotnie zaprojektowano pod kątem przesyłania serii stałych obrazów przez media strumieniowe, takie jak łącza telefoniczne czy satelitarne. JPEG pozwala na jednorazowe przesłanie najważniejszych nagłówków z informacjami o strumieniu, a następnie ciągłą transmisję dowolnej liczby obrazów z uwzględnieniem możliwości ponownego zsynchronizowania się w przypadku wystąpienia błędów transmisji. Interesujący nas format plików JFIF zakłada, że w strumieniu występuje zawsze tylko pojedyncza klatka obrazu. W standardzie JPEG nagłówki nazywane są markerami lub znacznikami. Każdy marker składa się z dwóch bajtów: 0xFF oraz przypisanego mu kodu. Po wszystkich rodzajach markerów (poza SOI, EOI i RSTn) występują dwa bajty precyzujące ilość zawartych w nich danych (z uwzględnieniem samych siebie, ale bez markera). Na **rys. 7** pokazano układ markerów w typowym pliku JFIF. Dokument standardu JPEG bardzo szczegółowo i dogłębnie opisuje ich format, znaczenie i wartości kodów, dlatego poprzestaniemy tutaj tylko na ich pobieżnym opisie (dotyczy on tylko omawianej cały czas odmiany Baseline DCT!).

SOI (Start Of Image) – prosty marker, niezawierający dodatkowych danych, zawsze musi znajdować się na początku pliku.

APPn, n=0...16 (Application Marker) – marker zawierający dane rozszerzające dla danej aplikacji. APP0 jest wykorzystywany przez format JFIF, lecz poza jego zidentyfikowaniem nie ma potrzeby dekodowania go



Rys. 7. Układ markerów w typowym pliku JPEG

w prostych aplikacjach. APP1-2 mogą zawierać dane w popularnym w aparatach cyfrowych formacie EXIF.

DQT (*Define Quantization Table*) – marker definiujący zawartość tablic kwantyzacji. Dekoder musi mieć cztery sloty, w których mogą być umieszczone cztery różne tablice. Jeden marker DQT może zawierać od 1 do 4 tablic, mogą one również mieć postać osobnych markerów. W praktyce kodery tworzą zazwyczaj tylko dwie tablice: jedną dla składowej luminancji i jedną wspólną dla składowych chrominancji.

SOFn (*Start Of Frame*) – marker definiujący parametry tzw. ramki. Dla Baseline DCT jest to marker SOF0, pozostałe definiują omówione wcześniej inne odmiany kompresji. Zawiera liczbę bitów na próbkę, rozmiary obrazu w pikselach oraz definicję składowych. JFIF narzuca tutaj trzy składowe o następujących numerach i kolejności: Y(1), Cb(2), Cr(3). Dla każdej składowej precyzowane są współczynniki subsamplingu oraz numer używanej tablicy kwantyzacji.

DHT (*Define Huffman Table*) – marker definiujący tablice pomocnicze do dekodowania Huffmana. Dekoder musi mieć w sumie cztery sloty, dwa dla tablic kodów DC i dwa dla tablic kodów AC. Tak jak w przypadku DQT, znacznik może zawierać od 1 do 4 tablic, mogą one także mieć postać osobnych markerów.

DRI (Define Restart Interval) – marker definiujący liczbę MCU, co którą wymagana jest obecność znacznika RSTn w strumieniu danych. Technika ta pozwala na ponowną synchronizację dekodera w przypadku, w którym dane w pliku są uszkodzone. Jest ona dość rzadko spotykana, jednak istnieje na rynku przynajmniej jeden popularny program do edycji grafiki korzystający z niej. Większość plików jest go pozbawiona i przyjmuje się, że znaczniki RSTn nie mogą pojawić się w strumieniu.

SOS (Start Of Scan) – marker rozpoczynający strumień zakodowanych danych. Zawiera definicję liczby zawartych w nim składowych, informację o kolejności ich występowania (posługując się numerami przyporządkowanymi w SOF) oraz informację o numerach przyporządkowanych im tablic Huffmana dla składników DC i AC. Bezpośrednio po markerze SOS następuje ciąg zakodowanych danych obrazu – od tego momentu, aż do czasu przesłania wszystkich bloków poza RSTn nie może już pojawić się żaden inny marker.

RSTn, n=0...7 (*Reset*) – marker pojawiający się w strumieniu zakodowanych danych w ściśle zdefiniowanych przez DRI odstępach. Każdy kolejny marker musi mieć numer modulo 8 o jeden większy od poprzedniego.

EOI (*End Of Image*) – marker występujący zawsze na końcu pliku.

Należy zaznaczyć, że wszystkie dane w formacie JPEG zapisywane są w postaci Big Endian, a kolejność występowania markerów nie jest zdefiniowana. Istotne jest jedynie to, aby przed rozpoczęciem skanu SOS pojawiły się wszystkie potrzebne informacje.

Warto również wspomnieć o jeszcze jednym markerze zdefiniowanym w standardzie – DNL. Ma on za zadanie zdefiniowanie liczby linii w obrazie, gdy wartość ta jest równa 0 w markerze SOF. Znacznik DNL powinien wystąpić zaraz po zakończeniu transmisji wszystkich bloków pierwszego skanu. Rozwiązanie to jest praktycznie martwe w przypadku plików. Standard pozwala także na wystąpienie dalszych skanów (markerów SOS wraz z zakodowanymi blokami obrazu) zamiast markera EOI, lecz jest to niespotykane w przypadku Baseline DCT i JFIF. Jedynie wcześniej opisana progresywna odmiana JPEG przesyła w ten sposób kolejne zestawy współczynników dla całego obrazu.

Zakodowane dane podlegają kilku regułom. Ponieważ może w nich wystąpić wartość 0xFF, która jest zarezerwowana tylko dla znaczników, zastosowano technikę tzw. byte stuffingu. Po każdej wygenerowanej wartości 0xFF w ciągu danych umieszczony zostaje dodatkowy bajt 0x00 (zasada ta nie dotyczy danych umieszczonych w markerach!). Dekoder natrafiając na 0xFF sprawdza wartość następnego bajtu i jeśli jest to 0x00, to zostaje on zignorowany. Może to być także jeden ze znaczników RST, lecz w każdym innym przypadku będzie to oznaczało błąd. Ponieważ kodowanie Huffmana operuje na kodach o zmiennej długości, cały strumień danych jest interpretowany w dekoderze bit po bicie, zawsze z najstarszym bitem wczytywanym jako pierwszy, bez zaokrąglania do pełnych bajtów. Zaokrąglenie występuje tylko i wyłącznie w przypadku wystąpienia markerów RSTn – nawet jeśli w bajcie poprzedzającym marker pozostały niewczytane bity, muszą one zostać zignorowane.

Biblioteka

Główny nacisk w projekcie biblioteki dekodującej został położony na prostotę, ograniczenie ilości kodu, zapotrzebowania na pamięć RAM oraz względnie dużą wydajność. Projekt został stworzony w czystym języku ANSI C, bez wykorzystania jakichkolwiek funkcji bibliotecznych. Dzięki temu jest on całkowicie przenośny między platformami sprzętowymi oraz różnymi kompilatorami. Jednak należy zaznaczyć, że testy wykonane zostały tylko na kompilatorach Borland Turbo C++ oraz GCC. Inne bardziej egzotyczne środowiska mogą stwarzać potencjalnie pewne problemy związane np. z przesuwaniem bitowym zmiennych typu *signed* lub mnożeniem liczb. Biblioteka składa się z pliku nagłówkowego *easyjpeg.h* oraz właściwego kodu zawartego w *easyjpeg.c*.

Wykorzystany został fakt, że format JPEG pierwotnie był przeznaczony dla mediów strumieniowych. Biblioteka nie wymaga wczytania całego pliku wejściowego do pamięci – przekazywany jest do niej tylko wskaźnik na dowolną funkcję zdeklarowaną przez użytkownika, pobierającą kolejny bajt strumienia. Takie rozwiązanie pozwala na dekodowanie w locie plików umieszczonych na nośnikach podzielonych na bloki, takich jak karty pamięci. Przykładowa funkcja w razie wysycenia bufora jest odpowiedzialna za wczytanie następnej porcji, np. 512-bajtowego sektora. Jest to operacja całkowicie przeźroczysta z punktu widzenia dekodera i zapewnia dużą uniwersalność.

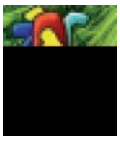
Biblioteka ma własne, zdefiniowane w pliku nagłówkowym zestawy typów prostych oraz wyliczeniowych. Wszystkie typy w bibliotece poprzedzone są prefiksem *ej_* np. *ej_error_t*, *ej_s8*, *ej_u32* itp. Dla nietypowych kompilatorów i architektur może zajść konieczność modyfikacji typów prostych, tak aby odpowiadały założonym rozmiarom bitowym. Dodatkowo zdefiniowany jest prefiks *INLINE*, który może mieć różną postać dla różnych kompilatorów (domyślnie podany dla GCC).

Trzonem biblioteki jest struktura *ej_info* zawierająca wszystkie informacje o aktualnie przetwarzanym pliku. Dokładny opis pól struktury znajduje się w pliku nagłówkowym. Obecna wersja ma jedną globalną strukturę, z której korzystają wszystkie funkcje. Wynika to z założenia, że użytkownik w prostych aplikacjach nie będzie chciał dekodować więcej niż jednego pliku jednocześnie. Podstawową rzeczą, którą należy wykonać przed rozpoczęciem korzystania z biblioteki, jest ustawienie w strukturze wskaźnika **ej_info.get_byte* na wyżej wspomnianą funkcję pobierającą bajty danych. Użytkownik jest sam odpowiedzialny za ustawienie pozycji w swoim medium na początek pliku, który chce przetwarzać. Struktura ma także pole *ej_info.error* zawierające kod ostatniego napotkanego błędu, a lista możliwych błędów jest zdefiniowana w typie wyliczeniowym *ej_error_t*.

Rozpoczęcie przetwarzania pliku następuje poprzez wywołanie funkcji *ej_info()*. Interpretowane są wszystkie markery występujące przed strumieniem zakodowanych danych, a w polach struktury kompletowane są wszelkie potrzebne informacje. W razie wystąpienia błędu funkcja zwróci wartość –1, a w przypadku powodzenia 0. W tym momencie użytkownik może odczytać ze struktury takie informacje, jak rozmiar całego obrazu (*ej_info.ver*, *ej_info.hor*) oraz rozmiar pojedynczego MCU (*ej_info.ver_mcu*, *ej_info.hor_mcu*).

Właściwe dekodowanie następuje dopiero po wywołaniu funkcji *ej_decode(ej_mode_t mode, ej_u?? *bmp)*. Funkcja przyjmuje

Tab. 3. Dostępne tryby i zachowanie się funkcji

Obraz	 Obraz przed kompresją Rozmiar 44×51 MCU 16×16	 Rzeczywisty obraz w pliku JPEG z naniesionym podziałem na MCU Rozmiar 48×64
Tryb dekodowania	**EJ_SINGLE_MCU**	**EJ_LINE_OF_MCU** / **EJ_ALL_MCU**
Bitmapa wejściowa	 16×16	 44×51 /  44×51
1. wywołanie		 / 
2. wywołanie		 / 
3. wywołanie		 / 
4. wywołanie		 / 

jako parametry: tryb działania oraz wskaźnik do bitmapy, a zwraca 0 w przypadku powodzenia lub –1 w przypadku wystąpienia błędu. **Tab. 3** prezentuje dostępne tryby i zachowanie się funkcji przy kolejnych wywołaniach.

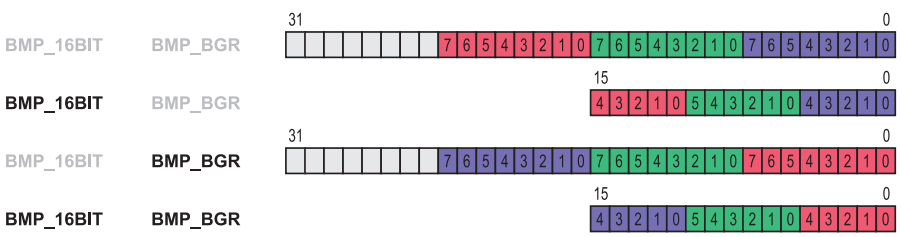
Tryby drugi i trzeci operują na pełnej bitmapie. Funkcji musi zostać przekazany wskaźnik do tablicy zawierającej co najmniej *ej_info.ver * ej_info.hor* elementów. Piksele w rzędach zapisywane są od lewej do prawej, a kolejne rzędy następują po sobie od góry do dołu. W przypadku obrazów, których rozmiar nie jest wielokrotnością rozmiaru MCU, obcięcie nadmiarowej krawędzi następuje wewnętrznie i jest całkowicie przeźroczyste dla użytkownika. Tryb drugi różni się od trybu trzeciego tym, że w jednym wywołaniu wykonuje dekodowanie tylko jednego rzędu kompletnych MCU. Takie zachowanie pozwala np. na zaimplementowanie paska postępu dekodowania i nie blokuje permanentnie głównego kodu programu. W trybie tym funkcja w każdym wywołaniu rozpoczyna zapis danych od tego samego miejsca w tablicy, na którym zakończył się on w poprzednim.

Tryb pierwszy jest preferowany w aplikacjach, które nie mają dostatecznej ilości pamięci RAM, aby zdekodować całą bitmapę w jednym przebiegu (przykładowo dla bitmapy 320×240 jest konieczny bufor o pojemności ponad 300 kB!). Zamiast tego funkcja pracuje na małych bitmapach o rozmiarach odpowiadających jednemu MCU (maksymalnie 16×16, co daje już bufor o rozmiarze 1 kB). Należy pamiętać, że dekoder nie wie niczego o rozmiarach przekazanego mu bufora. Przykładowo dla pliku z MCU 16×8 w każdym wywołaniu zapisanych zostanie pierwszych 128 elementów tablicy (8 wierszy po 16 elementów). Dla MCU leżących na krawędziach obrazu zdekodowane zostaną tylko piksele użyteczne, lecz tablica wyjściowa dalej jest traktowana jako bitmapa o rozmiarach MCU. Piksele leżące za krawędzią obrazu pozostaną niezmienione, dlatego może zajść konieczność wyzerowania tablicy przed jej przekazaniem do funkcji, aby uzyskać czarne wypełnienie tak jak w tab. 3. Nadmiarowe wywołania funkcji, gdy dekodowanie zostało zakończone, są ignorowane, a zawartość przekazywanej tablicy pozostaje niezmieniona. W trybach pierwszym i drugim postęp dekodowania można monitorować za pomocą pól *ej_info.ver_pos* oraz *ej_info.hor_pos*. Zawierają one współrzędne lewego górnego piksela MCU, który będzie dekodowany w następnym wywołaniu *ej_decode(…)*.

Biblioteka ma możliwość konfiguracji formatu generowanej bitmapy. Służą do tego dwie (domyślnie wyłączone znakiem komentarza) definicje umieszczone w plikach nagłówkowych BMP_16BIT oraz BMP_BGR. Pierwsza z nich definiuje, czy bitmapa będzie miała 16-bitowy kolor piksela w formacie 5:6:5 zamiast 32-bitego w formacie 8:8:8. Zależy od niej także typ wskaźnika (*ej_u16** lub *ej_u32**), który jest przekazywany do funkcji *ej_decode(…)*. Druga definicja powoduje odwrócenie kolejności składowych kolorów z RGB na BGR. **Rys. 8** przedstawia wizualizację komórek tablicy (pikseli) dla wszystkich kombinacji ustawień. Format 24-bitowy (3 bajty na piksel) został celowo pominięty, ze względu na małą wydajność i niepraktyczność przetwarzania tego typu danych na architekturach 32-bitowych. Dane 16-bitowe 5:6:5 mogą być bardzo wygodne w stosowaniu z małymi wyświetlaczami TFT, których kontrolery zazwyczaj przyjmują bitmapy właśnie w takim formacie.

Szczegóły implementacji, testy wydajności

Podstawowy element biblioteki, czyli transformata IDCT, został opracowany na podstawie jednej z implementacji stałoprzecinkowych dostępnych w bibliotece IJG. Opiera się ona na popularnym algorytmie AAN (nazwa pochodząca od nazwisk autorów: Arai, Agui i Nakajima). Jego główną zaletą jest fakt, że do wykonania jednowymiarowej, 8-punktowej transformaty potrzebnych jest tylko pięć mnożeń i ok. 40 dodawań/odejmowań. Uzyskano to poprzez „wyprowadzenie" z transformaty prawie wszystkich współczynników stałych i zintegrowanie ich z macierzą kwantyzacji, przez którą i tak muszą zostać wymnożone dane wejściowe. W celu wykonania transformaty dwuwymiarowej przeprowadzane jest złożenie transformat jednowymiarowych – najpierw przetwarzane są kolumny, a następnie



Rys. 8. Komórki tablicy pikseli

Tab. 4. Przykłady pomiarów czasu dekodowania i wyświetlania obrazów

Obraz					
Rozmiar	640×480 123 KB	640×480 20,5 KB	320×240 17,9 KB	125×183 17,8 KB	2048×1536 883 KB
Ustawienie jakości	95/100	50/100	95/100	nieznane, obraz z WWW	nieznane, obraz z aparatu fotograficznego
Czas inicjalizacji	PC: <1 ms ARM: 10 ms	PC: <1 ms ARM: 10 ms	PC: <1 ms ARM: 10 ms	PC: <1 ms ARM: 14 ms	PC: <1 ms ARM: 14 ms
Czas dekodowania	PC: 109 ms ARM: 1777 ms	PC: 47 ms ARM: 796 ms	PC: 16 ms ARM: 359 ms	PC: 7 ms ARM: 116 ms	PC: 391 ms ARM: 12839 ms

wiersze. W implementacji biblioteki proces wymnażania współczynników transformaty i macierzy kwantyzacji przebiega w funkcji *ej_init()*, a wynik jest zapisywany bezpośrednio w polach struktury *ej_info.qt[]*. Kosztem takiego rozwiązania jest dwukrotnie większa wielkość zajmowanej przez nie pamięci, ponieważ wymnożone dane potrzebują co najmniej typu 16-bitowego. Większość spotykanych plików JPEG korzysta tylko z dwóch tablic kwantyzacji i dlatego można ograniczyć liczbę slotów poprzez definicję *QT_CNT* umieszczoną w pliku nagłówkowym. Powoduje to ograniczenie użycia pamięci o ok. 260 B. Procedura IDCT wykonuje obliczenia na danych o precyzji 5 bitów po przecinku (precyzja jest zwiększana poprzez odpowiednio mniejsze przesunięcie bitowe po mnożeniu tablicy kwantyzacji). Z właściwości transformaty wynika zwiększenie precyzji o kolejne 2 bity, dlatego ostateczne wyniki są dzielone poprzez przesunięcie o 7 bitów w prawo.

Biblioteka powinna dać się łatwo modyfikować do nietypowych aplikacji, np. o innej przestrzeni kolorów lub nietypowych współczynników subsamplingu. Podstawą jest funkcja *decode_unit()*, która dekoduje ze strumienia kolejny blok obrazu 8×8 i zwraca go w postaci 64-elementowej tablicy o wartościach 16-bitowych ze znakiem. Wartości nie mogą być 8-bitowe, ponieważ drobne błędy wynikające z precyzji obliczeń IDCT nakładają się i w efekcie niektóre współczynniki wychodzą poza zakres jednego bajtu. Podobne efekty mogą powstać w funkcji *ycc2rgb(…)*, która dokonuje konwersji przestrzeni kolorów, dlatego konieczne jest przeprowadzenie tzw. saturacji do 8 bitów. Odbywa się ona właśnie w *ycc2rgb()* i niweluje wszystkie skumulowane wyjścia poza zakres. Właściwe składanie obrazu z płaszczyzn Y Cb i Cr odbywa się już w funkcji *ej_decode(…)*. Dla każdego typu subsamplingu powstał oddzielny fragment prostszego kodu, niż byłoby to w przypadku funkcji uniwersalnej.

Testy, podsumowanie

W celu przetestowania biblioteka została wstępnie zaimplementowana w środowisku Borland Turbo C++, na komputerze PC z procesorem Intel Core Duo 1,66 GHz. Prosta aplikacja konsolowa (załączona do artykułu wraz z kodem biblioteki) przyjmuje jako argument wywołania ścieżkę do pliku *.jpg, wyświetla informacje na jego temat, czasy trwania wywołania funkcji *ej_init()* i *ej_decode(EJ_ALL_MCU, …)*, a następnie generuje plik bitmapy *.bmp ze zdekodowanym obrazem. Drugi test został przeprowadzony w środowisku GCC 4.3 na procesorze ARM Cortex M3, zaimplementowanym w pewnym urządzeniu z wyświetlaczem TFT 320×240, nad którym pracuje autor. Procesor był taktowany zegarem 72 MHz. Wyniki testów przedstawiono w **tab. 4**. Prędkość dekodowania jest mocno zależna nie tylko od rozdzielczości obrazu, ale także od stopnia jakości (kwantyzacji), z jakim został wygenerowany oraz jego szczegółowości. Różnica prędkości dekodowania między PC a ARM mieści się w granicach od 15 do 30 razy, co względnie odpowiada rzeczywistej różnicy częstotliwości zegarów obu procesorów.

W przypadku kompilacji GCC z optymalizacją – O2 kod biblioteki zajmował **ok. 6,5 kB**, a zapotrzebowanie na pamięć RAM wyniosło **ok. 1,5 kB** pamięci zajmowanej wprost (struktura *ej_info*) oraz **ok. 1 kB** pamięci zajmowanej lokalnie poprzez zmienne i tablice umieszczane na stosie (nie można o tym zapominać!). Kod źródłowy to ok. 800 linii programu oraz 200 linii pliku nagłówkowego (wliczając komentarze).

Projekt biblioteki wydaje się być udany pod względem funkcjonalnym, choć można mieć zastrzeżenia co do jej wydajności. Nie potrzeba specjalnych pomiarów, aby zauważyć, że dekodowanie tych samych obrazów w niektórych przeglądarkach obrazów na PC trwa krócej niż wywołanie programu demonstracyjnego. Wynika to zapewne z wykorzystania rozszerzeń MMX i silnej optymalizacji kodu pod procesory x86.

Biblioteka na pewno stanowi dobrą bazę do wykonania w przyszłości podobnej optymalizacji np. pod ARM Cortex M3 bądź procesor DSP. Przytoczone w artykule informacje powinny ułatwić to zadanie bardziej zaawansowanym Czytelnikom, a także pozwolić na własne modyfikacje i ulepszenia. Jednakże autor zaleca, aby przed zagłębieniem się w kod biblioteki zapoznać się, choćby pobieżnie ze standardem ITU. Bardzo pomocne w zrozumieniu zasad kodowania Huffmana i generowania tablic pomocniczych mogą być również artykuły z przytoczonej na początku strony internetowej.

Kod biblioteki został udostępniony na zasadach prostej licencji w stylu MIT/X11, co oznacza, że może być wykorzystywany w dowolnych celach zamkniętych lub otwartych, także komercyjnych, pod warunkiem umieszczenia treści licencji w wynikowym kodzie (jeśli jest on otwarty) lub w dokumentacji oprogramowania korzystającego z biblioteki.

Michał Wysocki
mos.wysocki@gmail.com