OSTRAVSKÁ UNIVERZITA V OSTRAVĚ
PŘÍRODOVĚDECKÁ FAKULTA
KATEDRA INFORMATIKY A POČÍTAČŮ

Zpracování a komprese obrazu založená na platformě Arduino

DIPLOMOVÁ PRÁCE

Autor práce: Bc. Jan Lazar
Vedoucí práce: RNDr. Martin Kotyrba, Ph.D.

2016

UNIVERSITY OF OSTRAVA
FACULTY OF SCIENCE
DEPARTMENT OF INFORMATICS AND COMPUTERS

Processing and image compression based on the platform Arduino

DIPLOMA THESIS

Author: Bc. Jan Lazar
Supervisor: RNDr. Martin Kotyrba, Ph.D.

2016

ČESTNÉ PROHLÁŠENÍ

Já, níže podepsaný/á student/ka, tímto čestně prohlašuji, že text mnou odevzdané závěrečné práce v písemné podobě i na CD nosiči je totožný s textem závěrečné práce vloženým v databázi DIPL2.

Ostrava dne 22. 4. 2016

……
podpis studenta/ky

OSTRAVSKÁ UNIVERZITA V OSTRAVĚ
Přírodovědecká fakulta
Akademický rok: **2014/2015**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Jan LAZAR**
Osobní číslo:	**P14077**
Studijní program:	**N1801 Informatika**
Studijní obor:	**Informační systémy**
Název tématu:	**Zpracování a komprese obrazu založená na platformě Arduino**
Zadávající katedra:	**Katedra informatiky a počítačů**

Zásady pro vypracování:

1. Principy a možnosti komprese obrazu
2. Platforma Arduino (hardware, IDE)
3. Možnosti komprese založené na platformě Arduino
4. Návrh architektury a softwarového řešení pro zpracování a kompresi obrazu
5. Experimenty a reálná ověření, závěry

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

ČAPEK, Jan a Peter FABIAN. Komprimace dat: principy a praxe : principy a metody komprimace, ztrátové i bezeztrátové algoritmy, obrázky, animace, video, hudba a zvuk, formáty JPEG, MPEF a mnoho dlaších. Vyd. 1. Praha: Computer Press, 2000, viii,173 s. ISBN 80-7226-231-9.

SOBOTA, Branislav a Ján MILIÁN. Grafické formáty. 1. vyd. České Budějovice: Kopp, 1996, 157 s. ISBN 80-85828-58-8.

MURRAY, James Dickson. Encyklopedie grafických formátů. Praha: Computer Press, 1995. ISBN 80-85896-18-4.

ŽÁRA, Jiří. Moderní počítačová grafika. 2., přeprac. a rozš. vyd. Brno: Computer Press, 2004, 609 s., 16 s. obr. příl. ISBN 80-251-0454-0.

Vedoucí diplomové práce: **RNDr. Martin Kotyrba, Ph.D.**
Katedra informatiky a počítačů

Datum zadání diplomové práce: **3. listopadu 2014**
Termín odevzdání diplomové práce: **22. dubna 2016**

RNDr. Martin Kotyrba, Ph.D.
vedoucí diplomové práce

Doc. Ing. Cyril Klimeš, CSc.
vedoucí katedry

V Ostravě dne 3. listopadu 2014

ABSTRAKT

Diplomová práce se zabývá kompresí a dekompresí rastrového obrazu na vývojové platformě Arduino. Byla vytvořena přídavná deska (shield) s externí pamětí, kompatibilní s 8 a 32 bitovou verzí vývojové platformy. Dále bylo vytvořeno softwarové řešení, které se stará o obsluhu přídavné paměti a zpracování obrazu ve formátu BMP a JPEG. Celé řešení bylo experimentálně ověřeno na obrazech s různými vlastnostmi a s různým rozlišením.

Klíčová slova: Arduino, JPEG, BMP, komprese obrazu, dekomprese obrazu

ABSTRACT

The diploma thesis is about compression and decompression of raster image on the Arduino development platform. It was created additional circuit board (shield) with external memory, compatible with 8 and 32-bit versions of the development platforms. It was also created a software solution that takes care of the servicing of additional memory and processing images in BMP and JPEG format. The whole solution was experimentally verified in images with different properties and different resolutions.

Keywords: Arduino, JPEG, BMP, image compression, image decompression

PODĚKOVÁNÍ

Děkuji svému vedoucímu diplomové práce RNDr. Martinu Kotyrbovi, Ph.D. za cenné připomínky, pomoc, odborné vedení a trpělivost při vypracovávání diplomové práce. Dále děkuji své rodině a přátelům za podporu během studia.

Prohlašuji, že předložená práce je mým původním autorským dílem, které jsem vypracoval/a samostatně. Veškerou literaturu a další zdroje, z nichž jsem při zpracování čerpal/a, v práci řádně cituji a jsou uvedeny v seznamu použité literatury.

V Ostravě dne 22. 4. 2016

. . . . . .

Jan Lazar

OBSAH

ÚVOD

Již nějakou dobu se zabývám programováním mikro kontrolérů na platformě Arduino. Na internetu se dá najít velká spousta rozšíření pro desky Arduino, ať už se jedná o moduly s tlačítky, ovladače motorů, displeje a jiné. Začal jsem se zabývat prací s barevnými LCD displeji, které lze k Arduino také připojit. Obsluha LCD displeje zabírá hodně programové a operační paměti, ale lze jej pro zobrazování použít. Proto jsem se rozhodl, že si udělám řešení, které by umožnilo na platformě Arduino zobrazit obrázek, popřípadě ho převést do jiného formátu.

Hlavním stěžením je problematika uložení dat v souborech (grafické soubory se liší uložením či použitou kompresí) a také nemalé nároky na výkon a paměť. Arduino samo o sobě obsahuje mikro kontrolér, který neobsahuje velké množství paměti ani výkonu, proto jsou některé operace problematické nebo náročné na čas.

První část práce se zabývá platformou Arduino, konkrétně hardwarem jednotlivých vývojových desek, programovacím jazykem či prostředím, ve kterém lze pro Arduino programovat. V druhá část obsahuje informace o grafických formátech BMP a JPEG. V třetí části je popsán návrh hardwaru a softwaru pro komprimaci a dekomprimaci BMP a JPEG, obsluha přídavné paměti a potřebných periferií. Závěr práce se zabývá otestováním celého řešení, zhodnocením daného řešení, možnostmi využití a případným vylepšením.

1 CÍLE PRÁCE

Cílem této práce je vytvořit ucelené řešení pro kompresi a dekompresi rastrového obrazu na vývojové platformě Arduino. Řešení bude obsahovat hardwarové řešení v podobě přídavné paměti, softwarové řešení pro zpracování obrazu a obsluhu přídavné paměti. Podporován bude formát souboru BMP s podporou základní komprese a sekvenční JPEG s barevnou hloubkou 24 bitů. Celé hardwarové a softwarové řešení bude odzkoušeno na 8bitové a 32bitové vývojové verzi platformy Arduino.

2 VÝVOJOVÁ PLATFORMA ARDUINO

Arduino je elektronická vývojová deska a vývojové prostředí, na které se vztahuje open-source licence. Tento projekt se díky zveřejněnému schématu zapojení stal velice populární, jelikož si lidé mohou vývojové desky volně upravovat, zvelebovat, přidávat nové možnosti, nebo desky naopak minimalizovat. Za posledních pár let bylo díky tomuto přístupu vytvořeno mnoho rozšiřujících desek hardwaru (shieldy), a tím se zvýšila možnost využití. Arduino se pomalu dostává také do výuky mnohých škol, díky nízké pořizovací ceně, jednoduché rozšiřitelnosti, jednoduchému programovacímu jazyku a internetovému fóru, kde je řešeno mnoho projektů a také případné neduhy hardwaru [1].

2.1 Arduino jako nástroj komprese a dekomprese obrazu

Ačkoli je vývojová platforma v poslední době velice populární mezi mladými elektrotechniky a lidmi, kteří se zajímají o programování mikro kontrolérů nebo řízení, je velice problematické najít řešení, kde je řešena problematika komprese obrazu.

Na rozsáhlém oficiálním fóru je pár myšlenek ohledně komprese či použití v této oblasti, ale většinou jsou zamítnuty z důvodu malé paměti nebo malého výkonu mikro kontroléru. Ohledně formátu BMB jsou dohledatelné i části kódů, které byly aplikovány na platformě Arduino, ale většinou nepodporují žádnou kompresi, pouze oddělují hlavičku od dat. V případě grafického formátu JPEG je hledání ještě komplikovanější.

Ačkoli existuje minimalistická knihovna pro dekompresi *picojpeg*, kterou zpracoval Rich Geldreich [2], na platformu Arduino stále portována nebyla. Autor tuto knihovnu napsal pro staré počítače v jazyce C/C++, v dokumentaci je uveden záznam, že knihovna byla přizpůsobena pro vývojovou platformu mikro kontrolérů MSP430[1], ta se ale hodně liší od platformy Arduino.

Andy Brown [3] na svém osobním webu přizpůsobil knihovnu *picojpeg* tak, aby mohl otevřít JPEG soubor načtený z Flash paměti nebo z proudu dat skrze sériovou linku. Obraz zobrazoval na TFT displeji, který byl používán v mobilním telefonu Nokia N82. Bohužel celá konstrukce řešení není lehce modifikovatelná na řešení, které by umožňovalo jinou manipulaci s obrazem, než jen zobrazení na displeji.

[1] MSP430 je řada 16 bitových mikro kontrolérů od společnosti Texas Instruments

2.2 Hardware

Vývojová deska Arduino obsahuje pouze základní věci, které programátor potřebuje. Na desce nalezneme mikroprocesor, základní součástky, které jsou potřebné pro jeho funkci (krystal, rezistory, kondenzátory), konektory pro připojení externích periferií, napájecí konektor se stabilizátorem napětí, USB konektor pro připojení k počítači. K USB konektoru je připojen virtuální sériový port FT232 nebo ATmega8U2/ATMega16U2, který se stará o emulaci sériového portu, jelikož v mikroprocesoru s osmibitovou architekturou není zakomponován řadič USB, ale pouze sériová linka. U verze s architekturou ARM je nativně zakomponován USB řadič, ale pro komunikaci a programování bývá osazen i převodník připojený na sériovou linku. Pro signalizaci komunikace a přítomnosti napájení jsou na deskách většího rozměru LED diody [1].

Díky malému výkonu osmibitových mikro kontrolérů, se komunita, která se stará o vývoj a podporu platformy Arduino, rozhodla vyvinout desky s 32bitovou architekturou ARM. Díky tomuto řešení vznikla vývojová deska Arduino DUE a menší Arduino Zero. Dosti velkým problémem se stala kompatibilita rozšiřujících desek, jelikož nové desky nejsou tolerantní k TTL logice využívající napětí 0 až 5 V, z čehož vyplývá, že shieldy, které vývojář použil u desek s osmibitovou architekturou, u architektury ARM nemůže využít.

2.2.1 Arduino Mega 2560

Arduino Mega 2560 vychází z desky Arduino Mega, která se řadí mezi starší desky, ale díly veliké popularitě a velkému množství I/O pinů je hojně využíváno.

Srdcem této vývojové desky je osmibitový procesor AVR od firmy Atmel. Tyto procesory disponují redukovanou instrukční sadou RISC s harvardskou architekturou. Tato deska byla standardně osazována dvěma typy mikroprocesorů. Nejstarším z nich byl mikroprocesor ATmega1280 (označení Arduino Mega), který byl později nahrazen mikroprocesorem ATmega2560. Tyto procesory se většinou liší jen velikostí paměti pro data a program.

Tabulka 1 Základní vlastnosti mikro kontroléru ATMega2560

Vlastnost	**Hodnota**
Napájecí napětí	5 V
Počet I/O pinů	86 (díky zapojení použitého na desce je využitelných pouze 70)
Počet analogových vstupů	16
Počet PWM[2] výstupů	15
Zatížitelnost pinu	při 5 V až 20 mA, při 3,3 V až 50 mA
Paměť Flash	256 KB, z níž 8 KB zabírá zavaděč
Paměť SRAM	8 KB
Paměť EEPROM	4 KB
Pracovní frekvence	Až 16 MHz
Rozhraní	TWI, SPI, USART

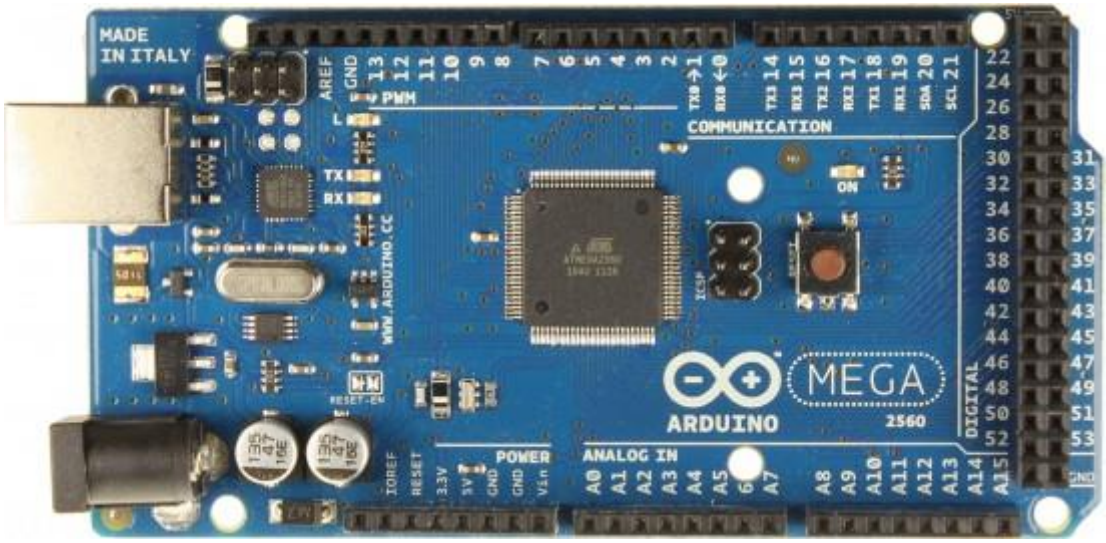
Vývojová deska Arduino Mega 2560 má rozměry 101,5 mm x 53,3 mm. Obsahuje pouze základní součástky a konektory pro připojení periferií.

Deska prošla již třemi revizemi. Největší změny, které byly provedeny, jsou:

- záměna převodníku FTDI za ATMega8u2/ATMega16u2,
- rozšíření lišty s piny pro zajištění budoucí kompatibility s deskami s 3,3V logikou.

Deska Arduino Mega 2560 je hojně využívána jako srdce 3D tiskáren.

[2] Pulse Width Modulation



Obrázek 1 Vývojová deska Arduino Mega 2560 [4]

2.2.2 Arduino Due

Arduino Due je první vývojová deska společnosti Arduino osazená 32bitovým procesorem, konkrétně jde o Atmel SAM3X8E ARM Cortex-M3. Rozložení I/O pinů je oproti Arduino Mega 2560 lehce pozměněno. Velkou změnou je použití napětí 3,3 V. Proto není možno s Arduino Due použít shieldy kompatibilní se staršími deskami. Při použití 5 V může dojít k poškození mikroprocesoru.

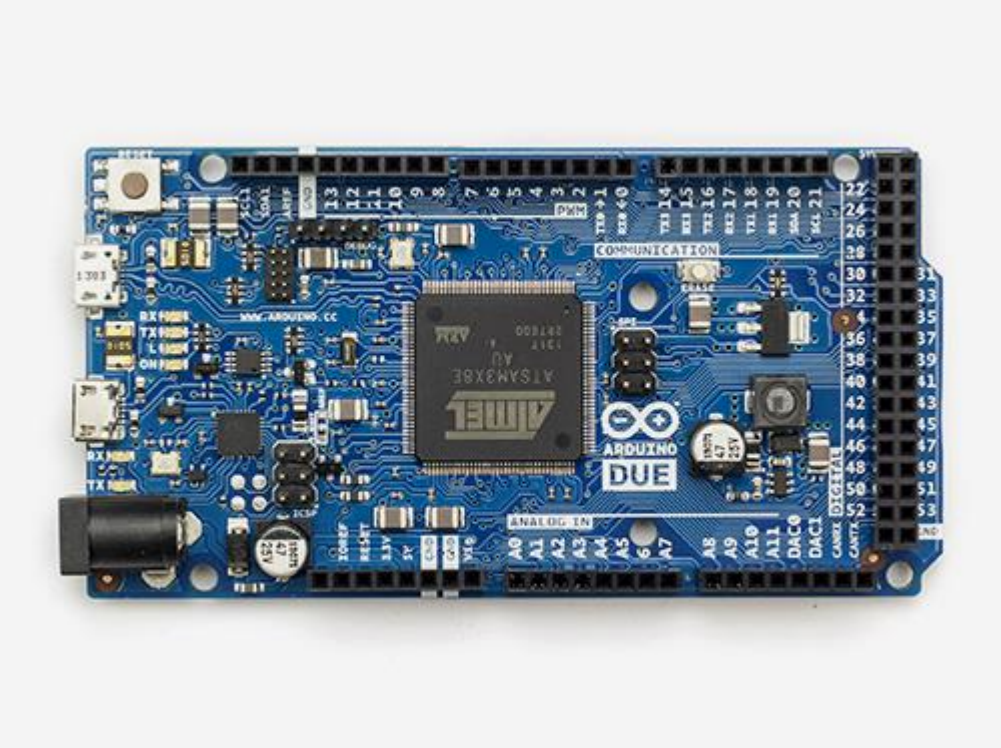
Jak už bylo zmíněno, vývojová deska je osazena mikroprocesorem Atmel SAM3X8E ARM Cortex-M3. Procesory z rodiny Cortex-M3 jsou určeny do levných embedded zařízení, kde je požadován vysoký výkon a vysoká efektivita. Tento mikroprocesor se výborně hodí do oblasti průmyslové automatizace, ovládacích systémů či do automobilového průmyslu.

Tabulka 2 Základní vlastnosti mikro kontroléru AT91SAM3X8E

Vlastnost	**Hodnota**
Napájecí napětí	3,3 V
Počet I/O pinů	103 (díky zapojení na desce je využitelných pouze 73)
Počet analogových vstupů	12
Počet analogových výstupů	2

Počet PWM výstupů	12
Paměť Flash	512 KB
Paměť SRAM	96 KB
Pracovní frekvence	Až 84 MHz
Rozhraní	USB, Ethernet, CAN, TWI, SPI, USART

Vývojová deska Arduino Due má rozměry 101,5 mm x 53,3 mm. Obsahuje pouze základní součástky a konektory pro připojení periferií.



Obrázek 2 Vývojová deska Arduino Due [5]

2.3 Software

2.3.1 Arduino IDE

Vývojové prostředí pro Arduino je multiplatformní, takže není žádný problém programovat na jiném operačním systému než ve Windows. Prostředí je velice jednoduché, neobsahuje žádné speciální funkce, jen základní knihovny pro programování základních

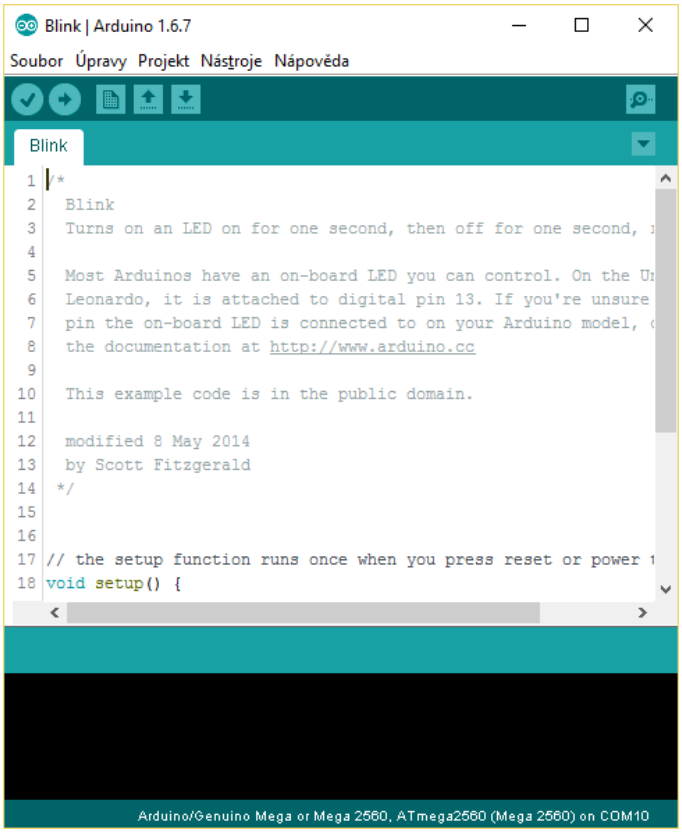
desek Arduino a také základní knihovny pro některé shieldy. To ale neznamená, že se další knihovny do prostředí nedají přidat. Stačí je vložit do patřičných složek od programu.

Vývojové prostředí je psáno v jazyce JAVA, ale jedná se o mírně přepracované prostředí projektu Processing, který se používá pro vizualizaci a řízení mikroprocesorového hardwaru přímo z počítače. V prostředí najdeme jen několik tlačítek a nabídku pro práci se souborem, výběr desky, záložku sketch - pod tímto názvem se skrývá zdrojový kód pro platformu Arduino, a záložka pomoc. V prostředí je zakomponován monitor sériového portu pro základní komunikaci při vývoji a stavová konzole informující o chybách, kompilaci a velikosti kódu. I když je toto prostředí napsáno v jazyce JAVA, díky knihovně třetí strany podporuje sériový port.

Programovací jazyk, který čerpá ze základů jazyka Wiring, který byl původně navržen pro programování vývojové desky AVR, je hodně podobný jazykům C a C++. Stavba zdrojových souborů od knihoven odpovídá strukturám těchto jazyků. V Arduino IDE lze při použití patřičné direktivy psát kód v jazyce C nebo C++, jelikož celý kód se při kompilaci převádí do jednoho z těchto jazyků. V útrobách tohoto vývojového prostředí najdeme standardní kompilátor běžně používaný pro kompilaci zdrojových kódu pro mikroprocesory rodiny AVR a ARM.

Velkou výhodou tohoto programovacího jazyka je jeho jednoduchost. Začátečník nemusí vědět příliš informací o architektuře mikroprocesoru, zacházení s jednotlivými registry a problematikou ohledně zacházení s pamětí. Programátoři v tomto prostředí budou marně hledat funkci našeptávání, která v tomto prostředí schází. Zvýrazňování příkazů, které se nacházejí v implementovaných knihovnách, je samozřejmost.

Jednou z možných nevýhod tohoto programovacího jazyka je velikost kódu pro kompilaci, často bývá o hodně větší a pomalejší než stejný kód v jazyce C či C++, avšak pro naučení základu programování mikroprocesorů, které nepotřebuje příliš vysokou přesnost, je to přijatelný kompromis [1].



Obrázek 3 Pracovní prostředí Arduino IDE

2.3.2 Visual Micro plugin pro Microsoft Visual Studio

Visual micro plugin je rozšíření pro Microsoft Visual studio, díky kterému se programování pro Arduino a podobné jazyky, které z jazyka Arduino vycházejí, stává pohodlnější a hlavně přehlednější.

Plugin je možno používat bezplatně pro domácí použití, bez debuggeru, ostatní varianty použití jsou zpoplatněny. Toto rozšíření je možno nainstalovat do MS Visual Studia 2012 a novějších verzí. Poté stačí nastavit výchozí složku, kde je Arduino IDE nainstalováno a celé prostředí je nakonfigurováno.

Mezi největší výhody rozšíření je našeptávání, se kterým je programování rychlejší a pohodlnější, a správa projektu, kterou v Arduino IDE nenalezneme. Díky tomu, že prostředí MS Visual Studia je odladěné a není psáno v jazyce JAVA, je realizace změn daleko rychlejší.

Obrázek 4 Hlavní nabídka MS Visual Studio s rozšířením Visual Micro

3 KOMPRESE RASTROVÉHO OBRAZU

Rastrové obrazy se vyznačují vysokou paměťovou náročností, která roste kvadraticky s jejich rozlišením. Kompresi obrazů je tedy po právu věnována značná pozornost. Na rozdíl od komprese obecných souborů lze vycházet z vlastností a charakteristických rysů konkrétního rastrového obrazu. Nejprve popíšeme základní kompresní metody pro rastrové obrazy a poté i některé formáty. Počet grafických rastrových formátů je překvapivě vysoký, ačkoliv jejich společným cílem je ve většině případů pouze úschova dvourozměrného pole pixelů, reprezentujících obrázek. Existence mnoha formátů má několik příčin:

- Historické důvody - formáty odrážejí technický vývoj, zejména postupně se zvyšující barevné možnosti zařízení jak pro snímání obrazu, tak pro jeho zobrazení,
- Vazba na program - podle druhu aplikace vznikaly specializované formáty, například pro úschovu skic a kreseb (PCX), černobílých dokumentů (TIFF) či pro přenos barevných fotografií a jejich prezentaci na WWW (GIF, JPEG),
- Technické důvody - mnoho formátů bere ohled na rozlišení skenerů, pomocí kterých jsou obrazy zaznamenávány, na obrazová rozlišení v různých osách, na odlišné architektury obrazové paměti v grafických kartách, či na interpretaci dvojice bytů v jednom 16bitovém slově – některé procesory ukládají do bytu s nižší adresou nižší řády 16bitového čísla (little endian), jiné je ukládají do bytu s vyšší adresou (big endian).
- Metoda komprese - vzhledem k velkému paměťovému objemu barevných obrazů je žádoucí uchovávat obraz v komprimované podobě. Volba vhodné kompresní metody je často závislá na charakteru obrazu a na jeho dalším použití. Základní dělení rozlišuje kompresní metody na ztrátové (lossy) a bezeztrátové (lossless).

Uvedený přehled nezaznamenává zcela všechny důvody existence široké škály obrazových formátů. Existuje celá řada dalších aplikačních požadavků, jakými je například uložení více obrázků v jednom souboru, zápis sekvence obrazů pro animaci, multimediální obrázky rozšířené o informace potřebné pro interakci (oblasti zájmu), vícerozměrné snímky z medicínských aplikací (prostorové řezy MRl, CT), prokládání při přenosu po síti s cílem ukázat obrázek již po přenesení části dat (interlacing), uložení jednoho obrazu

v několika stupních kvality (multiresolution), přidávání poznámek (anotací) a dalších informací týkajících se daného obrazu (metadata) apod.

Většina formátů je schopna uchovávat obrázky jak černobílé, tak barevné. Zvyšující se počet barev vede pochopitelně k větším paměťovým nárokům. Základní typy barevných obrazů ukládaných v souborech jsou uvedeny v následující tabulce:

Tabulka 3 Základní typy barevných obrazů [6]

Obrázek		**bit/pixel**	**poznámka**
černobílý	B/W	1	délka řádku zaokrouhlována na celé byty, (padding)
v odstínech šedi	gray scale	8	někdy jen 6 bitů
s paletou	palleted	8	256 barev v paletě, někdy jen 128, 64,…
plně barevný	color	24	RGB, CMY, YUV, YCBCR
		32	RGBA, CMYK
s vysokým dynamickým rozsahem	HDR	48	až 96 bitů na pixel

Mezi další varianty patří tzv. high color obrázky s přímým barevným kódováním barevných složek po 5 bitech (16bitové slovo na pixel) či paletované obrázky s kódováním CMYK. Velký objem dat a zároveň specifický tvar obrazových informací jsou podnětem pro používání různých druhů kompresí. Pokud zmenšíme objem dat tak, aby informace zůstala nezměněna, hovoříme o bezeztrátové kompresi. Bezeztrátová komprese nedosahuje takové úspory paměti jako komprese ztrátová, při níž se odstraňuje informace, která není příliš významná. Pokud například posloupnost 77876778778 nahradíme řadou ze samých sedmiček, ztratíme část informace, ale novou posloupnost budeme moci komprimovat podstatně lépe. V počítačové grafice je důležitá tzv. psychovizuální redundance. Označuje tu část informace, jejíž nepřítomnost nepostřehneme, a proto ji můžeme zanedbat.

Tabulka 4 Přehled vlastností kompresních metod [6]

Kompresní metoda	**Zkratka**	**Ztrátová**	**Příklad formátu**
Run length encoding	RLE	Ne	PCX (ZSoft PaintBrush)
Huffmanovo kódování	CCITT	Ne	TIFF
Slovníkové kódování	LZW	Ne	GIF, PNG, ZIP, ARJ
Diskrétní kosinová transformace	DCT	Ano	JPEG

Poznamenejme, že u všech obrazových formátů se komprese týká pouze vlastních obrazových dat. Hlavička souboru, definice palety a další doplňující informace se nekomprimují.

3.1 Rastrový formát BMP

Bitmapové soubory Windows jsou uloženy v (DIB) formátu tj. formát nezávislý na zařízení, který umožňuje Windows zobrazit bitmapy na jakémkoli zobrazovacím zařízení. Termín „device independent“ značí, že bitmapy specifikují barvu pixelu v nezávislé formě od metod používaných na zobrazování barev. Implicitní přípona Windows DIB souboru je *.BMP [7].

3.1.1 Struktura BMP souboru

Každý soubor BMP obsahuje hlavičku BMP souboru, hlavičku s informacemi, tabulku barev (nemusí být použita) a samotné pole bytů, které určují bitmapu.

Tabulka 5 Struktura souboru BMP [7]

Název	**Význam**
BITMAPFILEHEADER	Struktura obsahující informace o souboru BMP
BITMAPINFOHEADER	Struktura obsahující informace o obraze v uloženém soboru
RGBQUAD[0]	0. položka barevné palety – poměr mezi složkami RGB
RGBQUAD[1]	1. položka barevné palety – poměr mezi složkami RGB

…	
RGBQUAD[N]	N. položka barevné palety – poměr mezi složkami RGB
BITS	Samotné obrazové údaje

3.1.1.1 Hlavička souboru BMP - BMPFILEHEADER

Hlavička souboru obsahuje informace o typu a velikosti tohoto souboru. Hlavička souboru je definována strukturou v následující tabulce.

Tabulka 6 Hlavička souboru BMP [7]

Délka	Název	Význam
2 byty	bfType	Identifikátor formátu BMP – obsahuje znaky „BM“
4 byty	bfSize	Velikost souboru s obrazovými údaji (typ long)
2 byty	bfReserved1	Rezervované – nastavené na 0
2 byty	bfReserved2	Rezervované – nastavené na 0
4 byty	bfOffbits	Udává posun struktury BITMAPFILEHEADER od začátku souboru (typ long)

3.1.1.2 BMP INFO hlavička – BITMAPINFOHEADER

Info hlavičky souborů BMP se liší dle verzí. Novější verze zpravidla obsahuje starší verzi a přidává k ní nové údaje. Momentálně se můžeme ve Windows setkat s následujícími verzemi:

- BITMAPINFOHEADER – délka 40 bytů, podpora 16 a 32 bitů na pixel, RLE,
- BITMAPV4HEADER – délka 108 bytů, přidány barevné prostory a gamma korekce,
- BITMAPV5HEADER – přidány ICC barevné prostory.

Tabulka 7 Struktura BITMAPV5HEADER [8]

Délka	**Název**	**Význam**
4 bajty	bV5Size	Velikost BITMAPINFOHEADER
4 bajty	bV5Width	Šířka obrazu
4 bajty	bV5Height	Výška obrazu
2 bajty	bV5Planes	Počet rovin pro výstupní zařízení, musí být nastaveno na 1
2 bajty	bV5BitCount	Počet bitů na pixel, musí být 1, 2, 4, 8, 16, 24 nebo 32
4 bajty	bV5Compresion	Udává typ komprese: 0 – BI_RGB, 1 – BI_RLE8, 2 – BI_RLE4, 6 – BI_ ALPHABITFIELDS
4 bajty	bV5SizeImage	Velikost obrazu v bytech
4 bajty	bV5XPelsPerMeter	Horizontální rozlišení výstupního zařízení na metr
4 bajty	bV5YPelsPerMeter	Vertikální rozlišení výstupního zařízení na metr
4 bajty	bV5ClrUsed	Počet použitých barev, pokud je 0, je použit maximální počet barev dle bV5BitCount
4 bajty	bV5ClrImportant	Počet barev potřebný pro vykreslení bitmapy
Zde končí hlavička BITMAPINFOHEADER		
4 bajty	bV5RedMask	Maska pro červenou složku
4 bajty	bV5GreenMask	Maska pro zelenou složku
4 bajty	bV5BlueMask	Maska pro modrou složku
4 bajty	bV5AlphaMask	Maska pro alfa kanál
4 bajty	bV5CSType	Typ barevného prostoru
36 bajtů	bV5Endpoints	Hodnoty pro převod mezi RGB a CIE-XY
4 bajty	bV5GammaRed	Gamma faktor pro červenou složku
4 bajty	bV5GammaGreen	Gamma faktor pro zelenou složku

4 bajty	bV5GammaBlue	Gamma faktor pro modrou složku
Zde končí hlavička BITMAPV4HEADER		
4 bajty	bV5Intent	Typ barevného profilu
4 bajty	bV5ProfileData	Zpravidla název souboru s barevným profilem
4 bajty	bV5ProfileSize	Délka barevného profilu
4 bajty	bV5Reserved	Rezervováno, hodnota 0

3.1.1.3 Barevná paleta RGBQUAD

Pokud je bitmapa uložena v bitové hloubce 1,2,4 nebo 8 bitů na pixel, je bitová mapa použita. Tabulka barev se nenachází ve formátu RGB, který je reprezentován 24 bity na pixel, ale strukturou 32 bitů, kde je vždy 8 bitů určeno na jeden kanál. Počet barev uložených v daném souboru závisí na proměnné bV5ClrUsed. Pokud je tato hodnota nulová, je počet barev v souboru 2^n, kde n je hodnota bV5BitCount. V samotných datech je konkrétní barva zastoupena číslem, díky kterému se vybere správná barva z bitové mapy. Barvy v bitové mapě by měly být seřazeny dle důležitosti.

3.1.2 Barevný mód souboru BMP

Jak už bylo zmíněno výše, bitmapa může být uložena s různou barevnou hloubkou. Počet použitých nebo maximální počet barev je určen parametrem bV5BitCount v BITMAPINFOHEADER struktuře. Tento parametr určuje počet bitů, který je potřebný pro uložení jednoho pixelu, hodnoty a vlastnosti jsou uvedeny v následující tabulce.

Tabulka 8 Určení počtu barev v bitmapě

Hodnota	Popis
1	Monochromatická bitmapa, tabulka barev obsahuje pouze 2 barvy, bit s hodnotou 0 odpovídá první barvě v barevné paletě, bit s hodnotou 1 odpovídá druhé barvě z barevné palety.
2	Bitmapa obsahuje maximálně 4 barvy. Do jednoho bytu jsou zapsány 4 pixely tak, že významnější bit je vždy uveden vlevo. Dvojice bitů je dekódována na hodnotu 0 až 3.
4	Bitmapa má maximálně 16 barev. Každý pixel odpovídá jednomu záznamu v bitové mapě dle indexu, který se skládá ze 4 bitů. Jeden byte reprezentuje 2 pixely. Například hodnota 0x1F reprezentuje index 1 a F (15), jedná se o 2. a 16. záznam v bitové mapě. S tímto barevným režimem je možné použít komprimaci algoritmem RLE4.
8	Bitmapa má maximálně 256 barev. Pixel je reprezentován 1 bytem, který je odkazem do bitové mapy. Například hodnota 0x1F odpovídá 32. záznamu v bitové mapě. Tento barevný režim je možné kombinovat s komprimačním algoritmem RLE8.
16	Bitmapa má maximálně 65536 barev, člen bV5ClrUsed je nastaven na 0. Jednomu pixelu odpovídají 2 byty, data jsou uložena ve zpravidla ve formátu 5.5.5.1, kde barevné složky jsou zastoupeny po 5 bitech, jeden bit je možné využít pro alfa kanál.
24	Bitmapa má maximálně 2^{24} barev, člen bV5ClrUsed je nastaven na 0. Každá trojice bytů zastupuje jeden pixel.
32	Bitmapa má maximálně 2^{32} barev, člen bV5ClrUsed je nastaven na 0. Každá čtveřice bytů zastupuje jeden pixel. V tomto režimu bývá definován alfa kanál. Všechny kanály mohou být definovány bitovou maskou.

3.1.3 Komprimace BMP

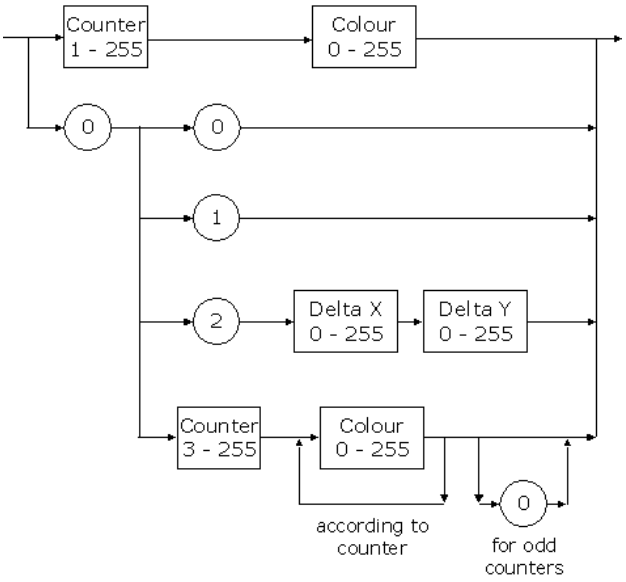
Ve formátu BMP je možno využít komprimaci pomocí algoritmu RLE. Tento režim je k dispozici pro bitmapy s 16 a 256 barvami. Komprimace zmenšuje celkovou velikost souboru. V souboru BMP je možno použít následující režimy:

- **Komprimace bitmap s 256 barvami** – v tomto případě je hodnota parametru bV5Compresion nastavena na BI_RLE8.
- **Komprimace bitmap s 16 barvami** - v tomto případě je hodnota parametru bV5Compresion nastavena na BI_RLE4.

3.1.3.1 Algoritmus RLE8

Algoritmus RLE8 umožňuje komprimovat obraz, který obsahuje až 256 barev. Pracuje ve dvou režimech:

- **Kódovací režim** – skupina informací se skládá z dvojice bajtů. První byte určuje počet pixelů, které budou mít index barvy určený druhým bytem. První byte může být nastaven na nulu, což je úniková hodnota, podle druhého bytů se identifikuje interpretace úniku, hodnoty druhého bytu mohou být:
  - 0 – Konec řádku.
  - 1 – Konec bitmapy.
  - 2 – Delta – dva následující byty za touto hodnotou obsahují bezznaménkové hodnoty horizontálního a vertikálního posunutí pixelů od aktuální pozice.
- **Absolutní režim** – tento mód je signalizován hodnotou nula v prvním bytu páru, druhý byte určuje, kolik následujících bytu není komprimováno. Pokud je počet nekomprimovaných bytů lichý, přidává se jeden byte s nulovou hodnotou pro zarovnání. Sekvence 00 03 00 01 02 00 bude dekódováno jako 00 01 02, sekvence 00 04 00 01 02 03 jako 00 01 02 03.



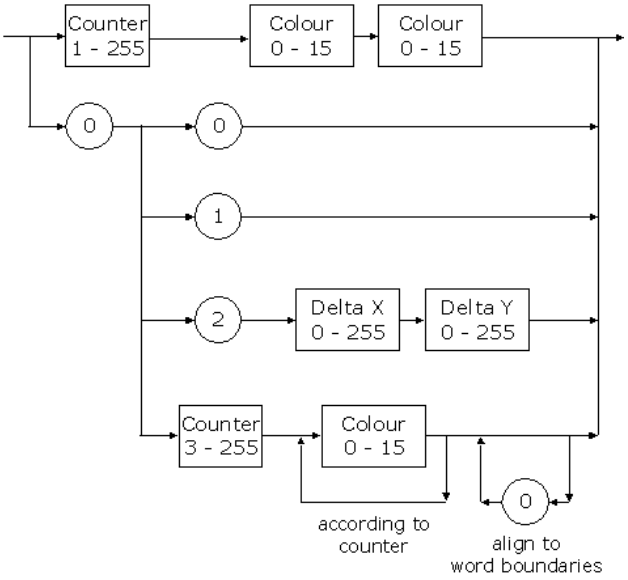
Obrázek 5 Schéma dekódování pomocí RLE8 [9]

3.1.3.2 Algoritmus RLE4

Algoritmus RLE4 umožňuje komprimovat obraz, který obsahuje až 16 barev, tudíž jeho použití není moc rozšířené. Režimy kódování jsou obdobné jako v podání RLE8. RLE4 využívá 2 režimy kódování:

- **Kódovací režim** – skupina informací se skládá z dvojice bajtů. První byte určuje počet pixelů, které budou mít indexy barev určených druhým bytem. Druhý byte obsahuje dva indexy barev. První barva je zahrnuta v čtyřech bitech s vyšší významností, druhá ve čtyřech bitech s nižší významností. První byte může být nastaven na nulu, což je úniková hodnota. Podle druhého bytu se identifikuje interpretace úniku, hodnoty druhého bytu mohou být:
  - 0 – Konec řádku.
  - 1 – Konec bitmapy.
  - 2 – Delta – dva následující byty za touto hodnotou obsahují bezznaménkové hodnoty horizontálního a vertikálního posunutí pixelů od aktuální pozice.
- **Absolutní režim** – tento mód je signalizován hodnotou nula v prvním bytu páru, druhý byte určuje, kolik následujících bytu není komprimováno. Pokud je počet bytů použitých pro nekomprimované hodnoty lichý, přidává se jeden byte

s nulovou hodnotou pro zarovnání. Sekvence 00 04 12 34 bude dekódováno jako 1 2 3 4, sekvence 00 05 12 34 50 00 jako 1 2 3 4 5.



Obrázek 6 Schéma dekódování pomocí RLE4 [10]

3.1.4 Obrazová data v souboru BMP

O formátu a systému uložení dat už byla zmínka v předchozích kapitolách. Jednou ze zvláštností uložení dat v souboru je směr zápisu. Data jsou zapisována zdola nahoru. První řádek zapsaný v souboru je posledním řádkem bitmapy.

Podobně, jako tomu bylo u RLE8 či RLE4, dochází i v případě nekomprimovaných dat v zarovnávání. V případě nekomprimovaných dat se zarovnává na 4 byty. Počet bytů na řádek je určen následujícím vzorcem (1).

$$\text{Velikost řádku} = \left(\frac{\text{Počet bitů na pixel} * \text{šířka obrazu} + 31}{32}\right) * 4 \qquad (1)$$

3.2 JPEG

JPEG je zkratka z Joint Photographic Experts Group, což je název komise, která byla ustanovena v roce 1986 a v roce 1992 vytvořila standard pro ukládání a kompresi obrazů. JPEG používá ztrátovou kompresi, neboli výsledek je jiný než originál! Rozdíl je ale okem nepostřehnutelný a navíc stupeň komprese je možné v poměrně širokém rozsahu měnit. JPEG tak vyhoví jak tam, kde je potřeba maximální kvalita fotografie (např. tisk), tak tam,

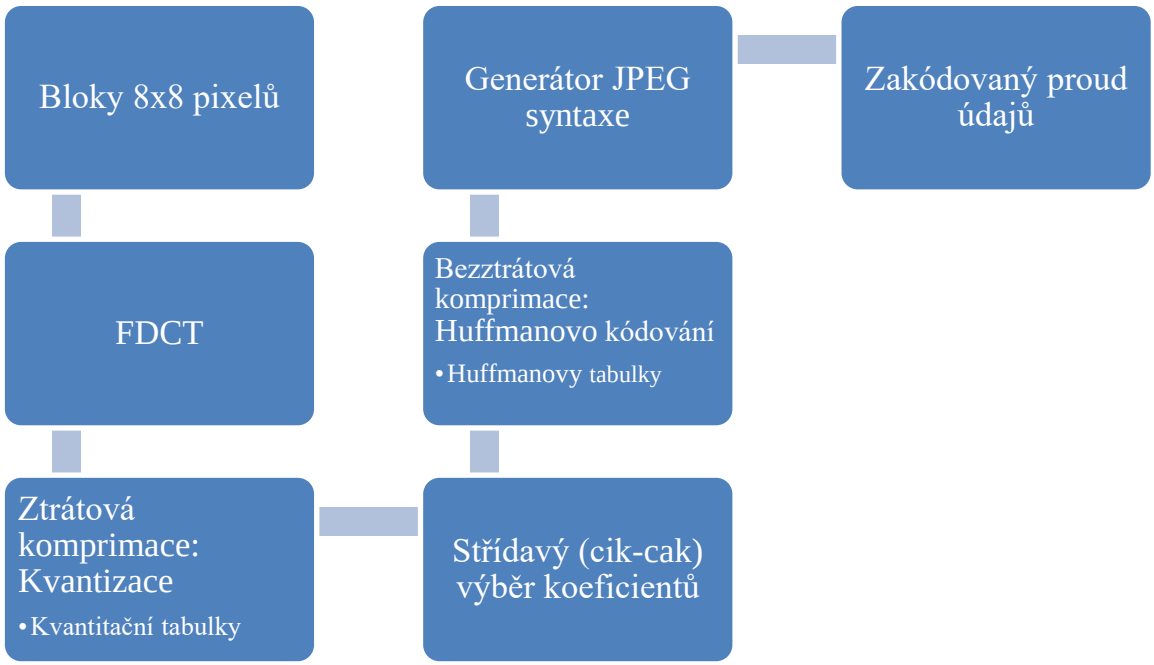
kde je preferována velikost souboru (internet, mail).

Finální úspora velikosti souboru záleží nejen na nastaveném stupni komprese, ale i na obsahu fotografie. Ostré fotografie plné jemných detailů (např. pole nebo tráva) lze zkomprimovat mnohem méně než např. jemný portrét s rozostřeným pozadím [11].

Standard JPEG popisuje 4 způsoby činnosti (módy):

- sekvenční DCT kódování, při kterém je každý prvek obrázku zakódován v jediném zleva doprava, seshora dolů,
- progresivní (postupné) DCT kódování, při kterém je obrázek zakódován v rámci několika přechodů přes originál, kterého cílem je poskytnout urychleně alespoň hrubý náčrt obrázku, pokud je celkový čas přenosu dlouhý,
- bezeztrátové kódování, v případě, že chceme zachovat věrný obraz originálu,
- hierarchické kódování, pokud chceme získat obrázek s různými rozlišeními [12].

3.2.1 Sekvenční JPEG kodér



Obrázek 7 Blokové schéma sekvenčního kodéru JPEG [12]

Vzorky originálu v rozmezí <0, 2^{p-1}> jsou posunuty o jeden bit, čímž se změní jejich vyjádření (v aritmetice se znaménkem) o hodnotu -2^{p-1}. Pro černobílé obrázky s p=8 se původní rozsah vzorků z intervalu <0, 255> zobrazí do intervalu <-128, +127>.

Tyto hodnoty se poté transformují do frekvenčního oboru pomocí přímé diskrétní kosinové transformace (FDCT) za pomoci následujícího vztahu (2):

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)cos\frac{(2x+1)u\pi}{16}cos\frac{(2y+1)v\pi}{16} \qquad (2)$$

kde

$C(u) = \frac{1}{\sqrt{2}}$ pro u=0

$C(u) = 1$ pro u>0

$C(v) = \frac{1}{\sqrt{2}}$ pro v=0

$C(v) = 1$ pro v>0

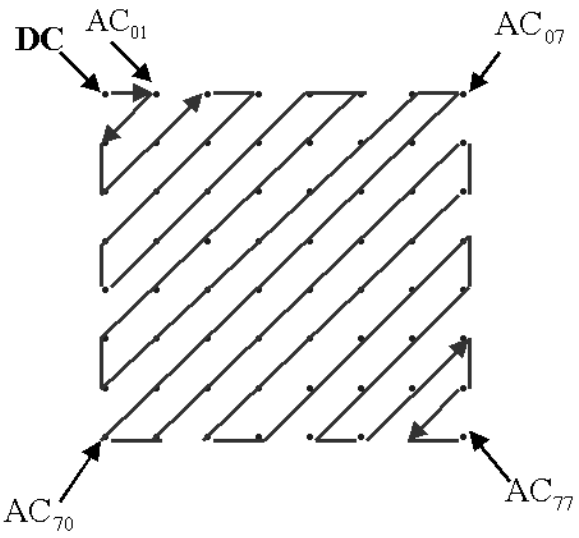
Transformovaný 64bodový diskrétní signál je funkcí dvou prostorových rozměrů *x* a *y* a jeho komponenty jsou nazývány prostorovými frekvencemi anebo DCT koeficienty. Koeficient F(0,0) je nazýván DC koeficientem a zbývajících 63 koeficientů AC koeficienty.

Pro typický blok obrázku 8 x 8 pixelů má většina prostorových koeficientů nulové anebo téměř nulové hodnoty a nemusí být kódována. Tato skutečnost je základem komprimace. V dalších kroku se všech 64 koeficientů kvantizuje podle 64prvkové kvantizační tabulky, specifikované aplikací. Kvantizace redukuje amplitudu koeficientů, které buď nepřispívají, anebo přispívají jen málo ke kvalitě obrázku, za účelem zvětšení počtu koeficientů s nulovou hodnotou. Účelem kvantizace je také odstranění informace, která není vizuálně pozorovatelná. Kvantizace se vykonává podle následujícího vztahu (3):

$$F_q(u,v) = round\left\lfloor\frac{F(u,v)}{Q(u,v)}\right\rfloor \qquad (3)$$

kde Q(u,v) jsou kvantizační koeficienty specifikované v kvantizační tabulce. Každý prvek Q(u,v) je číslo z intervalu 1 až 255, které specifikuje krok kvantizace pro odpovídající DCT koeficient.

Po kvantizaci se 63 AC koeficientů uspořádá do „cik-cak“ posloupnosti, jak je ukázáno na následujícím obrázku.



Obrázek 8 Cik-cak upořádání AC koeficientů [13]

Toto uspořádání umožní další fázi kódování (entropické kódování) tím, že koeficienty příslušné nižším frekvencím, které budou pravděpodobněji nenulové, budou zpracovány dříve než koeficienty odpovídající vyšším frekvencím.

DC koeficienty reprezentující průměrné hodnoty 64 vzorků obrázku jsou zakódovány s použitím prediktivních metod. Použití prediktivního kódování pro DC koeficienty je zdůvodněno tím, že mezi DC koeficienty sousedních bloků s rozměrem 8 x 8 je obvykle silná korelace. V důsledku toho je možné ještě zvýšit stupeň komprimace. Poslední částí kodéru JPEG je entropické kódování, které umožňuje zvýšení komprimace tím, že zakóduje kvantizované DCT koeficienty do kompaktnější podoby. JPEG standard definuje dvě metody entropického kódování: Huffmanovo kódování a aritmetické kódování. Běžné sekvenční kodéry JPEG používají Huffmanovo kódování.

Huffmanův kodér konvertuje DCT koeficienty získané kvantizací do kompaktní binární posloupnosti ve dvou krocích:

a) jako mezikrok je vytvořena posloupnost symbolů,
b) z posloupnosti symbolů je vytvořena binární posloupnost pomocí Huffmanových tabulek.

V posloupnosti symbolů každý pár symbolů reprezentuje AC koeficient

Symbol 1	Symbol 2
(délka, počet)	**(amplituda)**

kde:

- **délka** je číslo určující počet za sebou jdoucích AC koeficientů s nulovou hodnotou, předcházejících nenulový AC koeficient. Hodnota parametrů je v rozmezí 0 až 15, což si vyžaduje pro její reprezentaci 4 bity.
- **počet** je číslo udávající počet bitů, pomocí kterých je vyjádřena velikost koeficientu amplituda. Počet bitů je v rozmezí 0 až 10, takže pro reprezentaci potřebujeme 4 bity.
- **amplituda** udává velikost nenulového AC koeficientu v rozmezí od -1023 po 1023, což si vyžaduje až 10 bitů.

Například, pokud by posloupnost AC koeficientů byla (0,0,0,0,0,0,476), její symbolická reprezentace bude (6,9) (476), protože posloupnost obsahuje 6 nulových hodnot, tj. délka=6, počet=9 a amplituda=476.

Pokud by hodnota parametru délka byla větší než 15, potom hodnota prvního symbolu (15,0) je interpretována jako symbol rozšíření s hodnotou délka=16.

Za sebou mohou následovat nejvíce tři tato rozšíření. Například, pokud bychom měli za sebou symboly (15,0) (15,0) (7,4) (12), znamená to, že délka má hodnotu 16+16+7, počet hodnotu 4 a amplituda hodnotu 12.

Symbol (0,0) označuje konec bloku (EOB) a slouží k ukončení každého bloku 8x8 bodů.

Pro DC koeficienty se symbolická reprezentace skládá ze symbolů:

Symbol 1	Symbol 2
(počet)	**(amplituda)**

Koeficienty DC se kódují diferenciálně a jejich rozsah je dvojnásobný vůči rozsahu AC koeficientů, je to interval <-2047,2047>.

Druhým krokem Huffmanova kódování je konverze meziproduktu, posloupnosti symbolů do binární posloupnosti. V této fázi jsou symboly nahrazeny kódem proměnlivé délky, nejprve se zakódují DC koeficienty a poté AC koeficienty.

Každý Symbol1 (pro DC i AC koeficienty) se zakóduje pomocí kódu s proměnlivou délkou VLC (variable-length code) podle Huffmanovy tabulky, vytvořené speciálně pro každý prvek obrázku. Symbol 2 je zakódován pomocí kódu variabilní délky pro celá čísla VLI (variable-length integer), kterého délka je uvedena v následující tabulce 9.

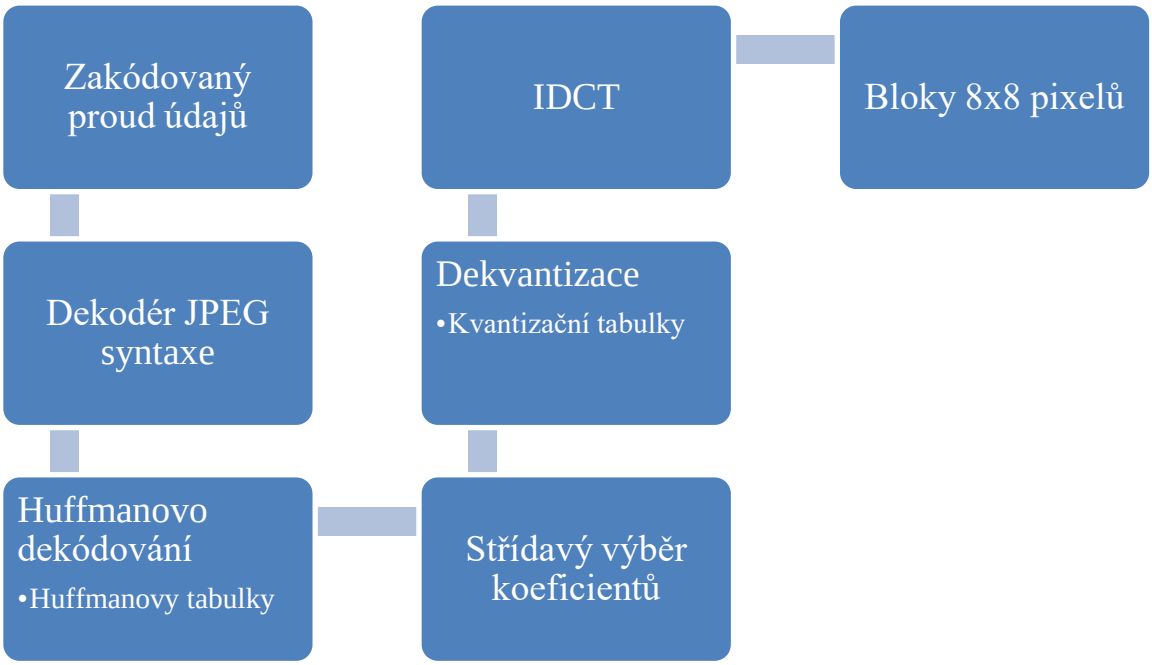
Tabulka 9 Hodnota Huffmanových kódů pro Symbol 2 pro AC koeficienty [12]

Počet	**Rozsah amplitud**
1	<-1,1>
2	<-3,-2><2,3>
3	<-7,-4><4,7>
4	<-15,-8><8,15>
5	<-31,-16><16,31>
6	<-63,-32><32,63>
7	<-127,-64><64,128>
8	<-255,-128><128,255>
9	<-511,-256><256,511>
10	<-1023,-512><512,1023>

Například AC koeficient reprezentovaný jako symboly (1,4) (12) se zakóduje do binární posloupnosti (1111101101100), kde (111110110) je hodnota VCL získaná z Huffmanových tabulek a (1100) je hodnota VLI kódu pro číslo 12 [12].

3.2.1 Sekvenční JPEG dekodér

Při sekvenčním JPEG dekódování jsou kroky, které jsme udělali při kódování obrazu, prováděny v opačném pořadí, jak je uvedeno v následujícím schématu.



Obrázek 9 Blokové schéma sekvenčního dekodéru JPEG [12]

Nejprve se na zakódovaná binární data použije entropický dekodér (např. Huffmanův). Ten převede binární posloupnost na posloupnost symbolů pomocí tabulek Huffmanových kódů (VLC kód) a VLI dekódování a poté symboly převede na posloupnost DCT koeficientů. Poté se provede dekvantizace pomocí funkce: $F(u,v) = F_q(u,v) * Q(u,v)$

Pak se na dekvantizované koeficienty aplikuje inverzní diskrétní kosinová transformace (IDCT), která převede koeficienty z frekvenčního oboru zpět do prostorového oboru. IDCT je definovaná vztahem (4):

$$F(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)f(u,v)cos\frac{(2x+1)u\pi}{16}cos\frac{(2y+1)v\pi}{16}\right] \quad (4)$$

kde

$C(u) = \frac{1}{\sqrt{2}}$ pro u=0

$C(u) = 1$ pro u>0

$C(v) = \frac{1}{\sqrt{2}}$ pro v=0

$C(v) = 1$ pro v>0

Poslední krok spočívá v posunu hodnot dekomprimovaných vzorků do intervalu $<0,2^{p-1}>$ [12].

3.2.2 Struktura souboru JPEG

Soubory typu JFIF/JPEG jsou rozděleny do takzvaných segmentů (*segments*), kde každý segment je představován proudem bytů o maximální délce 65535. Každý segment začíná takzvanou značkou (*marker*), což je dvoubytová hodnota, kde první byte má vždy hodnotu **0×ff** (tj. 255 decimálně) a druhý byte nabývá podle typu značky hodnot od **0×01** do **0×fe**. Hodnoty **0×00** a **0×ff** u druhého byte značky nejsou použity, aby se zaručila synchronizace v případě, že při přenosu obrázku dojde k chybě. Pokud k takové situaci dojde, postačuje, aby dekodér našel v proudu bytů hodnotu **0×ff** následovanou hodnotou **0×01–0×fe** a může provést synchronizaci s tím, že předchozí data sice nebude umět rekonstruovat, ale zbytek obrázku už ano.

Z tohoto důvodu se v samotných datech nikdy nesmí objevit samostatná hodnota **0×ff**, ale musí být doplněna bytem s hodnotou **0×00**. Pro nás má tato struktura jednu výhodu – při zkoumání obsahu souborů JFIF/JPEG nám postačí v hexadecimálním editoru hledat výskyt bytů **0×ff** následovaných nenulovou hodnotou a podle této dvojice bytů již víme, jaký segment na tomto místě začíná. Máme totiž jistotu, že takto uspořádaná dvojice bytů vždy značí začátek segmentu, v datech by se nikdy neměla objevit [14].

Tabulka 10 Nejdůležitější značky v souborech JPEG [14]

Značka	Plný název	Hodnota	Význam
SOI	Start Of Image	0×ffd8	značka umístěná na začátku souboru
EOI	End Of Image	0×ffd9	značka umístěná na konci souboru
RSTi	Restart Marker i	0×ffd0+i	značky určené pro synchronizaci (pokud jsou značky použity, tak se periodicky opakují)
SOF0	Start Of Frame 0	0×ffc0	specifikace tvaru obrazových dat
SOS	Start Of Scan	0×ffda	začátek vlastních kódovaných obrazových dat
APP0	Application Marker	0×ffe0	značka identifikující soubor JFIF
COM	Comment	0×fffe	jednoduchý textový komentář se zadanou délkou
DNL	Define Number Of Lines	0×ffdc	počet řádků (dnes se již nepoužívá)
DRI	Define Restart Interval	0×ffdd	interval výskytu synchronizačních značek RSTi
DQT	Define Quantization Table	0×ffdb	začátek segmentu koeficientů kvantizační tabulky
DHT	Define Huffman Table	0×ffc4	začátek segmentu kódových slov Huffmanova kódování

3.2.2.1 Hlavička souborů JPEG

Samotná hlavička souboru typu JPEG začíná dvojicí bytů s hodnotou **0×ff** a **0×d8**. Jak již víme z předchozího popisu, dvojice bytů začínající hodnotou **0×ff** představuje značku, v tomto případě se jedná o značku SOI (*Start Of Image*). Za touto značkou následuje segment nazvaný APP0, který začíná značkou s hodnotou **0×ff** a **0×e0**. Za touto značkou je uložena dvoubytová hodnota segmentu, která musí být větší nebo rovna šestnácti [14].

Hlavička souboru JPEG obsahuje základní informace o obrazu, označení souboru, verzi JPEG použitou ke komprimaci, údaje o rozlišení a velikost náhledového obrázku.

Tabulka 11 Rozložení bytů v hlavičce souboru JPEG [14]

Offset	Význam sekvence
00	značka SOI – Start Of Image
02	značka APP0 – začátek stejnojmenného segmentu
04	délka segmentu APP0 (včetně samotné délky)
06	nulou ukončený řetězec „JFIF“
11	číslo verze
12	číslo subverze
13	jednotky rozlišení obrázku
14	horizontální rozlišení
16	vertikální rozlišení
18	šířka náhledového
19	výška náhledového

3.2.2.2 Kvantizační tabulky

Jednou z velmi důležitých informací, které je možné v těchto souborech nalézt, jsou tabulky kvantizačních koeficientů použitých při kvantizaci prováděné ihned po DCT. Tyto tabulky začínají značkou **DQT**, která je v souboru identifikovatelná dvojicí bytů **0×ffdb**. Po této dvojici bytů následuje délka, opět uložená jako dvoubytové slovo. Ihned po délce je umístěn velmi důležitý byte, ve kterém je zakódován jak typ tabulky, tak i počet bitů použitých pro uložení jednotlivých koeficientů. Horní polovina bytu určuje počet bitů na koeficient (0=8 bitů, jinak 16 bitů), dolní polovina bytu obsahuje číslo kvantovací tabulky (0-3).

Po těchto údajích již následují hodnoty jednotlivých kvantizačních koeficientů, které jsou uspořádány podle cik-cak sekvence. Těchto koeficientů je vždy 64 a jsou tedy uloženy buď na 64 bytech (8 bitů na koeficient) nebo na 128 bytech (16 bitů na koeficient). Teoreticky

je sice možné do souboru uložit až 16 kvantovacích tabulek, ve skutečnosti jsou však ukládány pouze tabulky dvě: první pro barvové koeficienty *Y*, druhá pro barvové koeficienty C_b a C_r [14].

3.2.2.3 Huffmanovy kódovací tabulky

Huffmanovy kódovací tabulky začínají značkou **DHT**, která je v souboru identifikovatelná dvojicí bytů **0×ffc4**. Za touto dvojicí bytů následuje délka segmentu (dva byte) a posléze jeden byte se zakódovaným číslem a typem tabulky.

Tabulka 12 Popis bytu specifikující Huffmanovu tabulku [14]

Bity	Význam
0..3	číslo Huffmanovy kódovací tabulky z rozsahu 0 až 3
4	typ kódovací tabulky 0=DC, 1=AC
5..7	tyto bity musí být nastaveny na nulu

Po těchto údajích následuje sekvence šestnácti bytů, které obsahují počty symbolů pro délky kódových slov 1 až 16. Součet hodnot těchto bytů udává celkový počet kódů, který musí být menší než 256 (maximální délka kódového slova je však šestnáct bitů, protože Huffmanův kód je prefixový). Následuje *n* bytů (*n*=počet kódů), ve kterých jsou postupně uloženy symboly, ze kterých se při inicializaci dekodéru vytváří binární strom [14].

3.2.2.4 Informace o obrazových datech

Informace o obrazových datech jsou specifikována blokem **SOF0** (*Start of Frame*), ten je v souboru identifikován dvojicí bytů **0xffc0**. Za touto dvojicí bytů následuje délka segmentu, bitové rozlišení na pixel, výška a šířka v pixelech, počet barevných komponent a samotné údaje o jednotlivých komponentách a podvzorkování.

3.2.2.5 Obrazová data

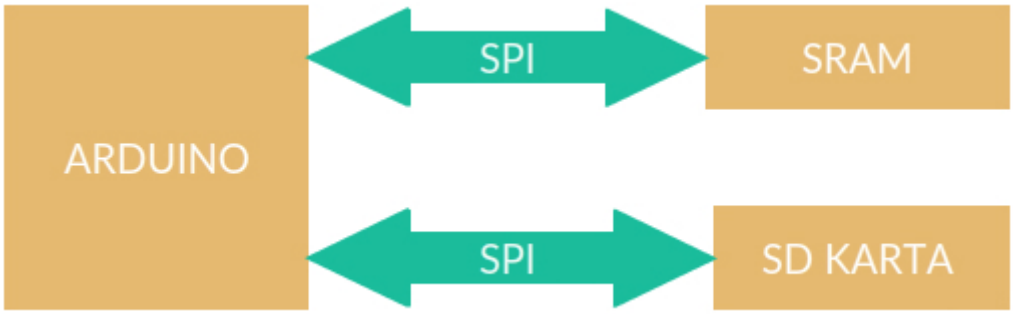
Obrazová data se nachází v souboru pod blokem SOS (Start of Scan), blok je identifikován dvojicí bytů **0xffda**. Za touto dvojicí se tradičně nachází délka segmentu, následuje počet komponent a informace o pořadí komponent a údaje o příslušnosti Huffmanových tabulek. Za těmito údaji jsou umístěna komprimovaná data obrazu.

4 VLASTNÍ NÁVRH A ŘEŠENÍ

Celá tato kapitola se bude zabývat vývojem hardwaru a softwaru pro Arduino, pomocí kterého bude možno komprimovat a dekomprimovat rastrový obraz. Řešení bude kompatibilní s 8bitovou a 32bitovou verzí Arduino. V závěru kapitoly bude otestována funkčnost celého řešení.

4.1 SRAM shield - rozšiřující deska pro Arduino

Jelikož se již v začátcích vyskytly problémy s velikostí SRAM paměti, kterou disponuje Arduino Mega, Bylo rozhodnuto, že bude k Arduino desce přidán externí SRAM čip. Data, která budou dekomprimována či komprimována budou uložena na SD kartě, proto bude na do návrhu přidán i SD slot.



Obrázek 10 Hardwarová architektura řešení

4.1.1 SPI SRAM 23LC1024

Při důkladném porovnávání různých čipů pamětí a možných zapojení byla nalezena následující řešení:

- SRAM s adresací pomocí I/O pinů – tyto paměťové čipy nabízejí v této variantě daleko větší úložný prostor, avšak k adresaci je potřeba vytvořit řadič, který by se staral o samotnou adresaci. Návrh takového řadiče není náročný z pohledu softwaru. Z pohledu hardwaru nastává problém v komplikovaném zapojení na plošném spoji. Takový plošný spoj je problematické vyrobit v domácích podmínkách,

- SRAM s adresací pomocí integrované sběrnice SPI nebo I2C - ačkoliv je paměť těchto čipů značně omezena, obsluha paměti je o hodně jednodušší než v předchozí variantě. Po sběrnici, která má 2 - 4 vodiče, se zašle adresa buňky, kterou chceme obsloužit a následně se provede čtení nebo zápis. Zapojení je jednoduché, jelikož tyto pamětí mívají zpravidla osm pinů, proto je návrh plošného spoje jednodušší a následná výroba levnější.

Při procházení oficiálního fóra Arduino bylo nalezeno zapojení SRAM paměti 23LC1024 od uživatele Riva, u zapojení byla přiložena i jednoduchá knihovna SpiRAM [15], která byla otestována a použita v řešení.

Tabulka 13 Základní parametry paměti 23LC1024

Vlastnost	**Hodnota**
Velikost paměti	1 Mb (128K x 8b)
Pracovní frekvence	Až 20 MHz
Napájecí napětí	2,5 – 5 V
Sběrnice	SPI

Díky napájecímu napětí 2,5 – 5 V a I/O pinům tolerantních k 5V logice je tato paměť vhodná bez jakýchkoli úprav zapojení pro Arduino Mega i Due.

4.1.2 SD slot

Pro čtení a zápis je použita SD karta, proto bylo nutné do navrhovaného návrhu zahrnout i SD slot, do kterého bude vložena SD karta.

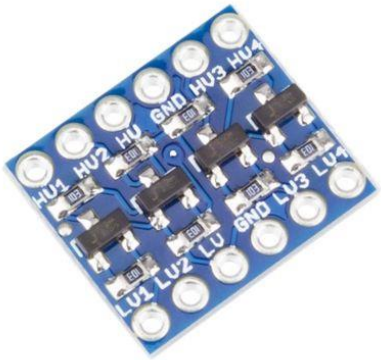
SD karta využívá pro komunikaci sběrnici SPI, ale pouze v logice s napětím 3,3 V, připojení 5 V na piny SD karty může kartu nenávratně poškodit. Proto je nutné zapojení slotu součástky pro převod logiky na 3,3 V.

4.1.3 Převodník pro SD kartu

Pro zachování kompatibility rozšiřující desky bylo nutné mezi Arduino a SD kartu zapojit převodník logiky. Pokud by deska byla používána pouze s Arduino Due, logické oddělení by nutné nebylo, jelikož Arduino Due využívá logiku 3,3 V.

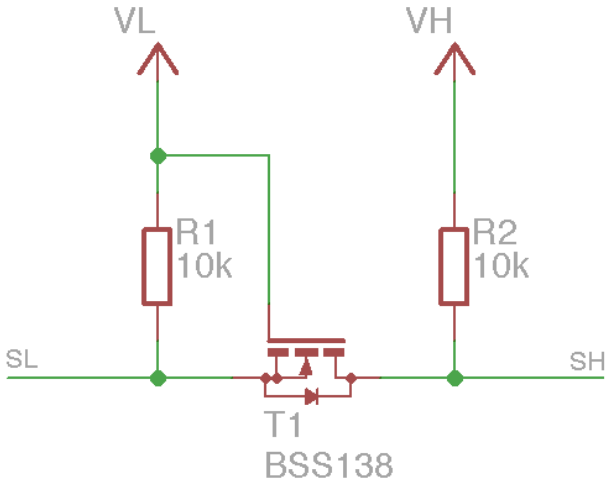
V prvotních návrzích jsem zkoušel na plošný spoj navrhnout zapojení převodníků pomocí rezistorů a tranzistorů přímo na plošném spoji, ale podotýkal jsem se s problémy mnohonásobného křížení cest. Proto jsem využil externího převodníku, který je zapojen na jiném plošném spoji a má vstupy a výstupy vyvedeny na konektorové liště.

Použitý převodník disponuje čtyřmi kanály s obousměrnou komunikací. Jelikož zapojení sběrnice vyžaduje pouze čtyři vodiče, je tento převodník optimální.



Obrázek 11 Použitý čtyř kanálový logický převodník [16]

Zapojení každého kanálu je realizováno MOSFET tranzistorem BSS138 a dvěma rezistory.



Obrázek 12 Zapojení jednoho kanálu logického převodníku [17]

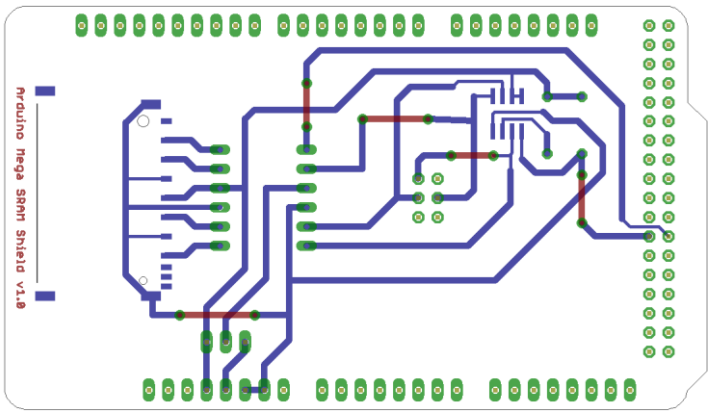
Princip obvodu je docela jednoduchý. *VL* označuje nižší napájecí napětí a *VH* zase vyšší. *SL* je signál s nižší (např. 3,3V) logikou a *SH* zase vyšší (např. 5V). Pokud ani jedna strana nekomunikuje, pull-up rezistory způsobí, že je linka na obou stranách v logické jedničce. Tedy SL=3,3V a SH=5V. Rozdíl napětí mezi vývody *source* a *gate* tranzistory je blízké nule a tranzistor je zavřen.

Jakmile levá strana (ta s nižším napětím) spojí linku se zemí (logická nula), rozdíl napětí mezi source a gate tranzistoru stoupne, a tím se otevře. Logická nula se objeví i na pravé straně.

Pokud pravá strana spojí linku se zemí, dioda mezi *source* a *drain* tranzistoru způsobí, že se zvětší rozdíl napětí mezi *source* a gate a tranzistor se otevře. Tím se na levé straně objeví logická nula [17].

4.1.4 Zapojení SRAM shieldu

Návrh rozšiřující desky vychází z Arduino Mega protoshieldu[3]. Z protoshieldu byly odebrány veškeré nepotřebné kontakty, tlačítka a přebytečné konektorové lišty. Arduino Mega má sběrnici SPI vyvedenou na pinech 50, 51, 52, 53 a konektoru ISCP, Arduino Due pouze na konektoru ISCP. Z tohoto důvodu bylo nutné zapojit konektorovou lištu do středu shieldu, aby byla komunikace skrze SPI možná na obou deskách. Kvůli zapojení převodníku úrovní byl na desku přidán přepínač (jumper) vstupního napětí. Pro zapojení paměti 23LC1024 bylo použito katalogové zapojení.



Obrázek 13 Plošný spoj SRAM shieldu.

[3] https://www.adafruit.com/products/192

Externí paměť i SD karta komunikují s deskou Arduino po stejné sběrnici, jedinou odlišností je zapojení *SS* (Slave Select) pinu.

Po zkoumání rozšířených shieldů, které by mohly být použity zároveň s tímto, jsem se rozhodl vyvést SS piny na následující piny Arduino:

- SS pin pro SD kartu – pin číslo 43,
- SS pin pro SRAM paměť – pin číslo 42.

Pro využití všech nepoužitých pinů jsou v návrhu použity dutinkové lišty s prodlouženými kontakty. Lišta se deskou prostrčí a připájí, zbytek kontaktu je možné zasunout do konektoru na desce Arduino. Z pohledu ze shora je prodloužená dutinková lišta totožná s tou, která je použita na Arduino desce.

Schéma a zapojení je do diplomové práce vloženo jako příloha.

4.2 Softwarové řešení

Vývojové desky Arduino neoplývají příliš velkým výpočetním výkonem, hodí se převážně pro řízení. Obraz se na nich ve většině případů pouze reprezentuje, proto byl celý vývoj dosti problematický. Ze zdrojů bylo velice obtížné vyčíst informace, které by pro celý projekt byly přínosné. Problematiku komprese či dekomprese BMB či JPEG souboru na Arduinu komplexně nikdo neřešil, proto celá práce vznikala na zelené louce.

Ze začátku bylo nutné nastudovat normu, ve které je popsán JPEG formát. Bohužel i ta se podotýká s některými nedostatky. Některé informace se v ní vyskytují dosti zkresleně, některé vůbec, proto bylo nutné informace obtížně dohledávat.

Programy psané pro počítačové platformy je možno různými nástroji testovat a validovat. U platformy Arduino je to problém. Jediná možnost, jak si ověřit funkčnost, správné výpočty a operace, je výpis do konzole, eventuálně do souboru na SD kartu. Celý tento proces není možno automatizovat. Změněný kód je nutno znovu zkompilovat, nahrát do Arduina a data následně kontrolovat z výpisu. Pokud jsou výpočty náročné a v případě JPEG formátu jsou, je celý tento proces velice zdlouhavý. Jelikož byl celý projekt psán pomocí rozšíření, které je přidáno do MS Visual Studia, projekt neobsahuje diagram tříd. Ačkoli byla snaha tento diagram vygenerovat, kvůli nekompatibilitě prostředí a struktuře Arduino knihoven, generování vždy skončilo chybou.

Celé softwarové řešení bylo vytvořeno postupným prototypováním. Operace s BMP formátem nejsou nijak náročné, proto vývoj části pro operace s BMP souborem nebyl příliš dlouhý. V případě JPEG formátu byl vývoj o dost náročnější a tedy i podstatně delší. Vývoj také brzdil výpočetní výkon platformy Arduino. Operace s JPEG formátem tvoří převážnou část této práce.

4.2.1 Soubory pro ukládání dekomprimovaných dat

Dekomprimovaná data je třeba ukládat do externí paměti, jelikož není možné je držet ve vnitřní paměti mikroprocesoru. Z tohoto důvodu jsem navrhl jednoduchý binární formát souboru, v kterém se nachází základní údaje o obraze. Jedná se o rozměry či barevný formát, a samotná dekomprimovaná data.

Soubor podporuje název ve formátu 8.3 (8 znaků názvu a 3 znaky přípony), který je nativně podporován knihovnou pro obsluhu SD karty. Dle dostupných zdrojů, je knihovna

pro obsluhu SD karty schopna obsloužit i delší názvy, ale na deskách s mikroprocesory AVR bývají s delšími názvy problémy, proto jsem použil stabilní řešení ve formátu 8.3.

Šířka a výška obrazu, který může být do souboru uložen, musí být menší nebo rovna 65 535 bodů, což odpovídá datovému typu *unsigned int* (odpovídá 2 bytům) a zároveň je to maximální rozměr obrazu ve formátu JPEG.

V souboru byl jeden byte rezervován pro barevný formát, díky kterému je možné rozlišit, jak barevné údaje zpracovávat. Momentálně je používán pouze pro formát RGB.

Struktura souboru je následující:

- 2 B – šířka obrazu (*unsigned int*),
- 2 B – výška obrazu (*unsigned int*),
- 1 B – barevný formát (*unsigned char*),
- Obrazová data – blok bytů, kde pro jeden pixel jsou použity 3 byty (*unsigned char*), v datech není použit žádný oddělovat řádků.

O obsluhu vytváření souboru eventuální zápis údajů se stará třída *ExportFile*. V této třídě je také vytvořena metoda pro vytvoření prázdného souboru pro BMP a JPEG formát. Soubory jsou standardně ukládány do složky *EXPORT* na SD kartě. Místo, kam se mají soubory ukládat, je možné změnit úpravou maker *EXPORT_FOLDER* a *EXPORT_PATH* v hlavičkovém souboru *CommonMacros*. Hodnoty obou maker musí být totožné. Pouze u makra *EXPORT_PATH* je navíc lomítko, jelikož slouží pro vytvoření cesty výsledného souboru. Hodnota *EXPORT_FOLDER* je použita pro zjištění existence složky pro export. V případě, že složka neexistuje, je následně vytvořena.

4.2.2 Bitové čtení a zápis

Ze specifikací komprimovaného obrazu v souboru BMP či JPEG vyplývá, že data jsou komprimována tak, aby zabírala co nejméně datového prostoru. Obraz v BMP může být uložen tak, že v jednom bytu je obsažen jeden či více pixelů. V případě JPEG je kódování jiné, ale také využívá zápis i čtení po jednotlivých bitech. Z těchto důvodů jsem napsal třídy *BitRead* a *BitWrite*, které se starají o čtení a zápis bitů.

Třída *BitRead* je využívána pro čtení BMP i JPEG souboru, v případě BMP je použita pro čtení souboru s rozlišením 1 a 2 bity na pixel, pro čtení obrazových dat z JPEG souboru je

použita vždy. Jedinou odlišností je konstruktor třídy. Pro BMP soubor je použit konstruktor s parametrem *file* (soubor). Pro JPEG je konstruktor přetížen přidáním parametru *type* (pro JPEG nastaven na 1).

Zápis obrazových JPEG je specifický tím, že pokud se v datech vyskytuje hodnota 0xFF, musí být následován hodnotou 0, aby nebyl rozeznán jako marker pro jednotlivé segmenty, které se v JPEG vyskytují. Tato hodnota 0 se při dekódování vypouští, proto je v této třídě pravidlo vypuštění hodnoty 0 jestliže následuje po hodnotě 0xFF.

Přečtení jednoho bitu se provádí metodou *readBit()*, která vrací hodnotu datového typu *unsigned char* s hodnotou 0 nebo 1. Pro vyčištění načtených proměnných slouží metoda *flush()*.

Třída *BitWrite* je využita pouze pro zápis JPEG souboru. Konstruktor je tvořen parametrem *file* (soubor). Jelikož jsem třídu navrhoval pro zápis dat pro JPEG, má vestavěnou logiku pro zápis hodnoty 0 po hodnotě 0xFF, z důvodu, který je rozebrán výše. Zápis obsluhuje metoda *writeValue()* s parametry *value* a *length*. Proměnná *value* slouží pro předání zapisované hodnoty a proměnná *length* pro předání bitové délky zápisu. Pro zápis hodnot, které jsou momentálně v bufferu, slouží metoda *writeLast()*, používá se převážně k zápisu posledních hodnot.

4.2.3 Čtení BMP souboru

O čtení BMP souboru se stará stejnojmenná třída BMP. Čtení se iniciuje metodou *open(char *inputFile)*, která požaduje odkaz na název souboru. Daný soubor je následně otevřen, ze souboru jsou vyčteny hlavičky, které uchovávají informaci o souboru, pro přehlednost a jednodušší operaci jsou hlavičky uloženy do struktur *BMPHeader_Segment* a *BITMAPV5HEADER_Segment*. Data ve struktuře *BMPHeader_Segment* obsahují pouze základní informace o souboru (identifikátor, velikost souboru a odsazení hlavičky nesoucí informace o obrazových datech od začátku souboru). Struktura *BITMAPV5HEADER_Segment* nese informace o obrazu (rozměry, komprese, bitová hloubka a jiné). Data z těchto hlaviček jsou dále použita pro správné přečtení či dekomprimování obrazových dat.

Pro dekompresi dat jsou nejdůležitější data ve struktuře *BITMAPV5HEADER_Segment*, kterými se bude řídit celá dekomprese. Nejdříve dojde k porovnání informace o kompresi, která je načtena do proměnné *bV5Compression*. Zde se rozhodne, zda jsou data

komprimována či nikoliv. Pokud jsou data komprimována, je tato hodnota dále porovnána k rozeznání typu komprese (RLE4 či RLE8).

Dále je detekována bitová hloubka a počet barev použitých v obraze. V případě bitové hloubky 1, 2, 4 či 8 bitů na pixel, jsou obrazová data zastoupena číslem, které odpovídá záznamu v barevné paletě, která je v souboru uložena. Je tedy nutné tuto paletu také načíst. Paleta je pro svou velikost (až 256 barev) načtena do paměti SRAM na rozšiřujícím shieldu.

Následuje samotné dekódování obrazových dat. Data jsou uložena do souboru, který byl pro tento účel navrhnut. Pokud je použita komprese RLE4 nebo RLE8, data se nejprve dekomprimují a poté se dle příslušných odkazů uloží odpovídající hodnoty RGB z palety do souboru. Pokud se jedná o nekomprimovaný obraz, dochází ke čtení hodnot patřičné bitové délky a následné identifikaci RGB barvy z palety. Poté je tato RGB barva zapsána do souboru. Jedinou výjimku tvoří barevná hloubka 24 bitů, při této bitové hloubce není použita komprese ani barevná paleta. Data se tedy pouze načtou a uloží do souboru. Z důvodu, že čtení a následný zápis je velice pomalé, načte se nejprve jeden řádek obrazu do SRAM paměti a až poté je zapsán do souboru.

Jedinou zvláštností BMP souboru je zápis od spodní části obrazu, proto je nutné data ze souboru, kde jsou již dekódována, zapsat do výsledného v opačném pořadí řádků, jinak by byl obraz při reprezentaci zrcadlově převrácen.

4.2.4 Zápis BMP souboru

Pro zápis obrazu do BMP souboru je použit 24bitový formát RGB bez komprese. Tento barevný formát je nejpoužívanější. Pro informace o obrazových datech je použita hlavička BITMAPFILEHEADER o délce 40 bytů. Ačkoli mohla být vybrána i novější verze hlavičky, nebylo to potřeba. Položky této hlavičky plně vyhovují obsahu, jelikož máme o obraze pouze informace o rozměru a barevném formátu, popřípadě hodnoty, které se počítají z těchto informací.

O uložení se stará metoda *writeFile(char *inputImage)*, která požaduje odkaz na jméno dekomprimovaného souboru s příponou .bin. Ze vstupního souboru se načtou informace o rozměru obrazu, ze kterých je vypočítána délka jednoho řádku v bytech a také počet nulových bytů, které se použijí pro zarovnání řádku (počet bytů na řádek musí být násobek 4). Následně je vytvořen soubor s příponou .bmp, do kterého se zapíše hlavička

souboru s patřičnými informacemi a odsazením obrazových dat od začátku souboru. Dále následuje hlavička, která nese informace o obrazových datech (formát, komprese, rozměry). Jelikož jsou řádky v BMP souboru zapsána v opačném pořadí, je potřeba číst vstupní data od konce. V každém cyklu je nutné vypočítat offset ve vstupním souboru a řádek obrazových dat přepsat do výstupního souboru. Jakmile jsou data přepsána, jsou všechny použité soubory uzavřeny a tím je zápis dokončen.

4.2.5 Čtení a dekódování JPEG

JPEG formát se vyznačuje použitím pro obrazy, ve kterých se vyskytuje velké množství barev. Pro maximální kompresi a robustnost výsledného souboru používá velké množství operací. Vstupní obraz je podroben podvzorkování barevného kanálu, diskrétní kosinové transformaci, kvantizaci, cik-cak kódování, diferenčnímu kódování a nakonec Huffmanovu kódování.

Na platformě s mikrokontrolérem se dekódování obrazu z JPEG řeší zřídka, a většinou je takové řešení omezeno na konkrétní hardware. Na platformě Arduino tato problematika řešena nebyla a informace, které jsou k dispozici praví, že výpočetní výkon vývojové desky je na dekompresi obrazu z JPEG souboru příliš malý. Výpočetní výkon sice malý být může, ale není řečeno, že tato problematika není řešitelná. Proto bylo vyvinuto následující řešení, které umožňuje dekompresi obrazu z JPEG formátu s pomocí externí SRAM paměti.

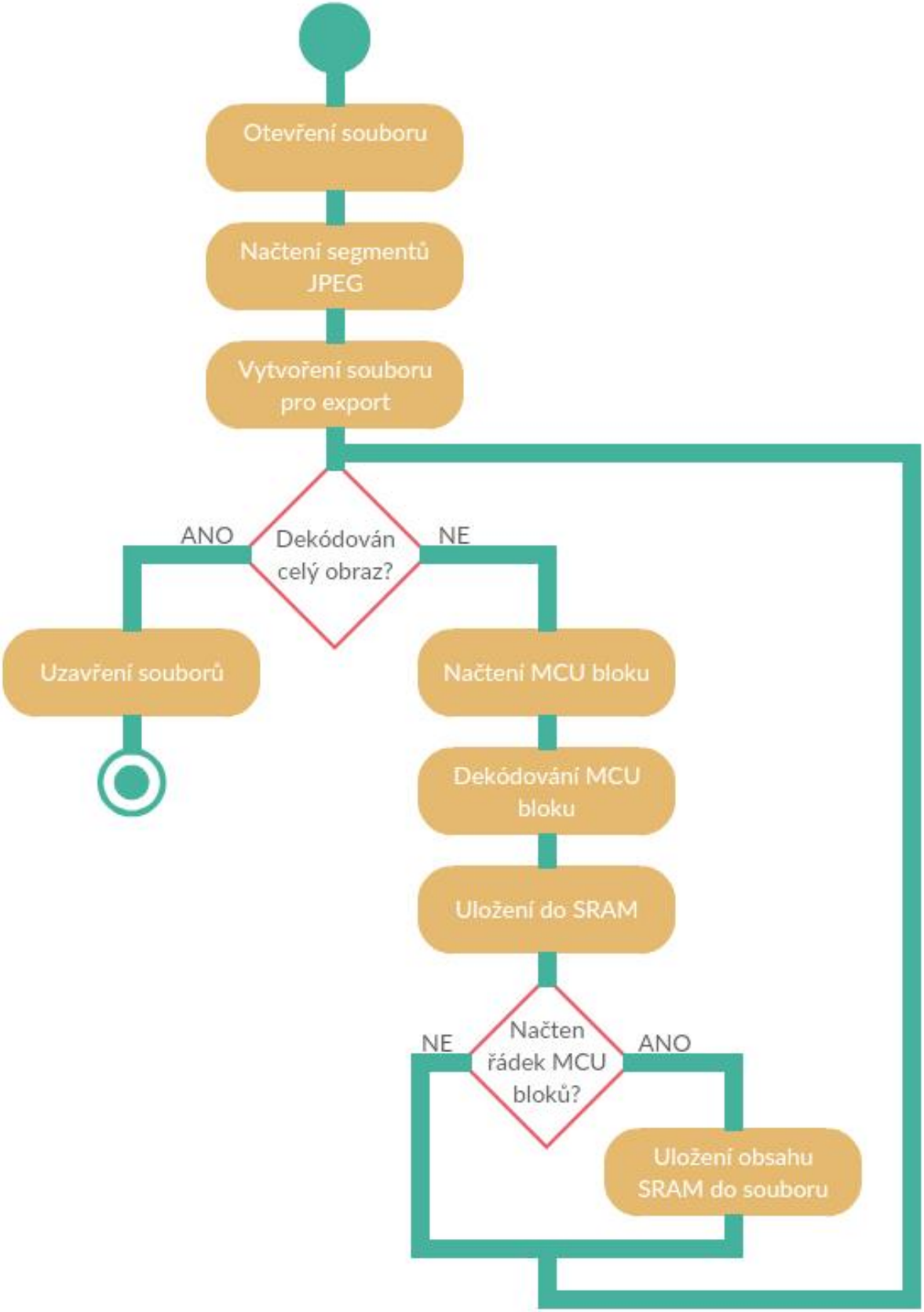
O dekompresi obrazu se stará třída *JPEG*. Dekomprimování obrazu začíná voláním metody *open(char * inputFile)*, která má vstupní parametr název souboru s příponou .jpg. Soubor se skládá z několika segmentů, ve kterých jsou ukryty informace o souboru, obraze, tabulky pro dekódování a také tabulky pro kvantizaci. Všechna důležitá data je potřeba před zahájením dekomprese načíst.

Ze souboru je nejdříve načten segment APP0 nebo APP1, který nese informace o souboru, a jeho identifikátor, rozlišení, verzi JPEG, případně náhledový obrázek. Následuje segment DQT, který obsahuje kvantizační tabulky. Ty mohou být až 4, ale zpravidla bývají použity pouze 2. Jedna pro luminiscenční složku a druhá pro barvonosné složky. Tabulky je nutno dekódovat pomocí cik-cak algoritmu, který obstarává třída *ZigZag*. Dalším segmentem je segment SOF0, který obsahuje informace o obraze. Nejdůležitější informace jsou rozměry obrazu, bitová přesnost, počet barevných komponent, pořadí komponent, příslušné

označení tabulek pro dekódování a informace o podvzorkování. Poslední segment, který je nutno načíst před dekompresí je segment DHT, který obsahuje Huffmanovy kódovací tabulky. Ty jsou zpravidla 4, 2 pro DC složky a 2 pro AC složky obrazu. Hodnoty těchto tabulek jsou načteny na začátek SRAM paměti, jelikož hodnot v jedné tabulce může být až 255.

Nyní následuje samotné dekódování obrazových dat. Celý obraz v JPEG formátu je kódován pomocí MCU[4], což je blok 8x8 pixelů při nepodvzorkované složce, případně 16x16 při podvzorkované. Pokud je obraz nepodvzorkován, jsou načteny 3 bloky MCU (Y, Cb a Cr) pro obrazovou plochu 8x8 pixelů. V případě, že je použito podvzorkování barvonosných kanálů, je načteno celkem 6 MCU bloků. Bloky luminiscenční složky se nepodvzorkovávají, tudíž na ploše 16x16 pixelů jsou 4. Dále následuje blok Cb a Cr. Než se dostaneme k samotným číselným informacím, je nutné MCU blok dekódovat z binárního proudu. Hodnoty jsou zakódovány pomocí binárního stromu, takže je nejdříve nutné detekovat VLC[5], pomocí kterého zjistíme, kolik následujících bitů nese hodnotu aplitudy, která je zakódována pomocí VLI[6]. Pokud máme načteny všechny hodnoty z binárního proudu, doplníme příslušné nuly na patřičná místa. Toto pole 64 hodnot převedeme pomocí cik-cak algoritmu na pole 8x8. Toto pole je dekvantizováno a následně je provedena inverzní diskrétní kosinová transformace. Pokud jsou načteny všechny složky obrazu, převedou se z barevného formátu YCbCr do RGB. Data jsou následně zapsána do SRAM paměti. Jakmile je načten celý řádek MCU bloků (8 nebo 16 obrazových řádků obrazu), je zapsán do výsledného souboru. V případě, že byly složky podvzorkovány, jsou při zápisu do souboru expandovány na původní rozměr. Tato operace se opakuje, dokud není dekódován celý obraz. Následně jsou všechny otevřené soubory uzavřeny a dekomprese je u konce.

[4] Minimal Code Unit
[5] Variable Length Code
[6] Variable Length Integer



Obrázek 14 Diagram dekomprese JPEG formátu

4.2.6 Zápis a kódování JPEG

Kódování JPEG souboru je inverzní operací k dékódování. Používají se stejné operace pouze v opačném pořadí.

O kompresi obrazu se stará třída *JPEG*, která je použita i pro dekódování. Komprimace obrazu začíná voláním metody *open (char * inputFile)*, která má jako vstupní parametr název souboru s dekomprimovanými údaji s příponou .bin. V tuto chvíli je vytvořen soubor JPEG se stejným názvem.

Do výsledného souboru je nutné zapsat veškeré informace o souboru. Pro urychlení operací a ušetření datového prostoru byly části JPEG souboru, které se shodují, uloženy do dvou externích souborů uložených do složky HEADERS. Soubory jsou pojmenovány Header1.bin a Header2.bin. Soubory je možné uložit jinam, popřípadě je i přejmenovat. V tomto případě je nutné provést změny v hlavičkovém souboru *JPEG_Macros.h*, konkrétně u položek JPEG_HEADER_FILE1 a JPEG_HEADER_FILE2.

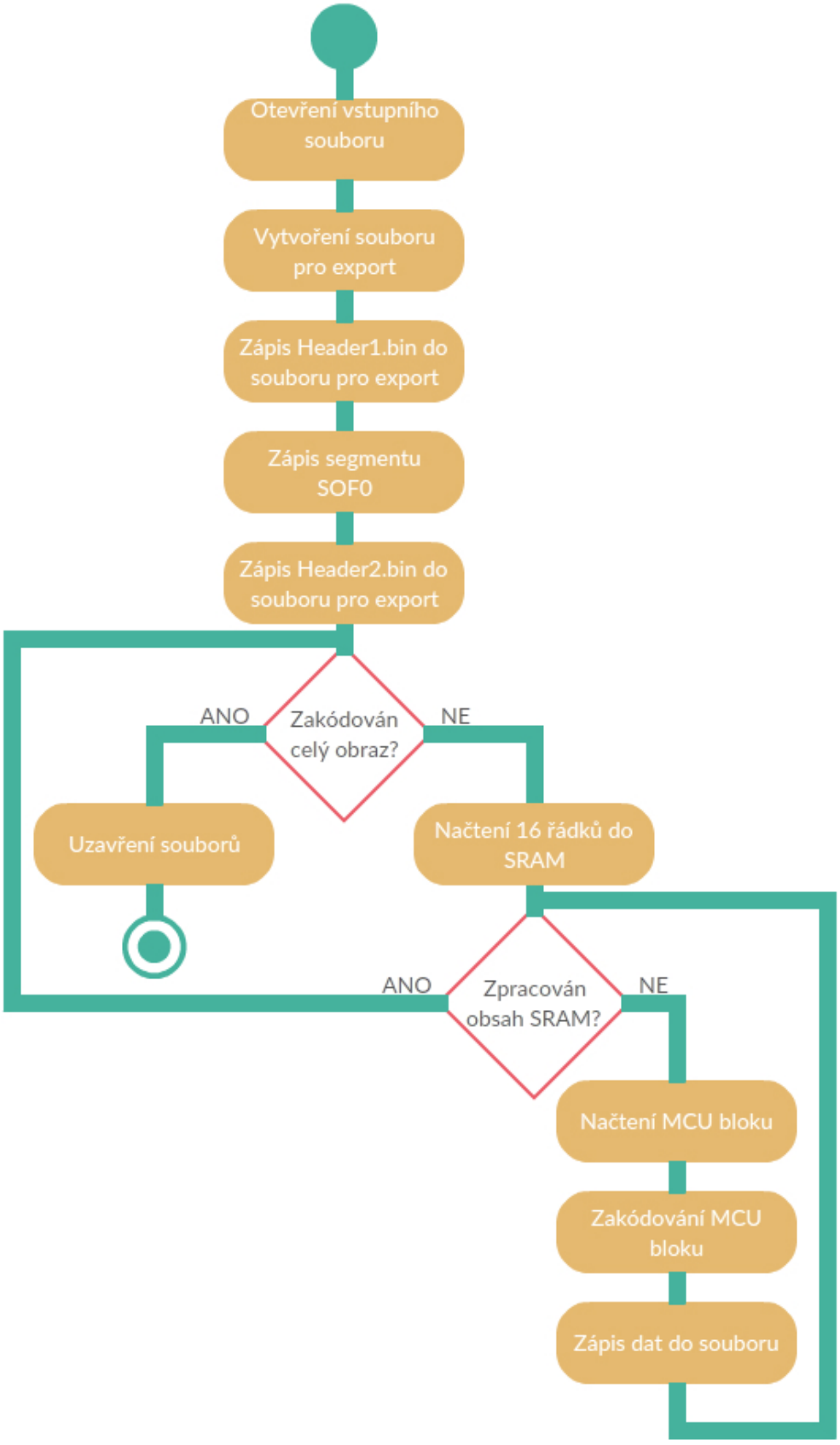
Do výsledného souboru je nejdříve zapsán první hlavičkový soubor, obsahuje informace o segmentech APP0 a kvantizačních tabulkách DQT. V tuto chvíli je otevřen nekomprimovaný vstupní soubor a jsou z něj vyčteny informace o rozlišení souboru. Tyto informace jsou použity k zapsání segmentu SOF0. Za tímto segmentem jsou uloženy Huffmanovy kódovací tabulky, které jsou uloženy v druhém hlavičkovém souboru. Tento soubor je nahrán za blok SOF0. V tuto chvíli je nutné zpracovat samotná obrazová data a uložit je do výstupního souboru.

Nejdříve jsou vypočteny rozměry obrazu, které jsou dělitelné číslem 16, jelikož výsledný soubor má podvzorkované barvonosné kanály. Dle výpočtu se pak řeší, kolik pixelů je nutné dopočítat či zkopírovat, aby byla operace úspěšná. V případě tohoto řešení probíhá kopírování posledního pixelu či řádku. Do paměti SRAM je načteno 16 řádků obrazu, zároveň dochází k eventuálnímu dokopírování potřebných pixelů. Následně je do polí načteno potencionálních 6 MCU bloků (oblast 16x16 pixelů). Bloky jsou převedeny z barevného modelu RGB do YCbCr. Další operací je podvzorkování kanálů Cb a Cr. Na každý tento MCU blok je aplikována dopředná dvourozměrná kosinová transformace. Pro urychlení následujících operací jsou výstupní hodnoty zaokrouhleny a převedeny na celočíselný datový typ. Následuje kvantizace pomocí kvantitačních tabulek. V tomto řešení

zvoleny s kvalitou 75 %. MCU bloky jsou podrobeny střídavému výběru koeficientů pomocí cik-cak algoritmu, kdy dojde k seskupení nulových hodnot. V tuto chvíli zbývá výsledné pole hodnot zakódovat do výstupního binárního proudu pomocí statických Huffmanových tabulek, které jsou uvedeny v [18].

Původním záměrem řešení bylo ušetřit co nejvíce programové paměti. 8bitová platforma totiž nabízí dosti velké úložiště v podobě EEPROM, které by se hodilo pro uložení statických Huffmanových tabulek. Bohužel z tohoto řešení sešlo. 32bitový mikro kontrolér použitý na vývojové desce pamětí EEPROM nedisponuje, tudíž by řešení byla odlišná a navzájem nekompatibilní. Došlo tedy k uložení tabulek do části programového kódu, který sice narostl, ale stále zůstalo dost programové paměti volné pro jiné funkcionality.

Z výsledného pole 64 hodnot je nejdříve zakódována stejnosměrná složka (DC), ta je v souboru zároveň kódována diferenčním způsobem, tudíž nejdříve dojde k výpočtu rozdílu mezi předchozí DC hodnotou a současnou. Nynější hodnota se uloží, bude použita při zpracování dalšího MCU bloku stejného kanálu. Nyní je potřeba DC hodnotu zakódovat pomocí VLC a VLI kódu. VLC kód nese informaci o tom, na kolik následujících bitů bude uložena ukládaná hodnota. VLC kód je definován 12 kategoriemi pro luminiscenční kanál a 12 kategoriemi pro barvonosné kanály. VLI kód nese informaci o amplitudě, která je ukládána. Střídavé složky AC jsou kódovány pomocí algoritmu, který je do jisté míry podobný s RLE. Každý koeficient je kódován pomocí dvou hodnot. První hodnota vznikne složením počtu předcházejících nulových hodnot a počtu bitů potřebných pro zápis hodnoty VLI kódem. Pomocí třídy *HuffmanCoding* se veškeré informace přepočítají a zpět se vrací pouze výsledné kódy a jejich patřičná délka. Následuje samotný zápis pomocí třídy *BitWrite*. Tyto operace se provádí cyklicky tak dlouho, dokud není zpracována celá obrazová část. Na závěr se do souboru zapíší zbylé hodnoty bitového proudu a ukončovací segmenty. Použité soubory jsou následně uzavřeny.



Obrázek 15 Diagram komprese do JPEG formátu

4.2.7 Instalace do Arduino IDE

Celé řešení bylo doplněno o možnost přidání mezi knihovny v Arduino IDE. Jediným úskalím, které bylo třeba vyřešit, bylo zvýrazňování metod a tříd. Arduino toto řeší dosti nešťastným způsobem v podobě externího textového souboru *keywords.txt*. Datové typy se označují klíčovým slovem KEYWORD1, metody klíčovým slovem KEYWORD2. Pro zvýraznění byly vybrány pouze datové typy BMP a JPEG stejnojmenných tříd a jejich metody, které slouží k otevření a uložení souboru.

Pro instalaci knihovny je potřeba celou složku s projektem zkopírovat do složky *libraries,* která bývá standardně ve Windows uložena na adrese *C:\Program Files (x86)\Arduino\libraries*. Pokud bylo při kopírování Arduino IDE spuštěno, je potřeba jej restartovat. Následně by projekt měl být viditelný v sekci *File-Examples.* Pro správnou funkci komprimace do JPEG souboru je nutné na kartu do kořenového adresáře zkopírovat složku HEADERS se soubory Header1.bin a Header2.bin.

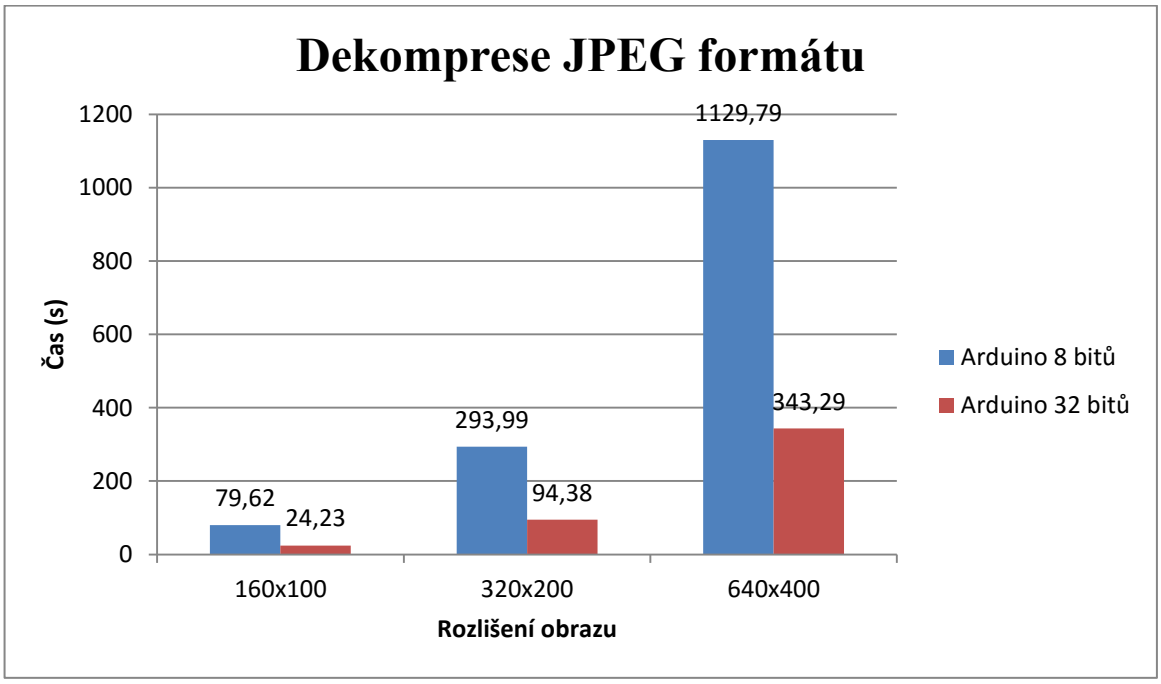
5 TESTY A EXPERIMENTY

Celé řešení bylo testováno na obrázcích v rozlišení 160x100, 320x200 a 640x400 pixelů. Testována byla dekomprese obrazu ze souboru BMP s různými počty barev, kompresí RLE, dále dekomprese těchto obrázků do souboru s dekomprimovanými daty a komprese do formátu JPEG ze souboru s dekomprimovanými daty.

Soubor s dekomprimovanými daty byl otestován na LCD displeji pomocí sketche *TestDecompresedFile.ino*. Pro správnou funkci tohoto kódu je potřeba správně nainstalovaná knihovna UTFT [19].

5.1 Testy dekomprese JPEG

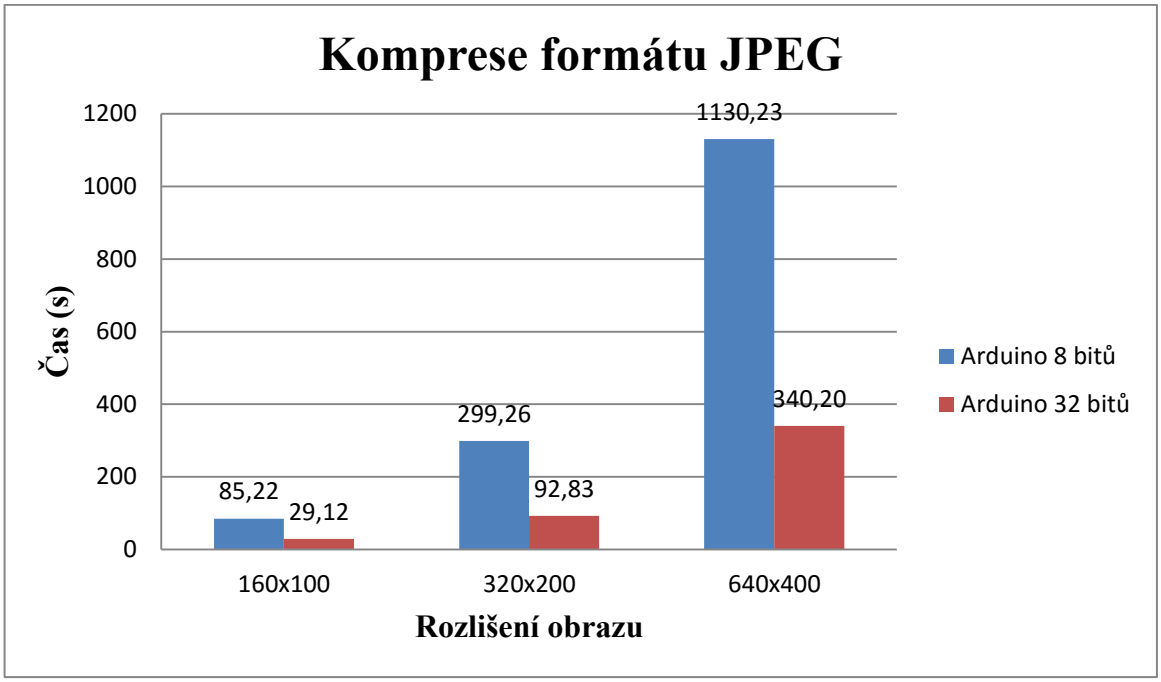
Bylo provedeno celkem 15 testů na 8bitové platformě a 15 testů na 32bitové platformě. Z průměrných časů, které jsou v níže uvedeném grafu, vyplývá, že nemá moc velký smysl testovat obrázky s vyšším rozlišení než je 640x400 pixelů. Obrázky ve větším rozlišení samozřejmě lze dekomprimovat ale čas, který je k dekomprimaci potřeba, značně narůstá.



Obrázek 16 Graf času dekomprese v závislosti na velikosti obrazu

5.2 Testy komprese do JPEG

Komprimace do formátu JPEG byla otestována na obrázcích v rozlišení 640x400, 320x200 a 160x100 pixelů na obou platformách. Výsledný obraz byl otestován náhledem v počítači a analyzován nástrojem JPEGsnoop ve verzi 1.7.5 [20]. V tomto programu je možné vyčíst detailní informace i postup, jak byly komprimovány/dekomprimovány jednotlivé bloky MCU. Komprese a dekomprese JPEG jsou navzájem opačné operace. Naměřené časy jsou hodně podobné těm, které byly naměřeny u dekomprese JPEG formátu.

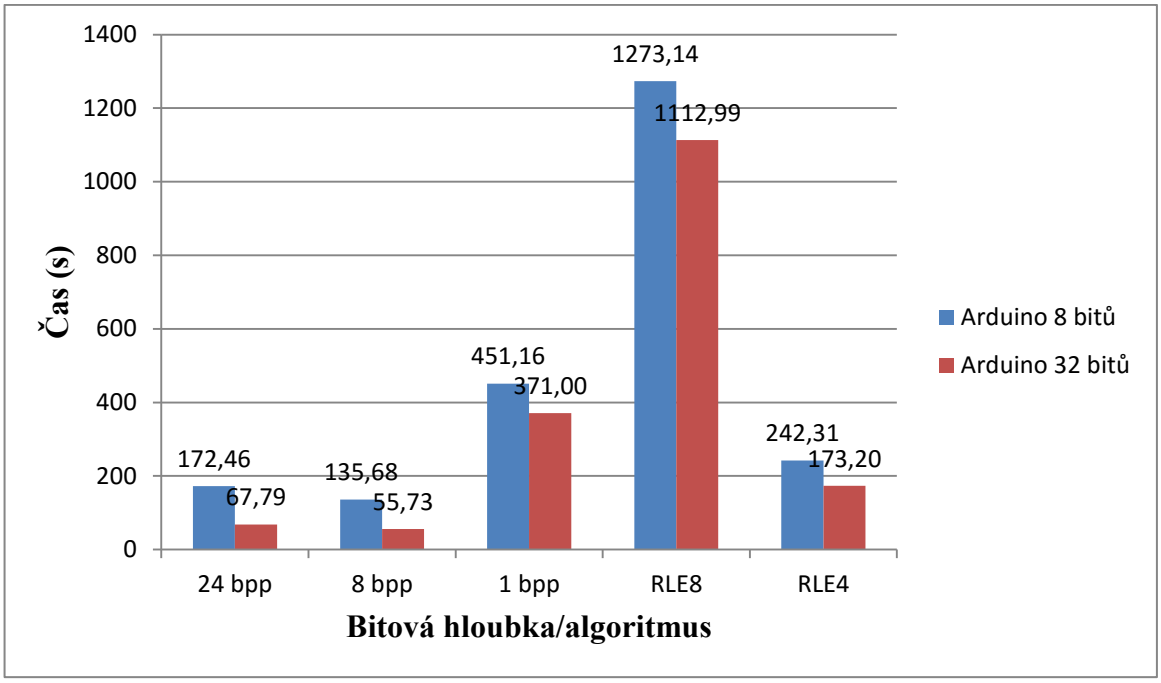


Obrázek 17 Graf času dekomprese JPEG v závislosti na rozlišení obrazu

5.3 Testy dekomprese BMP formátu

Dekomprese formátu byla testována na obrázku 640x480 pixelů v různých barevných hloubkách. Byla testována barevná hloubka 1, 4, 8 a 24 bitů na pixel a obraz uložen s podporou algoritmu RLE8 a RLE4, kde číslo odpovídá barevné hloubce. Ačkoli podpora obrázků, které mají barevnou hloubku menší než 24 bitů, není moc častá, je tato možnost zahrnuta do řešení. U dekomprese BMP formátu hraje největší roli rychlost čtení a zápisu na SD kartu, která v podání Arduino není nějak ohromující.

V následujícím grafu je uvedeno porovnání časů dekomprese obrazu v rozlišení 1, 8 a 24 bitů na pixel a časů dekomprese při použití RLE8 a RLE4 algoritmu na obou platformách.



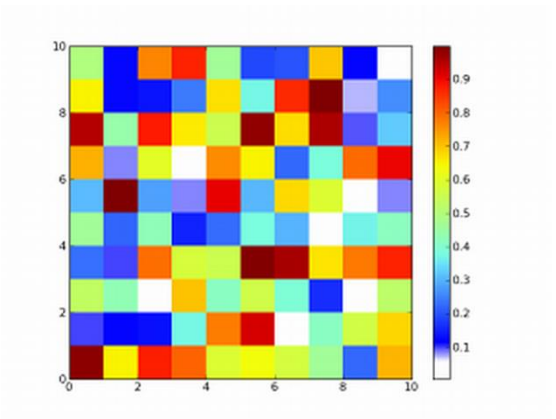
Obrázek 18 Graf času dekomprese BMP v závislosti na bitové hloubce a použitém algoritmu komprimace

5.4 Testy komprese do BMP formátu

V případě komprese do BMP formátu byly otestovány obrázky, které byly dekomprimovány pomocí JPEG algoritmu. Byly tedy uloženy jako nekomprimovaná data v souboru.

Komprese nebo spíše zápis je v řešení diplomové práci aplikován jako zápis bitmapy v bitové hloubce 24bitů na pixel, takže rychlost uložení odpovídá rychlosti čtení a zápisu SD karty.

5.5 Ukázky použitých obrázků pro experimenty



6 ZÁVĚR

Cílem diplomové práce bylo vytvoření uceleného řešení pro kompresi a dekompresi obrazu na vývojové platformě Arduino. Řešení obsahuje softwarovou a hardwarovou část. Hardwarová část v podobě shieldu, který je kompatibilní s 8 a 32 bitovou platformou Arduino. Na shieldu měla být přítomna přídavná paměť. Softwarové řešení mělo danou paměť obsluhovat a také se starat o kompresi a dekompresi obrazů. Celé řešení bylo otestováno na obou platformách. Řešení bylo vloženo na oficiální fórum Arduino [21].

Pro vytvoření shieldu s externí pamětí a slotem na SD kartu jsem použil CAD software EAGLE Light, který je sice omezen velikostí desky plošného spoje a možnou dokumentací, ale pro tohle řešení byl dostačující. Jediným omezením bylo vyřešení kompatibility mezi 8 a 32 bitovou platformou ze stránky operačního napětí. Externí paměť pracuje v rozmezí 2,5 až 5,5 V, takže ji bylo možné připojit bez starostí. SD karta má operační napětí 3,3 V, proto bylo nutné zapojit převodník napěťových úrovní.

Již z počátku vývoje se objevily problémy v nekompatibilitě zdrojového kódu mezi jednotlivými platformami. Ve většině případů šlo o datový typ *integer*, který se na 32bitové platformě nechoval jako znaménkový, ačkoliv by měl. Oprava naštěstí nebyla složitá. Postačilo dané proměnné, kde byla potřeba využít záporné hodnoty, jako *signed integer*.

Řešení části kódu, který se staral o BMP formát, se obešla bez problémů ve vývoji. Problém byl s některými grafickými programy. V soboru BMP s hlavičkou verze 4 a starší není možné rozlišit pořadí barevných kanálů, proto se může stát, že obraz bude správně dekomprimován, ale může dojít k záměně barevných kanálů. Nejrozšířenější programy ukládají barevné kanály ve správném pořadí.

Řešení dekomprese JPEG byla dosti problematická. Většina dostupných zdrojů informuje pouze o principech a algoritmech použitých v JPEG formátu. Doporučení, které se váže k normě ISO [18], zdánlivě vypadá kompletní, ale některé důležité informace je obtížné najít nebo se v dokumentu vůbec nenacházejí.

Velice cenným nástrojem, který byl využíván po celou dobu vývoje, se stal program JPEGsnoop [20]. Díky tomuto programu bylo možné odhalovat chyby už od počátku vývoje, jelikož umožnuje detailní dekompresi s popisy operací.

Celý proces spojený s implementací byl velice zdlouhavý, jelikož Arduino má nízký výkon a veškeré testy a výpočty musely být distribuovány skrze sériovou konzoli, která snižovala výkon. V závěru návrhu dekomprese JPEG se objevily problémy s diferenčním kódováním, které je použito pro DC komponenty obrazu. Tyto problémy byly odstraněny.

Původním záměrem bylo ušetření co nejvíce Flash a SRAM paměti v mikro kontroléru. Bohužel mikro kontrolér, který je použit na desce 32bitové platformy nedisponuje EEPROM pamětí. Proto bylo nutné naimplementovat Huffmanovy kódovací tabulky do zdrojového kódu, který se patřičně zvětšil. Při přepisování těchto kódů mohla vzniknout chyba, ačkoli byly kódy několikrát kontrolovány. Několik chyb bylo opraveno při testování.

Celé řešení bylo otestováno na sérii obrázků v různých rozlišeních a v případě BMP formátu i různé barevné hloubce. Testům byla věnována celá předchozí kapitola. Rychlosti komprese a dekomprese není sice nějak závratná, ale je funkční.

Z časů, které byly získány při testování, vyplývá, že 8bitová platforma není díky nízkému výkonu vhodná pro operaci s rastrovým obrazem. Hlavní příčinou je absence jednotky pro operace s plovoucí desetinnou čárkou. S desetinnými čísly je možno pracovat, kompilátor absenci jednotky řeší delším kódem, díky čemuž klesá výpočetní výkon. Výkon 32bitové platformy je daleko vyšší nejen díky taktu 84 MHz, ale i délce zpracovaného slova. Bohužel mikro kontrolér také postrádá jednotku pro práci s desetinnými čísly a řeší tyto operace hrubou silou.

Přínos své diplomové práce vidím v prohloubení znalostí převážně o JPEG formátu a algoritmech v něm použitých. Velký přínos může být pro komunitu, která se soustředí okolo platformy Arduino, jelikož kód i s dokumentací bude sdílen na fóru a kdokoli se bude moct zapojit do modifikací či optimalizací. Celá tato práce může najít také uplatnění v nízkonákladových zařízeních pro zaznamenávání obrazu, aplikace má však své limity. Pokud takováto zařízení budou schopna řešit složité problémy, bude možné je používat v náročném prostředí, jelikož spotřebovávají méně energie než výkonné počítače. Již teď se nacházejí ve velkém množství spotřebičů, kde řeší složité operace.

RESUMÉ

Diplomová práce je zaměřena na problematiku komprese a dekomprese obrazu na vývojové platformě Arduino. Práce je členěna do čtyř částí. První část je věnována samotné vývojové platformě Arduino. V úvodu se nachází shrnutí implementací platformy Arduino jako nástroj komprese či dekomprese rastrového obrazu. Dále je zde popsáno vývojové prostředí a také vývojové desky, které byly použity pro tuto práci. V druhé části je rozebrána problematika komprese obrazu. Podrobně je zde popsán rastrový formát BMP a JPEG včetně operací, které probíhají během komprimace.

Další kapitola se zabývá samotným návrhem řešení, které umožňuje realizovat kompresi a dekompresi obrazu na platformě Arduino. Je zde popsán návrh rozšiřující desky (Shieldu) obsahující přídavnou SRAM paměť, která je kompatibilní s 8 a 32bitovou verzí Arduino. Také je zde popsáno softwarové řešení, které se stará o obsluhu shieldu a samotnou realizaci s rastrovým obrazem.

V závěru práce se nachází testy a experimenty, které byly provedeny jako ověření funkčnosti a výkonnosti celého řešení.

Cílem této práce bylo vytvořit ucelené řešení pro kompresi a dekompresi rastrového obrazu na vývojové platformě Arduino. Podporován je formát souboru BMP s podporou základní komprese a sekvenční JPEG s barevnou hloubkou 24 bitů.

Celé hardwarové a softwarové řešení bylo odzkoušeno na 8bitové a 32bitové vývojové verzi platformy Arduino.

Z testů a experimentů vyplývá, že komprese a dekomprese rastrového obrazu na platformě Arduino je realizovatelná. Výkon platformy (hlavně 8bitové) je hodně nízký, proto časová náročnost komprimace či dekomprimace obrazu je velice vysoká. Velkým problémem je absence jednotky pracující s desetinnou čárkou, což způsobuje rapidní nárůst času při provádění diskrétní kosinové transformace.

SUMMARY

The diploma thesis is focused on the compression and decompression of the image on the Arduino development platform. The work is divided into four parts. The first part is dedicated to the development platform Arduino. The introduction is a summary of the implementation of the Arduino platform as a tool for compression or decompression of raster image. Furthermore, there is disclosed a development environment and development boards, which were used for this work. The second part discusses the problem of image compression. There are described in detail raster format BMP and JPEG, including operations that take place during compression.

Another chapter deals with the design solution that allows realizing image compression and decompression on this platform. There is described a proposal expansion boards (Shield) containing additional SRAM memory that is compatible with 8 and 32-bit versions of Arduino. Also disclosed is a software solution that takes care of the operation of the Shield and realization of a raster image.

In conclusion, there are tests and experiments that were conducted as to verify the functionality and performance of the solution.

The aim of this work was to create a comprehensive solution for compression and decompression of raster image on the Arduino development platform. It supports BMP file format and basic sequential JPEG compression with color depth of 24 bits.

The entire hardware and software solutions have been tested at 8-bit and 32-bit version of the development platform Arduino.

From tests and experiments suggest that compression and decompression of raster image platform Arduino is realizable. Platform performance (especially 8-bit) is very low, therefore, time-consuming compression or decompression of the image is very high. The big problem is the lack of units working with a decimal point, causing a rapid rise time in the implementation of discrete cosine transform.

SEZNAM POUŽITÉ LITERATURY

[1] LAZAR, Jan. *Oživení souřadnicového zapisovače ALFI ze stavebnice Merkur pod Windows*. Ostrava, 2014. Bakalářská práce. Ostravská univerzita. Vedoucí práce RNDr. Marek Vajgl, Ph.D.

[2] GELDREICH, Rich. Picojpeg. *Google Code Archive* [online]. 2010 [cit. 2015-01-01]. Dostupné z: https://code.google.com/p/picojpeg/

[3] BROWN, Andy. Nokia QVGA TFT LCD for the Arduino Mega. Graphics Library (part 2 of 2). *Andy´s workshop* [online]. 2012 [cit. 2015-09-10]. Dostupné z: http://andybrown.me.uk/2012/06/04/nokia-qvga-tft-lcd-for-the-arduino-mega-graphics-library-part-2-of-2/

[4] MARIAN, P. Arduino Mega 2560 Pinout. *Electronics Projects Circuits* [online]. [cit. 2016-01-10]. Dostupné z: http://www.electroschematics.com/7963/arduino-mega-2560-pinout/

[5] Arduino: ArduinoBoardDue. *Arduino.cc* [online]. [cit. 2016-01-05]. Dostupné z: https://www.arduino.cc/en/Main/ArduinoBoardDue

[6] ŽÁRA, Jiří. *Moderní počítačová grafika*. 2., přeprac. a rozš. vyd. Praha: Computer Press, 2004. ISBN 80-251-0454-0.

[7] SOBOTA, Branislav a Ján MILIÁN. *Grafické formáty*. 1. vyd. České Budějovice: Kopp, 1996. ISBN 80-858-2858-8.

[8] BITMAPV5HEADER structure. *Microsoft* [online]. [cit. 2016-03-05]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/dd183381%28v=vs.85%29.aspx

[9] MS-Windows .bmp RLE8. *BinaryEssence* [online]. [cit. 2016-02-20]. Dostupné z: http://www.binaryessence.com/dct/en000053.htm

[10] MS-Windows .bmp RLE4. *BinaryEssence* [online]. [cit. 2016-02-20]. Dostupné z: http://www.binaryessence.com/dct/en000072.htm

[11] PIHAN, Roman. Formáty pro ukládání fotografií - 2.díl: jpeg. *Digimanie* [online]. 2007 [cit. 2016-03-20]. ISSN 1214-2190. Dostupné z:

http://www.digimanie.cz/formaty-pro-ukladani-fotografii-2dil-jpeg/1970

[12] ČAPEK, Jan a Peter FABIÁN. *Komprimace dat: principy a praxe*. Vyd. 1. Praha: Computer Press, 2000. Internet. ISBN 80-722-6231-9.

[13] Encoding process. *CMLaboratory* [online]. [cit. 2016-03-20]. Dostupné z: http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/jpeg/jpeg/encoder.htm

[14] TIŠNOVSKÝ, Pavel. Programujeme JPEG: Interní struktura souborů typu JFIF/JPEG. *ROOT.CZ* [online]. 2007 [cit. 2016-03-20]. ISSN 1212-8309. Dostupné z: http://www.root.cz/clanky/programujeme-jpeg-interni-struktura-souboru-typu-jfifjpeg/

[15] RIVA. 23LC1024 SRAM Library. *Arduino.cc* [online]. 2013, 28.12.2013 [cit. 2015-10-09]. Dostupné z: http://forum.arduino.cc/index.php?topic=182918.0

[16] Bi-Directional Logic Level Converter (4 Channel). *Addicore.com* [online]. [cit. 2016-03-05]. Dostupné z: http://www.addicore.com/Logic-Level-Converter-Bi-Directional-5V-to-3-3V-p/227.htm

[17] SLINTÁK, Vlastimil. Konverze mezi 5V a 3,3V logikou. *UArt.cz* [online]. 2011 [cit. 2015-11-12]. Dostupné z: http://uart.cz/253/konverze-mezi-5v-a-3v-logikou/

[18] CCITT REC. T.81 (1992 E). *INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES*. INTERNATIONAL TELECOMMUNICATION UNION, 1992.

[19] Library: UTFT. *Rinky-Dink Electronics* [online]. 2016 [cit. 2016-04-07]. Dostupné z: http://www.rinkydinkelectronics.com/library.php?id=51

[20] HASS, Calvin. JPEGsnoop 1.7.5 - JPEG File Decoding Utility. *ImpulseAdventure* [online]. 2015 [cit. 2016-04-11]. Dostupné z: http://www.impulseadventure.com/photo/jpeg-snoop.html

[21] LAZAR, Jan. Arduino JPEG and BMP library with SRAM Shield. *Arduino.cc* [online]. [cit. 2016-04-20]. Dostupné z: http://forum.arduino.cc/index.php?topic=395102.msg2719055#msg2719055

SEZNAM POUŽITÝCH SYMBOLŮ

TTL	Transistor-transistor Logic
AVR	Architektura 8bitových mikrokontrolérů společnosti ATMEL
ARM	Architektura 32bitových mikrokontrolérů
ISCP	In System Circuit Programing
SPI	Serial Peripheral Interface
TWI	Two Wire serial Interface
USART	Universal Synchronous / Asynchronous Receiver and Transmitter
SD	Secure Digital
DCT	Diskrétní kosinová transformace
FDCT	Dopředná diskrétní kosinová transformace
IDCT	Inverzní diskrétní kosinová transformace
EEPROM	Electrically Erasable Programmable Read-Only Memory
RLE	Run Length Encoding
PWM	Pulse Width Modulation
MCU	Minimal Code Unit
VLC	Variable Length Code
VLI	Variable Length Integer

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

Příloha 1: Schéma zapojení SRAM shieldu

Příloha 2: Schéma plošného spoje SRAM shieldu

Příloha 3: CD obsahující:

- Elektronickou verzi práce,
- zdrojové kódy,
- schémata zapojení z programu EAGLE Lite.