

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta elektrotechnická
Katedra elektroniky a informačních technologií

DIPLOMOVÁ PRÁCE

Využití IR kamerových modulů

Autor práce: **Bc. Jan Pillmann**
Vedoucí práce: **Ing. Petr Weissar,
Ph.D**

2024

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta elektrotechnická
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: **Bc. Jan PILLMANN**
Osobní číslo: **E22N0046P**
Studijní program: **N0714A060013 Elektronika a informační technologie**
Specializace: **Elektronika**
Téma práce: **Využití IR kamerových modulů**
Zadávací katedra: **Katedra elektroniky a informačních technologií**

Zásady pro vypracování

Prozkoumejte a prakticky implementujte IR kamerové moduly. Využijte dodané HW prostředky:

1. Vývojový kit pro STM32 a kamerové moduly pro IR a viditelné světlo
2. IR snímače využívané v projektu CubeSAT

Pro dodaný HW připravte vhodné příklady a dokumentaci umožňující jejich další nasazení v projektech.

Rozsah diplomové práce: **40-60**
Rozsah grafických prací:
Forma zpracování diplomové práce: **elektronická**

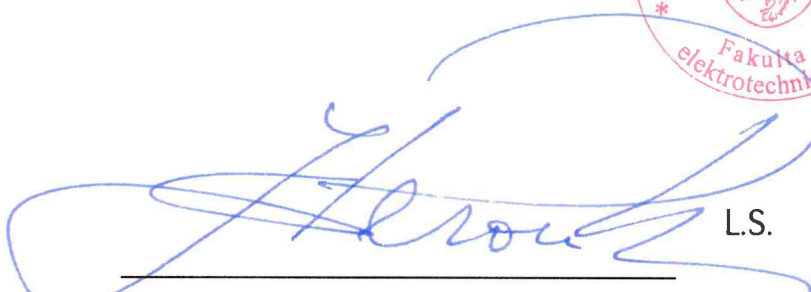
Seznam doporučené literatury:

1. Yiu, Joseph. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors, Elsevier Science & Technology, 2013.
2. Dokumentace výrobců.

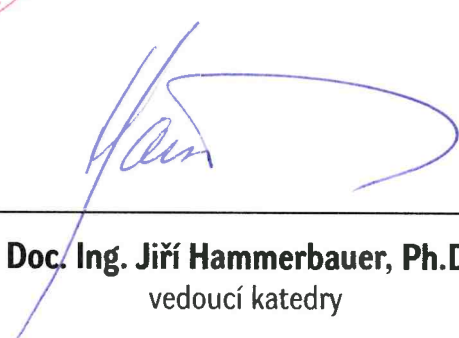
Vedoucí diplomové práce: **Ing. Petr Weissar, Ph.D.**
Katedra elektroniky a informačních technologií

Datum zadání diplomové práce: **6. října 2023**
Termín odevzdání diplomové práce: **24. května 2024**




L.S.

Prof. Ing. Zdeněk Peroutka, Ph.D.
děkan



Doc. Ing. Jiří Hammerbauer, Ph.D.
vedoucí katedry

V Plzni dne 6. října 2023

Abstrakt

Práce se zaměřuje na možnosti použití běžně dostupných kamerových modulů snímajících v infračerveném a viditelném spektru pro projekty vestavěné elektroniky. Cílem bylo porovnat jednotlivé moduly, popsat datovou komunikaci a nakonec vytvořit ukázkové příklady použití. Výsledkem práce je ukázkové řešení programu pro mikrokontrolér platformy STM32, aplikačního protokolu založeném na UDP pro přenos snímků po počítačové síti a počítačové aplikaci v jazyce C#, která obrazová data dekoduje, vizuálně zobrazí analyzuje. Vykreslování snímků v reálném čase zajišťuje grafická knihovna OpenGL, přes kterou bylo zároveň implementováno několik vizuálních funkcí zvyšujících informační hodnotu pořízených dat, a to zejména aplikování několika vybraných typů palet barev, plynulé prolínání spekter snímku nebo dynamické sledování teplotních extrémů. Dále je součástí řešení podrobný popis softwarové části u hlavních principů a algoritmů. Jednotlivé dílčí výsledky byly koncipovány obecně a lze je modulárně použít pro řešení vlastního problému.

Klíčová slova

Kamerový modul, termální kamera, infračervené spektrum, viditelné spektrum, FLIR Lepton, MLX90640, AMG8833, Arducam, STM32, TCP/IP, Ethernet, lwIP, UDP, C#, Winforms, OpenGL, OpenTK, paleta barev, vykreslování proudu snímků

Abstract

The thesis focuses on the possibilities of using commonly available infrared and normal camera modules for embedded electronics projects. The aim was to compare individual modules, describe data communication, and ultimately create usage examples. The solution consists of example program for STM32 microcontroller, application protocol based on UDP for image transmission over a computer network, and a computer application written in C# that decodes image data, visually displays and analyzes them. OpenGL provides real-time image rendering including several visual functions, which enhance the informational value of the captured data. Some of these visual functions include: application of color palettes, smooth blending of the captured spectrums or dynamic tracking of temperature extremes. Furthermore, the solution includes a detailed description of the software part regarding main principles and algorithms. Individual partial results were implemented generally and can be modularly used to solve own specific problems.

Keywords

Camera module, thermal camera, infrared spectrum, visible spectrum, FLIR Lepton, MLX90640, AMG8833, Arducam, STM32, TCP/IP, Ethernet, lwIP, UDP, C#, Winforms, OpenGL, OpenTK, color palette, real-time image rendering.

Obsah

Seznam použitých symbolů a zkratk	vi
Seznam obrázků	vii
Seznam tabulek	ix
Úvod	1
1 Přehled vybraných kamerových modulů	2
1.1 Požadavky	2
1.1.1 Rozlišení, formát a snímková frekvence	3
1.1.2 Komunikační rozhraní	3
1.1.3 Prostředí	4
1.1.4 Záření v prostoru vesmíru	4
1.2 Arducam	6
1.2.1 Camera Parallel Interface (CPI)	7
1.2.2 Serial Camera Control Bus (SCCB) specifikace	8
1.2.3 SCCB komunikační protokol pro 8bitové registry	9
1.2.4 SCCB rozšíření komunikačního protokolu pro 16bitové registry	10
1.2.5 Čtení snímků	10
1.2.6 Arducam Mini 2MP Plus	12
1.3 Arducam Mini 5MP Plus	13
1.4 FLIR Lepton	13
1.5 AMG88XY	14
1.5.1 Porovnání jednotlivých variant	15
1.5.2 Komunikační protokol	16
1.5.3 Inicializace	16
1.5.4 Přerušení	17
1.5.5 Průměrování	17
1.5.6 Termistor	18
1.5.7 Čtení snímků	18
1.6 MLX90640 a MLX90641	19
1.6.1 Snímková frekvence	20
1.6.2 Komunikační protokol, požadavky na ovladač sběrnice	21
2 Komunikace v počítačových sítích	22
2.1 ISO/OSI referenční model	22

2.2	TCP/IP	22
2.2.1	Vrstva síťového přístupu	23
2.2.2	Síťová vrstva	23
2.2.3	Transportní vrstva	24
2.2.4	Aplikační vrstva	25
2.2.5	Maximální velikost aplikačních dat UDP datagramu	25
2.3	Ethernet	25
3	Použití kamerových modulů s STM32	27
3.1	Vývojová deska a přípravek pro připojení kamerových modulů	27
3.1.1	Ethernet	28
3.1.2	Hodinové takty	28
3.2	lwIP TCP/IP stack	29
3.2.1	Paketový buffer	29
3.2.2	Konfigurace	31
3.2.3	Inicializace rozhraní	31
3.2.4	Kontrola stavu rozhraní	32
3.2.5	UDP Protocol Control Block (PCB)	33
3.2.6	Příjem UDP datagramu	33
3.2.7	Odeslání UDP datagramu	34
3.3	Aplikační protokol	35
3.4	UDP spojení	36
3.5	Struktura programu	36
3.6	Arducam Mini 2MP Plus	37
3.6.1	Ovladač SCCB sběrnice	38
3.6.2	Čtení snímků	39
3.7	FLIR Lepton 2.5	40
3.7.1	Ovladač SPI sběrnice	40
3.7.2	Čtení snímků	43
3.8	AMG8833	44
3.8.1	Čtení snímků	44
3.8.2	Čtení teploty termistoru	45
3.9	MLX90640	46
3.9.1	Ovladač I ² C sběrnice	46
3.9.2	Inicializace	48
3.9.3	Čtení snímků	48
4	OpenTK	50
4.1	Principy vykreslování	50
4.1.1	Textura	51
4.1.2	Shader	52
4.1.3	Vertex shader	53
4.1.4	Fragment shader	53
4.1.5	Kompilace shaderů a vytvoření programu grafické karty	54

4.1.6	Nahrání souřadnic do vertex shaderu	55
4.1.7	Nahrání textury do fragment shaderu	57
4.1.8	Vykreslení obdélníku s texturou	59
4.2	Vykreslení fontu	59
4.2.1	Bitmapová textura fontu	60
4.2.2	Fragment shader	61
4.3	Způsoby nahrávání globálních dat do shaderu	61
4.3.1	Textura	62
4.3.2	Uniform Buffer Object (UBO)	62
4.3.3	Shader Storage Buffer Object (SSBO)	64
5	Aplikace pro zpracování dat	65
5.1	Uživatelské rozhraní	65
5.2	Struktura aplikace	66
5.2.1	IRClientSettings	66
5.2.2	IRClientConnection	67
5.2.3	IRClientRendering	68
5.3	Dekódování snímků v přijatých paketech	69
5.3.1	Arducam Mini 2MP Plus	69
5.3.2	FLIR Lepton 2.5	70
5.3.3	AMG8833	71
5.3.4	MLX90640	72
5.4	Palety barev termálních snímků	73
5.4.1	Příprava dat	73
5.4.2	Aplikování palety ve fragment shaderu	74
5.5	Vykreslování	75
5.5.1	Struktura vykreslovací logiky	76
5.5.2	Vykreslování snímků	76
5.5.3	Překryvné uživatelské rozhraní	77
6	Závěr	79
	Seznam použité literatury	84
	Přílohy	A
6.1	Ukázka grafického výstupu aplikace pro zpracování dat	A

Seznam použitých symbolů a zkratek

Značka	Popis	Jednotka
<i>GPIO</i>	General-Purpose Input/Output	-
<i>CPI</i>	Camera Paralel Interface	-
<i>SPI</i>	Serial Paralel Interface	-
<i>TCP</i>	Transmission Control Protocol	-
<i>UDP</i>	User Datagram Protocol	-
<i>IP</i>	Internet Protocol	-
<i>MAC</i>	Medium Access Control	-
<i>MII</i>	Medium Independent Interface	-
<i>RMII</i>	Reduced Medium Independent Interface	-
<i>LLC</i>	Logical Link Control	-
<i>LLCPDO</i>	Logical Link Control Protocol Data Unit	-
<i>IP</i>	Internet Protocol	-
<i>ICMP</i>	Internet Control Message Protocol	-
<i>IGMP</i>	Internet Group Management Protocol	-
<i>ARP</i>	Address Resolution Protocol	-
<i>PCB</i>	Protocol Control Block	-
<i>VBO</i>	Vertex Buffer Object	-
<i>VAO</i>	Vertex Array Object	-
<i>UBO</i>	Uniform Buffer Object	-
<i>SSBO</i>	Shader Storage Buffer Object	-

Seznam obrázků

1	Blokové schéma vnitřních obvodů snímače OV2640, převzato z [5]	7
2	Časování signálů CPI obrazového rozhraní snímače OV5642, převzato z [8]	8
3	Časový diagram datového přenosu po SCCB sběrnici, převzato z [10]	8
4	Rezistor eliminující poškození zařízeníů připojených na SCCB sběrnici při kolizi (zkratu) na datovém vodiči <i>SIO_D</i> , převzato z [10]	9
5	Zapisovací cyklus SCCB sběrnice pro 8bitový registr [10]	9
6	Čtecí cyklus SCCB sběrnice pro 8bitový registr [10]	10
7	Časový diagram SPI zapisovacího cyklu do registru, převzato z [11]	11
8	Časový diagram základního SPI čtecího cyklu z registru, převzato z [11]	11
9	Časový diagram SPI dávkového/čtecího cyklu z framebufferu, převzato z [11]	12
10	Blokové schéma modulu z řady AMG88XY, převzato z [16]	14
11	Diagram zapisovacího cyklu do registru modulu AMG88XY po I ² C sběrnici, převzato z [17]	16
12	Diagram čtecího cyklu registru modulu AMG88XY po I ² C sběrnici, převzato z [17]	16
13	Nákres modulu AMG88XY s vyznačením snímaného pole s uspořádáním pixelů, převzato z [16]	19
14	Blokové schéma modulů MLX90640 a MLX90641, převzato z [19]	19
15	Obnovovací frekvence modulů MLX90640 a MLX90641, převzato z [19]	20
16	Zapisovací cyklus I ² C sběrnice u modulů MLX90640 a MLX90641, převzato z [19]	21
17	Čtecí cyklus I ² C sběrnice u modulů MLX90640 a MLX90641, převzato z [19]	21
18	Návrh desky plošného spoje přípravku pro připojení kamerových modulů k STM32 vývojové desce.	28
19	Konfigurace hodinových taktů jádra procesoru a periférií.	29
20	Vývojový diagram základních stavů řídicího programu kamerového modulu Arducam	38
21	Vývojový diagram základních stavů řídicího programu kamerového modulu FLIR Lepton	40
22	Záznam z osciloskopu přerušovaného SPI přenosu slova 0x05 (<i>CPOL</i> = 1, <i>CPHA</i> = 1)	42
23	Vývojový diagram algoritmu programu kamerového modulu AMG8833	44
24	Vývojový diagram základních stavů řídicího programu infračerveného kamerového modulu MLX90640	46
25	Aproximace koule s různým počtem trojúhelníků, převzato z [31]	50
26	NDC souřadný systém, ve kterém leží obdélník vytvořený ze dvou trojúhelníků	51
27	Pipeline struktura OpenGL, převzato z [32]	52

28	Účinek jednotlivých filtrů textury při zvětšování, pro levý obrázek je použito nastavení <i>nearest</i> a pro pravý <i>linear</i>	59
29	Ukázka definice znaku fontu přes matematicky definované křivky, převzato z [37]	60
30	Aproximace znaku fontu trojúhelníky, převzato z [38]	60
31	Ukázka části textury fontu	60
32	Ukázka uživatelského rozhraní aplikace pro zpracování dat	65
33	Přehled hlavních objektů aplikace	66
34	Přehled objektů s konfigurací aplikace	67
35	Přehled objektů řízení UDP komunikace	68
36	Přehled objektů vykreslování	68
37	Diagram třídy <i>rawImage</i> a <i>FragmentReceiver</i>	69
38	Zdrojová data palet barev (seshora Ironbow, Glowbow, Rainbow), převzato z [43]	73
39	Závislost normalizované velikosti barevné složky na normalizované velikosti teploty u palety barev Ironbow	74
40	Přehled objektů zabalujících volání OpenTK	76
41	Diagramy tříd pro dynamické vykreslování textu	78
42	Textura ukazatele teplotních bodů	78
43	Ironbow paleta barev se zobrazením pouze infračerveného spektra	A
44	Ironbow paleta barev s prolnutím infračerveného spektra s viditelným spektrem .	B
45	Zobrazení pouze viditelného spektra	C
46	Ironbow paleta barev s pevně nastaveným mapováním na teplotní rozsah	D
47	Glowbow paleta barev	E
48	Rainbow paleta barev	F
49	Greyscale paleta barev	G

Seznam tabulek

1	Přehled parametrů kamerového modulu Arducam Mini 2MP Plus [13]	13
2	Přehled parametrů kamerového modulu Arducam Mini 5MP Plus [14]	13
3	Přehled parametrů různých verzí modulu Lepton [15]	14
4	Přehled jednotlivých označení modulů AMG88XY [16]	15
5	Teplotní rozsahy variant modulu s vysokým a nízkým zesílením [16]	15
6	Přehled společných parametrů všech variant kamerových modulů AMG88XY [16]	15
7	Parametry infračervených kamerových modulů MLX90640 a MLX90641 [19][20] .	20
8	Srovnání vrstev ISO/OSI referenčního modelu s TCP/IP zásobníkem protokolů [23][24]	23
9	Typy paketů aplikačního protokolu	35
10	Typy akcí systémového paketu aplikačního protokolu	36
11	Struktura aplikačního protokolu pro přenos snímků kamerového modulu Arducam Mini 2MP Plus	40
12	Struktura aplikačního protokolu pro přenos snímků infračerveného kamerového modulu FLIR Lepton 2.5	44
13	Struktura aplikačního protokolu pro přenos snímků a teploty termistoru infračerveného kamerového modulu AMG8833	45
14	Struktura aplikačního protokolu pro přenos snímků infračerveného kamerového modulu MLX90640	49
15	Struktura zdrojového pole pro nahrání dat do UBO s rozložením std140 v případě skalárních datových typů	63
16	Struktura zdrojového pole pro nahrání dat do UBO s rozložením std140 v případě vektorových datových typů	63
17	Struktura zdrojového pole pro nahrání dat do UBO s rozložením std140 v případě pole skalárních a vektorových datových typů	64

Úvod

Kamerové moduly nabízí možnost efektivně integrovat funkci snímání obrazu do projektů s vestavěnou elektronikou. Kromě zaběhlých kamerových modulů snímajících ve viditelném spektru existují i termální kamerové moduly snímající v infračerveném spektru. Spojením snímání obrazu v obou spektrech vzniká mocný nástroj pro sledování okolního prostoru a společně se strojovým zpracováním získaných dat lze okolní dění automaticky analyzovat a vyhodnocovat. Mezi možnosti využití se řadí například pokročilá detekce osob, zvířat, či jiných objektů, prevence vzniku požáru u nehlídaných spotřebičů v domácnosti, automatická detekce poruch průmyslových zařízení a mnoho dalších. Výsledky práce budou využity k nasazení kamerových modulů ve studentském satelitu Pilsen CubeSAT s cílem detekovat horizont Země z pořízeného snímku a podle toho určit orientaci satelitu v prostoru.

Diplomová práce navazuje na bakalářskou práci [1], kterou rozšiřuje o další infračervené kamerové moduly, kamerové moduly viditelného spektra, přenos snímků po počítačové síti a pokročilé zpracování obrazových dat s hardwarově akcelerovaným vykreslováním pomocí grafické karty. Cílem práce je porovnat vybrané kamerové moduly, popsat jejich komunikační rozhraní a připravit ukázkové příklady implementace. Práce popisuje proces od získání snímku s mikrokontrolérem STM32, až po konečné vizuální reprezentování a analýzu dat v počítačové aplikaci. Byl kladen důraz na to, aby konkrétní výsledky práce byly řešeny obecným způsobem, a díky tomu je možné výsledky upravit a aplikovat na řešení vlastního problému. Ve druhé části práce je podrobně rozebráno řešení praktické části a jsou vysvětleny použité principy a algoritmy.

1 Přehled vybraných kamerových modulů

V nejjednodušší formě je kamerový modul elektrooptické zařízení integrující všechny potřebné součástky pro zajištění správné funkce obrazového snímače. Kromě samotného snímače dále integruje zejména čočku, řídicí elektroniku, komunikační rozhraní a vyrábí se v pouzdrech k přímému pájení na desku plošných spojů. Často ale stále vyžaduje řadu externích součástí, jako jsou regulátory napětí, krystalové oscilátory a pasivní součástky. Existují ale i moduly, které kromě kamerového modulu integrují veškeré externí součástky a v některých případech i přídatné integrované obvody poskytující rozšířené funkce nad rámec samotného modulu. Takovéto moduly jsou vyráběny v podobě plošného spoje s konektorem pro připojení do nadřazeného systému.

Kamerové moduly jsou obecně koncipovány k univerzálnímu použití v různorodých projektech a značně zjednodušují implementaci snímání obrazu. V případě použití na embedded platformách jsou uzpůsobené mikrokontrolerům s relativně nízkým výpočetním výkonem a proces snímání obrazu značně zjednodušují. Integrují běžně používaná rozhraní, nejčastěji SPI, I²C nebo UART a obvykle řeší veškerý postprocessing zaznamenaného snímku.

1.1 Požadavky

Volba modulu závisí na konkrétních požadavcích aplikace, mezi hlavní patří například:

- snímané spektrum
- rozlišení
- formát snímků, podpora komprese
- komunikační rozhraní, rychlost přenosu
- zorné pole, možnost výměny čočky
- velikost vnitřní paměti (framebuffer)
- odolnost vůči vlivům prostředí (teplota, vlhkost, záření)
- spotřeba energie

Jedno z předpokládaných využití této práce je nasazení kamerových modulů v projektech pro strojové vyhodnocování zachycených snímků, například detekce a určení polohy horizontu Země v satelitu CubeSat nebo sledování polohy osob a vybraných předmětů ve vnitřních prostorech. Implementace musí být možná i na systémech s nízkým výpočetním výkonem a malou operační pamětí, zejména na mikrokontroléru STM32F103.

1.1.1 Rozlišení, formát a snímková frekvence

Pro účely strojového vyhodnocování snímků ve zmíněných projektech není vysoké rozlišení příliš důležité. Naopak, mít možnost snížit rozlišení na velikost okolo 320x240 pixelů je potřeba, jelikož mikrokontrolér STM32F103 v satelitu CubeSAT nemá dostatečný výpočetní výkon ani paměť pro zpracování snímku ve vyšším rozlišení. Výhodou kamerového modulu je možnost dynamicky přepínat rozlišení snímače, kdy je snímek nejprve vyhodnocen v nízkém rozlišení a případně je poté uložen na externí paměťové médium ve vyšším rozlišení.

Satelit disponuje komunikačním rozhraním s pozemní stanicí o nízké přenosové rychlosti, a proto je klíčové omezit objem přenášených dat. Datovou velikost snímku ovlivňuje kromě zvoleného rozlišení také formát snímku a po kamerovém modulu je požadována podpora komprese. Zamýšlený postup je takový, že se snímek nejdříve zachytí v nízkém rozlišení, vyhodnotí se buď přímo na satelitu, nebo se odešle do pozemní stanice, a na základě toho bude snímek případně uložen v plném rozlišení na externí paměť pro pozdější vyhodnocení.

Tyto požadavky na rozlišení a kompresi dat jsou prakticky relevantní jen pro kamery viditelného spektra, které dosahují rozlišení až jednotek megapixelů. Rozlišení termokamer je naopak výrazně nižší a objem dat jednoho snímku je i v plném rozlišení bez komprese přijatelný.

Se zvyšující se snímkovou frekvencí vzrůstá i objem získaných dat, nepředpokládá se tedy příliš časté snímání a vyhodnocování snímků. Postačující snímková frekvence kamerového modulu je 1 snímek za sekundu a v případech satelitu i méně.

1.1.2 Komunikační rozhraní

Komunikace s moduly musí probíhat přes standardní komunikační rozhraní, pro které má výše zmíněný mikrokontrolér dostupné periferie, a to SPI, I²C nebo UART. Na komunikačním protokolu také záleží, preferovány jsou protokoly s nízkými nároky na časování a náročnost zpracování dat.

Převážná většina kamerových modulů implementuje určitou formu framebufferu, neboli paměti pro dočasné uložení získaného snímku, než je přenesen po sběrnici do vlastní paměti mikrokontroléru, kde je zpracován. Avšak některé moduly snímají při konstantní snímkové frekvenci f_{frame} a obsah framebufferu je periodicky přepisován. Z toho vyplývá, že přečtení dat z framebufferu musí proběhnout v době jedné periody $T(s)$ vypočtené v rovnici 1.

$$T = \frac{1}{f_{frame}} \quad (1)$$

Zde nastává komplikace, protože když nebudou všechna data včasně přečtena, dojde k přepsání aktuálně zpracovávaného snímku novým snímkem a původní data jsou ztracena. Příklad takového kamerového modulu je FLIR Lepton.

Do druhé kategorie spadají moduly s proměnlivou snímací frekvencí, která se odvíjí od schopnosti

mikrokontroléru číst a zpracovávat data. Místo toho, aby ke snímání docházelo automaticky, je snímání nutno iniciovat příkazem z mikrokontroléru, a až v ten moment jsou předchozí data framebufferu přepsána. Příkladem této kategorie modulů je Arducam.

1.1.3 Prostředí

Při výběru kamerového modulu musí být bráno v potaz prostředí, kde bude zařízení používáno, jelikož všechna zařízení jsou ovlivňována fyzikálními vlivy daného prostředí. Pro zajištění správného a bezporuchového provozu je důležité dodržet provozní podmínky stanovené v datových listech, zejména teplotu. Uvnitř satelitu CubeSAT dochází k cyklování teploty elektroniky v rozsahu 0 až 40 °C.

Nasazení kamerových modulů ve vnitřních prostorech s běžnou pokojovou teplotou a relativní vlhkostí vzduchu nebývá obvykle problematické. Na druhou stranu při venkovním použití nastávají problémy, jelikož okolní teplota běžně dosahuje hodnot pod 0 °C v zimním období nebo naopak vyšších desítek stupňů Celsia při vystavení přímému slunečnímu záření. Relativní vlhkost vzduchu také běžně dosahuje 100 % při dešti. Další problematické situace zahrnují přímý kontakt s vodou ať už vlivem deště, nebo kondenzací vodních par.

Zvýšit odolnost vůči vysoké relativní vlhkosti vzduchu a přímého kontaktu s vodou lze docílit například lakováním desky plošných spojů se součástkami nebo zapouzdrěním zařízení do vodotěsného krytu. V případě zapouzdrění modulu termální kamery je důležité brát v potaz propustnost infračerveného záření u použitého materiálu, jelikož sklo a běžně používané čiré polymery sice dobře propouštějí záření ve viditelném spektru, ale infračervené záření propouštět nemusí.

1.1.4 Záření v prostoru vesmíru

Atmosféra Země velmi dobře stíní dopadající vysokoenergetické záření z prostoru vesmíru. Z toho vyplývá, že s rostoucí nadmořskou výškou toto stínění přestává být účinné. V extrémním případě, při opuštění atmosféry, začínají být úrovně záření velmi vysoké a elektronika vystavená tomuto záření je negativně ovlivněna. Lze tedy očekávat, že u kamerových modulů, včetně řídicího mikrokontroléru, budou v satelitu nastávat nepředvídatelné jevy.

Záření v prostoru vesmíru je složeno ze 3 základních složek [2]:

1. **Kosmické záření** je proud částic s extrémně vysokou energií, které vznikají mimo naši sluneční soustavu. Skládá se zejména z **protonů**, které tvoří většinu celkového záření a jejich směr pohybu je izotropický. Většina částic dosahuje kinetické energie o velikosti přibližně 1 GeV, vyskytují se ale i částice s mnohonásobně vyšší energií, nicméně četnost výskytu částice klesá s velikostí její energie.
2. Slunce je dalším zdrojem částicového záření ve sluneční soustavě. Při jaderné fúzi dochází vlivem extrémně vysokých teplot k úniku a urychlení částic z korony Slunce, a to zejména

protonů, alfa částic a elektronů. Vzniká tím **sluneční vítr**, jehož částice však nedosahují takových energií, aby zásadním způsobem ovlivňovaly elektroniku. Na druhou stranu, při **slunečních erupcích** jsou částice akcelerovány na mnohonásobně vyšší rychlosti a také je vyzařováno širokospektrální elektromagnetické záření pokrývající radiové vlny až gamma záření. Energie částic je při erupci několikanásobně zvýšena, a to až v řádu několika milionů. Maximální velikosti energií leží poté v rozsahu 0,1 až 1 GeV.

3. Vesmírná tělesa s magnetickým polem zachycují pohybující se částice, čímž okolo nich vznikají **Van Allenovy radiační pásy**. Na rovníku jsou pásy širší a u pólů naopak zcela zanikají.

Střet pohybující se částice s mikroelektronikou způsobí přenos energie v různých formách do polovodičové struktury. Dochází tím buď k dočasným změnám elektrického stavu, nebo k permanentnímu poškození a změně elektrických parametrů součástky. Při průchodu elektricky nabitě částice polovodičovým substrátem vznikají páry elektron-díra, neboli volné nosiče náboje. Rovnovážný stav je po čase znovu dosažen samovolnou rekombinací nebo přenesením volných nosičů v substrátu difuzním či driftovým proudem [2]. Vzniká tím přechodový děj a pokud změna nahromaděného náboje v místě citlivého uzlu obvodu a dosáhne dostatečné velikosti, vznikne **single-event effect (SEE)**.

SEE může být buď dočasný, nebo způsobit permanentní poškození. Nejčastěji dochází k tzv. **bit-flipu**, kdy je překlopen obsah buňky paměti nebo registru sekvenční logiky, k fyzickému poškození ale nedochází. Horší případ zahrnuje v CMOS technologii současné otevření obou komplementárních tranzistorů, čímž vzniká nízkoimpedanční cesta mezi napájecí větví a při nevhodném odpojení obvodu od napájení může dojít k přehřátí a permanentnímu poškození součástky. Je proto vhodné sledovat proudový odběr a při jeho překročení automaticky odpojit napájení ovlivněné součástky či modulu. Nejhorší stav zahrnuje přímé poškození polovodičové struktury vysokoenergetickou částicí.

I když nedojde při střetu částice s obvodem k SEE, záření má akumulární charakter a elektrické vlastnosti materiálů v integrovaném obvodu jsou soustavně degradovány až do bodu, kdy zařízení přestane splňovat požadované parametry a fungovat. Jedním z projevů je zvyšování proudu u izolantů, což zejména degraduje vlastnosti MOSFET tranzistorů. Dalším mechanismem je přímá změna krystalické struktury materiálu [2]. Tyto jevy lze relativně rychle pozorovat u CMOS kamerových senzorů, které nejsou certifikovány pro vesmírné projekty, a u fotocitlivého prvku pixelu průběžně roste velikost **temného proudu**. V důsledku roste i jas pixelu, než dosáhne saturované hodnoty a svítí maximálním jasnem, je tedy zničen. Na druhou stranu infratermočlánek senzoru termokamery by z principu neměl být příliš náchylný na toto poškození, avšak žádné volně dostupné práce zkoumající odolnost nebyly nalezeny a pro podrobnější analýzu je vyžadován další výzkum.

Pro aplikace v prostředí se zvýšenými úrovněmi radiace musí být počítáno již při návrhu zařízení. Odolnost lze zvýšit výběrem certifikovaných součástek pro tato prostředí, existují například speciální paměti s opravnými kódy, které jsou schopny odolat několika SEE způsobujících bit-flip. Další možností ochrany proti záření je stínění celého zařízení.

1.2 Arducam

Vybrané Arducam kamerové moduly popsané v následujících podkapitolách spadají do kategorie modulů s rozšiřujícími integrovanými obvody nad rámec modulu snímače (dále jen snímač). Samotný snímač od společnosti OmniVision integruje již mnoho funkcí, jako je například přepínání rozlišení, několik typů výstupních formátů včetně JPEG komprese, korekci mrtvých pixelů, korekci zkreslení způsobené čočkou, automatické gamma nastavení, minimalizaci šumu, úpravu ostrosti, kompenzaci černého bodu a další [3][4]. Na obrázku 1 je znázorněno blokové schéma takového snímače.

Čtení připraveného snímku z vnitřních obvodů snímače probíhá přes paralelní kamerové rozhraní *CPI* (podrobněji viz podkapitolu 1.2.1), které však není příliš vhodné pro zvolený mikrokontrolér, protože nemá hardwarově integrovanou periférii pro toto rozhraní, a kvůli tomu by musel být přenos řešen softwarově skrze univerzální vstupně výstupní piny (GPIO). Vedlo by to na neoptimální způsob přenosu dat s příliš vysokými časovými nároky na zpracování. Kamerový modul proto dále obsahuje FPGA a externí RAM paměť (framebuffer), společně tvořící převodník z *CPI* na běžněji používané *SPI* v kontextu mikrokontrolérů použitých v této práci.

Vzniká tím ale několik nevýhod, zejména snížení přenosové rychlosti a vznik prodlevy mezi zahájením zachycení snímku a počátku jeho přijetí:

1. Hodinový signál *SPI* je omezen na $f_{clk} = 8 \text{ MHz}$.
2. Data jsou přenášena sériově po jednom bitu místo původního paralelního přenosu.
3. Nejdříve jsou data stažena ze snímače do externího framebufferu a až poté jsou přenášena do mikrokontroléru.

Vlivem toho dochází ke snížení maximální dosažitelné snímkové frekvence v závislosti na velikosti dat snímků, resp. jejich rozlišení a formátu. Obecně platí, že čím vyšší je zvolené rozlišení, větší barevná hloubka a nižší stupeň komprese, tím klesá maximální dosažitelná snímková frekvence, přestože je toho snímač schopen dosáhnout. Rovnice 2 udává přibližně periodu T_{frame} zachycení a přenesení jednoho snímku ze snímače do mikrokontroléru (s).

$$T_{frame} = t_{sensor} + t_{cpi} + \frac{n_{bit}}{f_{clk}} \quad (2)$$

Kde t_{sensor} je doba zachycení a zpracování snímku snímačem (s), t_{cpi} je doba přenosu snímku mezi snímačem a FPGA (s), n_{bit} je velikost přenášeného snímku v bitech a f_{clk} je frekvence hodinového signálu *SPI* (Hz). Například pro snímač OV2640 je udávána snímková frekvence pro UXGA rozlišení (1200x1600 pixelů) $f_{frame} = 15 \text{ Hz}$ při JPEG kompresi [5]. Aby bylo možné dosáhnout udávané snímkové frekvence, celková perioda z rovnice 2 musí být menší než perioda této snímkové frekvence, viz rovnice 3.

$$T_{frame} \leq \frac{1}{f_{frame1}} \leq \frac{1}{15} \approx 66,7 \text{ ms} \quad (3)$$

Jinými slovy řečeno, je nutné zajistit včasné zpracování současného snímku před tím, než je možné začít zachytávat následující snímek. Dále lze předpokládat, že součet prvních dvou členů rovnice 2 je určitě nižší než tato doba. Při předpokladu kompresního poměru 25 mezi snímek bez komprese o barevné hloubce 16 bitů na 1 pixel a snímekem s JPEG kompresí je předpokládaná velikost snímku s UXGA rozlišením a JPEG kompresí:

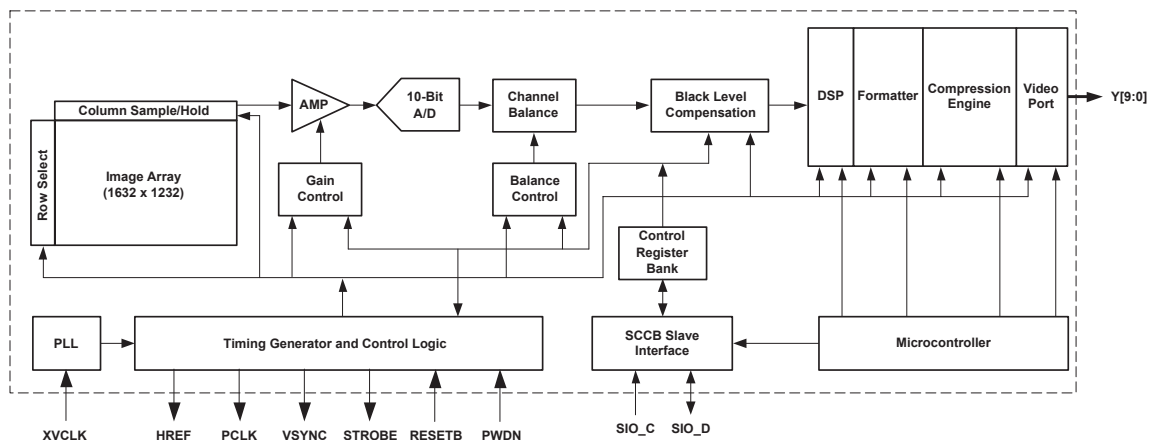
$$n_{bit_JPEG} = \frac{n_{bit_raw}}{25} = \frac{1200 \cdot 1600 \cdot 16}{25} = 1\,228\,800 \text{ b} \quad (4)$$

Samotná doba přenosu t_{spi} snímku s bitovou délkou n_{bit_JPEG} přes SPI o hodinovém taktu $f_{clk} = 8 \text{ MHz}$ je vypočtena v rovnici 5.

$$t_{spi} = \frac{n_{bit_JPEG}}{f_{clk}} = \frac{1\,228\,800}{8 \cdot 10^6} = 153,6 \text{ ms} \quad (5)$$

Jak je vidět, zpoždění způsobené přenosem přes SPI je více než dvojnásobek maximální přípustné doby celého řetězce z rovnice 3. Zavedením pouze tohoto zpoždění, se zanedbáním vlivu ostatních členů v rovnici 2, dojde ke snížení původní maximální snímkové frekvence $f_{frame1} = 15 \text{ Hz}$ na:

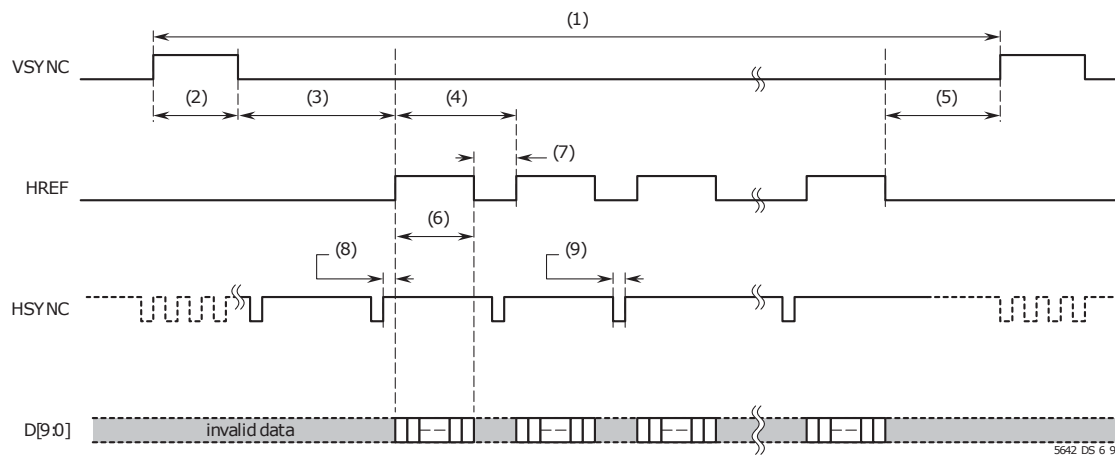
$$f_{frame2} = \frac{1}{t_{spi}} = \frac{1}{153,6 \cdot 10^{-3}} \approx 6,51 \text{ Hz} \quad (6)$$



Obrázek 1: Blokové schéma vnitřních obvodů snímače OV2640, převzato z [5]

1.2.1 Camera Parallel Interface (CPI)

Kamerové rozhraní CPI definovala společnost MIPI Alliance v roce 2004, jde tedy o poměrně zastaralé rozhraní, které bylo postupem času nahrazeno sériovým rozhraním CSI [6]. Rozhraní se dělí na dvě samostatná rozhraní, první využívá SCCB sběrnici pro nastavování snímače skrz čtení a zapisování vnitřních registrů a podrobněji je popsáno v podkapitole 1.2.2. Druhé rozhraní, ilustrované na obrázku 2, slouží k paralelnímu přenosu obrazových dat. Využívá k tomu tři základní signály. Prvním je *VSYNC*, který indikuje počátek nového snímku, signál *HREF* stanovuje počátek řádku snímku a s náběžnou hranou *PCLK* (na diagramu není znázorněno) jsou přenášena jednotlivá data na paralelní sběrnici *D*. [7]



Obrázek 2: Časování signálů CPI obrazového rozhraní snímače OV5642, převzato z [8]

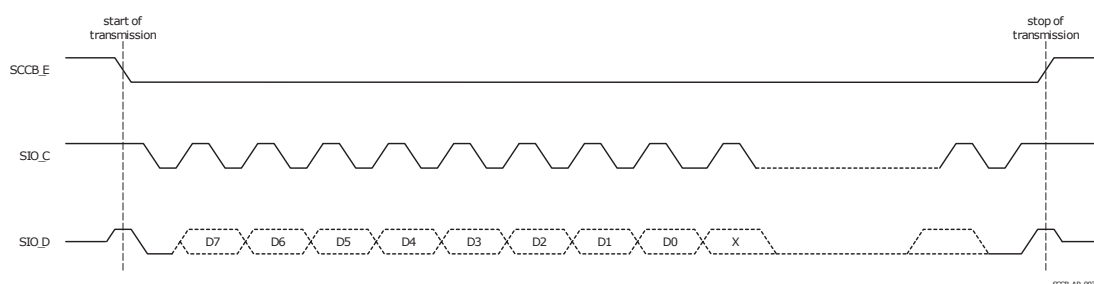
Práce se bude dále věnovat pouze SCCB rozhraní pro nastavování snímače, protože časování a čtení dat snímků na paralelní sběrnici je implementováno skrze FPGA v Arducam modulech.

1.2.2 Serial Camera Control Bus (SCCB) specifikace

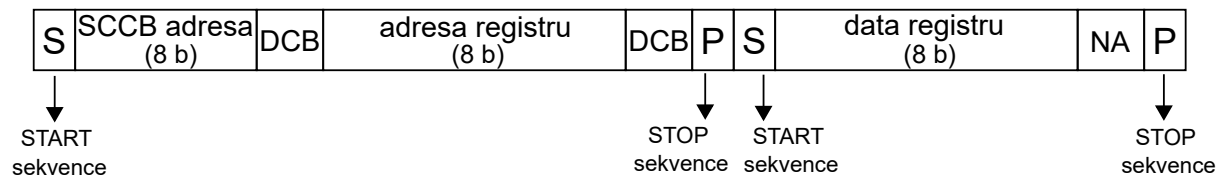
SCCB sběrnice byla vyvinuta a nasazena společností OmniVision pro komunikaci s kamerovými snímači jejich výroby. Sběrnice je se svými specifikacemi téměř totožná s I²C sběrnici ve standardním rychlostním režimu o $f_{scl} = 100 \text{ kHz}$. Tato podkapitola porovnává specifikace obou sběrnic [9][10] a pojednává především o jejich odlišnostech.

Snímače mají pevně danou adresu, na jednu sběrnici je možné tedy přímo připojit pouze 1 snímač. Existují ale i snímače s 3. vstupním výběrovým signálem, přes který mikrokontrolér vybere 1 z n snímačů. Časování zobrazuje časový diagram na obrázku 3 včetně dodatečného 3. vodiče *SCCB_E*.

SCCB místo topologie s otevřeným kolektorem používá **dvojčinné push-pull** zapojení koncových stupňů, což je zásadní rozdíl oproti I²C sběrnici. Plyne z toho riziko **kolize** na sběrnici a je doporučeno vložit ochranný rezistor na signálový vodič *SIO_D*, jak je vidět na obrázku 4. Kolize může nastat i v případě, kdy je master zařízení připojeno přes otevřený kolektor, když snímač nastaví na *SIO_D* vysokou úroveň a master zařízení nízkou úroveň. Další rozdíl spočívá v klidové úrovni signálu hodinového taktu *SIO_C*, který I²C definuje ve vysoké úrovni a SCCB ve stavu vysoké impedance.



Obrázek 3: Časový diagram datového přenosu po SCCB sběrnici, převzato z [10]



Obrázek 6: Čtecí cyklus SCCB sběrnice pro 8bitový registr [10]

1.2.4 SCCB rozšíření komunikačního protokolu pro 16bitové registry

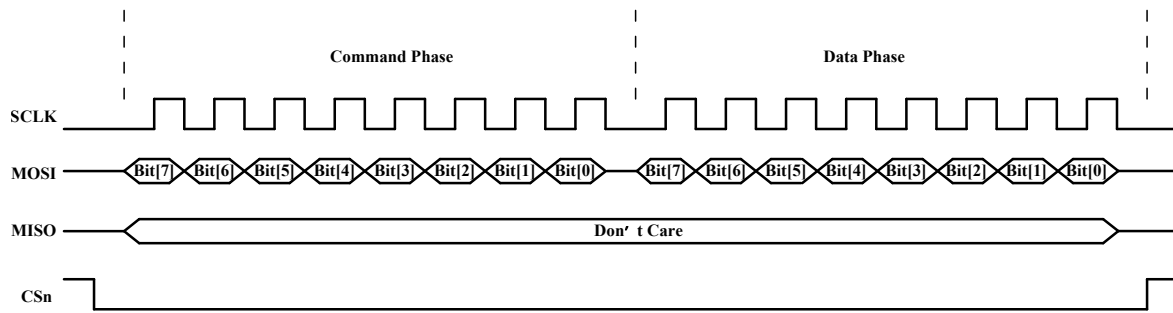
Přestože SCCB specifikace [10] nepovoluje cykly s větším počtem datových bajtů než 2, které jsou určeny pro snímače s 8bitovými registry, existují i novější snímače s 16bitovými registry. Některé snímače implementují pouze 16bitové adresování a šířka slova zůstává 8bitová, existují ale i snímače s 16bitovým adresováním včetně 16bitové šířky slova. Pro tyto snímače již základní přenosové cykly nestačí a je nutno zavést rozšířené cykly.

Rozšířené cykly mají stejnou strukturu jako původní zapisovací cyklus na obrázku 5 a čtecí cyklus na obrázku 6, akorát místo jednoho datového bajtu jsou odeslány 2 bajty s tím, že nejvýznamnější bajt 16bitové hodnoty je odeslán jako první.

1.2.5 Čtení snímků

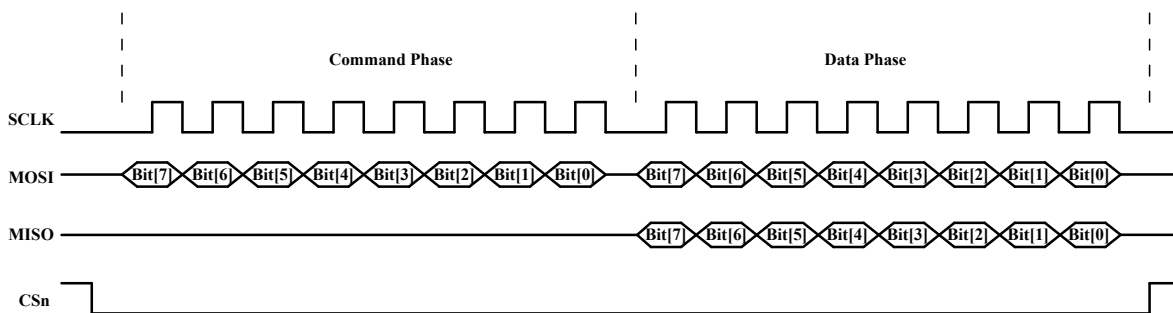
Integrované FPGA na Arducam modulech zajišťuje časování a čtení snímků ze snímače po paralelním CPI rozhraní. Přečtený snímek ukládá do externího framebufferu, ze kterého snímek následně vyčítá po SPI sběrnici. Díky tomuto uspořádání nejsou kladeny náročné časovací požadavky přímo na mikrokontrolér, který místo toho řídí SPI přenos vyhovujícím tempem, při kterém stíhá příchozí data zpracovat.

Podobně jako u SCCB protokolu, i SPI komunikační protokol je založen na čtení/zápisu adresovaných registrů v FPGA. Specifikace komunikačního protokolu definuje aplikační poznámka výrobce [11]. Základní přenosový cyklus je rozdělen do 2 bajtů, z čehož první bajt určuje adresu registru a nejvýznamnější bit adresy rozhoduje, zda proběhne zápis či čtení. Při zápisu je nejvýznamnější bit adresy nastaven na vysokou úroveň, druhý bajt přenosu je odeslán z mikrokontroléru a představuje hodnotu, která bude do vybraného registru zapsána. Časový diagram zapisovacího cyklu je vidět na obrázku 7. Rozhraní je konfigurováno v režimu SPI mode 0 ($CPOL = 0$ a $CPHA = 0$).



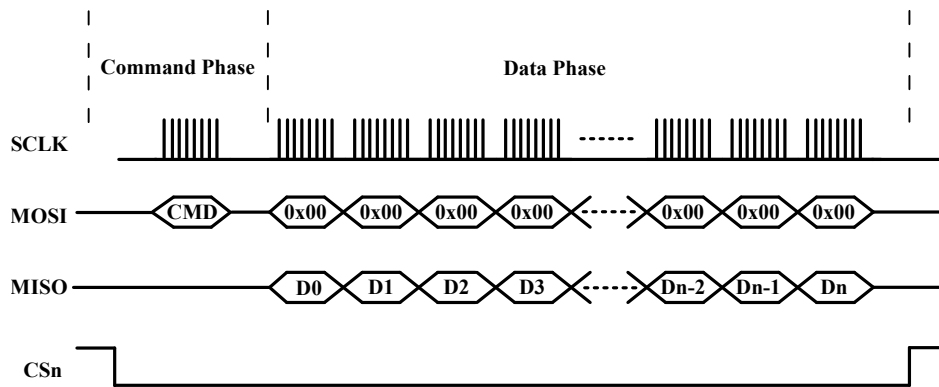
Obrázek 7: Časový diagram SPI zapisovacího cyklu do registru, převzato z [11]

Při čtecím cyklu je nejvýznamnější bit adresy nastaven na nízkou úroveň a druhý bajt přenosu vysílá FPGA, jehož hodnota reflektuje aktuální hodnotu ve zvoleném registru. Časový diagram základního čtecího cyklu je vidět na obrázku 8.



Obrázek 8: Časový diagram základního SPI čtecího cyklu z registru, převzato z [11]

Základní čtecí cyklus je složen ze 2 bajtů, přičemž pouze 1 bajt představuje užitečná čtená data. Při občasném čtení 1bajtové hodnoty z registru je tato neefektivita přenosu zanedbatelná, ale při čtení dlouhých dat snímku, například o velikosti 100 kB, dojde ke značnému zpomalení přenosu. Z tohoto důvodu je zaveden upravený čtecí cyklus pracující v dávkách a slouží pouze ke čtení dat z framebufferu. Výhoda spočívá v tom, že čtení ze speciální adresy registru přepne FPGA do režimu dávkového čtení a následně je s taktováním hodinového signálu SPI přenášén proud bajtů snímku bez nutnosti znovu specifikovat adresu registru. Dojde tím ke zdvojnásobení propustnosti. Po odeslání adresy dávkového čtení musí být úroveň signálu MOSI při taktování hodinového signálu na nízké úrovni. Časový diagram dávkového čtení z framebufferu je vidět na obrázku 9.



Obrázek 9: Časový diagram SPI dávkového čtecího cyklu z framebufferu, převzato z [11]

Capture Control registr nastavuje počet po sobě jdoucích snímků k sekvenčnímu zachycení, které budou uloženy do framebufferu. Jelikož kapacita externího framebufferu převyšuje mnohonásobně velikost jednoho snímku i při plném rozlišení s JPEG kompresí, může do něho být uloženo mnoho snímků bez potřeby okamžitého zpracování mikrokontrolérem. *FIFO Control* registr spouští snímání a ovládá framebuffer. Jakmile je snímek zachycen ve framebufferu, příznak *Camera Write FIFO Done* v *Trigger* registru je nastaven a ve trojici registrů *Camera write FIFO size* je uložena velikost snímku v bajtech. Data snímku z framebufferu jsou čtena buď opakováním *Single FIFO Read* operace (viz obrázek 8), nebo dávkovou čtecí operací *Burst FIFO Read* (viz obrázek 9). Podrobnější informace o SPI registrech se nacházejí v aplikační poznámce [11].

1.2.6 Arducam Mini 2MP Plus

Modul kamery viditelného spektra Arducam Mini 2MP Plus integruje CMOS snímač OV2640 s maximálním rozlišením 1600x1200 pixelů, externí framebuffer o velikosti 8 MB a FPGA pro řízení procesu snímání a konverzi CPI rozhraní snímače na SPI.

Snímač disponuje 8bitovými registry s 8bitovou adresací po SCCB sběrnici, která je popsána v podkapitole 1.2.3. Tyto registry jsou rozděleny do dvou sad a je přes ně řešena veškerá konfigurace snímače. Registr na adrese 0xFF určuje, která sada registrů je aktivní. Zápis hodnoty 0x00 vybere první sadu registrů a zápis hodnoty 0x01 vybere druhou sadu. Zda komunikace po SCCB sběrnici probíhá správně, lze ověřit přečtením registrů s produktovou identifikací typu snímače, které obsahují pevně danou konstantní hodnotu určenou datovým listem. Jedná se o dvojici registrů *PIDH* a *PIDL* ve druhé sadě registrů.

Inicializace snímače spočívá v nastavení mnoha vnitřních registrů a jednotlivé kroky konfigurace nebudou podrobně vysvětleny. Funkce registrů jsou stručně popsány v datovém listu snímače [5], avšak u některých registrů není funkce zřejmá. Aplikační poznámky snímače [3] a zdrojové kódy knihoven od výrobce Arducam [12] obsahují ukázkové sady hodnot registrů pro určité konfigurace snímače, které byly v práci použity, a to zejména přepnutí výstupního formátu do režimu JPEG komprese a změna výstupního rozlišení.

Souhrn parametrů kamerového modulu je v tabulce 1.

Tabulka 1: Přehled parametrů kamerového modulu Arducam Mini 2MP Plus [13]

Rozsah napájecího napětí	3,3 až 5,0 V
Proudový odběr (low power)	20 mA
Proudový odběr (normální činnost)	70 mA
Rozsah provozní teploty	−10 až 55 °C
Maximální rozlišení	1600 x 1200 pixelů
Výstupní formát	JPEG, RGB565
Rozhraní	I ² C, SPI (8 MHz)
Framebuffer	8 MB
Zorný úhel	60° (čočka je vyměnitelná)

1.3 Arducam Mini 5MP Plus

Modul kamery viditelného spektra Arducam Mini 5MP Plus integruje snímač *OV5642*, který poskytuje lepší parametry obrazu oproti snímači 2megapixelové varianty, provedení obou modulů je ale stejné včetně procesu čtení snímků po SPI sběrnici. Hlavním rozdílem je vyšší rozlišení, jehož maximální hodnota je 2592x1944 pixelů. Snímač *OV5642* používá pro svoji konfiguraci 16bitové registry s 16bitovou adresací a proces zápisu se čtením blíže popisuje podkapitola 1.2.4. V tabulce 2 jsou uvedeny parametry, které se liší od 2megapixelové varianty a zbylé parametry jsou shodné s tabulkou 1.

Tabulka 2: Přehled parametrů kamerového modulu Arducam Mini 5MP Plus [14]

Proudový odběr (low power)	20 mA
Proudový odběr (normální činnost)	300 mA
Rozlišení	2592x1944 pixelů

1.4 FLIR Lepton

Řada infračervených kamerových modulů Lepton od společnosti FLIR nabízí ze všech infračervených kamerových modulů popsanych v této práci nejvyšší rozlišení. Místo infratermočlánků integrují **VOx mikrobolometr** a dosahují rozlišení až 160x120 [15]. Mikrobolometr je pole odporových sensorů, které je integrované v čipu snímače. Velikost odporu sensorů je teplotně závislá na množství absorbovaného dopadajícího infračerveného záření.

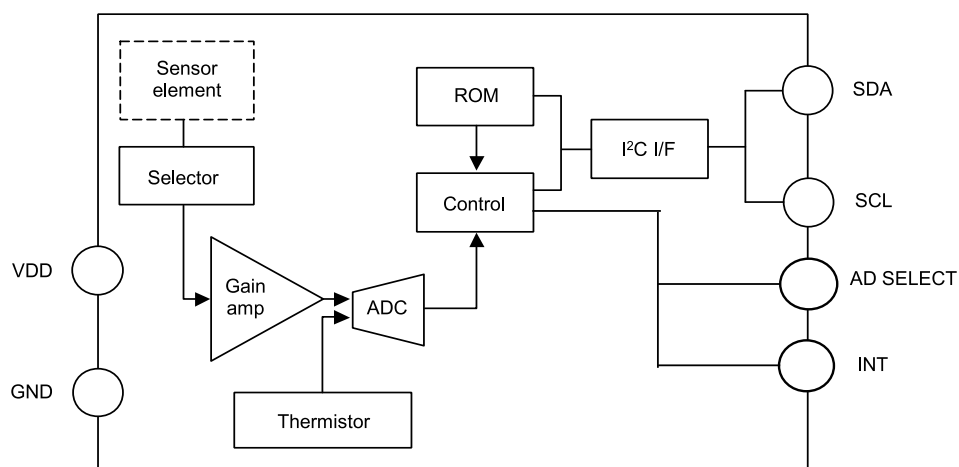
Rozlišení a výstupní formát závisí na konkrétní verzi modulu, zejména verze 2.5 a 3.5 poskytují režim radiometrie, ve kterém jsou výstupní hodnoty senzoru kalibrovány a přepočteny na absolutní teplotu ve stupních Celsia. Výstupní hodnoty zbylých verzí jsou relativní a nelineárně závislé na teplotě modulu. Souhrn parametrů modulů je uveden v tabulce 3 a práce [1] podrobněji popisuje jednotlivé verze a komunikační protokol.

Tabulka 3: Přehled parametrů různých verzí modulu Lepton [15]

Napájecí napětí	2,8 V senzor, 1,2 V jádro a 2,5 až 3,1 V IO
Příkon (low power)	5 mW
Příkon (normální činnost)	160 mW
Rozlišení	80x60 (4800 pixelů) pro verze < 3.0 a 160x120 (19200 pixelů) pro verze ≥ 3.0
Výstupní formát	Absolutní teplota ve stupních Celsia pro verze 2.5 a 3.5, pro ostatní nelineárně závislá hodnota na teplotě modulu
Rozhraní	SPI (20 MHz)
Snímková frekvence	8,7 snímků za sekundu
Teplotní rozlišení	<0,05 °C
Přesnost teploty	v nejlepší případě ±5°
Rozsah teplot měřeného objektu	-10 až 140 °C nebo -10 až 400 °C (dle nastavení zesílení)
Rozsah provozní teploty	-10 až 80 °C
Zorný úhel	Typicky 50° nebo 57°

1.5 AMG88XY

Řada infračervených kamerových modulů AMG88XY od společnosti Panasonic disponuje nejmenším sensorovým polem ze všech modulů termálních kamer popsanych v této práci, a to o rozlišení 8 x 8. Technické parametry všech variant modulů jsou uvedeny v datovém listu [16]. Princip snímání teploty v jednotlivých bodech je založen na MEMS technologii infratermočlánků (anglicky thermopile), jejichž výstupní napětí je závislé na množství absorbované energie dopadajícího infračerveného záření. Analogové hodnoty napětí jednotlivých sensorů jsou zesíleny a po digitalizaci analogově-digitálním převodníkem uloženy do registrů. Na straně modulu probíhá veškeré zpracování hodnot sensorů, což je výhodné, protože na výstupu jsou již digitální hodnoty absolutních teplot ve stupních Celsia a není potřeba žádný dodatečný přepočet. Blokové schéma modulu je vidět na obrázku 10.

**Obrázek 10:** Blokové schéma modulu z řady AMG88XY, převzato z [16]

Komunikace probíhá po sběrnici I²C jak pro nastavování modulu, tak pro čtení snímkových dat.

Modul také podporuje režim přerušení, které je vyvoláno v případě detekce teploty snímaného objektu v uživatelsky definovaném teplotním rozsahu. Dále je v modulu integrován termistor, který měří vnitřní teplotu obvodů.

1.5.1 Porovnání jednotlivých variant

Existují různá provedení toho modulu, která se liší posledními dvěma číslicemi v označení AMG88XY. První číslice, označena písmenem **X**, udává velikost napájecího napětí a druhá číslice, označena písmenem **Y**, uvádí zesílení vnitřních obvodů. Přehled XY hodnot je vidět v tabulce 4.

Tabulka 4: Přehled jednotlivých označení modulů AMG88XY [16]

X	V _{dd} (V)	Y	Zesílení
3	3,3	3	Vysoké
5	5,0	4	Nízké

Velikost zesílení určuje především rozsah provozní teploty a rozsah teplot měřeného objektu, také je ale ovlivněna přesnost změřených teplot. Jak je vidět v tabulce 5, varianta s vysokým zesílením nevyhovuje nasazení v prostředí s teplotou nižší než 0°C. Možnosti použití varianty s vysokým zesílením jsou tedy více omezeny než u varianty s nízkým zesílením. Pokud dojde k překročení maximální teploty měřeného objektu, až už kladné nebo záporné, modul vyhodnotí přetečení A/D převodu. [16]

Tabulka 5: Teplotní rozsahy variant modulu s vysokým a nízkým zesílením [16]

	Vysoké zesílení	Nízké zesílení
Rozsah teplot měřeného objektu	0 až 80 °C	-20 až 100 °C
Rozsah provozní teploty	0 až 80 °C	-20 až 80 °C
Přesnost teploty	Typ. ±2,5°C	Typ. ±3,0°C

Kromě výše uvedených rozdílů jsou zbylé parametry všech variant modulu stejné a přehled je uveden v tabulce 6.

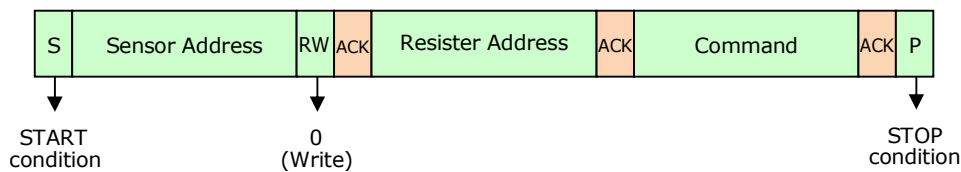
Tabulka 6: Přehled společných parametrů všech variant kamerových modulů AMG88XY [16]

Proudový odběr (low power)	0,2 mA
Proudový odběr (normální činnost)	4,5 mA
Rozlišení	8 x 8 (64 pixelů)
Výstupní formát	Stupně Celsia kódované ve dvojkovém doplňku
Rozhraní	I ² C
Snímková frekvence	1 nebo 10 snímků za sekundu
Teplotní rozlišení	0,25 °C
Zorný úhel	60°

1.5.2 Komunikační protokol

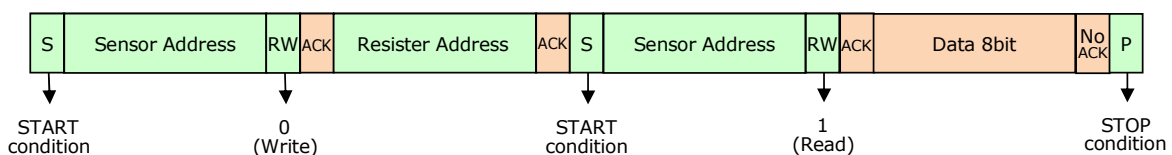
Čtení snímků i konfigurace modulu probíhá po I²C sběrnici ve fast režimu o $f_{scl} = 400$ kHz formou čtení a zapisování registrů na zvolených adresách. Modul pracuje ve slave režimu a nabízí výběr mezi 2 adresami v 7bitovém adresovacím režimu pomocí vstupního pinu *AD_SELECT*. Při nízké úrovni je použita adresa 0x68 a při vysoké 0x69. Na jednu I²C sběrnici je tedy možné přímo připojit 2 tyto moduly. Komunikační protokol dovoluje v jedné I²C transakci 8bitově adresovat registr s 8bitovou šířkou slova, avšak některé vnitřní registry modulu jsou delší než 8 bitů. V tomto případě se každý bajt registru jeví jako samostatný registr s vlastní adresou a zápis či čtení musí být opakováno pro každý bajt registru zvlášť.

Data jsou do registru modulu zapsána zapisovacím cyklem zobrazeným na obrázku 11. První datový bajt určuje adresu registru a následující bajt hodnotu, která má být do registru zapsána.



Obrázek 11: Diagram zapisovacího cyklu do registru modulu AMG88XY po I²C sběrnici, převzato z [17]

Čtecí cyklus, zobrazen na obrázku 12, je rozdělen do 2 I²C transakcí. Nejprve master zařízení odešle adresu zvoleného registru a ve druhé čtecí transakci modul odešle přečtenou hodnotu registru.



Obrázek 12: Diagram čtecího cyklu registru modulu AMG88XY po I²C sběrnici, převzato z [17]

1.5.3 Inicializace

Pro zahájení aktivního snímání musí být modul přepnut do **normal** režimu, ve kterém dosahuje proudového odběru typicky 4,5 mA. Naopak v úsporném **sleep** režimu jsou vnitřní obvody modulu vypnuty, snímání zastaveno a proudový odběr klesá typicky na 0,2 mA. Režimy jsou řízeny registrem *PCLT* (power control register).

Po přepnutí do normal režimu trvá po dobu 50 ms, než je modul připraven přijímat další příkazy. V tomto okamžiku jsou vnitřní registry v nedefinovaném stavu a je potřeba provést **softwarový reset** přes registr *RST*, při kterém jsou příznaky ve status registru *STAT* resetovány a současně se nahrají korekční hodnoty senzoru z ROM paměti modulu.

Registr *FPSC* přepíná snímkovou frekvenci mezi 1 nebo 10 snímků za sekundu. Nyní je modul inicializován a nezávisle na master zařízení začíná snímat a zpracovávat data při zvolené snímkové frekvenci. Ustálení čtených hodnot ze sensorového pole nastává po 15 s. [17]

1.5.4 Přerušení

Modul podporuje funkci detekování, zda jednotlivé teplotní body snímku spadají do předem definovaného rozsahu teplot. V případě překročení stanovené hranice, ať už dolní nebo horní, dojde k vyhodnocení přerušení. Indikace přerušení nastává nastavením nízké logické úrovně na výstupním pinu *INT* a nastavením bitu *INTF* ve status registru *STAT*.

Pro zapnutí funkce přerušení je potřeba v registru *INTC* povolit výstup na *INT* pinu a zvolit, jaký režim porovnávání dat pro vyhodnocení bude použit. Dostupné jsou 2 možnosti, buď budou porovnávány absolutní velikosti teplot pixelů, nebo rozdíl velikostí teplot stejných pixelů mezi dvěma po sobě jdoucími snímky. Podle předchozí volby bude každý pixel snímku porovnán s předem definovanými hranicemi teplot a pokud velikost teploty jakéhokoliv pixelu je větší než horní hranice nebo je menší než spodní hranice, přerušení je vyvoláno.

Horní hranice je definována v registru *INTH*, dolní hranice v registru *INTL* a hystereze v registru *IHYS*. Všechny zmíněné registry jsou 2bajtové s 12bitovými hodnotami v celočíselném formátu se znaménkem ve dvojkovém doplňku. Teplotní rozlišení nejméně významného bitu je $\Delta t = 0,25^\circ\text{C}$. Rovnice 7 ukazuje příklad výpočtu hodnoty registru pro stanovení horní hranice teploty $t = 30^\circ\text{C}$.

$$INTH = \frac{t}{\Delta t} = \frac{30}{0,25} = 120, \quad (7)$$

kde t je požadovaná teplotní hranice ve stupních Celsia. Kolísání teploty okolo nastavených hranic může způsobit opakované vyvolávání přerušení. Pokud je to nežádoucí jev, je možné nastavit hysterezi v registru *IHYS* a jakmile dojde k vyvolání přerušení, dolní a horní hranice pro resetování přerušení je posunuta o zvolenou velikost ve stupních Celsia.

Podrobnější informace o tom, který pixel přesně vyvolal přerušení, je dostupná v sadě registrů *INT0* až *INT7*. Každý bit v registru označuje jeden pixel a udává, zda jeho velikost teploty přesáhla stanovenou hranici a tudíž vyvolala přerušení. Jeden registr pokrývá 8 pixelů, celá sada registrů pokrývá všech 64 pixelů. Resetování příznaku přerušení ve status registru *STAT* a resetování příznaků v sadě registrů *INT0* až *INT7* je provedeno buď automaticky, kdy ve snímku není žádný pixel, který přesahuje stanovené hranice, nebo softwarově přes reset registr *RST*. Druhá varianta může být užitečná v případě použití hystereze, když je například hysterezní smyčka příliš široká a přerušení není po dlouhou dobu samo resetováno. [17][18]

1.5.5 Průměrování

Další podporovanou funkcí modulu je filtr klouzavého průměru snímaných dat. Pokud je zvolena snímková frekvence 10 snímků za sekundu, je na výstupu průměr předchozího snímku a součas-

ného. Při snímkové frekvenci 1 snímek za sekundu je průměrování dvojnásobné, protože ačkoliv má snímač sníženou četnost obnovy dat na výstupu, uvnitř stále snímá při snímkové frekvenci 10 snímků za sekundu. Za jednu periodu, tj. 1 s, snímač zprůměruje 10 snímků a výstup vytvoří ještě jedním průměrem z předchozího průměru 10 snímků a současného průměru 10 snímků.

Výhodou průměrování je snížení šumu pixelů za cenu zvýšení prodlevy mezi změnou snímané scény a reakcí na výstupu snímače. Průměrovací filtr je ovládán dvěma registry. Prvním z nich je *AVE* a druhý je blíže nespecifikovaný registr na adrese 0x07. [16][18]

1.5.6 Termistor

Teplotu modulu měří integrovaný termistor s teplotním rozlišením $\Delta t = 0,0625^\circ\text{C}$ a rozsahem měřitelných teplot -20 až 80°C . Vnitřní obvody snímače zajišťují vzorkování a převod velikosti napětí termistoru na 12bitovou celočíselnou hodnotu se znaménkem kódovanou v přímém kódu. Znaménko je uloženo v nejvýznamnějším bitu a data jsou rozdělena do dvou registrů *TTHL* a *TTHH*. Výpočet výsledné teploty t ($^\circ\text{C}$) z přečtené hodnoty x v dekadickém tvaru probíhá dle rovnice 8.

$$t_{\text{termistor}} = 0,0625 \cdot x \quad (8)$$

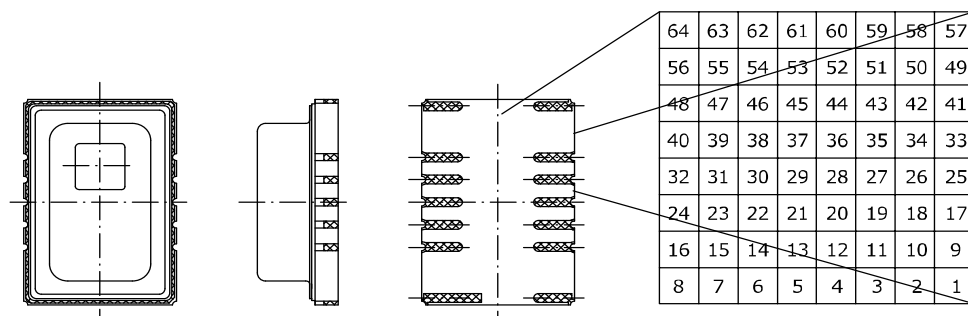
1.5.7 Čtení snímků

Pixel snímku je tvořen 12bitovou celočíselnou hodnotou se znaménkem kódovanou ve dvojkovém doplňku. Na každý pixel připadají 2 registry TnL a TnH , kde n označuje pozici pixelu dle obrázku 13. Pro získání jednoho celého snímku je tedy potřeba vyčíst data ze 128 registrů. Teplotní rozlišení senzorů je $\Delta t = 0,25^\circ\text{C}$ a výpočet probíhá obdobně jako při výpočtu teploty termistoru. Přesný postup pro výpočet teploty t ($^\circ\text{C}$) jednoho pixelu je uveden v rovnici 9.

$$t_{\text{pixel}} = 0,25 \cdot x, \quad (9)$$

kde x je převedená hodnota ze dvou příslušných registrů do dekadického tvaru.

Data pixelů jsou v poli registrů periodicky aktualizována dle zvolené snímkové frekvence a aktualizace všech registrů probíhá najednou. Nemůže tedy nastat situace, kdy by pole registrů současně obsahovalo data více než jednoho snímku.

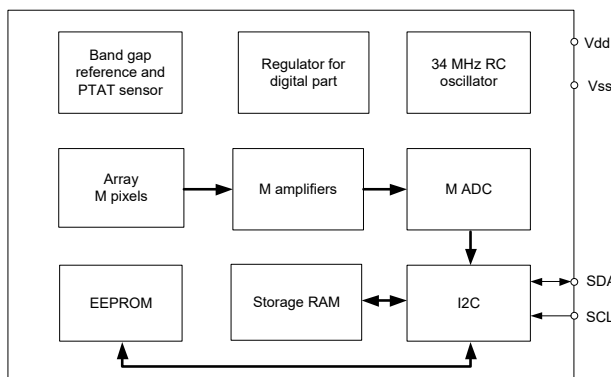


Obrázek 13: Nákres modulu AMG88XY s vyznačením snímaného pole s uspořádáním pixelů, převzato z [16]

1.6 MLX90640 a MLX90641

Moduly termálních kamer MLX90640 a MLX90641 nabízejí vyšší rozlišení než řada modulů AMG88XY, ale zároveň nižší než Flir LEPTON, ocitají se tedy uprostřed běžně dostupných technologií infračervených kamerových modulů. Rozdíl mezi dvěma variantami spočívá v rozlišení a v rozsahu provozní teploty. Podrobnější souhrn parametrů obou variant je v tabulce 7.

Princip snímání teploty je založen na stejném principu jako u řady modulů AMG88XY popsaných v podkapitole 1.5, tj. každý pixel je tvořen infratermočlánkem, jehož výstupní napětí reaguje na množství absorbované energie dopadajícího infračerveného záření. Napětí je dále zpracováno analogovými obvody a poté zdigitalizováno analogově-digitálním převodníkem. Blokové schéma modulu je vidět na obrázku 14.



Obrázek 14: Blokové schéma modulů MLX90640 a MLX90641, převzato z [19]

Další rozdíl oproti AMG88XY je, že výstupní hodnoty nejsou kalibrovány a neudávají absolutní teplotu ve stupních Celsia. Místo toho je v MLX modulech EEPROM paměť s kalibračními konstantami, které jsou unikátní pro každý čip a jsou uloženy do paměti při výrobě. Kalibrační hodnoty z EEPROM paměti musí být při startu programu vyčteny a následně dosazeny do mnoha výpočtů, které převedou nezpracované hodnoty senzoru na absolutní teplotu ve stupních Celsia. Mezi výpočty spadá například výpočet napájecího napětí senzoru, kompenzace zesílení, kompenzace teplotních dat pixelů (offset, teplota okolí, napájecí napětí senzoru, emisivita, teplotní gradient), normalizování citlivosti a další. Rovnice výpočtů jsou uvedeny v datových listech

[19][20] a nebudou dále v této práci popisovány.

Výrobce poskytuje knihovnu v jazyce C s řízením modulu a implementací přepočtu nezpracovaných hodnot senzoru [21][22]. Knihovna je rozdělena na výpočetní část, která nezávisí na platformě mikrokontroléru, a na část ovladače hardwaru I²C sběrnice. Díky této struktuře stačí vytvořit pouze ovladač pro zvolený mikrokontrolér a zbytek knihovny lze použít bez úprav.

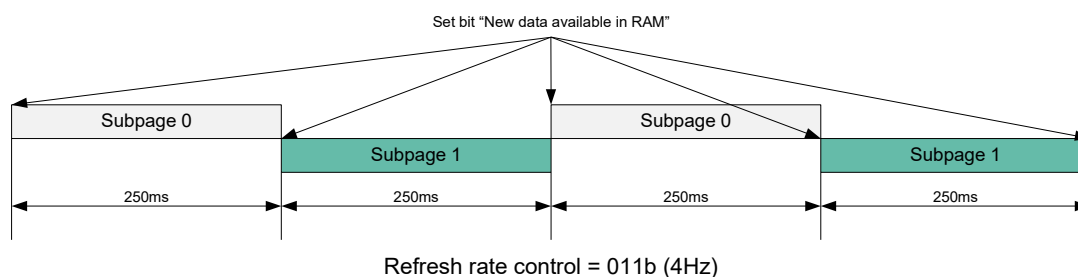
Tabulka 7: Parametry infračervených kamerových modulů MLX90640 a MLX90641 [19][20]

	MLX90640	MLX90641
Napájecí napětí	3,3 V	
Proudový odběr	20 mA	12 mA
Rozlišení	32x24 (768 pixelů)	16x12 (192 pixelů)
Výstupní formát	Nezpracované hodnoty senzoru	
Rozhraní	I ² C	
Snímková frekvence	0,25 až 32 snímků za sekundu	
Teplotní rozlišení	18 bit ADC	
Přesnost teploty	v nejlepším případě $\pm 1^\circ$	
Rozsah teplot měřeného objektu	-40 až 300 °C	
Rozsah provozní teploty	-40 až 85 °C	-40 až 125 °C
Zorný úhel	55° nebo 110°	

1.6.1 Snímková frekvence

V datových listech modulů [19][20] je uvedena obnovovací frekvence 0,5 až 64 Hz, nejedná se však o snímkovou frekvenci, která je ve skutečnosti **poloviční**. Snímání probíhá ve dvou krocích, kdy v každém kroku je zaznamenána polovina snímku a obnovovací frekvence určuje právě četnost obnovy tohoto **jednoho kroku** (viz obrázek 15).

Přepočet nezpracovaných hodnot senzoru je relativně náročný a výsledná snímková frekvence se odvíjí od výpočetního výkonu použitého mikrokontroléru. Přestože modul dosahuje až 32 snímků za sekundu, výsledná snímková frekvence může být kvůli nedostatečnému výpočetnímu výkonu i několikanásobně nižší.

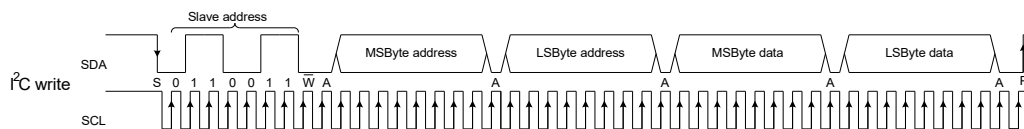


Obrázek 15: Obnovovací frekvence modulů MLX90640 a MLX90641, převzato z [19]

1.6.2 Komunikační protokol, požadavky na ovladač sběrnice

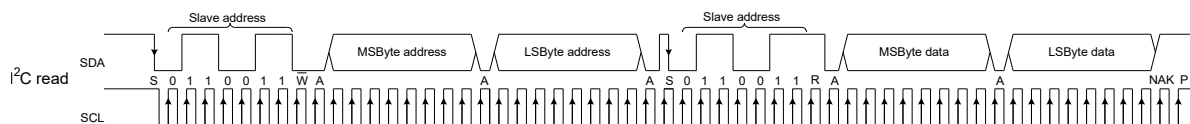
Modul pracuje ve slave režimu s výchozí I²C adresou 0x33 v 7bitovém adresovacím režimu. Adresa je konfigurovatelná přes EEPROM paměť a na jednu sběrnici lze připojit až 127 těchto modulů. Minimální frekvence hodinového signálu sběrnice je $f_{scl} = 400$ kHz a při čtení z EEPROM paměti musí být nastavena na tuto hodnotu. Pro čtení snímku může být frekvence zvýšena až na 1 MHz. Registry jsou adresovány 16bitově a mají 16bitovou šířku slova, přičemž je nejdříve přenesen nejvýznamnější bajt.

Ovladač I²C sběrnice musí zajistit funkce pro čtení a zápis dat po sběrnici. Zapisovací cyklus (viz obrázek 16) zahájí ovladač zapisovací I²C transakcí, ve které nejdříve odešle 16bitovou adresu z mapy paměti modulu, po které následuje 16bitová hodnota k zapsání. V mapě paměti jsou zahrnuty paměti ROM, RAM, EEPROM a vnitřní registry.



Obrázek 16: Zapisovací cyklus I²C sběrnice u modulů MLX90640 a MLX90641, převzato z [19]

Čtecí cyklus (viz obrázek 17) je složen ze 2 I²C transakcí. V první transakci odešle ovladač 16bitovou adresu z mapy paměti modulu a poté zahájí čtecí transakci, ve které modul odešle přečtenou 16bitovou hodnotu. Mezi zapisovací a čtecí transakcí nesmí být ovladačem přivedena na sběrnici stop sekvence, místo toho musí být zopakována start sekvence.[19]



Obrázek 17: Čtecí cyklus I²C sběrnice u modulů MLX90640 a MLX90641, převzato z [19]

2 Komunikace v počítačových sítích

Tato kapitola pojednává o základním návrhu a rozdělení obecné počítačové sítě dle referenčního **ISO/OSI modelu** a následně o jeho implementaci do **TCP/IP sady protokolů**. Některé části budou popsány jen okrajově, protože cílem kapitoly je představit vybrané části teorie, která je potřebná pro pochopení kapitoly 3.9.3, kde budou informace využity k praktickému vývoji programu pro mikrokontrolér s komunikací přes Ethernet.

2.1 ISO/OSI referenční model

Komunikace v počítačových sítích probíhá na několika úrovních, které popisuje referenční ISO/OSI model. Ten definuje 7 vrstev komunikace, kdy jednotlivé vrstvy mají přesně stanovené funkce, které poskytují následující vyšší vrstvě a docilují toho využitím služeb předchozí nižší vrstvy. Nepojednává však o detailech implementace pro specifický systém, místo toho stanovuje teoretické požadavky a všeobecné principy na to, jaké funkce by jednotlivé vrstvy měly zahrnovat. Výhoda vrstveného modelu spočívá v možnosti implementovat jednotlivé vrstvy modulárně. V případě změn pak stačí upravit pouze jednu vrstvu, zatímco zbylé zůstávají v původním stavu. Například změna metalického přenosového média na optické znamená přepracovat jen 1 vrstvu.

První **fyzická** vrstva ISO/OSI modelu stanovuje, které fyzické médium bude použito k přístupu do sítě a detailně jakým způsobem. Popisuje fyzikálními veličinami proces datového přenosu, ale i další aspekty, jako jsou třeba použité materiály a jejich uspořádání. U metalického přenosového média jsou například definovány velikosti napětí logických úrovní, časování a tvar průběhu signálu, rozložení pinů v konektoru nebo vlastnosti kabelu.

Popis fyzického média je pouze první krok k realizaci datového přenosu, zbývá ještě definovat komunikační protokol. Jakým způsobem budou data uspořádána, jak bude probíhat přístup z více zařízení ke sdílenému médiu, zda bude použita detekce/oprava chyb v datech nebo způsob adresace cílového přijímače, to vše závisí na druhé **linkové** vrstvě ISO/OSI modelu. [23][24]

2.2 TCP/IP

Praktickým nasazením ISO/OSI modelu se v současných počítačových sítích a internetu stala sada protokolů TCP/IP. Sada uvedla již přesné definice komunikačních protokolů na jednotlivých vrstvách, zejména protokol *TCP* na transportní vrstvě a *IP* protokol na síťové vrstvě, ze kterých vychází samotný název sady protokolů. Místo původních 7 vrstev ISO/OSI modelu, definuje TCP/IP pouze 4, resp. některé vrstvy ISO/OSI modelu spojuje do jedné. Porovnání struktury vrstev ISO/OSI modelu a TCP/IP protokolového zásobníku je vidět v tabulce 8.

Tabulka 8: Srovnání vrstev ISO/OSI referenčního modelu s TCP/IP zásobníkem protokolů [23][24]

Pořadí vrstvy	Název vrstvy ISO/OSI	Název vrstvy TCP/IP
7	Aplikační	Aplikační
6	Prezentační	
5	Relační	
4	Transportní	Transportní
3	Síťová	Síťová
2	Linková	Síťového přístupu
1	Fyzická	

2.2.1 Vrstva síťového přístupu

Spojením fyzické a linkové vrstvy ISO/OSI modelu vzniká v TCP/IP **vrstva síťového přístupu**. Přistupuje se zde k přenosovému médiu přes síťové rozhraní systému a dochází k přenosu dat na té nejnižší úrovni, a to libovolným způsobem, protože TCP/IP nestanovuje žádné konkrétní protokoly. Jediné, co TCP/IP definuje, je způsob přístupu k funkcím vrstvy síťového přístupu z vyšší síťové vrstvy. Díky tomu lze TCP/IP použít téměř se všemi běžně používanými síťovými rozhraními. Realizace datového přenosu závisí až na zvolené technologii, jako je například Ethernet, X.25, Token Ring nebo bezdrátové WiFi sítě pod standardem IEEE 802.11. [24]

Vrstva síťového přístupu pokrývá jeden síťový segment, ve kterém jsou všechna připojená zařízení schopna mezi sebou navázat komunikaci. Tento rozsah dále upravují aktivní síťové prvky, jako jsou most, rozbočovač a přepínač, které operují pouze na úrovni síťového rozhraní a pro vyšší vrstvy je jejich činnost transparentní.

2.2.2 Síťová vrstva

Síťová vrstva, někdy také označována jako internetová vrstva, slouží k vzájemnému propojování většího množství síťových segmentů, které rozděluje do sítí. Nabízí komunikaci se zařízeními adresovanými napříč několika sítěmi a směřuje datagramy po nejoptimálnější dostupné síťové cestě. Hlavním protokolem pro přenos dat z vyšších vrstev je **IP protokol**, který nenavazuje spojení, nezajišťuje úspěšné doručení dat a neřídí tok dat¹. Zařízení v síti jsou adresována **IP adresami** a přenášená informace se nazývá **IP datagram**.

Pokud je datagram koncovým zařízením odeslán mimo lokální síť, musí být **směrovačem** předán do následujícího bodu v síti na cestě do cílové sítě. Směrovače udržují směrovací tabulku s přímo dostupnými cestami do lokálně připojených sítí nebo s nepřímými dostupnými cestami skrz jednu či více bran. Pokud není v tabulce nalezena cesta do cílové sítě, je datagram odeslán do výchozí cesty.

Struktura propojení sítí může být libovolná a navíc časově nestálá. Mezi dvěma zařízeními se

¹Rízení toku dat řídí odesílání dat takovým způsobem, aby přijímač stihl všechna data zpracovat a nedošlo k jeho přehlcení.

běžně vyskytuje několik dostupných cest a kdykoliv může dojít ke vzniku nové cesty a nebo k zániku již existující cesty. Je tedy potřeba průběžně kontrolovat všechny dostupné síťové cesty a podle toho aktualizovat směrovací tabulku. Pro výměnu informací ohledně stavu a struktury sítě používá TCP/IP zásobník protokolů další protokoly, jako jsou ICMP, IGMP nebo ARP. [25]

2.2.3 Transportní vrstva

V počítačových systémech je běžně provozováno současně několik aplikací spuštěných v procesech, které komunikují po síti a u příchozích dat je nutno rozlišit, které aplikaci jsou určena. Jeden proces může navazovat jedno či více spojení. K jednoznačnému rozlišení zdroje a cíle dat zavádí transportní vrstva porty a sokety.

Zdrojový **port** určuje, který zdrojový proces ve vysílacím počítači data odesílá a cílový port určuje cílový proces v přijímacím počítači, kterému jsou data určena. Ve vybraných případech může komunikace probíhat mezi dvěma procesy na stejném počítači a data nejsou po síti vůbec přenesena. Port je 16bitové číslo, jehož hodnota v některých případech dle konvence určuje přímo aplikaci procesu, a to zejména u serverových aplikací, jako je například webový server (80 nebo 443), telnet (23) nebo FTP (20 a 21). Naopak porty klientských procesů jsou obvykle dynamicky přiřazovány TCP/IP zásobníkem protokolů a nelze podle nich přímo určit aplikaci.

Soket v systému počítače jednoznačně přiřazuje specifický port k procesu a není dále přenášen po síti. Port může být přiřazen v jeden okamžik pouze k jednomu procesu.

Mezi další volitelné funkce transportní vrstvy se řadí zejména spolehlivé doručování spojově orientovaným přenosem dat, segmentace, ochrana proti zahlcení sítě, oprava chyb přenosu nebo řízení toku dat. TCP/IP definuje 2 základní protokoly: TCP a UDP. Volba závisí na charakteru aplikace.

TCP navazuje spojení a poskytuje spolehlivé doručování dat. Řeší automatické potvrzování, znovu odesílání, segmentaci a příjem duplicitních paketů. Nevýhodou je ale zvýšená latence způsobená těmito mechanismy. Uplatnění nalezne například při přenosu souborů, kdy jakákoliv chyba přenosu dat může způsobit poškození celého souboru. [24]

UDP naopak nezajišťuje žádnou z výše uvedených funkcí, které nabízí TCP. V případě potřeby musí být tyto funkce implementovány protokolem v aplikační vrstvě. Absence pokročilých funkcí vede na zjednodušení protokolu a přináší několik výhod oproti TCP:

- **Nižší režie síťového provozu**, protože nedochází k potvrzování paketů, kontrole stavu spojení nebo řízení přenosu dat v rámci zamezení přetížení sítě. Dále má UDP kratší hlavičku paketu, a tedy s každým odeslaným paketem je přenesen nižší objem dat. Mimo snížení režie provozu sítě může dojít i ke **zvýšení přenosové rychlosti**.
- **Latence** mezi odesláním a přijmutím paketu je nižší. To může být užitečné při streamování videa, zvuku či jiných dat.

- Nižší počet stavů a funkcí přenosu dat vede ke **snížení nároků na výpočetní výkon** koncových síťových zařízení, což může být významné pro embedded systémy s relativně nízkým výpočetním výkonem.
- Nepotvrzování paketů přináší možnost **jednosměrného přenosu dat**.

2.2.4 Aplikační vrstva

Nejvyšší vrstva představuje již protokoly aplikací, mezi které spadá například HTTP, DNS, FTP, TELNET nebo SSH. Ve většině případů je protokol aplikační vrstvy realizován přes TCP, UDP nebo jejich kombinací.

2.2.5 Maximální velikost aplikačních dat UDP datagramu

Aplikační data, neboli **payload**, je část datagramu, kde se nachází přenášená informace. Jak již bylo řečeno v podkapitole 2.2.3, protokol UDP nezajišťuje fragmentaci. Pokud je payload příliš dlouhý na přenesení v jediném datagramu, fragmentaci musí řešit aplikační vrstva. Otázkou je, jakou volit maximální payload délku při návrhu aplikačního protokolu.

Maximální délka vychází z **MTU** linkové vrstvy, což je v případě Ethernetu nejčastěji 1500 bajtů. Od této délky je nutno odečíst všechny délky hlaviček protokolů vyšších vrstev a zbývající délka je dostupná pro payload. Při komunikaci přes internet může být však po síťové cestě datagram zabalen dalšími protokoly, čímž dojde k navýšení délky. A nebo MTU linkových vrstev mohou být nižší než MTU ve zdrojové síti. A proto TCP/IP definuje minimální délku IP datagramu, při které je zaručeno úspěšné přenesení datagramu celou sítí bez nutnosti fragmentace, a to 576 bajtů. IP datagramy delší než 576 bajtů mohou být zahozeny jakýmkoliv směrovačem po síťové cestě.

Hlavička IP datagramu při využití všech volitelných polí dosahuje až 60 bajtů a hlavička UDP datagramu je dlouhá 8 bajtů. Výsledná maximální délka payload dat se zaručeným doručením skrz jakoukoliv síť respektující TCP/IP specifikaci je tedy **508 bajtů**. Pokud ale bude komunikace provozována pouze na lokální Ethernetové síti se známými parametry, není důvod se omezovat touto hodnotou a místo toho využít delší délku datagramu přibližující se MTU.

2.3 Ethernet

Mezi běžně používané technologie pro kabelová spojení v počítačových sítích se řadí skupina technologií zvaná Ethernet. Zařízení jsou spojena kroucenou dvojlinkou, optickými kabely nebo dříve i koaxiálními kabely.

Standard *IEEE 802.3* rozděluje linkovou vrstvu ISO/OSI modelu na 2 podvrstvy. První podvrstva linkové vrstvy **MAC** ovládá zařízení na fyzické vrstvě označované jako **PHY** přes **MI**

rozhraní a abstrahuje řízení datového přenosu. Základní úkony MAC podvrstvy jsou například vysílat a přijímat datové rámce, adresovat stanice, detekovat chyby nebo řídit datový tok.

Druhá podvrstva linkové vrstvy **LLC** se vyskytuje nad MAC podvrstvou a ovládá ji. Představuje rozhraní mezi síťovou vrstvou a MAC podvrstvou. Příchozí data ze síťové vrstvy mohou být tvořena libovolným protokolem a svoji velikostí přesahovat kapacitu jednoho rámce. LLC podvrstva rozdělí příchozí data ze síťové vrstvy na *LLC PDU*, které pak následně multiplexuje do jednotlivých rámců a odešle přes MAC podvrstvu. Na přijímací straně jsou LLC PDU demultiplexována a následně jsou obnovena data původního síťového protokolu. LLC může dále poskytovat řízení datového toku a znovu odesílat chybně přijatá data, nicméně v případě Ethernetu nejsou tyto funkce přítomny. [25]

3 Použití kamerových modulů s STM32

V této kapitole je rozvedena praktická implementace vybraných kamerových modulů popsaných v předchozích kapitolách. Výsledkem je ukázkový zdrojový kód pro platformu STM32 včetně podrobnějšího popisu vybraných částí algoritmů a použitých knihoven.

Cílem návrhu je vyvinout hardware pro připojení kamerových modulů k vývojové desce a software s algoritmy pro konfiguraci modulů a následné čtení zachycených snímků, které budou přes Ethernet odeslány do počítačové aplikace, kde proběhne konečné zpracování a grafická prezentace dat. Mikrokontrolér připojený do počítačové sítě má charakter IP kamery. Svoji adresu získá buď automaticky z DHCP serveru, nebo staticky z předem určené konfigurace. Aplikace pro zpracování dat je blíže popsána v kapitole 5.

Díky síťovému připojení lze snadno nasadit více těchto IP kamer a například centrálně sledovat vnitřní nebo vnější prostory, a to jak ve viditelném spektru, tak i v infračerveném, sloužícím pro pokročilejší detekci osob, předmětů nebo zvířat. Tímto se však práce podrobněji nezabývá a bylo by nutné vytvořit mimo jiné vlastní hardwarový návrh odolný proti vlivům zvoleného prostředí.

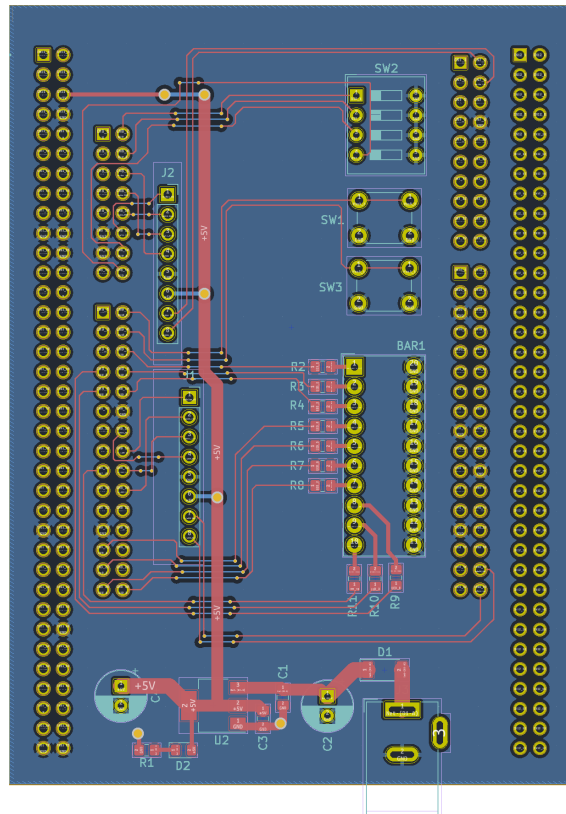
3.1 Vývojová deska a přípravek pro připojení kamerových modulů

Všechny zdrojový kód pro implementaci komunikace s kamerovými moduly byl vyvinut v jazyce C pro mikrokontrolér **STM32F429ZI** na vývojové desce **NUCLEO-F429ZI** ve vývojovém prostředí STM32CubeIDE. S minimálními úpravami je ale možné zdrojový kód použít i pro jiné STM32 mikrokontroléry, které disponují SPI, I²C nebo UART periferiemi. Pokud by byla použita zcela odlišná platforma, tato práce může sloužit jako ukázkový příklad algoritmů a principů komunikace s kamerovými moduly pro vyvinutí vlastního zdrojového kódu.

Pro snadné připojení kamerových modulů k vývojové desce byl v softwaru KiCAD 7.0 navržen a následně vyroben přípravek osazený konektorem pro připojení k univerzálním vývodům vývojové desky. Z druhé strany se nachází 2 konektory pro nezávislé připojení až 2 různých kamerových modulů. Komunikace s kamerovými moduly probíhá přes rozhraní SPI, I²C nebo UART, přičemž některé moduly využívají dvě sběrnice zároveň. Piny mikrokontroléru všech těchto sběrnic jsou přivedeny na kamerové konektory přípravku. Moduly Arducam Mini a FLIR Lepton sdílí stejné rozložení pinů, a proto bylo zvoleno pro oba konektory. Pro připojení ostatních modulů s rozlišeným pinovým uspořádáním bylo potřeba vyrobit redukce.

Na levé polovině přípravku (obrázek 18) jsou vedle sebe umístěny 2 konektory za účelem připojení jedné kamery snímající ve viditelném spektru a druhé kamery snímající v infračerveném spektru. Tímto je umožněno snímat stejnou scénu v obou spektrech zároveň a následně v aplikaci pro

zpracování dat zobrazit obě spektra překrytá přes sebe. V případě snímání termokamerou může být někdy obtížné rozpoznat, o jaký objekt se jedná, a proto je příhodné mít možnost zobrazit si daný objekt i ve viditelném spektru. Na přípravku se dále nachází LED bar graf, 4pólový DIP přepínač a 2 tlačítka pro ovládání programu a signalizaci stavů během vývoje. Napájení vývojové desky a kamerových modulů je řešeno přes kulatý napájecí DC konektor a lineární 5voltový regulátor.



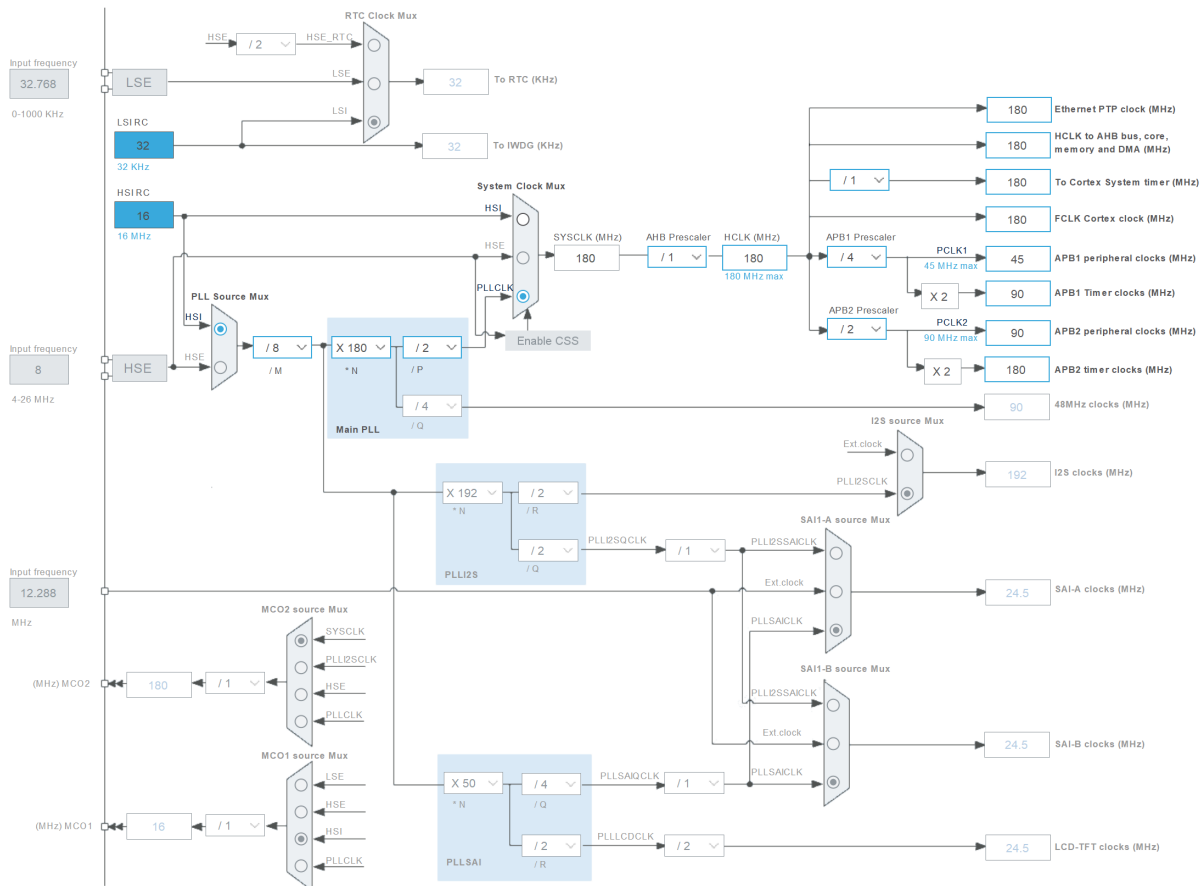
Obrázek 18: Návrh desky plošného spoje přípravku pro připojení kamerových modulů k STM32 vývojové desce.

3.1.1 Ethernet

Součástí mikrokontroléru STM32F411ZI je periferie poskytující funkce linkové vrstvy Ethernetu. Implementuje MAC ze standardu IEEE 802.3 a přes MII rozhraní komunikuje s externím PHY, které řídí fyzickou vrstvu Ethernetu [26]. Vývojová deska NUCLEO-F429ZI integruje jako PHY integrovaný obvod LAN8742. Z původního MII rozhraní vychází upravená varianta RMII s redukováním počtem propojovacích signálů mezi MAC a PHY, které tento transceiver používá [27].

3.1.2 Hodinové taktý

Všechny hodinové taktý mikrokontroléru byly nastaveny na maximální frekvenci. Podrobná konfigurace taktů periferií je vidět na obrázku 19.



Obrázek 19: Konfigurace hodinových taktů jádra procesoru a periférií.

3.2 lwIP TCP/IP stack

Vývojová deska implementuje vrstvu síťového rozhraní z TCP/IP zásobníku protokolů a zbylé vrstvy musí být implementovány softwarově. Jednou z takových implementací je open-source projekt **lwIP - A Lightweight TCP/IP stack**. Jedná se o plnohodnotný zásobník TCP/IP protokolů určený pro embedded zařízení. Klade důraz na rychlost, univerzalitu a nízké nároky na paměti ROM a RAM. Zdrojový kód není závislý na cílové platformě, kvůli tomu ale vyžaduje ovladače pro abstrahování přístupu k MAC periférii na konkrétním systému. Vývojové prostředí CubeIDE má přímou podporu lwIP včetně veškerého podpůrného softwaru a konfigurace. Dále obsahuje volitelná rozšíření pro lwIP, jako je například DHCP server.[28]

LwIP zásobník protokolů byl v této práci použit pro realizování komunikace mezi vývojovou deskou a počítačem přes protokol UDP na transportní vrstvě. Kapitola 2.3 popisuje podrobněji komunikaci v počítačových sítích.

3.2.1 Paketový buffer

K odesílání a přijímání dat v lwIP jsou určeny paketové buffery definované strukturou **pbuf**. Buffer kromě ukazatele na aplikační data obsahuje i ukazatel na následující zřetěžený buffer.

K řetězení bufferů dojde v okamžiku, kdy délka aplikačních dat přesáhne alokovanou velikost jednoho bufferu a data se rozdělí do více bufferů. Pro alokaci paketového bufferu je dostupných několik možností:

- **RAM:** Payload bufferu i struktura pbuf je uložena na haldě v RAM jako jeden nepřerušovaný blok paměti. Dokumentace lwIP doporučuje pro odesílání volit tento typ alokace.
- **POOL:** Pool je sada předem alokovaných bloků paměti v RAM, a proto při alokaci bufferu nemusí být prohledávána celá halda jako v předchozím typu alokace, místo toho je vybrán některý z volných bloků paměti v poolu. Vede to na rychlejší alokaci, ale nevýhodou je, že i při alokaci velmi krátkého bufferu je obsazen celý buffer z poolu, přestože je většina paměti nevyužitá. V RAM jsou uloženy payload i pbuf struktura. Dokumentace lwIP nedoporučuje pro odesílání volit tento typ alokace, protože při TCP přenosu může dojít k zařazení paketů k odeslání do fronty bez toho, aniž by byly buffery z poolu průběžně uvolňovány. Tím dojde k vyčerpání poolu a přestanou být přijímány režijní TCP pakety.
- **ROM:** Pro pbuf strukturu je alokována paměť v poolu a payload bufferu odkazuje do ROM. Před odesláním nejsou data zkopírována, protože se nepředpokládá změna v průběhu odesílání.
- **REF:** Podobně jako u ROM je struktura uložena v poolu, ale data jsou před odesláním zkopírována, protože by mohla být změněna.

Pokud je buffer alokovaný na haldě v RAM, k zřetězení nedojde. Když je ale alokovaný z poolu a je požadována větší délka bufferu, než je délka elementárního bufferu v poolu, bude alokováno více zřetězených bufferů za sebou. Pokud je pro aplikaci vyžadováno alokovat nezřetězený buffer z poolu o větší velikosti, než je nastavena v základu, je toho možné docílit konfiguračním parametrem *TCP_MSS*, který podrobněji popisuje podkapitola 3.2.2.

V práci byla měřena doba trvání alokace 800 bajtů nejdříve na haldě v RAM a poté z poolu při taktu jádra procesoru 180 MHz. Alokační na haldě v RAM trvala 2,7 μ s a z poolu 4,3 μ s. Alokační z poolu bufferů je rychlejší z toho důvodu, protože všechny buffery jsou stejně dlouhé a algoritmus pro nalezení volného prostoru v paměti je jednodušší než při alokaci z haldy o různě velkých blocích. Přestože lwIP dokumentace nedoporučuje alokovat paketový buffer z poolu pro odchozí pakety, v některých případech, kdy jsou kladeny velké požadavky na časování programu, toho lze využít. V práci bylo vyzkoušeno odesílání UDP paketů alokovaných z poolu a nebyly zjištěny chybové stavy.

Konfigurační parametr *MEMP_NUM_PBUF* stanovuje počet alokovaných bufferů v poolu. Pokud aplikace alokuje buffery převážně v RAM, není třeba zbytečně obsazovat RAM paměť mikrokontroléru alokacími bufferů v poolu, které nebudou využity. Naopak při četnější alokaci bufferů v poolu o větší délce aplikačních dat může dojít k vyčerpání poolu a je vhodné počet alokovaných bufferů zvýšit. [28]

3.2.2 Konfigurace

Výchozí lwIP konfigurace se nachází převážně v souboru *opt.h*, kde jsou definice všech konfiguračních parametrů s popisy. Změna výchozí hodnoty je provedena předefinováním konfiguračního parametru v souboru **lwipopts.h**. Pro účely této práce byl povolen DHCP klient a provedeny následující změny v konfiguraci:

```
1 #define MEM_SIZE 10000
2 #define MIN_SIZE 200
3 #define MEMP_NUM_PBUF 4
```

V podkapitole 2.2.5 byla odvozena a blíže popsána maximální velikost aplikačních dat UDP datagramu, při které je zaručeno doručení datagramu napříč libovolnou počítačovou sítí respektující TCP/IP bez nutnosti fragmentace. Komunikace v této práci je předpokládána po lokální Ethernetové síti, kde není problém přenášet i delší datagramy, a proto byl doporučený konzervativní limit zvýšen. TCP obdobně určuje limit velikosti uživatelských dat na jeden segment, který je v lwIP nastaven parametrem *TCP_MSS* původně na 536 bajtů. Přestože je parametr určen pro TCP, lwIP z něho vypočítává **velikost paketových bufferů v poolu**, které jsou použity pro TCP i UDP. V případě, kdy je požadováno alokovat jeden buffer jako celistvý blok paměti bez toho, aniž by došlo k zřetězení více bufferů za sebe, musí být tato hodnota zvětšena.

Velikost paketového bufferu v poolu lze změnit i přímo přepsáním původní definice, která je uvedena níže. Je ale potřeba brát v potaz velikosti všech hlaviček protokolů, pro které musí být včetně aplikačních dat paměť alokována.

```
1 #define PBUF_POOL_BUFSIZE ...
    LWIP_MEM_ALIGN_SIZE(TCP_MSS+40+PBUF_LINK_ENCAPSULATION_HLEN+PBUF_LINK_HLEN)
```

Počet předem alokovaných bufferů v poolu je nastaven parametrem *MEMP_NUM_PBUF*, který byl v této práci snížen, protože jediná situace, kdy bude buffer alokován z poolu, je při přijímání řídicích paketů z počítače, které přichází velmi zřídka a nedojde tedy k vyčerpání poolu. V ostatních případech jsou paketové buffery alokovány v RAM.

V případě alokace paketových bufferů na haldě v RAM je důležitý parametr *MEM_SIZE*. Ten určuje celkovou velikost haldy pro dynamickou alokaci paměti v rámci lwIP funkcí a při současném přenášení většího množství dat je vhodné tento parametr zvětšit. Parametr *MIN_SIZE* dále stanovuje minimální velikost elementárního bloku paměti na haldě, kdy nižší hodnoty vedou na efektivnější využití paměti za cenu větší fragmentace paměti. [28]

3.2.3 Inicializace rozhraní

Rozhraní v lwIP přiřazuje IP adresu, masku podsítě a IP adresu výchozí brány k fyzickému síťovému rozhraní systému a alespoň jedno rozhraní musí být nastaveno před tím, než je možno

navázat komunikaci po síti. Inicializaci a přidání rozhraní do lwIP ukazuje následující útržek kódu.

```
1 lwip_init();
2 struct netif gnetif;
3 netif_add(&gnetif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, ...
           &ethernet_input);
4 netif_set_default(&gnetif);
5 netif_set_up(&gnetif);
6 netif_set_link_callback(&gnetif, ethernet_link_status_updated);
```

Prvním krokem inicializace je spuštění lwIP zásobníku protokolů přes funkci *lwip_init*, tím dojde zejména k inicializaci systému pro alokaci paměti. Nyní již lze přidat rozhraní, jehož konfiguraci obsahuje struktura **netif**. Funkce **lwip_add** nejdříve tuto strukturu připraví a poté zaregistruje nově nakonfigurované rozhraní do lwIP. Funkce přijímá parametry pro nastavení IP adresy, masky podsítě, IP adresy výchozí brány a parametry pro konfiguraci ovladače síťového přístupu. Mezi tyto parametry pro ovladač spadají dodatečné stavové informace a odkazy na 2 funkce. První funkce je volána lwIP zásobníkem po připravení struktury *netif* a obsahuje specifický inicializační kód ovladače. Druhou funkci volá ovladač vždy při přijmutí nového Ethernetového rámce a data předá do lwIP.

Následně je funkcí **netif_set_up** rozhraní softwarově spuštěno. Ke změně dostupnosti linkové vrstvy na síťovém rozhraní může dojít kdykoliv, například odpojením kabelu, a proto lwIP dostupnost periodicky kontroluje. Funkce **netif_set_link_callback** přiřadí k vybranému rozhraní zpětné volání uživatelské funkce, která je automaticky volána při změně dostupnosti síťového rozhraní.

V případě dynamického nastavení rozhraní DHCP serverem je IP adresa, maska a výchozí brána nastavena při vytváření lwIP rozhraní na nulovou hodnotu a po skončení inicializace je zavolána funkce **dhcp_start**. Informaci o tom, zda již došlo k nakonfigurování rozhraní, poskytuje návratová hodnota funkce **dhcp_supplied_address**. Po úspěšném získání nastavení z DHCP serveru jsou původně nastavené hodnoty ve struktuře *netif* nahrazeny novými údaji a IP adresu lze například vypsát do terminálu. [28]

3.2.4 Kontrola stavu rozhraní

V předchozí podkapitole bylo zmíněno, že ke změně stavu dostupnosti síťového rozhraní může dojít kdykoliv. Struktura rozhraní *netif* obsahuje statusovou proměnnou *flags*. Příznak *NETIF_FLAG_UP* informuje o softwarovém stavu rozhraní v lwIP, zda-li je povoleno a připraveno pro síťový provoz. Druhý příznak *NETIF_FLAG_LINK_UP* nastavuje přímo ovladač vrstvy síťového přístupu a oznamuje dostupnost linkové vrstvy. Makra *netif_is_up* a *netif_is_link_up* usnadňují přístup k vybraným příznakům.

V závislosti na konkrétní implementaci ovladače vrstvy síťového přístupu vychází i kontrola

stavu rozhraní a průběh nastavování zmíněných příznaků se může lišit. Některé PHY například nepodporují detekci stavu linkové vrstvy. V lwIP implementaci v STM32CubeIDE pro hardware použitý v této práci je periodicky volána funkce `ethernet_link_check_state`, která kontroluje stav linkové vrstvy u externího PHY a podle toho oba příznaky současně nastavuje. V tomto případě tedy mezi příznaky není rozdíl a dostupnost rozhraní lze zjistit pomocí libovolného příznaku. [28]

3.2.5 UDP Protocol Control Block (PCB)

Před zahájením UDP komunikace mezi 2 zařízeními musí být v lwIP nastaven lokální a vzdálený koncový síťový bod. K tomu slouží struktura `udp_pcb`, pro kterou je alokována paměť funkcí `udp_new`. Následně je nastavena IP adresa a port lokálního koncového bodu mikrokontroléru přes funkci `udp_bind`.

Lokální IP adresa je adresa síťového rozhraní, které bylo vytvořeno v podkapitole 3.2.3. Pokud je přítomno více rozhraní, tímto je možné filtrovat, na kterém z nich bude nasloucháno pro příchozí UDP datagramy. V případě, kdy je požadováno naslouchat na všech rozhraních, musí být do funkce parametrem předána nulová IP adresa (nebo zjednodušující makro `IP4_ADDR_ANY`). Pokud je lokální port nulový, lwIP automaticky přidělí jeden z volných portů v rozsahu definovaném konfiguračními parametry `UDP_LOCAL_PORT_RANGE_START` a `UDP_LOCAL_PORT_RANGE_END`.

Obdobně jsou IP adresa a port vzdáleného koncového bodu nastaveny funkcí `udp_connect`. Všechny provedené kroky doposud nezpůsobily zatím žádný síťový provoz, ale pouze jen inicializaci všech potřebných prvků. Zároveň je vhodné kontrolovat návratové hodnoty všech volaných funkcí, které dávají informaci o tom, zda akce proběhla úspěšně. Následující útržek kódu ilustruje příklad vytvoření a inicializace `udp_pcb` struktury. [28]

```
1 struct udp_pcb* udpPcb = udp_new();
2 if (udpPcb == NULL)
3     return MEMORY_ERROR;
4
5 err_t error_code = udp_bind(udpPcb, IP4_ADDR_ANY, localPort);
6 if (err_code != ERR_OK)
7     return UDP_BIND_ERROR;
8
9 error_code = udp_connect(udpPcb, destIp, destPort);
10 if (error_code != ERR_OK)
11     return UDP_CONNECT_ERROR;
```

3.2.6 Příjem UDP datagramu

Po úspěšném vytvoření a inicializaci řídicí struktury `udp_pcb` pro UDP spojení je dále potřeba specifikovat uživatelskou funkci pro zpracování příchozích UDP datagramů, která bude systémem

lwIP automaticky volána. Toho je docíleno funkcí `udp_recv`, která přijme odkaz na vybranou funkci pro zpracování datagramu a argument, který bude funkci předán s každým voláním. Při příjmu UDP datagramu lwIP systém kontroluje všechny připravené PCB k příjmu a přijímací funkci vyvolá u těch, u kterých se lokální síťový bod shoduje s cílovým bodem příchozího datagramu.

Přijímací funkci je v parametrech předán argument, PCB definující UDP spojení, paketový buffer s aplikačními daty, IP adresa koncového bodu vysílače a port. Před opuštěním funkce musí být paketový buffer uvolněn voláním funkce `pbuf_free`, jinak by došlo k úniku paměti. [28]

```
1 udp_recv(udpPcb, UdpReceiveCallback, arg);
2
3 void UdpReceiveCallback(void *arg, struct udp_pcb *upcb, struct pbuf *p, ...
   const ip_addr_t *addr, u16_t port)
4 {
5     uint8_t* data = (uint8_t*)p->payload;
6     ProcessPayload(data);
7     pbuf_free(p);
8 }
```

3.2.7 Odeslání UDP datagramu

Odeslání UDP datagramu začíná vytvořením struktury paketového bufferu `pbuf` a alokací paměti pro aplikační data ve zvoleném paměťovém prostoru, který podrobněji popisuje podkapitola 3.2.1. Funkce `pbuf_alloc` přijme v prvním parametru součet délek hlaviček všech protokolů odvíjejících se od vrstvy, na které bude datagram odeslán. LwIP definuje makra s výpočtem délky hlaviček v závislosti na vrstvě:

- PBUF_RAW
- PBUF_RAW_TX
- PBUF_LINK
- PBUF_IP
- PBUF_TRANSPORT

Druhý parametr alokační funkce přijme délku aplikačních dat, která budou odeslána a v posledním parametru přijme lokaci v paměti, kde bude prostor pro aplikační data alokován.

LwIP poskytuje pro nahrání dat do bufferu funkci `pbuf_take`, která i v případě zřetězení zajistí správně rozdělení a nahrání aplikačních dat do všech zřetězených bufferů. Druhým způsobem, jak nahrát data do bufferu, je vlastní správa paměti. Ukazatel `payload` ve struktuře `pbuf` ukazuje na začátek paměti s aplikačními daty. Buffer lze naplnit například funkcí `memcpy`, DMA řadičem nebo přímým ukládáním příchozích dat z SPI nebo I²C datového registru.

Poté, co jsou aplikační data nahrána v paketovém bufferu, může být UDP datagram odeslán funkcí `udp_send`. V prvním parametru přijímá PCB, který určuje zdrojový a cílový koncový bod v síti a ve druhém parametru paketový buffer. Nakonec musí být funkcí `pbuf_free` zajištěno uvolnění paměti po paketovém bufferu, aby nedošlo k úniku paměti.

Následující útržek kódu ukazuje příklad odeslání UDP datagramu s aplikačními daty v podobě ASCII textu a alokací bufferu na haldě v RAM.

```

1 const char* data = "UDP_datagram_payload";
2 struct pbuf* packetBuffer = pbuf_alloc(PBUF_TRANSPORT, strlen(data), PBUF_RAM);
3 strcpy(packetBuffer->payload, data);
4 udp_send(udpPcb, packetBuffer);
5 pbuf_free(packetBuffer);

```

3.3 Aplikační protokol

Pro přenos dat byl navržen v základu jednotný aplikační protokol, který je upraven pro každý typ datového přenosu. První datový bajt paketu vždy reprezentuje typ přenosu a je podle něho vybrána patřičná akce při jeho přijetí. V tabulce 9 je vidět přehled všech typů paketů.

Tabulka 9: Typy paketů aplikačního protokolu

Identifikační hodnota	Typ přenášených dat
0x01	Systém
0x02	FLIR Lepton
0x03	Arducam
0x04	AMG8833
0x05	MLX90640

Systémový paket přenáší buď režijní data pro řízení spojení mezi aplikací pro zpracování dat a mikrokontrolérem, nebo příkazy pro změnu nastavení kamerových modulů. Druhý datový bajt nese typ požadované akce, jejichž seznam je vidět v tabulce 10. Systémové pakety jsou mikrokontrolérem potvrzovány **ACKNOWLEDGE** pakety a v případě nepotvrzení nejsou znovu odesílány. Možným vylepšením by bylo pro systémové pakety použít TCP protokol, který zaručuje doručení paketu. Po bajtu nesoucím informaci o požadované akci mohou být ve zbylých bajtech paketu přenášeny dodatečné parametry akce, jako je například zvolené rozlišení u akce ARDUCAM REOSLUTION.

Tabulka 10: Typy akcí systémového paketu aplikačního protokolu

Identifikační hodnota akce	Systémová akce
0x01	CONNECT (připojení k MCU)
0x02	DISCONNECT (odpojení od MCU)
0x03	TEST CONNECTION (zkouška dostupnosti spojení s MCU)
0x04	ACKNOWLEDGE (potvrzení vykonání akce)
0x05	ARDUCAM RESOLUTION (změna rozlišení Arducam modulu)

3.4 UDP spojení

Spojení je navázáno odesláním systémového paketu z koncové aplikace s akcí **CONNECT** do mikrokontroléru, který připojení potvrdí akcí **ACKNOWLEDGE**. Dojde tím k přiřazení koncové IP adresy a UDP portu aplikace k PCB lwIP zásobníku a započne proces snímání a odesílání dat kamerových modulů. Speciálním případem je akce **TEST CONNECTION**, na kterou mikrokontrolér odpoví taktéž **ACKNOWLEDGE** akcí, nevytvoří ale aktivní spojení a nadále naslouchá na příchozí požadavky o navázání spojení. Odpojení koncové aplikace od mikrokontroléru nastává odesláním akce **DISCONNECT**, na kterou mikrokontrolér taktéž odpoví **ACKNOWLEDGE** akcí a následně zastaví zpracování kamerových dat do té doby, než je vytvořeno nové spojení.

3.5 Struktura programu

Ze zadaných požadavků plyne, že program mikrokontroléru musí zároveň obsluhovat 2 libovolné kamerové moduly bez toho, aniž by proces čtení snímku z jednoho modulu zcela zablokoval a zamezil čtení snímku z druhého modulu. Například FLIR Lepton snímá při konstantní snímkové frekvenci a pokud výstupní data nejsou průběžně vyčítána, vyhodnotí ztrátu synchronizace. Procesorový čas musí být tedy rovnoměrně rozdělen mezi oba 2 moduly, což vede na implementaci paralelního běhu **procesů**. Pokud by měl procesor mikrokontroléru více jader, každý kamerový modul by mohl být skutečně paralelně zpracováván na každém jádře procesoru zvlášť, nicméně procesor použitého mikrokontroléru je jednovádrový a běh procesů musí být na tomto jádře přepínán, jedná se tedy o **pseudoparalelní** běh procesů.

Existují operační systémy pracující v reálném čase (RTOS) pro embedded platformy, které tuto problematiku řeší. Jejich speciální vlastnost je, že umožňují reagovat na vnitřní i vnější události mikrokontroléru přesně v předem stanovených časových limitech a tím dodržet striktní časování programu. Pro platformu STM32 je ve vývojovém prostředí STM32CubeIDE předem připravená open-source implementace toho operačního systému **FreeRTOS**.

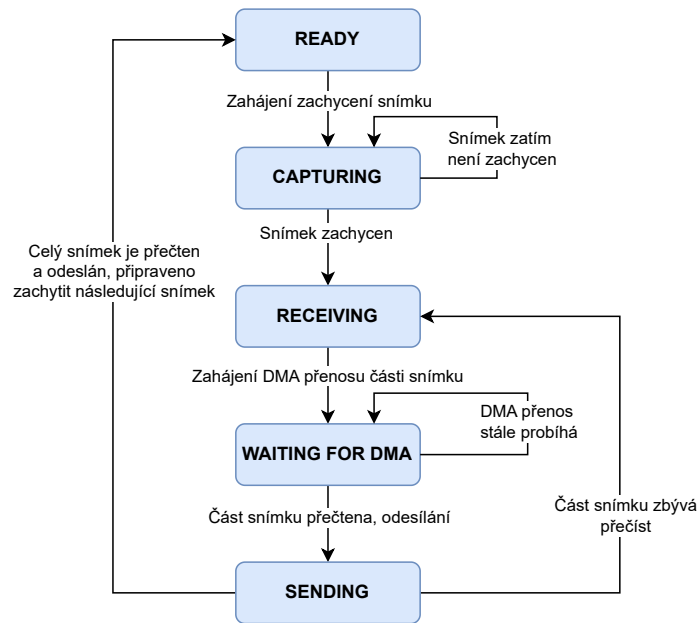
Pro realizaci praktické části této diplomové práce nakonec nebyl použit žádný hotový operační systém, protože současně bude spuštěn velmi nízký počet procesů bez potřeby pokročilejšího řízení. Místo toho byl vytvořen vlastní jednoduchý systém předávání procesorového času mezi jednotlivými procesy, který je založen na principu stavového automatu. Koloběh získávání snímků je rozložen do několika kroků (stavů), jejichž počet se odvíjí od náročnosti komunikace s daným kamerovým modulem. Obecně se jedná ale o tyto kroky:

- Přípraveno zahájit snímání
- Snímání zahájeno, čeká se na dokončení
- Čtení dílčí části snímku z modulu
- Odesílání UDP fragmentu s dílčí částí snímku

Hlavní smyčka programu periodicky volá v každém procesu vykonávací funkci, která vykoná dílčí úlohu podle aktuálního stavu procesu a na konci v současném stavu buď zůstane, nebo přejde do nového stavu. Poté je volání funkce ukončeno a je volána vykonávací funkce následujícího procesu. Tímto je zajištěno, že všechny procesy vykonávají průběžně jednotlivé dílčí části algoritmu a když například proces prvního kamerového modulu čeká, než bude snímek připraven ke čtení, proces druhého modulu tento čas může efektivně využít ke zpracování snímku. Avšak při návrhu je klíčové zajistit, aby některý stav netrval příliš dlouhou dobu k vykonání, protože přepínání mezi procesy neřídí operační systém v závislosti na uplynulém čase, ale procesy samy. Jakmile je u procesu vyvoláno vykonání stavu, ostatní procesy jsou blokovány do té doby, než je stav plně vykonán.

3.6 Arducam Mini 2MP Plus

K praktickému vývoji programu mikrokontroléru byl zvolen kamerový modul **Arducam Mini 2MP Plus**. Na obrázku 20 je vidět diagram základních stavů řídicího programu.



Obrázek 20: Vývojový diagram základních stavů řídicího programu kamerového modulu Arducam

3.6.1 Ovladač SCCB sběrnice

Přestože SCCB specifikace, popsaná v podkapiole 1.2.3, v některých ohledech odporuje specifikaci I²C sběrnice, nebyly zjištěny žádné chyby přenosu dat při použití I²C periferie mikrokontroléru ke komunikaci s modulem. Následující útržek kódu ukazuje implementaci ovladače **zapisovacího a čtecího cyklu** SCCB sběrnice s využitím HAL knihovny pro obsluhu I²C periferie mikrokontroléru. Zapisovací a čtecí funkce HAL knihovny nejsou příliš vhodné pro přenos dat o velikosti jednotek bajtů, jelikož režijní činnost přenosových funkcí knihovny dosahuje srovnatelné doby výkonání, jako samotný proces odesílání dat přes přímý přístup k registrům I²C periferie. Nicméně SCCB sběrnice slouží jen ke konfiguraci snímače, kdy největší objem dat je přenesen pouze jednou při startu zařízení pro počáteční konfiguraci. Poté jsou již přenášeny dílčí konfigurační data s mnohem nižší četností, například pro změnu rozlišení.

```

1  static uint8_t SccbReadReg8b(uint8_t regAddr)
2  {
3      uint8_t result;
4      HAL_I2C_Master_Transmit(i2cHandle, OV2640_I2C_WRITE_ADDR, &regAddr, 1, 1000);
5      HAL_I2C_Master_Receive(i2cHandle, OV2640_I2C_READ_ADDR, &result, 1, 1000);
6      return result;
7  }
8
9  static void SccbWriteReg8b(uint8_t regAddr, uint8_t regVal)
10 {
11     uint8_t buffer[2] = {regAddr, regVal};
12     HAL_I2C_Master_Transmit(i2cHandle, OV2640_I2C_WRITE_ADDR, buffer, 2, 1000);
13 }
  
```

3.6.2 Čtení snímků

Data snímku mohou být v závislosti na zvoleném rozlišení poměrně objemná a s relativně nízkou frekvencí hodinového taktu SPI sběrnice může přenos trvat dlouhou dobu. Aby nebyl procesor po dobu přenosu blokován, byl použit DMA řadič, který přenáší data z SPI sběrnice přímo do aplikačních dat paketového bufferu. Jelikož časovací obvody periférií mikrokontroléru neumožňovaly nastavit takovou kombinaci, aby frekvence jádra byla nastavena na 180 MHz a takt SPI sběrnice na 8 MHz, byla pro takt SPI sběrnice vyzkoušena následující možná vyšší hodnota, a to 11,25 MHz. Bylo zjištěno, že i se zvýšenou frekvencí hodinového signálu sběrnice nad limit stanovený v datovém listu, byl přenos pro účely této práce stabilní a bez chyb. Stabilita přenosu dat však dále zkoumána nebyla a nemusí platit ve všech možných případech.

Čtení dat snímku je rozděleno do n bloků o konstantní velikosti (s výjimkou posledního bloku, ten může být kratší), které jsou DMA řadičem ukládány do paketového bufferu a průběžně odesílány UDP protokolem. Velikost bloku, a s tím související velikost paketového bufferu, byla nastavena na 1200 B a je změnitelná v konfiguraci. Pokud není velikost snímku dělitelná velikostí bloku bezzbytku, poslední blok je stejně velký jako zbytek. Proměnná *readSize* určuje velikost aktuálně zpracovávaného bloku, pro který je alokován paketový buffer a následně spuštěn DMA přenos. V proměnné *imageSize* je po zachycení snímku uložena jeho velikost a indikuje, kolik bajtů zbývá přečíst a odeslat. S každým odeslaným blokem je proměnná dekrementována a po dosažení 0 je snímek kompletně zpracován. Proměnná *imageSize* je dále použita k nastavení příznaku paketu, že se jedná o poslední fragment snímku, neboť datová velikost JPEG snímku není konstantní, a nelze tedy na přijímací straně předpovídat, na kolik fragmentů bude snímek rozdělen.

```
1 uint32_t readSize = imageSize < ARDUCAM_FRAGMENT_SIZE ? imageSize : ...
   ARDUCAM_FRAGMENT_SIZE;
2 InitPacketBuffer(readSize);
3 HAL_SPI_Receive_DMA(spiHandle, (uint8_t*)packetBuffer->payload+8, readSize);
4 uint8_t lastFragment = (imageSize - ARDUCAM_FRAGMENT_SIZE) ≤ 0;
5 imageSize -= readSize;
```

Funkce *InitPacketBuffer* alokuje paketový buffer s ohledem na velikost aktuálního fragmentu a inicializuje hodnoty aplikačního protokolu. V tabulce 11 je vidět struktura paketu aplikačního protokolu pro odesílání snímků do aplikace pro zpracování dat.

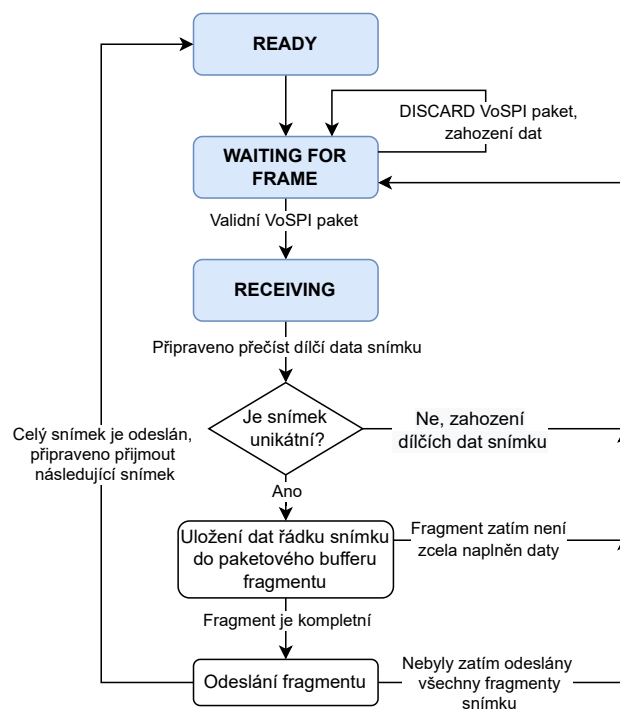
První výhodou zvoleného provedení je významné snížení nároků na RAM paměť mikrokontroléru, protože velikost celého snímku při maximálním rozlišení může dosahovat více než 100 kB. Druhá výhoda souvisí s celkovým návrhem procesového systému, kdy je průběžně s každým zpracovaným blokem dat snímku předáno řízení jinému procesu.

Tabulka 11: Struktura aplikačního protokolu pro přenos snímků kamerového modulu Arducam Mini 2MP Plus

offset	data
0	typ paketu = 3 (1 B)
1	číslo snímku (4 B)
5	číslo fragmentu (2B)
7	indikace posledního fragmentu (1 B)
8	data snímku (až 1200 B)

3.7 FLIR Lepton 2.5

K praktickému vývoji programu mikrokontroléru byla zvolena verze **2.5** kamerového modulu FLIR Lepton. Algoritmus obsluhy modulu vychází z práce [1] a byl upraven pro zvýšení propustnosti dat po SPI sběrnici, spolehlivosti udržení spojení s modulem a odesílání snímků přes UDP. Na obrázku 21 je vidět diagram základních stavů programu.

**Obrázek 21:** Vývojový diagram základních stavů řídicího programu kamerového modulu FLIR Lepton

3.7.1 Ovladač SPI sběrnice

Kombinace nastavení časovacích obvodů hodinových taktů periférií taktěz nebylo možné pro tento modul nastavit tak, aby bylo dosaženo maximální frekvence jádra procesoru a zároveň maximální povolené frekvence hodinového taktu SPI sběrnice pro FLIR Lepton, tj 20 MHz. Byla zvolena nejbližší možná frekvence hodinového taktu SPI sběrnice 22,5 MHz a v rámci vývoje

této práce nebyla zjištěna nestabilita datového přenosu, je však doporučeno nepřekračovat limity výrobce. Rozhraní bylo konfigurováno v režimu SPI mode 3 ($CPOL = 1$ a $CPHA = 1$).

Čtení dat z SPI sběrnice bylo z původních HAL knihoven nahrazeno přímým přístupem k registrům SPI periferie. Lepton po SPI sběrnici data pouze odesílá a v následující ukázce kódu je vidět funkce pro příjem dat z modulu. Z vybrané SPI periferie *spi* je přijat počet bajtů stanovený v proměnné *size* do pole *buffer*. Příznak *TXE* status registru *SR* značí, že vysílací buffer je prázdný. Když je buffer prázdný, je do něho zapsán nulový bajt skrz registr *DR*. Příznak *RXNE* oznamuje, že v přijímacím bufferu je přijat nový bajt, který je poté vyčten z registru *DR*.

```

1 void CameraSpiReceive(SPI_TypeDef* spi, uint8_t buffer[], uint32_t size)
2 {
3     for (uint32_t i=0;i<size;i++)
4     {
5         while(!(spi->SR & SPI_SR_TXE));
6         spi->DR = 0;
7         while(!(spi->SR & SPI_SR_RXNE));
8         buffer[i] = spi->DR;
9     }
10
11     while(spi->SR & SPI_SR_BSY);
12 }

```

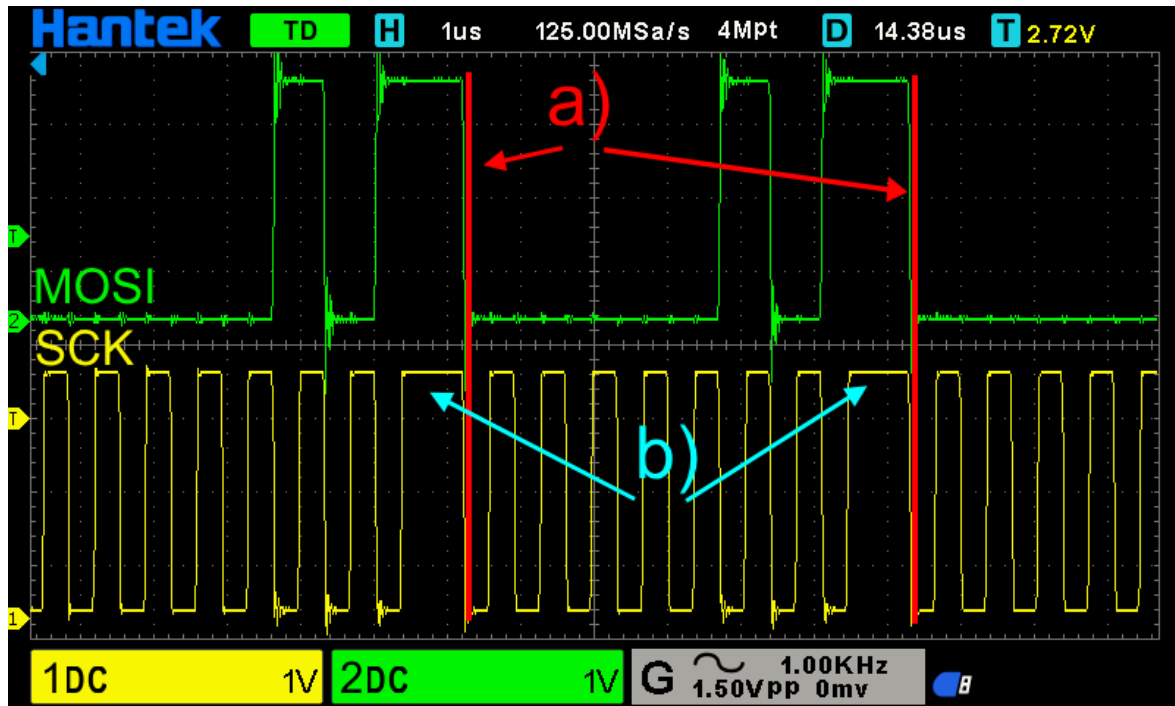
Přestože Lepton nic nepřijímá, do registru *DR* musí být stejně zapisováno, protože právě zápis spouští SPI přenos. Před tím, než je zapsán bajt $n + 1$ do *DR*, program vyčkává, než je bajt n přijat a vyčten z *DR*. Jenže v tento okamžik je již vysílací buffer prázdný a přenos je přerušeno do té doby, než je právě zapsán bajt $n + 1$ do *DR*, čímž dochází k **přerušovanému SPI přenosu**. Na obrázku 22 je vidět, kde k přerušování přenosu dochází. V bodech **a**) je počátek vysílaného bajtu a v bodech **b**) je vidět setrvání hodinového taktu v klidové úrovni po vyslání posledního bitu, než je vysílací buffer zaplněn následujícím bajtem. Zpoždění závisí na frekvenci jádra procesoru, počtu instrukcí k vykonání ve smyčce a nastavení fáze SPI.

Jednoduchou změnou lze kód upravit, aby byl přenos spojitý, a to zapsat bajt n do *DR* před tím, než je spuštěn for cyklus. Ve chvíli, kdy se program vrací na začátek smyčky, není vysílací buffer prázdný a přenos není přerušeno. Komplikace ale nastává v případě, kdy je software příliš pomalý na to, aby stihl vyčíst příchozí bajt z *DR* a mezitím dojde k přijmutí dalšího bajtu, který ten předchozí přepíše. Při vývoji bylo zjištěno, že při frekvenci hodinového taktu SPI 22,5 MHz procesor při taktu 180 MHz nestíhal včas vyčíst příchozí bajty kvůli zpoždění režijních instrukcí algoritmu smyčky, a proto byl zvolen přerušovaný přenos, jehož spolehlivost na taktu jádra procesoru nezávisí. Další možností, jak zajistit spojitý SPI přenos, je použít režim *RX – only* nebo DMA řadič.

```

1 spi->DR = 0;
2 size--;
3 for (uint32_t i=0;i<size;i++)
4 {
5     ...

```



Obrázek 22: Záznam z osciloskopu přerušovaného SPI přenosu slova 0x05 ($CPOL = 1$, $CPHA = 1$)

Lepton odesílá snímky trojnásobnou frekvencí, než snímá, tedy že každý třetí snímek je unikátní a zbylé dva jsou duplicitní. Tyto duplicitní snímky není třeba odesílat a jsou proto zahozeny, stejně jako nevalidní VoSPI pakety. Následující ukázka kódu funkce přečte a zahodí z SPI periferie *spi* počet bajtů stanovený v proměnné *size*. Periodicky je po sběrnici vysílán nulový bajt, čímž je spuštěn přenos. Jelikož není žádoucí žádná data přijímat, nemusí se čekat na příznak *RXNE*, příchozí data jsou v bufferu připsována a přenos probíhá bez přerušení. Po odeslání posledního bajtu je potřeba vyčkat po dobu nastavení příznaku *BSY*, což značí, že je poslední bajt stále přenášen. Protože nebyly příchozí bajty z registru DR vyčítány a došlo k jejich přepsání, byl nastaven příznak *OVR*, který je resetován přečtením registru *DR* a *SR*.

```
1 void CameraSpiDrop(SPI_TypeDef* spi, uint32_t size)
2 {
3     while(size)
4     {
5         while(!(spi->SR & SPI_SR_TXE));
6         spi->DR = 0;
7         size--;
8     }
9
10    while(spi->SR & SPI_SR_BSY);
11    (void) spi->DR;
12    (void) spi->SR;
13 }
```

3.7.2 Čtení snímků

Snímek je rovnoměrně rozdělen na několik UDP fragmentů, do kterých jsou data snímku ukládána po celých násobcích řádků. Pro výchozí nastavení bylo zvoleno dělení jednoho snímku po 5 řádcích, a tedy celkem na 12 fragmentů. Pokud není paketový buffer alokován, alokuje se a následně jsou čtená data snímku z SPI ukládána přímo do paketového bufferu. Pokud byl fragment naplněn, je paket odeslán.

```
1 if (packetBuffer == NULL)
2     InitPacketBuffer()
3
4 uint32_t payloadOffset = (lineNumber % 5)*(LEPTON_FRAME_SIZE - 4)+6;
5 CameraSpiReceive(SPI3, ((uint8_t *) packetBuffer->payload)+payloadOffset, ...
6     (LEPTON_FRAME_SIZE - 4));
7
8 if ((lineNumber+1) % 5 == 0)
9     SendUdpFragment();
```

Funkce *InitPacketBuffer* alokuje paketový buffer a inicializuje hodnoty aplikačního protokolu. V tabulce 12 je vidět struktura paketu aplikačního protokolu pro odeslání snímků do aplikace pro zpracování dat.

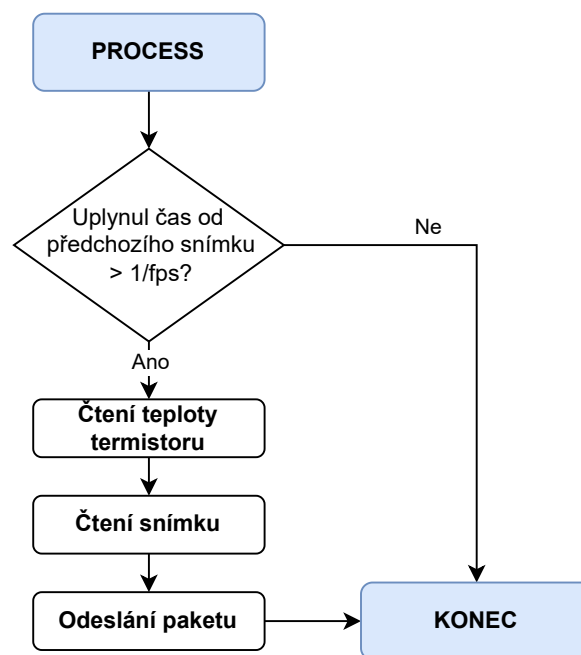
Pokud je potřeba přerušit čtení snímků z modulu, nesmí tak být učiněno během přijímání snímku, jinak může dojít k desynchronizaci s modulem a je vyžadován jeho reset. Při požadavku na odpojení aplikace pro zpracování dat od mikrokontroléru je vyčkáno, než stavový automat zaujme stav *READY* a až poté je čtení snímku přerušeno.

Tabulka 12: Struktura aplikačního protokolu pro přenos snímků infračerveného kamerového modulu FLIR Lepton 2.5

offset	data
0	typ paketu = 2 (1 B)
1	pořadí snímku (4 B)
5	pořadí fragmentu (1B)
6	data snímku (800 B)

3.8 AMG8833

Praktická část práce byla vytvořena pro variantu infračerveného kamerového modulu **AMG8833**, dle teoretické části by měl být kód mikrokontroléru kompatibilní i pro zbylé varianty, je to ovšem potřeba dále ověřit. Vzhledem k jednoduchosti použití modulu nebyl implementován stavový automat, místo toho je v jednom volání vykonávací funkce vyčten celý snímek s teplotou termistoru a následně jsou data odeslána v jediném UDP datagramu bez fragmentace.



Obrázek 23: Vývojový diagram algoritmu programu kamerového modulu AMG8833

Ovladač I²C sběrnice pro čtení a zápis registrů modulu je stejný, jako u Arducam Mini 2MP Plus (viz podkapitola 3.6.1).

3.8.1 Čtení snímků

Modul neposkytuje informaci o zachycení nového snímku, detekce nového snímku byla proto implementována v řídicím programu. Je kontrolováno, zda od předchozího čtení snímku uplynula

delší doba než perioda $T = 1/f_{fps}$. při volbě snímkové frekvence 1 snímek za sekundu je $T = 1000$ ms a při volbě 10 snímků za sekundu je $T = 100$ ms.

```

1  if (HAL_GetTick() < lastFrameTicks+100)
2  {
3      lastFrameTicks = HAL_GetTick()+100;
4      ... //novy snimek je pripraven ke cteni

```

Postupně jsou čteny všechny pixely, jejichž 12bitové celočíselné hodnoty se znaménkem kódované ve dvojkovém doplňku jsou rozděleny do 2 registrů. Přestože je původní hodnota 12bitová, musí být do 16bitové proměnné *temp* uložena tak, aby se její nejvýznamnější bit nacházel v nejvýznamnějším bitu proměnné *temp*. Tím je zajištěna správná reprezentace znaménka napříč změnou bitových délek čísel, ale výsledná hodnota je šestnáctkrát větší a musí být stejným číslem vydělena. Dělení 16 představuje stejnou operaci jako bitový posun vpravo o 4 bity s tím, že je zachováno znaménko.

```

1  static void AMG8833ReadPixels()
2  {
3      uint8_t pixelAddress = AMG8833_T01L;
4      for (uint8_t i=0;i<64;i++)
5      {
6          uint8_t uil = AMG8833ReadRegister(pixelAddress++);
7          uint8_t uih = AMG8833ReadRegister(pixelAddress++);
8          int16_t temp = ((uih << 12) | (uil << 4)) / (1<<4);
9          *((int16_t*)((uint8_t*)packetBuffer->payload+7+(2*i))) = temp;
10     }
11 }

```

V tabulce 13 je vidět struktura paketu aplikačního protokolu pro odesílání snímků a teploty termistoru do aplikace pro zpracování dat.

Tabulka 13: Struktura aplikačního protokolu pro přenos snímků a teploty termistoru infračerveného kamerového modulu AMG8833

offset	data
0	typ paketu = 4 (1 B)
1	číslo snímku (4 B)
5	teplota termistoru (2 B)
7	data snímku (128 B)

3.8.2 Čtení teploty termistoru

Kódovaná teplota termistoru je nejdříve složena ze spodní poloviny *tthl* a horní poloviny *tthh* vnitřního registru modulu. Poté je v proměnné *reg* maskováno 11 nejméně významných bitů, které představují absolutní hodnotu teploty. Ve 12. bitu je uloženo znaménko a pokud je bit nastaven, teplota je záporná.

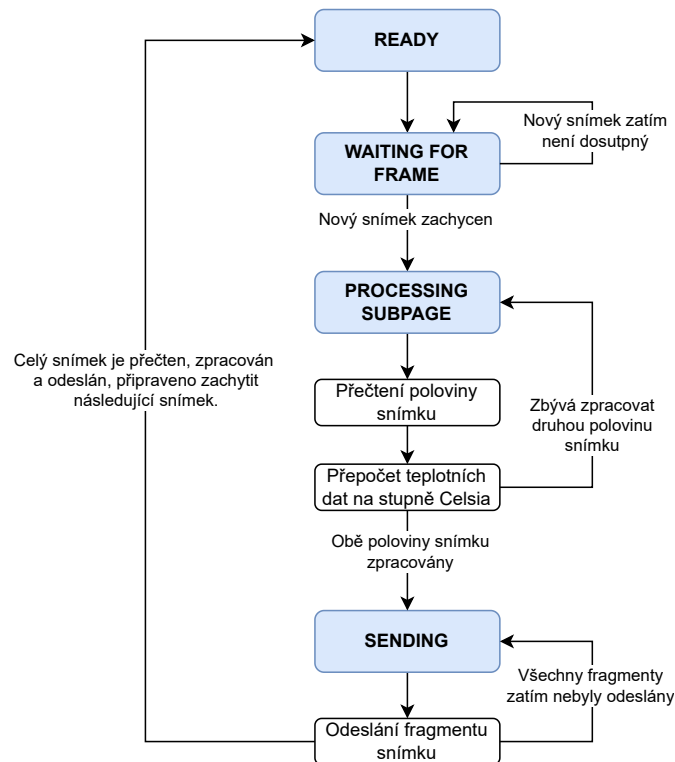
```

1  int16_t AMG8833ReadThermistor()
2  {
3      uint8_t tthl = AMG8833ReadRegister(AMG8833_TTHL);
4      uint8_t tthh = AMG8833ReadRegister(AMG8833_TTHH);
5      uint16_t reg = (tthl | tthh<<8) & 0x7FF;
6      return (tthh & 0x08) ? -1*(int16_t)reg : (int16_t)reg;
7  }

```

3.9 MLX90640

K praktickému vývoji programu mikrokontroléru byl ze skupiny MLX kamerových modulů zvolen model **MLX90640**. Na obrázku 24 je vidět diagram základních stavů řídicího programu.



Obrázek 24: Vývojový diagram základních stavů řídicího programu infračerveného kamerového modulu MLX90640

3.9.1 Ovladač I²C sběrnice

Knihovny výrobce modulu přistupují k I²C sběrnici prostřednictvím ovladače, který musí implementovat následující 2 funkce.

```

1 int MLX90640_I2CRead(uint8_t slaveAddr, uint16_t startAddress, uint16_t ...
    nMemAddressRead, uint16_t *data);
2 int MLX90640_I2CWrite(uint8_t slaveAddr, uint16_t writeAddress, uint16_t data);

```

První funkce ovladače **MLX90640_I2CRead** přečte z kamerového modulu s I²C adresou *slaveAddr* počet 16bitových hodnot *nMemAddressRead* začínajících na 16bitové adrese *startAddress* v paměťové mapě modulu a uloží je do pole *data*. Bylo vyzkoušeno použít samostatné funkce pro zápis adresy paměti a poté pro čtení dat jako u ostatních kamerových modulů využívajících I²C sběrnici, nicméně tyto funkce na konci přenosu vysílají stop kondici a kamerový modul nepokračuje v komunikaci. Místo toho musí být mezi transakcemi znovu vyslána start kondice. V HAL knihovnách je dostupná funkce **HAL_I2C_Mem_Read**, která správně implementuje požadovaný komunikační protokol.

HAL funkce přijímá v parametru počet bajtů k přečtení, oproti tomu parametr ovladače *nMemAddressRead* představuje počet 16bitových hodnot. Proměnná *nMemAddressRead* musí být proto vynásobena dvěma, čehož je efektivně dosaženo bitovým posunem vlevo o 1 bit.

```

1 uint8_t* pData = (uint8_t*) data;
2 HAL_I2C_Mem_Read(i2cHandle, slaveAddr << 1, startAddress, ...
    I2C_MEMADD_SIZE_16BIT, pData, nMemAddressRead << 1, 1000);

```

16bitová data jsou po sběrnici přenášena v big-endian zarovnání a jsou i stejně ukládána do cílového pole *data*. Na druhou stranu STM32 používá little-endian zarovnání paměti, a proto musí být oba bajty 16bitové hodnoty každého indexu v poli mezi sebou přehozeny.

```

1 for(uint16_t i=0; i < nMemAddressRead; i+=2)
2 {
3     uint8_t temp = pData[i+1];
4     pData[i+1] = pData[i];
5     pData[i] = temp;
6 }

```

Druhá funkce ovladače **MLX90640_I2CWrite** zapíše do kamerového modulu s I²C adresou *slaveAddr* 16bitovou hodnotu *data* na adresu v mapě paměti modulu *writeAddress*. Obdobně je použita upravená funkce HAL knihovny **HAL_I2C_Mem_Write** pro korektní realizaci I²C transakce.

```

1 uint8_t cmd[2] = {data >> 8, data & 0x00FF};
2 HAL_I2C_Mem_Write(i2cHandle, slaveAddr << 1, writeAddress, ...
    I2C_MEMADD_SIZE_16BIT, cmd, 2, 100);

```

Zapsaná data jsou nakonec ovladačem znovu přečtena a zkontrolována, zda-li se shodují s požadovanou hodnotou.

```

1 uint16_t dataCheck;
2 MLX90640_I2CRead(slaveAddr << 1, writeAddress, 1, &dataCheck);
3 if (dataCheck != data)
4     return -2;

```

3.9.2 Inicializace

Přepočtení výstupních hodnot sensorového pole na stupně Celsia vyžaduje kalibrační konstanty, které jsou uloženy v EEPROM paměti modulu a při startu programu musí být přečteny a uloženy do paměti mikrokontroléru. Na konci inicializace je nastavena požadovaná snímková frekvence.

```

1 uint16_t eeprom[832];
2 paramsMLX90640 mlxParameters;
3
4 MLX90640_DumpEE(MLX90640_I2C_ADDRESS, eeprom);
5 MLX90640_ExtractParameters(eeprom, &mlxParameters);
6 MLX90640_SetRefreshRate(MLX90640_I2C_ADDRESS, MLX_8_FPS);

```

3.9.3 Čtení snímků

Zda-li je připraven nový snímek k přečtení je zjištěno voláním funkce *IsSubpageReady*, která přečte status registr modulu a vrátí příznak signalizující nová data.

```

1 static uint8_t IsSubpageReady()
2 {
3     uint16_t statusRegister;
4     MLX90640_I2CRead(MLX90640_address, MLX90640_STATUS_REG, 1, &statusRegister);
5     return MLX90640_GET_DATA_READY(statusRegister);
6 }

```

Čtení dat probíhá ve dvou krocích a při každém je přečtena polovina pixelů snímku. Pixely jsou uspořádány buď do šachovnicového vzoru (výchozí nastavení), nebo vzoru proložených řádků. Přečtená data funkcí **MLX90640_GetFrameData** nejsou kalibrována a nelze je přímo použít pro určení přesné teploty měřeného tělesa, místo toho musí být přepočtena funkcí **MLX90640_CalculateTo** s využitím kalibračních konstant, které byly při inicializaci načteny z EEPROM paměti.

Dále je specifikována emisivita měřeného tělesa *emissivity* a odražená okolní teplota *tr*. Pokud nemá měřené těleso emisivitu 1, může odrážet okolní infračervené záření do snímače. Při výpočtu je tato odražená teplota kompenzována a měla by být její velikost do funkce zadána. Univerzálně lze zadat okolní teplotu, která může být zjištěna změřením teploty senzoru funkcí

MLX90640_GetTa, ta je však zvýšena o tepelné ztráty elektrickým proudem činnosti obvodů a musí být v závislosti na uložení modulu přepočtena. Pro umístění modulu ve volném vzduchu je výrobcem stanovena hodnota -8 °C [29], pokud by ale modul byl například uzavřen v krabici, musí být zjištěna konkrétní hodnota pro daný případ.

```

1 uint16_t rawFrame[834];
2 MLX90640_GetFrameData(MLX90640_I2C_ADDRESS, rawFrame);
3
4 float Ta = MLX90640_GetTa(rawFrame, &mlxParameters);
5 float tr = Ta - TA_SHIFT;
6 float emissivity = 0.95;
7 MLX90640_CalculateTo(rawFrame, &mlxParameters, emissivity, tr, mlxFrame);

```

Přepočet hodnot sensorového pole na kalibrované teploty ve stupních Celsia je náročný proces, který je složen z velkého množství floatových operací. Je proto doporučeno volit mikrokontrolér s hardwarovou implementací floating-point výpočetní jednotky. Přepočet dále klade nároky na RAM paměť mikrokontroléru, ve které musí být uloženy kalibrační konstanty sensorového pole, které se převážně vztahují zvláště k jednotlivým pixelům. Délka kalibračních hodnot je 4732 B, bufferu dat senzoru 1668 B a pole výsledných teplot 3072 B, tedy jen staticky alokovaná paměť dosahuje délky téměř 10 kB. Možnou úpravou by bylo přepočet implementovat v aplikaci pro zpracování dat na zařízení s vyšším výpočetním výkonem a mikrokontrolér použít pouze pro čtení sensorových dat modulu.

Zpracované teploty snímku jsou rovnoměrně fragmentovány do 3 paketů o stejných velikostech, jejichž struktura je vidět v tabulce 14.

Tabulka 14: Struktura aplikačního protokolu pro přenos snímků infračerveného kamerového modulu MLX90640

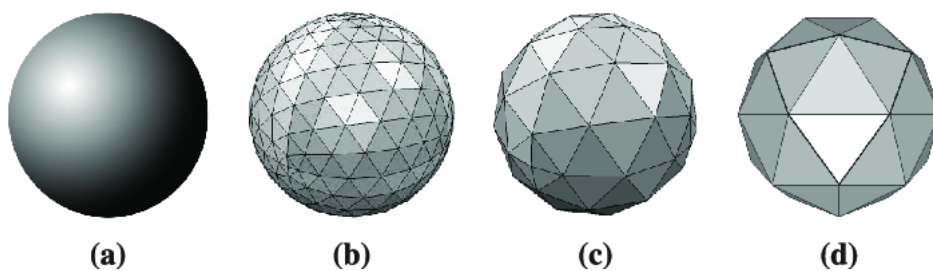
offset	data
0	typ paketu = 5 (1 B)
1	pořadí snímku (4 B)
5	pořadí fragmentu (1B)
6	data snímku (1024 B)

4 OpenTK

OpenTK je C# wrapper původních OpenGL knihoven v jazyce C. Umožňuje tvorbu 2D a 3D grafických aplikací s akcelеровaným výpočtem na grafické kartě, díky čemuž dosahuje vysokého paralelního výpočetního výkonu. Kromě tvorby grafických aplikací v samostatném okně, je součástí OpenTK balíček GLControl, který přináší grafický modul pro vykreslování do WinForms frameworku [30]. Oba softwarové balíčky byly v této práci použity pro vytvoření grafické aplikace pro vykreslení proudu snímků ve viditelném a infračerveném spektru v reálném čase, zvýšení kontrastu termálního snímku pomocí palety barev, zobrazení teplotních extrémů scény a prolínání termálního snímku se snímkem ve viditelném spektru.

4.1 Principy vykreslování

Moderní počítačová 3D grafika aproximuje všechny libovolné 3D objekty jedním typem elementárního tvaru, který musí být jednoduše matematicky definovatelný, rychle číselově zpracovatelný, jednoznačný a jeho spojením s dalším takovým tvarem nesmí vznikat nespojitosti. Elementární tvar, který splňuje tyto podmínky, je **trojúhelník** a stal se v této oblasti grafiky standardem. Vyšší počet obsažených trojúhelníků v objektu vede na přesnější aproximaci požadovaného tvaru, ale roste s ním i počet potřebných výpočetních operací. OpenGL interně pracuje pouze ve 3D režimu, rovnocenně však umožňuje vykreslovat 2D grafiku. Nastavením ortografické projekce a vykreslováním objektů do jedné roviny je ztracena informace o hloubce objektu a výsledek se jeví jako dvojrozměrný obraz.

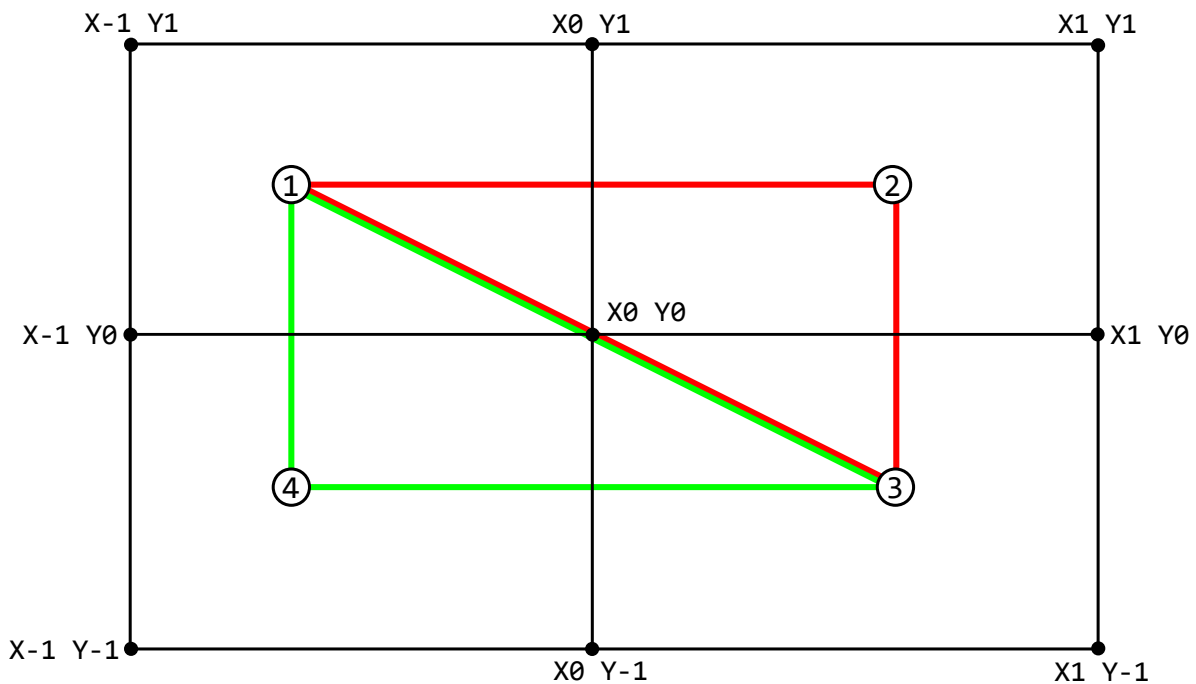


Obrázek 25: Aproximace koule s různým počtem trojúhelníků, převzato z [31]

Zbytek podkapitoly je věnován úvodu do 2D OpenGL na ukázkovém příkladu vykreslení obdélníku s texturou, čehož bude v kapitole 5.5.3 dále využito k vykreslení snímků kamerových modulů. Aby bylo zřejmé, co je přesně myšleno, bude nadále termín **vrchol** použit v souvislosti teoretického popisu geometrických útvarů a termín **vertex** v souvislosti praktické reprezentace vrcholu v OpenGL knihovnách.

Jak je vidět na obrázku 26, obdélník tvoří 4 vrcholy, které definují 2 trojúhelníky. OpenGL očekává, že všechny vertexy leží uvnitř souřadného systému **Normalized Device Coordinates**

(NDC), který se rozprostírá na intervalu $-1,0$ až $1,0$ pro složky x a y a cokoliv ležící mimo tento interval nebude vykresleno. NDC je na obrázku 26 znázorněn černou mřížkou s 9 vyznačenými body.



Obrázek 26: NDC souřadný systém, ve kterém leží obdélník vytvořený ze dvou trojúhelníků

Metoda **GL.ViewPort** mapuje NDC do zvolené velikosti okna aplikace měřené v pixelech. Parametry *width* a *height* určují šířku a výšku okna, měřenou od počátečního bodu v levém dolním rohu, který specifikují parametry x a y . NDC musí být přizpůsobeny po každé změně velikosti okna a vhodným přepočtem *width* a *height* lze například pevně nastavit poměr stran nezávisle na poměru stran okna aplikace.

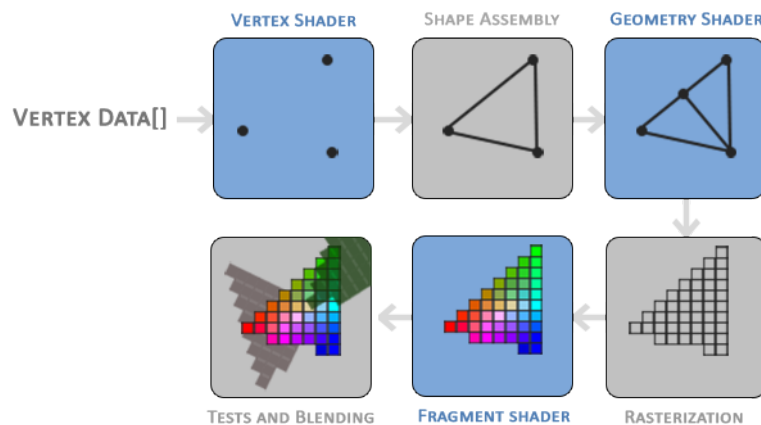
```
1 GL.Viewport(x, y, width, height);
```

4.1.1 Textura

Textury jsou v počítačové grafice nejčastěji použity k nastavení barvy jednotlivých pixelů vykreslovaného objektu, mohou ale nést další informace, například o výchylnkách hloubky určitých bodů objektu v podobě výškové mapy (heightmap). Umožňují vnést objektu mnoho detailů, pro které by jinak musel být definován velký počet vertexů. Textura je v paměti grafické karty nejčastěji realizována jako dvourozměrné pole, existují ale i jednorozměrné nebo trojrozměrné varianty. Textury jsou do paměti grafické karty nejčastěji nahrávány z rastrových obrázků různých formátů.

4.1.2 Shader

Program, který je určen pro spuštění na grafické kartě, je nazýván **shader**. Grafická karta je optimalizována pro paralelní běh velkého množství instancí různých typů shaderů v pipeline struktuře, která je zobrazena na obrázku 27. Modře označené stupně představují shader programy a jsou uživatelsky definovatelné, šedivě označené stupně jsou pevně dané. **Vertex shader** umísťuje do prostoru vrcholy tělesa a provádí matematické operace, jako je posun, rotace nebo změna velikosti. Je to počáteční bod v pipeline a musí být definován. Ve stupni *shape assembly* jsou vrcholy seskupeny do trojúhelníků. *Geometry shader* je volitelný shader, kterým je možné sestavenou geometrii z trojúhelníků modifikovat. *Rasterization* je stupeň, kde jsou trojúhelníky rozděleny na fragmenty o velikosti jednoho pixelu. Druhým povinným shaderem je **fragment shader**, který fragmentům přiřadí barvu. V posledním stupni *tests and blending* je výstup zařazen do zbytku scény.



Obrázek 27: Pipeline struktura OpenGL, převzato z [32]

Shadery jsou programovány v jazyce *GLSL*, který vychází z jazyka *C* a je mu velmi podobný. Instance vertex shaderu jsou paralelně spuštěny pro každý vertex na jednotlivých jádrech grafické karty, to samé platí i pro fragmenty (tj. pixely) u fragment shaderu. Výpočetní jednotky jader grafických karet jsou optimalizovány pro početní operace s čísly s plovoucí desetinnou čárkou (**float**), a proto je tento datový typ v shaderech nejčastěji zastoupen. Floaty jsou dále uspořádávány do **matic** a pro práci s vektory slouží datové typy jednorozměrných matic **vec2**, **vec3** a **vec4**. Přístupování ke složkám vektoru je ukázáno v následující ukázce kódu. Všechny 3 způsoby jsou ekvivalentní a liší se v souvislosti vektoru, kde je použit. Přístup ke složkám přes **xyzw** je v souvislosti souřadnic objektu, **rgba** v souvislosti barvy a **stpq** v souvislosti souřadnic textury. Kombinací jednotlivých složek lze získat vektory nižších velikostí, či přeuspořádat složky.[33]

```

1  vec4  vector4D;
2
3  vector4D.xyzw;
4  vector4D.rgba;
5  vector4D.stpq;
6
7  vec2  vector2D = vector4D.xy;    // preusporadani slozek
8  vector2D = vector4D.ga;        // xyzw, rgba a stpq nelze kombinovat, takže
9  vec3  vector3D = vector4D.qqs;  // napriklad vector2D.xg není validní

```

4.1.3 Vertex shader

Následující ukázka kódu představuje jednoduchý **vertex shader**, který z bufferu v paměti grafické karty načte pozice vertexů do proměnné *vertexPosition* a tuto proměnnou přiřadí do vstavené proměnné shaderu *gl_Position*, která je předána následujícímu stupni v pipeline struktuře. Jelikož proměnná očekává datový typ *vec4*, funkce **vec4** rozšíří dvousložkový vektor o zbylé 2 složky, které jsou předány v dalších parametrech. Složka $z = 0$ složka $w = 1$, protože ve 2D vykreslování nejsou použity a mají konstantní hodnoty. Shader dále načte hranice textury, která bude na obdélník aplikována, a předá je fragment shaderu skrz výstupní proměnnou *fragmentTexturePosition*.

Může být překvapivé, že vstupní datový typ proměnné *vertexPosition* není pole 2D vektorů, ale pouze 1 prvek. Je to z toho důvodu, jak již bylo zmíněno, protože pro každý vertex je paralelně spuštěna samostatná instance vertex shaderu.

```

1  #version 330 core
2  layout(location = 0) in vec2 vertexPosition;
3  layout(location = 1) in vec2 texturePosition;
4  out vec2 fragmentTexturePosition;
5
6  void main(void)
7  {
8      fragmentTexturePosition = texturePosition;
9      gl_Position = vec4(vertexPosition, 0.0, 1.0);
10 }

```

4.1.4 Fragment shader

Geometrie obdélníku je nyní připravena v pipeline struktuře a zbývá už jen definovat **fragment shader**, který určí výslednou barvu každého pixelu nacházejícího se uvnitř obdélníku, resp. dvou trojúhelníků. Informaci o barvách všech pixelů nese textura v uniform proměnné *texture0*. **Uniform** proměnné fungují podobně jako **statické** proměnné v C# třídě, tedy že jejich hodnota

je sdílena mezi **všemi instancemi** shaderů a je jednotná. Podrobněji jsou možnosti nahrání globálních dat do shaderu popsány v podkapitole 4.3.3.

Z textury *texture0* je funkcí **texture** vzorkována barva pro aktuálně zpracovávaný pixel v bodě textury určeném proměnnou *fragmentTexturePosition*. Jelikož jsou výstupní proměnné vektorů z vertex shaderu do fragment shaderu ve výchozím nastavení interpolovány, z původních bodů hranic textury se stalo pole bodů pokrývající celou texturu v ohraničené oblasti mezemi. Pozice $X_0;Y_0$ se nachází v levém dolním vrcholu textury a $X_1;Y_1$ v pravém horním vrcholu.

Navzorkovaná barva textury je uložena do výstupní proměnné *outputColor* typu *vec4*, jejíž složky reprezentují RGBA kanály a jsou normalizovány v rozsahu 0 až 1. Tato proměnná není přímo součástí fragment shaderu, jako to bylo v případě vertex shaderu u proměnné *gl_Position*, musí být ale samostatně deklarována. Důležité je dodržet, aby byla definována na lokaci atributu 0, kde ji očekává další stupeň v pipeline. Na jejím názvu nezáleží.[34]

```
1 #version 330
2 layout(location = 0) out vec4 outputColor;
3 in vec2 fragmentTexturePosition;
4 uniform sampler2D texture0;
5
6 void main()
7 {
8     outputColor = texture(texture, fragmentTexturePosition);
9 }
```

4.1.5 Kompilace shaderů a vytvoření programu grafické karty

Aby mohl být vertex shader a fragment shader spuštěn, musí být zkompileován. Jelikož je kompilace závislá na více faktorech, jako je typ grafické karty, její výrobce nebo systémový ovladač, shadery jsou kompilovány za běhu aplikace. Výstupem je objekt programu OpenGL, přes který jsou oba shadery spuštěny v pipeline grafické karty.

Následující ukázka kódu kompiluje zdrojový kód shaderů v textové podobě, který pak linkuje dohromady a nahrává do objektu programu. Zbylé mezivýsledky po kompilaci shaderů jsou zbytečné, jelikož jejich kopie byly nahrány do objektu programu, a je vhodné uvolnit jimi obsazenou paměť.

```
1 //kompilace shaderu
2 int vertexShaderHandle = GL.CreateShader(ShaderType.VertexShader);
3 GL.ShaderSource(vertexShaderHandle, vertexSourceCodeText);
4 GL.CompileShader(vertexShaderHandle);
5 int fragmentShaderHandle = GL.CreateShader(ShaderType.FragmentShader);
6 GL.ShaderSource(fragmentShaderHandle, fragmentSourceCodeText);
7 GL.CompileShader(fragmentShaderHandle);
8
9 //linkovani zkompileovanych shaderu do objektu programu
10 int programHandle = GL.CreateProgram();
11 GL.AttachShader(programHandle, vertexShaderHandle);
12 GL.AttachShader(programHandle, fragmentShaderHandle);
13 GL.LinkProgram(programHandle);
14
15 //uvolneni pameti
16 GL.DetachShader(programHandle, vertexShaderHandle);
17 GL.DetachShader(programHandle, fragmentShaderHandle);
18 GL.DeleteShader(vertexShaderHandle);
19 GL.DeleteShader(fragmentShaderHandle);
```

4.1.6 Nahrání souřadnic do vertex shaderu

Oba potřebné shadery jsou nyní připraveny, nicméně do nich zatím nebyla nahrána žádná vstupní data. V této podkapitole je popsáno, jakým způsobem jsou do vertex shaderu nahrány souřadnice vertexů, souřadnice mezí textury a samotná textura ze C# aplikace.

V prvním kroku je připraveno pole floatů se souřadnicemi vertexů a mezních bodů textury, které budou na sebe namapovány. Pro namapování celé textury přes celý obdélník musí být souřadnice vrcholů obdélníku a mezních bodů textury shodné, tj. v pravém horním vertexu obdélníku musí být namapován pravý horní mezní vrchol textury atd. pro zbylé vertexy.

```
1 private readonly float[] VERTICIES =
2 {
3 // souradnice vertexu    souradnice vrcholu textury
4     1.0f,  0.5f,          1.0f, 1.0f, // pravy horni vrchol
5     1.0f, -0.5f,         1.0f, 0.0f, // pravy dolni vrchol
6    -1.0f, -0.5f,         0.0f, 0.0f, // levy dolni vrchol
7    -1.0f,  0.5f,         0.0f, 1.0f, // levy horni vrchol
8     1.0f,  0.5f,          1.0f, 1.0f, // pravy horni vrchol
9    -1.0f, -0.5f,         0.0f, 0.0f, // levy dolni vrchol
10 };
```

Přestože obdélník tvoří 4 vrcholy, je v poli nadefinováno 6 vertexů. Je tak učiněno z toho důvodu, protože trojúhelníky tvořící obdélník mají společné vrcholy 1 a 3 (viz obrázek 26) a jelikož se jedná o nejjednodušší způsob vykreslení trojúhelníku, OpenGL není předána žádná informace,

které vertexy trojúhelníky sdílí, a proto musí být nadefinovány duplicitně. Opakovat vertexy ale není žádoucí, jelikož nenesou užitečnou informaci a zbytečně využívají paměť a výpočetní čas pro zpracování. U obdélníku jde o 2 zbytečné vertexy navíc a jelikož se předpokládá vykreslení nízkého počtu obdélníků v celé aplikaci, negativní dopady na výkon aplikace jsou zanedbatelné. Pokud bychom ale předpokládali místo obdélníku kvádr, který má 8 vrcholů, muselo by být nadefinováno **36** vrcholů a s rostoucí geometrickou složitostí vykreslovaného objektu tato neefektivita roste. Zlepšit situace může **Element Buffer Object (EBO)**, přes který lze definovat, které vertexy jsou společné, nicméně v této práci nebude použit.

OpenGL je **rozsáhlý stavový automat** a prostřednictvím vybraných metod jsou jeho stavy přepínány, což ovlivňuje následné volání jiných metod. Nejdříve je pro objekt vertexů **Vertex Buffer Object (VBO)** alokována paměť grafické karty. Metoda **GL.BindBuffer** nově vytvořený VBO globálně zvolí a všechna následující volání metod, které očekávají VBO, budou pracovat právě s tímto vybraným VBO.[35]

Metoda **BufferData** do VBO nahraje souřadnice vertexů a textury. Parametr *BufferUsageHint* určuje, jak často lze očekávat, že budou data v bufferu měněna. *StaticDraw* nepředpokládá změnu dat, *DynamicDraw* občasnou změnu a *StreamDraw* změnu s každým vykresleným snímkem. Jedná se ovšem pouze o optimalizační náznak pro OpenGL a i s možností *StaticDraw* není problém data bufferu opakovaně přehrávat.

```
1 int vertexBufferObject = GL.GenBuffer();
2 GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBufferObject);
3 GL.BufferData(BufferTarget.ArrayBuffer, VERTICES.LENGTH * sizeof(float), ...
   VERTICES, BufferUsageHint.StaticDraw);
```

Struktura nahraných dat v paměti je stejná, jako zdrojové pole dat, a musí být pro OpenGL specifikováno, jakým způsobem data převzít a do kterých proměnných vertex shaderu je nahrát. K tomu je určen **VertexArrayObject (VAO)**, který je vytvořen a přiřazen obdobně jako VBO.

Metoda **GL.VertexAttribPointer** nastaví VAO pro nahrání dat do vybrané vstupní proměnné vertex shaderu, v tomto případě do proměnné *vertexPosition* o počtu 2 složek (*vec2*) s datovým typem každé složky float. Poslední 2 parametry určují strukturu vstupního pole *VERTICES*, kde jsou souřadnice vertexů namíchány se souřadnicemi textury. Souřadnice vertexů jsou opakovány s krokem 4 floatů s nulovým offsetem. Lokaci vstupní proměnné *vertexPosition* lze zadat buď staticky ze zdrojového kódu vertex shaderu, nebo dynamicky metodou *GL.GetAttribLocation*.

Volání metody *GL.VertexAttribPointer* je vztaženo k aktuálně vybranému VAO a VBO a pro každou vstupní proměnnou vertex shaderu je opakováno. Vstupní proměnné jsou ve výchozím stavu vypnuty a musí být povoleny metodou **GL.EnableVertexAttribArray**, jinak nebudou naplněny daty.


```

1 int vertexArrayObject = GL.GenVertexArray();
2 GL.BindVertexArray(vertexArrayObject);
3
4 int vertPosLocation = GL.GetAttribLocation(programHandle, "vertexPosition");
5 // vertPosLocation = 0 viz zdrojovy kod vertex shaderu:
6 // layout(location = 0) in vec2 vertexPosition;
7
8 GL.VertexAttribPointer(vertexLocation, 2, VertexAttribPointer.Float, ...
   false, 4*sizeof(float), 0);
9 GL.EnableVertexAttribArray(vertexLocation);

```

Stejný proces je opakován pro vstupní proměnnou vertex shaderu *texturePosition*. Souřadnice textur jsou v poli *VERTICES* opakovány taktéž s krokem 4 floatů, nachází se ale před nimi souřadnice vertexů, takže je potřeba specifikovat ofset 2 floaty.

```

1 int texturePosLocation = shader.GetAttribLocation("texturePosition");
2 GL.VertexAttribPointer(texturePosLocation, 2, VertexAttribPointer.Float, ...
   false, 4*sizeof(float), 2*sizeof(float));
3 GL.EnableVertexAttribArray(texturePosLocation);

```

4.1.7 Nahrání textury do fragment shaderu

Paměť textury je alokována podobnými způsoby, jako byly alokovány VBO nebo VAO. Pokud fragment shader používá více textur současně, musí být pro každou vybrán vlastní slot v parametru metody **GL.ActiveTexture**.

Nahrání obrazových dat textury zajišťuje metoda **GL.Texture2D**. Parametr *PixelInternalFormat* definuje formát, v jakém budou data pixelů interně uložena po zpracování. Parametry *PixelFormat* a *PixelType* na druhou stranu udávají, v jakém formátu jsou data pixelů v proměnné *imageData* do metody předávána. Pokud se vnitřní formát od vnějšího liší, proběhne na pozadí převod, je tedy efektivní používat stejné formáty.

```

1 int textureHandle = GL.GenTexture();
2 GL.ActiveTexture(TextureUnit.Texture0);
3 GL.BindTexture(TextureTarget.Texture2D, Handle);
4
5 GL.Texture2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgb, width, ...
   height, 0, PixelFormat.Rgb, PixelType.UnsignedByte, imageData);

```

Parametr *PixelInternalFormat* nabízí několik základních možností: *RGBA*, *RGB*, *RG* a *R*. Každá možnost označuje, kolik barevných kanálů připadá na jeden pixel. Možnosti pak dále modifikují 2 přípony. První číselná přípona udává barevnou hloubku složek pixelu v bitech, po které následuje přípona s písmenem, která modifikuje datový typ, ve kterém budou data pixelu

uložena v paměti grafické karty. Modifikace datového typu zahrnují tyto možnosti [36]:

- **bez přípony datového typu:** Výchozí možnost, každý pixel textury je interně uložen celočíselně bez znaménka, avšak z pohledu shaderu, který k hodnotě přistupuje, se datový typ bude jevit jako **normalizovaný float**. To znamená, že celočíselná hodnota původního rozsahu 0 až $(2^n - 1)$, kde n je barevná hloubka v bitech, bude namapována na desetinné číslo v rozsahu 0 až 1.
- **_SNORM:** Normalizovaný celočíselný typ se znaménkem
- **f:** Desetinné číslo ve float formátu
- **ui:** Celočíselná hodnota bez znaménka bez normalizace
- **i:** Celočíselná hodnota se znaménkem bez normalizace

Ve zdrojovém kódu ukázky je pro *PixelFormat* zvolena možnost *RGB*, jde o 3 barevné složky s 8bitovou barevnou hloubkou, které budou normalizovány ve fragment shaderu na float. Tento formát je vhodný pro obrázky bez alfa kanálu (průhlednosti), například po dekódování JPEG obrázku.

Nakonec je slot textury **TextureUnit**, do kterého byla textura v předchozí ukázce kódu nahrána, uložen do proměnné *texture0* ve fragment shaderu.

```
1 int uniformLocation = GL.GetUniformLocation(programHandle, "texture0");
2 GL.ProgramUniform1(programHandle, uniformLocation, 0);
3                               //TextureUnit.Texture0 = 0
```

Znovu použitím metody *GL.TexImage2D* je možné data textury aktualizovat, dojde však k dealokaci původního prostoru v paměti grafické karty a alokaci nového, což je neefektivní. Pokud velikost dat nové textury nepřevyšuje velikost předchozí textury, pro kterou byl prostor v paměti alokován, paměť nemusí být znovu alokována a je možné použít odlehčenou metodu **GL.TexSubImage2D**.

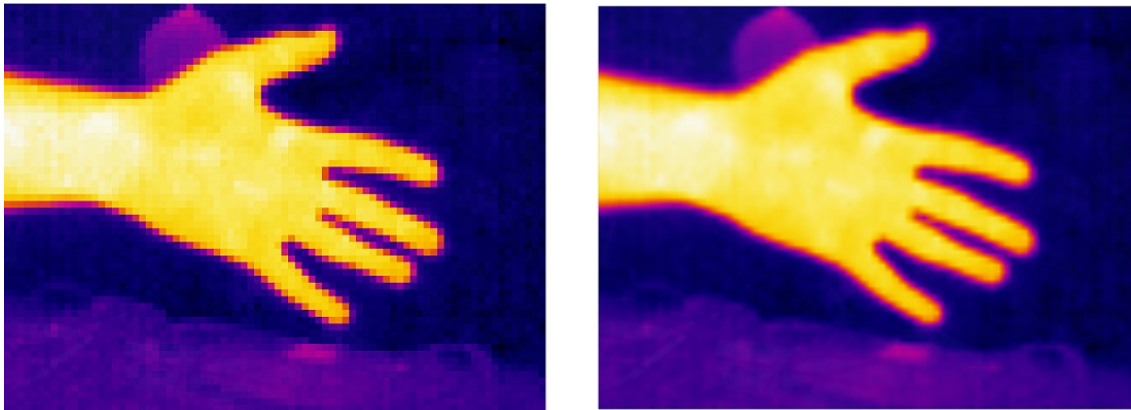
```
1 GL.TexSubImage2D(TextureTarget.Texture2D, 0, xOffset, yOffset, width, height, ...
   PixelFormat.Rgb, PixelType.UnsignedByte, newImageData);
```

Běžně může dojít k situaci, že rozlišení textury v pixelech neodpovídá v poměru 1 : 1 rozlišení při vykreslování a textura musí být zvětšena či zmenšena. Existují 2 základní možnosti:

- **Nearest:** Výsledná barva pixelu bude vybrána z nejbližšího pixelu textury, což vede na výrazně zvětšené hranaté pixely.
- **Linear:** Výsledná barva pixelu bude lineárně interpolována mezi několika nejbližšími pixely textury. Ve výsledném obrazu nebudou tolik patrné jednotlivé zvětšené pixely, ale hrany budou rozmazány.

Na obrázku 28 je vidět efekt obou možností a v následující ukázce kódu je uveden příklad nastavení lineárního režimu při zvětšování i zmenšování pro aktuálně vybranou texturu metodou `GL.BindTexture`.

```
1 GL TexParameter(TextureTarget.Texture2D, ...
   TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
2 GL TexParameter(TextureTarget.Texture2D, ...
   TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);
```



Obrázek 28: Účinek jednotlivých filtrů textury při zvětšování, pro levý obrázek je použito nastavení *nearest* a pro pravý *linear*

4.1.8 Vykreslení obdélníku s texturou

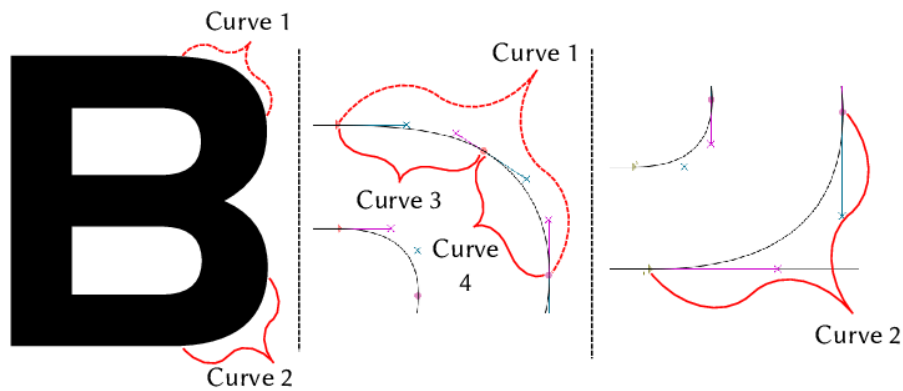
Data vertexů a textury jsou nahrána v paměti grafické karty, VAO je nakonfigurován pro nahrání vstupních dat do vertex shaderu, kompilací vertex a fragment shaderu byl vytvořen program grafické karty, a vše je tedy připraveno pro finální vykreslení obdélníku s aplikovanou texturou. Metoda `GL.DrawArrays` vykreslí 2 trojúhelníky, definované 6 vertexy, pomocí vybraného programu grafické karty a vybraného VAO.

```
1 GL.UseProgram(programHandle);
2 GL.BindVertexArray(vertexArrayObject);
3 GL.DrawArrays(PrimitiveType.Triangles, 0, 6);
```

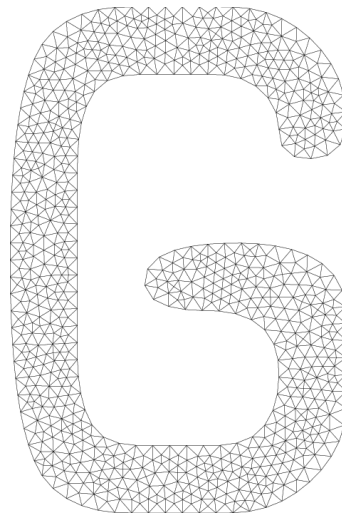
4.2 Vykreslení fontu

Jeden ze standardních formátů fontů v počítačové technice je **TrueType**, který definuje znaky matematickým popisem křivek (viz obrázek 29). OpenGL vyniká ve vykreslování elementárních tvarů, do kterých však křivky nespádají. Existují dva hlavní způsoby, jak font v OpenGL vykreslit:

1. Aproximovat znaky fontu trojúhelníky (viz obrázek 30)
2. Předvykreslit font do 2D textury a tu poté vykreslit na obdélník



Obrázek 29: Ukázka definice znaku fontu přes matematicky definované křivky, převzato z [37]



Obrázek 30: Aproximace znaku fontu trojúhelníky, převzato z [38]

4.2.1 Bitmapová textura fontu

V rámci této práce byl implementován druhý způsob. Texturu lze vytvořit buď dynamicky za běhu programu s využitím speciální vykreslovací knihovny (například FreeType [39]), nebo texturu staticky připravit do souboru a ten při startu aplikace načíst. Pro jednoduchost byl volen druhý způsob a textura byla vytvořena v programu Inkscape s fontem Consolas o výšce 128 pixelů. Nevýhodou této metody je, že font musí být neproporcionální, jinak by vykreslovací algoritmus byl značně zkomplikován. Část textury fontu je vidět na obrázku 31, textura v plné velikosti pokrývá všechny znaky základní ASCII tabulky a z jejího rozšíření znak pro stupeň.

() * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F

Obrázek 31: Ukázka části textury fontu

Výstupní obrázek 31 v plné velikosti ze softwaru Inkscape ve formátu PNG obsahoval barevnou informaci pouze v alfa kanálu. Obrázek byl skriptem v nástroji Octave dekodován, nulové RGB složky byly odstraněny a zbylý alfa kanál byl uložen do souboru v binárním formátu celočíselných 8bitových hodnot bez znaménka.

Princip vykreslení jednoho znaku je stejný, jako vykreslení obdélníku s texturou v předchozí podkapitole. Na každý znak spadá jeden obdélník, jehož souřadnice textury jsou posunuty na pozici odpovídajícího znaku v textuře. Pokud jsou znaky v textuře uspořádány ve stejném pořadí jako v ASCII tabulce a font je neproporcionální, ofset textury je vypočítán vztahem v rovnici 10.

$$x_{offset} = ASCII \cdot WIDTH, \quad (10)$$

kde x_{offset} je ofset textury v ose x , $ASCII$ je pořadí znaku v ASCII tabulce a $WIDTH$ je konstantní šířka jednoho znaku. Velikost písma je většinou udávána ve výšce a šířka je dopočítána dle specifického poměru délky stran daného fontu. Například poměr šířky ku výšce fontu Consolas je 0,5498, a tedy při výšce 128 pixelů je šířka $WIDTH = 128 \cdot 0,5498 = 70,374$ px.

4.2.2 Fragment shader

Jednokanálová textura fontu je do fragment shaderu načtena jako normalizovaný float. V místě, kde je nenulová průhlednost původního alfa kanálu textury, se nachází znak a je pro fragment vybrána bílá barva s původní hodnotou průhlednosti (hrany písma byly při rasterizaci vyhlazeny a ze syté barvy přechází do průhledné). Naopak pro fragment, kde je původní alfa kanál nulový, je vykresleno černé pozadí s 30% průhledností, aby byla zlepšena viditelnost na libovolném barevném podkladu.

```
1 float alpha = texture(texture0, texCoord).r;
2
3 if (alpha > 0.0)
4     outputColor = vec4(1, 1, 1, alpha);
5 else
6     outputColor = vec4(0, 0, 0, 0.3);
```

4.3 Způsoby nahrávání globálních dat do shaderu

V podkapitole 4.1.7 byla v shaderu použita proměnná typu **uniform** pro uložení globální hodnoty, která byla jednotná ve všech instancích shaderu. Uniform proměnná může být i pole vybraného datového typu pro nahrání většího množství dat, má to však svá omezení. OpenGL definuje minimální **počet složek** uniform proměnné pro každý typ shaderu, pro který je zaručena úspěšná alokace paměti. Složkou je myšlen jeden skalární datový typ: float, integer nebo boolean. U slo-

žených datových typů se počet složek počítá, například *vec3* má 3 složky. Minimální počet pro OpenGL 3.0+ je 1024 složek pro každý typ shaderu, ve skutečnosti tento limit bývá vyšší v závislosti na konkrétním hardwaru a softwaru počítače. Například na grafické kartě AMD Radeon RX 6650 XT a operačním systémem Windows 10 (dále jen **systém použitý v práci**) byl limit 8000, který lze zjistit následujícími příkazy:

```
1 GL.GetInteger(GetPName.MaxVertexUniformComponents); //8000
2 GL.GetInteger(GetPName.MaxFragmentUniformComponents); //8000
```

Využití samotných uniform proměnných je tedy poměrně omezené a pro nahrání většího množství dat existují další možnosti.

4.3.1 Textura

Textura může být použita i pro neobrazová data. Limit velikosti 1D a 2D textur je stanoven parametrem *MaxVertexUniformComponents*, který udává maximální rozměr každé dimenze textury. Na systému použitém v práci byl tento limit 16384.

```
1 GL.GetInteger(GetPName.MaxVertexUniformComponents); //16384
```

Data textury jsou uložena v globální paměti grafické karty, ke které musí být přistupováno, a nahodilé čtecí operace nemusí dosahovat neoptimálnějších výsledků.

4.3.2 Uniform Buffer Object (UBO)

Pro UBO je garantována minimální velikost 16 kB, konkrétní velikost bývá opět vyšší a pro systém použitý v práci byl zjištěn limit 4 MB. Další rozdíl oproti textuře je, že data UBO jsou zkopírována do lokální paměti shaderu a přístup k nim je efektivnější.[40]

```
1 GL.GetInteger(GetPName.MaxUniformBlockSize); //4194304
```

Vytvoření, alokace paměti a nahrání dat do UBO je obdobné, jako vytváření bufferů v předchozích případech.

```
1 byte[] uboData = {...};
2 uniformBufferObject = GL.GenBuffer();
3 GL.BindBuffer(BufferTarget.UniformBuffer, uniformBufferObject);
4 GL.BufferData(BufferTarget.UniformBuffer, uboData.Count, uboData, ...
   BufferUsageHint.StaticDraw);
```

V shaderu je definován **blok rozhraní** `dataBlock`, který sdružuje několik proměnných. Existují různá rozložení paměti, která určují, jakým způsobem budou data v paměti grafické karty uložena a zarovnána. Práce se ale bude zaměřovat pouze na rozložení `std140`, které zarovná data odlišně pro 3 případy.[35]

V prvním případě (viz následující ukázka kódu), pokud jsou v bloku rozhraní skalární datové typy o šířce slova N , budou v paměti uloženy s rozestupem násobku N . To znamená, že zdrojové pole dat pro nahrání bufferu metodou `GL.BufferData` musí být strukturováno dle tabulky 15.

```
1 layout (std140) uniform dataBlock
2 {
3     float data1;
4     float data2;
5     int data3;
6 };
```

Tabulka 15: Struktura zdrojového pole pro nahrání dat do UBO s rozložením `std140` v případě skalárních datových typů

ofset zdrojového pole dat	proměnná
0	float data1
4	float data2
8	int data3

Ve druhém případě (viz následující ukázka kódu), pokud jsou v bloku rozhraní vektory, jejichž základní složka je o šířce slova N , budou v paměti uloženy s rozestupem násobku $2N$ nebo $4N$. To znamená, že vektory `vec2` a `vec4` budou zarovnány bez toho, aniž by vzniklo nevyužitě místo v paměti. Naopak u `vec3` vznikne prázdné místo o velikosti $1N$. V tabulce 16 je vidět potřebné rozložení vstupního pole pro nahrání dat do bufferu.

```
1 layout (std140) uniform dataBlock
2 {
3     vec2 data1;
4     vec3 data2;
5     vec4 data3;
6     vec4 data4;
7 };
```

Tabulka 16: Struktura zdrojového pole pro nahrání dat do UBO s rozložením `std140` v případě vektorových datových typů

ofset zdrojového pole dat	proměnná
0	vec2 data1
2	vec3 data2
6	vec4 data3
10	vec4 data4

V posledním případě (viz následující ukázka kódu), pokud je v bloku rozhraní pole skalárních datových typů nebo vektorů, jsou všechny prvky zarovnány s rozestupem $4N$, jak může být vidět v tabulce 17. Způsobuje to komplikaci, protože u žádného pole, kromě *vec4*, nebude zarovnání zdrojového pole ze C# aplikace odpovídat cílovému zarovnání v paměti grafické karty a data zdrojového pole musí být přerovnána.

```

1 layout (std140) uniform dataBlock
2 {
3     vec2 data1[2];
4     float data2[2];
5 };

```

Tabulka 17: Struktura zdrojového pole pro nahrání dat do UBO s rozložením std140 v případě pole skalárních a vektorových datových typů

ofset zdrojového pole dat	proměnná
0	vec2 data1[0]
16	vec3 data1[1]
32	float data2[0]
48	float data2[1]

Po vytvoření a nahrání dat do UBO už zbývá jen buffer přiřadit k bloku rozhraní v shaderu následující ukázkou kódu. UBO může být v jeden okamžik přiřazen k několika shaderům zároveň a lze je za běhu přepínat, což může být výhodné pro přepínání mezi několika sadami dat.[40]

```

1 int blockId = GL.GetUniformBlockIndex(programHandle, "dataBlock");
2 GL.UniformBlockBinding(programHandle, blockId, 0);
3 GL.BindBufferBase(BufferRangeTarget.UniformBuffer, 0, uniformBufferObject);

```

4.3.3 Shader Storage Buffer Object (SSBO)

SSBO poskytuje alokaci největšího prostoru v paměti ze všech možností. OpenGL specifikace stanovuje minimální velikost 128 MB, není ale neobvyklé alokovat až celou paměť grafické karty. Data jsou uložena v globální paměti (podobně jako textury), do které je pomaleji přístupováno.[40]

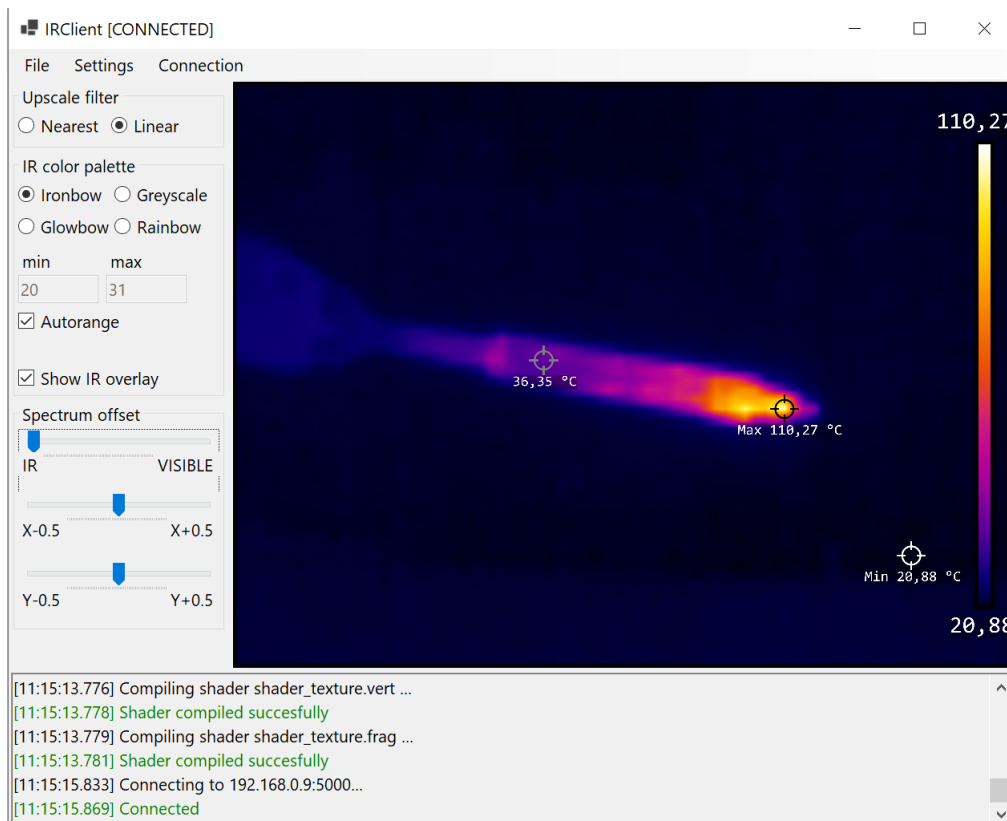
5 Aplikace pro zpracování dat

Pro příjem a zpracování snímků odeslaných z mikrokontroléru byla vytvořena počítačová aplikace v jazyce C# na platformě .NET 8.0 s Winforms frameworkem poskytujícím grafické rozhraní. K vykreslení a grafické úpravě snímků byl použit wrapper OpenGL knihoven OpenTK s balíčkem GLControl. K vývoji bylo použito vývojové prostředí Visual Studio 2022.

5.1 Uživatelské rozhraní

Obrazovka aplikace je rozdělena na 3 hlavní části viz obrázek 32. Levý panel obsahuje ovládací a informační prvky pro nastavování a sledování stavu aplikace, přičemž horní menu přepíná mezi několika sadami těchto prvků. Dolní panel zaujímá textové pole s výpisem vyvolaných událostí v čase a podrobnostmi o jejich výsledku. Uprostřed se nachází samotné okno pro vykreslování grafických dat snímků.

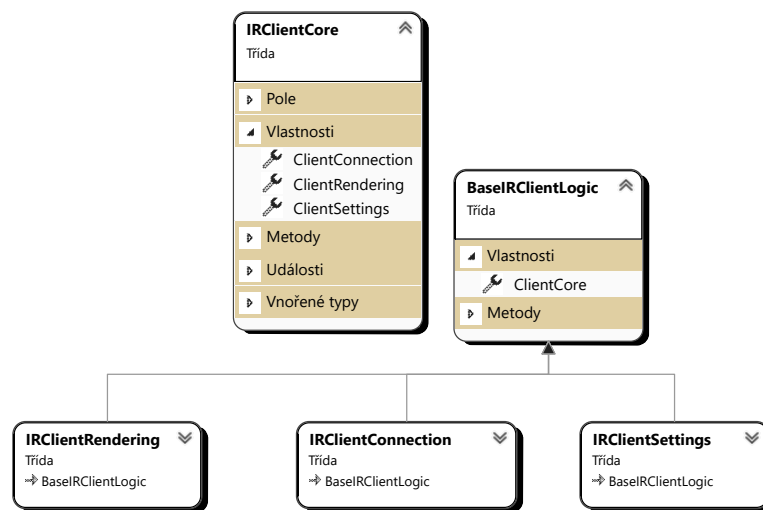
Při změně velikosti okna je udržován konstantní poměr stran vykreslovací plochy, který byl nastaven ve výchozí konfiguraci na 4:3, jelikož většina kamerových modulů použitých v této práci disponuje sensorovým polem s tímto poměrem stran.



Obrázek 32: Ukázka uživatelského rozhraní aplikace pro zpracování dat

5.2 Struktura aplikace

Aplikace implementuje OOP návrhové vzory a byla rozdělena do několika hlavních objektů, které vykonávají jednotlivé dílčí úkony. Po spuštění programu je vytvořena instance třídy **IRClientCore**, která vytvoří a iniciuje instance zbylých řídicích tříd, kterým předá svůj odkaz. Nové instance mohou vytvářet další instance, kterým podobně předají svůj odkaz, vzniká tím **stromová struktura** a z jakékoliv instance se lze zpátky odkázat na *IRClientCore* a tím pádem i na ostatní hlavní objekty. Následující podkapitoly popisují jednotlivé hlavní objekty, které jsou uvedeny v obrázku 33.

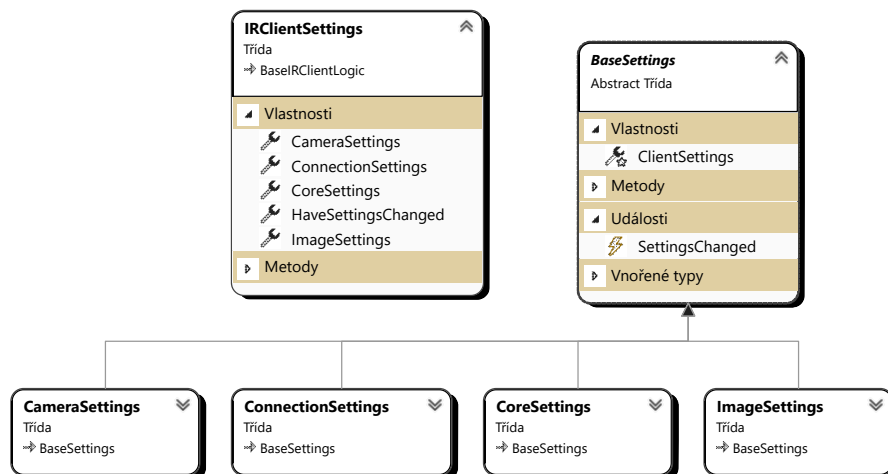


Obrázek 33: Přehled hlavních objektů aplikace

5.2.1 IRClientSettings

Aplikace byla navržena tak, aby umožňovala co největší konfigurovatelnost. Některá nastavení jsou uložena v konfiguračním souboru a některé parametry lze změnit pouze v kódu při kompilaci. Konfigurační parametry byly rozděleny do 3 tříd podle typu prvku, který konfiguruje. Přehled uspořádání konfiguračních tříd je vidět na obrázku 34.

- **CameraSettings**: Konstanty kamerových modulů. Rozlišení, počet fragmentů, délka fragmentů, datové ofsety v datagramu atd.
- **ConnectionSettings**: Konfigurace IP adresy a portu koncového síťového bodu
- **CoreSettings**: Konstanty jádra aplikace. Cesty ke zdrojovým kódům shaderů, dekodovaným texturám a paletám barev. Definice poměru stran vykreslovacího okna, velikosti dekodovaných textur, ASCII ofsetu textury fontu atd.
- **ImageSettings**: Parametry měnitelné pomocí ovládacích prvků aplikace za běhu. Výběr aplikované palety barev, prolínání a překryvu spekter snímků, výběr rozsahu mapování palety barev atd.



Obrázek 34: Přehled objektů s konfigurací aplikace

Změna nastavení vyvolá událost spadající pod patřičnou konfigurační třídu. Objekty aplikace se mohou přihlásit k odběru události a reagovat na změny konfigurace. Pro konfigurační parametry, které jsou uchovány v souboru, byla vytvořena třída pro ukládání a načítání konfiguračních dat ze souboru. Syntaxe konfiguračního souboru je uvedena v následující ukázce. Na začátku každého řádku je název parametru, jehož hodnotu odděluje znak rovná se.

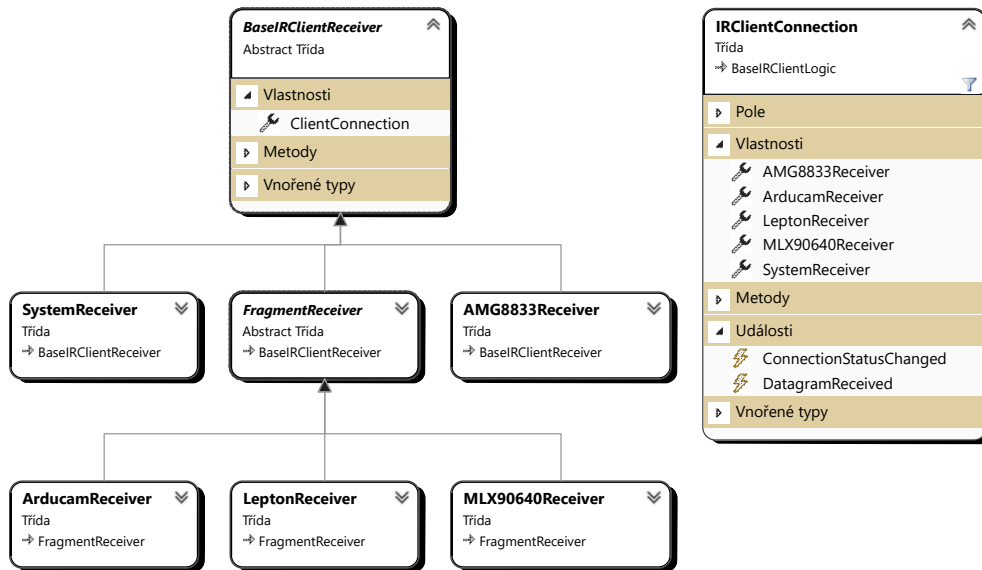
```

1 ir_color_palette=ironbow
2 ir_color_palette_preview=False
3 spectrum_overlap=0
  
```

5.2.2 IRClientConnection

Třída *IRClientConnection* obstarává síťovou komunikaci, diagram tříd je uveden na obrázku 35. Spravuje spojení s mikrokontrolérem a přijímá všechny typy paketů aplikačního protokolu. Přijaté pakety dekóduje a předá je ke zpracování instancím tříd jednotlivých typů paketů, které dědí ze třídy *BaseIRClientReceiver*. Základní abstraktní třída *FragmentReceiver* poskytuje dědicím třídám funkcionalitu pro skládání fragmentovaných paketů.

Přijímací třídy implementují **událostní** předávání zpracovaných dat do dalších částí aplikace. Každá část procesu zpracování příchozích dat vyvolává událost, kterou mohou dynamicky odebírat další třídy a reagovat tak na příchozí data. Například událost o přijetí snímku z termální kamery odebírá vykreslovací třída, která přijatý snímek vykreslí, a třída s počítadlem přijatých snímků, která inkrementuje informační počítadlo v uživatelském rozhraní s počtem přijatých snímků. Přijímače dále kontrolují, zda-li nedošlo ke ztracení některého snímku a v případě, že ano, je informace o počtu ztracených snímků předána voláním události.

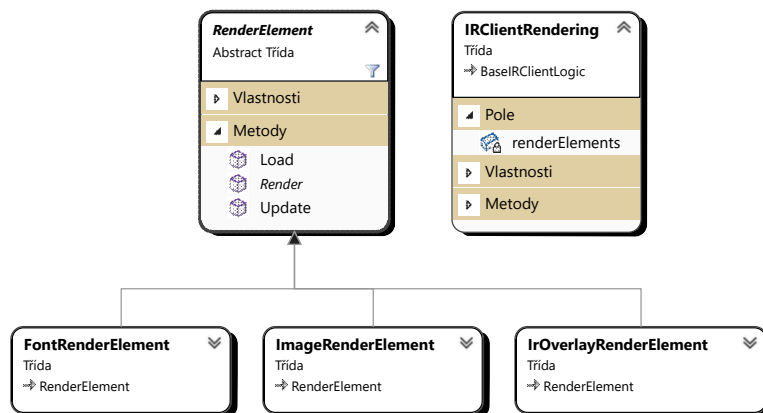


Obrázek 35: Přehled objektů řízení UDP komunikace

5.2.3 IRClientRendering

Vykreslovací úkony jsou modulárně rozděleny do potomků třídy *RenderElement*, tzv. **elementů**, které zastřešuje a řídí třída *IRClientRendering*. Diagram tříd je vidět na obrázku 36. Proces vykreslování je rozdělen do několika metod.

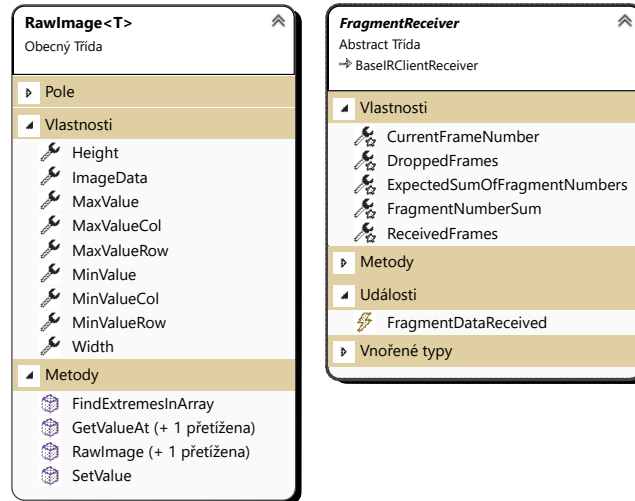
- **Load**: Inicializaci elementu, přípravu OpenGL bufferů, načtení textur apod.
- **Update**: Aktualizace stavů elementu před vykreslením
- **Render**: Vykreslení elementu s aktualizovanými daty



Obrázek 36: Přehled objektů vykreslování

5.3 Dekódování snímků v přijatých paketech

Příchozí snímky v paketech aplikačního protokolu jsou dekódovány do jednotného objektu třídy **RawImage<T>**, viz diagram na obrázku 37. Třída uchovává generická data snímku, rozlišení snímku a informace o maximální a minimální hodnotě.



Obrázek 37: Diagram třídy *rawImage* a *FragmentReceiver*

Dekódovací třídy fragmentovaných aplikačních protokolů dědí ze třídy *FragmentReceiver*, která řídí proces detekce příjmu nového snímku a detekce ztracených snímků. UDP fragmenty mohou přicházet v odlišném pořadí, než ve kterém byly odeslány, a proto musí být pro určení pozice přijatých pixelů ve snímku výlučně použito číslo pořadí fragmentu. Pořadí fragmentů začíná na čísle 1 a s každým přijetím nového fragmentu je toto pořadové číslo akumulováno v proměnné *FragmentNumberSum*. Tato suma je následně použita pro detekci přijetí všech fragmentů snímku, kdy je porovnána s výpočtem součtu aritmetické posloupnosti s_n dle vzahu 11

$$s_n = \frac{n \cdot (a_1 + a_n)}{2} = \frac{n \cdot (1 + n)}{2}, \quad (11)$$

kde n je počet členů (počet fragmentů), a_1 je první člen a a_n je poslední člen posloupnosti. V případě číslování fragmentů platí $a_1 = 1$ a $a_n = n$.

5.3.1 Arducam Mini 2MP Plus

Struktura aplikačního protokolu je uvedena v tabulce 11. Pro snímek je alokován 1 MB paměti v podobě pole bajtů *receivedData*, do kterého jsou příchozí data snímku ukládána.

```
1 private byte[] receivedData = new byte[1048576];
```

Velikost snímku není předem známa a odvíjí se až od přijmutí všech fragmentů. Poslední fragment snímku je označen příznakem a po jeho přijetí je vypočítán očekávaný součet všech pořadových čísel fragmentů.

```
1 if (data[CameraSettings.ARDUCAM_LAST_FRAGMENT_OFFSET] == 1)
2     ExpectedSumOfFragmentNumbers = SumOfFragments(fragmentNumber);
```

Počáteční index dat přijímacího bufferu *receivedData* je vypočítán v proměnné *bufferIndex* podle nastavené velikosti fragmentu a podle pořadí fragmentu, na jehož pozici jsou poté nahrána přijatá data fragmentu snímku. Výsledná velikost přijatých dat *finalSize* je součet všech dílčích velikostí dat fragmentů.

```
1 int bufferIndex = (fragmentNumber-1) * CameraSettings.ARDUCAM_FRAGMENT_SIZE;
2 int imageDataSize = data.Length - CameraSettings.ARDUCAM_IMAGE_DATA_OFFSET;
3 Buffer.BlockCopy(data, CameraSettings.ARDUCAM_IMAGE_DATA_OFFSET, ...
    receivedData, bufferIndex, imageDataSize);
4 finalSize += imageDataSize;
```

Po přijetí celého snímku jsou původní obrazová data v JPEG kompresi dekodována do objektu třídy *RawImage*. K dekodování byla použita knihovna **turbojpeg** [41] se C# wrapperem [42]. RGB složky pixelu jsou po dekodování uloženy ve 3 po sobě jdoucích bajtech.

```
1 byte[] imageData = receivedData.Take(finalSize).ToArray();
2 TJDecompressor dec = new TJDecompressor();
3 DecompressedImage di = dec.Decompress(imageData, TJPixelFormats.TJPF_RGB, ...
    TJFlags.NONE);
4 RawImage<byte> rawImage = new RawImage<byte>(di.Width, di.Height, di.Data);
```

5.3.2 FLIR Lepton 2.5

Struktura aplikačního protokolu je uvedena v tabulce 12. V jednom fragmentu se nachází 5 řádků snímku. Vertikální souřadnice dekodovaného snímku *rawImage* je pro prvního přijatý řádek vypočítána v proměnné *rowStart* a souřadnice 5. přijatého řádku v proměnné *rowEnd*. Dva vnořené cykly prochází přijatá obrazová data a skládají 14bitové hodnoty teplot uložené ve 2 po sobě jdoucích bajtech do jedné 16bitové hodnoty, ze které je následně vypočítána konečná teplota pixelu ve stupních Celsia v proměnné *finalTemp*.

```

1 int rowStart = CameraSettings.LEPTON_ROWS_PER_FRAGMENT * (fragmentNumber - 1);
2 int rowEnd = rowStart + CameraSettings.LEPTON_ROWS_PER_FRAGMENT;
3 int dataIndex = CameraSettings.LEPTON_IMAGE_DATA_OFFSET;
4 for (int row = rowStart; row < rowEnd; row++)
5 {
6     for (int col = 0; col < CameraSettings.LEPTON_WIDTH; col++)
7     {
8         UInt16 rawTemp = BEBitConverter.ToUInt16(data, dataIndex);
9         float finalTemp = (rawTemp / 100.0f) - 275.15f;
10        rawImage.SetValue(col, row, finalTemp);
11        dataIndex += 2;
12    }
13 }

```

Nakonec jsou po přijetí celého snímku nalezeny v objektu *rawImage* teplotní extrémy voláním metody *FindExtremesInArray*.

```

1 rawImage.FindExtremesInArray();

```

Kamerový modul odesílá data v big-endian zarovnání, ale x86 architektura ukládá data v little-endian zarovnání. Pro složení 2 bajtů do jedné 16bitové hodnoty proto nemůže být použita vestavěná třída frameworku *BitConverter*, jelikož by došlo k prohození nejméně významného bajtu s nejvýznamnějším. Byla proto vytvořena vlastní třída **BEBitConverter** (BE = big-endian), která deklaruje stejnou metodu se správným převodem. Lze očekávat úpadek efektivity převodu oproti původní optimalizované třídě, nicméně při nízké délce dat termálního snímku to nepředstavuje problém.

```

1 public static UInt16 ToUInt16(byte[] value, int startIndex)
2 {
3     return (UInt16)((value[startIndex] << 8) | (value[startIndex + 1]));
4 }

```

Po přijmutí celého snímku je objekt *rawImage* znovu iniciován a připraven pro příjem následujícího snímku.

```

1 rawImage = new RawImage<float>(CameraSettings.LEPTON_WIDTH, ...
    CameraSettings.LEPTON_HEIGHT);

```

5.3.3 AMG8833

Struktura aplikačního protokolu je uvedena v tabulce 13. Teplota termistoru je složena ze 2 bajtů a přenásobena konstantou pro dekódování na stupně Celsia.

```

1  Int16 thermistor = BitConverter.ToInt16(data, ...
    CameraSettings.AMG8833_THERMISTOR_OFFSET);
2  double thermistorTemperature = thermistor * 0.0625;

```

Pole obrazových dat prochází 2 vnořené for cykly a nastavují jednotlivé pixely objektu *rawImage*. Každý pixel je složen ze 2 bajtů a je přenásoben konstantou pro dekodování na stupně Celsia. Po dokončení dekodování snímku je volána metoda *FindExtremesInArray*, která najde nejvyšší a nejnižší teplotu snímku a uloží souřadnice pixelu, kde se tyto hodnoty nacházejí.

```

1  int resolution = CameraSettings.AMG8833_RESOLUTION;
2  RawImage<float> rawImage = new RawImage<float>(resolution, resolution);
3  int dataIndex = CameraSettings.AMG8833_IMAGE_DATA_OFFSET;
4
5  for (int y=0;y<resolution;y++)
6  {
7      for (int x=0;x<resolution;x++)
8      {
9          Int16 pixel = BitConverter.ToInt16(data, dataIndex);
10         rawImage.SetValue(x, y, pixel*0.25f);
11         dataIndex += 2;
12     }
13 }
14 rawImage.FindExtremesInArray();

```

5.3.4 MLX90640

Struktura aplikačního protokolu je uvedena v tabulce 14. Data snímku jsou rovnoměrně fragmentována a pro jejich přijetí je předem alokována paměť v poli floatů *receivedData*. Po přijetí fragmentu je z jeho pořadového čísla vypočítán ofset v proměnné *offset*, který je následně použit při kopírování přijatých dat snímku do pole *receivedData* na správnou lokaci.

```

1  int offset = (fragmentNumber - 1) * CameraSettings.MLX90640_FRAGMENT_SIZE;
2  Buffer.BlockCopy(data, 6, receivedData, offset, ...
    CameraSettings.MLX90640_FRAGMENT_SIZE);

```

Po přijetí celého snímku je z pole *receivedData* vytvořen výsledný objekt *rawImage*, ve kterém jsou následně nalezeny teplotní extrémy voláním funkce *FindExtremesInArray*.

```

1  RawImage<float> rawImage = new RawImage<float>(CameraSettings.MLX90640_WIDTH, ...
    CameraSettings.MLX90640_HEIGHT, receivedData);
2  rawImage.FindExtremesInArray();

```


Pole `receivedData` bylo do objektu `rawImage` předáno jako reference, a proto musí být pro přípravu příjmu následujícího snímku znovu alokováno, jelikož by jinak nový snímek přepsal data předchozího snímku.

```
1 receivedData = new float[CameraSettings.MLX90640_FRAME_SIZE];
```

5.4 Palety barev termálních snímků

Pixely termálního snímku jsou tvořeny hodnotami teplot, které sami o sobě neudávají žádnou barvu. Aplikování palety barev na termální snímek klade za úkol namapovat určitý rozsah barev na hodnoty termálního snímku tak, aby pro pozorovatele byly sledované detaily scény zřetelně rozeznatelné. Z toho vyplývá, že různé situace vyžadují odlišné palety barev. Obecně ale platí, že dynamický rozsah termální kamery běžně dosahuje až stovek stupňů Celsia, a proto barevný rozsah nemůže být mapován mezi nejnižší a nejvyšší možnou změřitelnou hodnotou, jelikož by většina detailů v mnoha případech byla nerozeznatelná. Kdybychom uvažovali rozsah teplot měřeného objektu -40 až 300 °C u modulu termální kamery MLX90640, na který by byl přiřazen barevný přechod od černé barvy k bílé a rozsah teplot měřené scény by byl 20 až 35 °C (vnitřní prostory), dynamický rozsah snímku by byl využit pouze ze 4,4 % a celý obraz by tvořily téměř stejné odstíny šedi. Situaci lze zlepšit omezením dynamického rozsahu termokamery a barevný rozsah mapovat pouze na rozsah teplot scény [1]. Aplikace nabízí dva režimy :

1. **Auto range:** Paleta barev je mapována mezi nejnižší a nejvyšší teplotu scény. Snímek dynamicky reaguje na změny teplotního rozsahu měřené scény.
2. **Fixed range:** Uživatel zadá nejnižší a nejvyšší teplotu, mezi které bude paleta barev mapována. Pokud některé pixely překračují tuto hranici, je pro ně vybrána krajní barva palety a dochází k saturaci.

5.4.1 Příprava dat

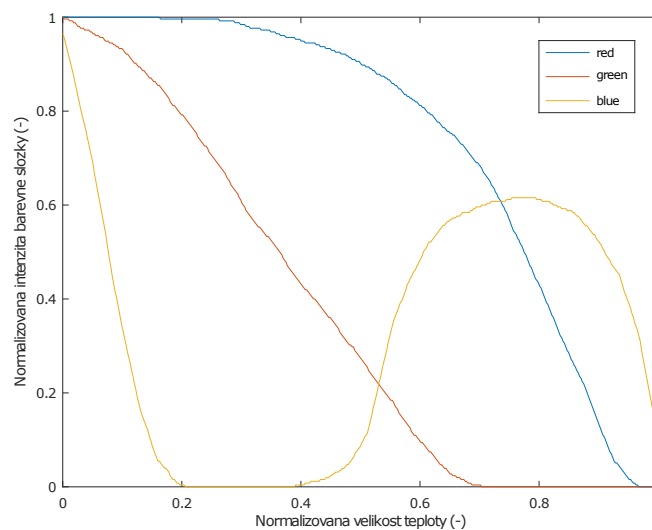
Byly vyzkoušeny 2 způsoby, kterými shader může aplikovat paletu barev na snímek. Buď je do shaderu nahrán předpis aproximující křivky barevných kanálů palet, na jejíž základě shader vypočítává výslednou barvu, nebo jsou průběhy barevných složek předem navzorkovány do souboru a shader vybírá barvu nejbližšího vzorku (případně lineárně aproximuje barvu ležící mezi 2 vzorky). Druhý způsob je založen na principu 1D textury.



Obrázek 38: Zdrojová data palet barev (seshora Ironbow, Glowbow, Rainbow), převzato z [43]

Jelikož průběhy barevných složek některých palet nejsou lineární (viz obrázek 39), byla aproximace provedena polynomy o řádu až 20 v softwaru Octave. Aproximace výsledné barvy výpočtem z polynomu byla implementována do shaderu. Bylo však zjištěno, že výpočet mocnin u takto vysokých řádků polynomů v základní přesnosti plovoucího čísla (32 bitů) není realizovatelný, protože akumulace nepřesností převýšila mnohonásobně použitelnost výsledku.

Ze získaných polynomů bylo proto předem vygenerováno 1000 vzorků v softwaru Octave s dvojnásobnou přesností, které byly uloženy do 1D textury v binárním formátu float čísel. barevné složky jsou v souboru uloženy po sobě, tedy vzorky 1 až 1000 představují červený kanál, vzorky 1001 až 2000 modrý kanál a vzorky 2001 až 3000 zelený kanál.



Obrázek 39: Závislost normalizované velikosti barevné složky na normalizované velikosti teploty u palety barev Ironbow

5.4.2 Aplikování palety ve fragment shaderu

Palety barev jsou ve fragment shaderu uloženy v UBO (podkapitola 4.3.2) jako pole datových typů *vec4*. Přestože jsou vzorky palet barev třísložkové, kvůli zarovnání *std140* byl volen čtyřsložkový vektor, přičemž 4. složka je nulová. Po startu aplikace jsou všechny binární soubory palet barev načteny do pole floatů, ze kterého je naplněn UBO.

```

1 uniform float minVal;
2 uniform float maxVal;
3 layout (std140) uniform paletteBlock
4 {
5     vec4 colorPalettes[3000];
6 };

```

Funkce *ColorPalette* aplikuje vybranou paletu na vstupní proměnnou *intensity*, jejíž hodnota musí ležet v rozmezí od *minVal* do *maxVal*, které společně určují celkový dynamický rozsah.

Do proměnných *lowerIndex* a *upperIndex* je vybrán nejbližší spodní a horní vzorek z palety, ze kterých je výsledná barva interpolována. Vestavěná funkce **mix** lineárně interpoluje mezi prvními dvěma vstupními parametry, přičemž výsledek interpolace váhuje třetí parametr. Jelikož jsou všechny palety barev uloženy ve stejném poli, vstupní proměnná *paletteOffset* vybírá, která paleta bude použita. Pro přehlednost byl z ukázky kódu odstraněn kód pro saturaci indexu, který zajišťuje, aby nedošlo k přístupu do paměti mimo hranice pole *colorPalettes*.

```
1  vec4 ColorPalette(float intensity, int paletteOffset)
2  {
3      float index = intensity/0.001;
4      int lowerIndex = 999 - int(floor(index));
5      int upperIndex = 999 - int(ceil(index));
6
7      lowerIndex += paletteOffset;
8      upperIndex += paletteOffset;
9      return mix(colorPalettes[lowerIndex], colorPalettes[upperIndex], ...
10         index-int(index));
11 }
```

Tímto způsobem byly implementovány 3 palety barev z obrázku 38. Poslední černobílá paleta Greyscale byla pro svoji jednoduchost implementována přímo v shaderu bez použití 1D textury se vzorky barev. Funkce *mix* lineárně interpoluje výsledný odstín šedi mezi černou a bílou barvou v závislosti na vstupní teplotní intenzitě *intensity*.

```
1  vec4 Greyscale(float intensity)
2  {
3      vec4 black = vec4(0.0, 0.0, 0.0, 0.0);
4      vec4 white = vec4(1.0, 1.0, 1.0, 0.0);
5      return mix(black, white, intensity);
6  }
```

5.5 Vykreslování

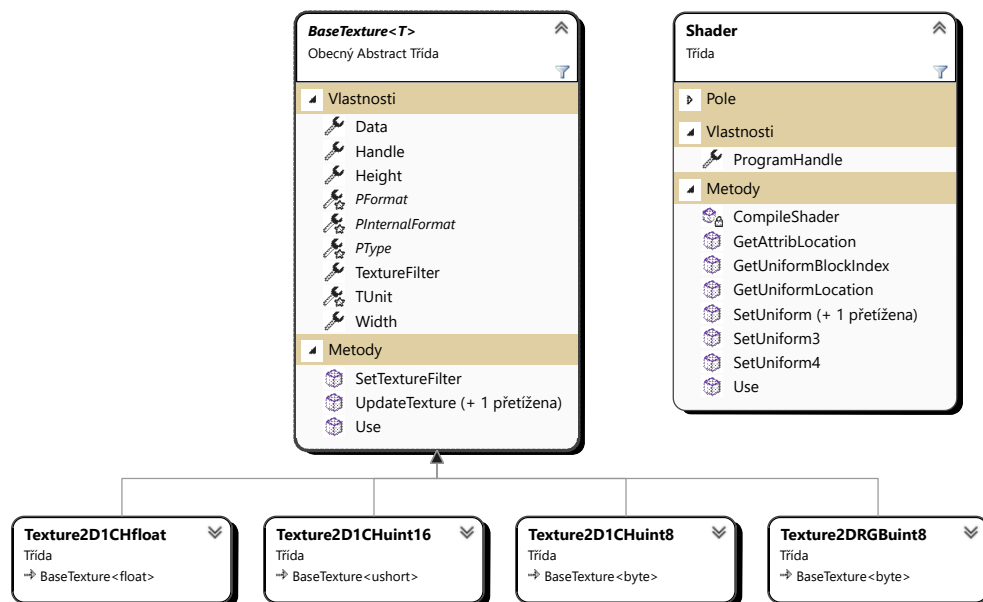
Vykreslování zajišťuje OpenGL s OpenTK wrapperem pro C#, princip vychází z vykreslení obdélníku přes celé vykreslovací okno, na které je aplikována textura s obrazovými daty snímku. S každým příchozím snímkem je textura aktualizována a je vyvoláno překreslení obrazovky. Detailní popis základní práce s OpenGL (resp. OpenTK) je rozveden v kapitole 4.3.3 a tato kapitola bude nadále tyto základy přeskaovat. Místo toho je zaměřena pouze na detaily implementace vybraných funkcionalit. Kromě vykreslování samotných snímků je vykresleno i překryvné uživatelské rozhraní s ukazateli míst teplotních extrémů, textovým výpisem teploty a legendou termální palety barev.

5.5.1 Struktura vykreslovací logiky

Práce se shadery, programem grafické karty a texturami byla zjednodušena deklarováním pomocných tříd, které uchovávají odkazy bufferů grafické karty a skrývají do jisté míry OpenTK volání. Diagram tříd pro grafické operace je vidět na obrázku 40.

Třída **Shader** načítá zdrojové kódy vertex a fragment shaderu, kompiluje je a vytváří z nich program grafické karty. Kontroluje při všech operacích chyby, které případně vypisuje do spodního informačního panelu aplikace. Zjednodušuje také nastavování uniform proměnných v shaderech.

Ze základní abstraktní třídy **BaseTexture** dědí jednotlivé třídy textur, které implementují specifické formáty dat. Zejména třída *Texture2D1CHfloat* je jednokanálová textura s nenormalizovanými 32bitovými floaty, které nesou teplotu termálních snímků. Dále pak třída *Texture2DRGBuint8* slouží k uchování barvy o třech kanálech s 8bitovou barevnou hloubkou pro pixely snímků ve viditelném spektru.



Obrázek 40: Přehled objektů zabalujících volání OpenTK

5.5.2 Vykreslování snímků

Současně je vykreslován snímek v infračerveném spektru a snímek ve viditelném spektru, přičemž oba jsou přes sebe překryty a uživatel plynule nastavuje, které spektrum převládá. Při pozorování termálního snímku někdy nemusí být úplně zřetelné, která část tělesa zobrazená na snímku odpovídá části tělesa ve skutečnosti, proto je příhodné sledovat těleso v obou spektrech. Další uživatelskou funkcí je možnost posunout snímky v ose x a y tak, aby snímaná tělesa byla opravdu přes sebe překryta, protože přestože jsou kamerové moduly umístěny v těsné blízkosti vedle sebe, nesnímají stejné zorné pole.

Proměnná fragment shaderu *texture0* obsahuje jednokanálovou texturu termálního snímku a

proměnné *offsetX* a *offsetY* představují uživatelsky nastavený posun spekter v obou osách. Termální snímek je posunut o polovinu nastaveného posunu v jednom směru a snímek viditelného spektra je posunut o polovinu druhým směrem. Velikost teploty je vzorkována v posunutém bodě a následně je normalizována mezi nejvyšší a nejnižší teplotu scény (podrobněji rozvedeno v podkapitole 5.4.2). Na normalizovanou teplotu bodu je aplikována zvolená paleta barev a výsledná barva pixelu termálního snímku je uložena do proměnná *irColor*.

```
1 vec2 updatedTexCoord = texCoord;
2 updatedTexCoord.x += offsetX / 2.0;
3 updatedTexCoord.y += offsetY / 2.0;
4
5 float val = texture(texture0, updatedTexCoord).r;
6 float normalizedIntensity = (val-minVal)/(maxVal-minVal);
7 vec4 irColor = ColorPalette(normalizedIntensity, selectedPalette);
```

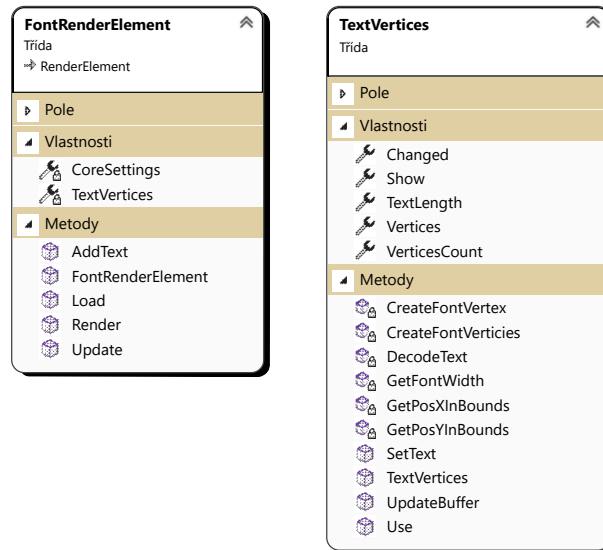
Ve druhém kroku je do proměnné *visibleColor* získána barva pixelu pro snímek viditelného spektra, který je uložen v proměnné *texture1*. Stejným způsobem, jako u termálního snímku, jsou souřadnice vzorkování *updatedTexCoord* posunuty ve druhém směru o polovinu nastaveného posunu (v ukázce kódu pro přehlednost vynecháno). Nakonec je výsledná barva pixelu interpolována mezi barvou pixelů obou snímků v závislosti na váhovacím faktoru *spectrumOverlap*, který je nastaven uživatelem.

```
1 vec4 visibleColor = texture(texture1, updatedTexCoord);
2 outputColor = mix(irColor, visibleColor, spectrumOverlap);
```

5.5.3 Překryvné uživatelské rozhraní

Přes snímky obou spekter je vykresleno překryvné uživatelské rozhraní, které informuje o teplotách termálního snímku. V pravé části okna je umístěna legenda aktuálně zvolené palety barev, ta vizuálně přiřazuje barvy obrazu ke konkrétním teplotám ve stupních Celsia. Dále jsou zobrazeny teplotní ukazatele, jenž průběžně sledují nejnižší a nejvyšší teplotu snímku a tyto body graficky ukazují. Třetí ukazatel sleduje uživatelův kurzor a vypisuje teplotu ve zvoleném bodě. Zmíněné grafické prvky je možné vidět v obrázku 32.

Způsobem uvedeným v podkapitole 4.2 bylo implementováno vykreslování fontu pro dynamické zobrazování textu v obrazu. Každá instance třídy **TextVerticies** reprezentuje jeden text na obrazovce. Kromě samotného textu lze specifikovat jeho velikost a umístění na obrazovce. Tato třída spravuje OpenGL buffery a všechny její instance jsou udržovány objektem třídy **FontRenderElement**, který řídí proces vykreslování. Diagramy obou tříd jsou vidět na obrázku 41.



Obrázek 41: Diagramy tříd pro dynamické vykreslování textu

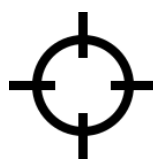
Fragment shader legendy teplotní palety barev je téměř identický jako shader pro aplikování palety barev na pixely termálního snímku v podkapitole 5.4.2. Místo z teploty pixelu termálního snímku je normalizovaná intenzita získána ze složky y souřadnic interpolované vstupní proměnné *positionInBoundingBox* z vertex shaderu.

```

1 in vec2 positionInBoundingBox;
2
3 void main()
4 {
5     float intensity = positionInBoundingBox.y;
6     vec4 color = ColorPalette(intensity, selectedPalette);
7 }

```

Grafické ukazatele zvýrazněných bodů jsou vykresleny jako čtverce s aplikovanou texturou na obrázku 42. Barva ukazatele je volena podle toho, jaký typ bodu sleduje, aby byla maximalizována jeho viditelnost. Implementované palety barev dosahují tmavých odstínů pro nízké teploty, proto byla pro ukazatel nejnižší teploty snímku zvolena bílá barva. Naopak vysoké teploty jsou světlé, a proto byla pro ukazatel nejvyšší teploty volena černá barva. Ukazatel sledující uživatelův kurzor byl nastaven na střední hodnotu obou barev, tj. odstín šedé. Pozice ukazatele je vypočítána ze souřadnic zvýrazněného pixelu, které jsou přepočteny na NDC. Nakonec je pod ukazatel umístěn text s výpisem teploty sledovaného bodu.



Obrázek 42: Textura ukazatele teplotních bodů

6 Závěr

Cílem diplomové práce bylo navázat na bakalářskou práci [1], rozšířit ji o další kamerové moduly a pokročilejší způsoby přenosu, zpracování a analýzy obrazových dat. Bylo zadáno zdokumentovat a vytvořit ukázkové příklady použití obdržených kamerových modulů snímajících v infračerveném a viditelném spektru s vývojovým kitem STM32 tak, aby bylo umožněno dosažené výsledky upravit a použít pro vývoj vlastního projektu.

V první kapitole byly určeny základní požadavky na kamerové moduly pro nasazení v embedded systémech a z toho i vyplývající limity použití. Zvláště byla kladena pozornost na jedno z předpokládaných využití práce ve studentském satelitu Pilsen CubeSAT a byl rozebrán úvod do problematiky provozu elektronického zařízení v prostoru vesmíru. Dále byly zdokumentovány jednotlivé kamerové moduly, kdy byly porovnány jejich technické parametry, popsána komunikační sběrnice s protokolem a nakonec podporované funkce a princip použití. Výsledkem je přehled většiny běžně dostupných kamerových modulů pro použití s mikrokontroléry.

Třetí kapitola implementovala poznatky z první kapitoly, se kterými bylo prakticky realizováno propojení kamerových modulů s předloženým kitem mikrokontroléru STM32. Byl navržen základní procesově orientovaný operační systém pro čtení snímků současně z kamerového modulu infračerveného spektra a modulu viditelného spektra. Pro přenos získaných obrazových dat do aplikace pro zpracování po Ethernetu byl navržen aplikační protokol založený na UDP. Pro implementaci TCP/IP zásobníku protokolů pro STM32 byl použit projekt lwIP, přičemž se část kapitoly věnovala, včetně ukávek zdrojových kódů, základům nastavení a práce s lwIP pro navázání komunikace po počítačové síti. Ve druhé kapitole byly uvedeny teoretické předpoklady síťové komunikace s bližším zaměřením na embedded zařízení. Výsledkem je obecné ukázkové řešení navázání komunikace a čtení snímků z předložených kamerových modulů a odesílání získaných dat přes počítačovou síť. Dohromady byly vypracovány ukázkové příklady pro moduly: **Arducam Mini 2MP Plus**, **FLIR Lepton 2.5**, **AMG8833** a **MLX90640**. Kapitola zahrnuje vývojové diagramy, ukázky zdrojových kódů a podrobné vysvětlení vybraných principů. Ukázkové řešení lze upravit a použít při vývoji vlastního projektu.

Pátá kapitola byla zaměřena na problematiku přijímání snímkových dat a následné dekódování, vizuální zobrazení a analýzu. Byla vytvořena počítačová aplikace v jazyce C# s Winforms frameworkem a grafickou knihovnou OpenTK pro hardwarově akcelerované vykreslování pomocí grafické karty. Čtvrtá kapitola uvedla základní principy vykreslování s OpenTK, které byly vysvětleny na ukázkovém příkladu včetně ukávek zdrojových kódů. Na těchto základech byla později vytvořena vizuální část počítačové aplikace, která vykreslila data snímků a aplikovala grafické úpravy zvyšující jejich informační hodnotu. Zejména byly realizovány tyto grafické funkce: prolínání spekter snímků, aplikování několika palet barev s automatickým nebo statickým rozsahem, překryvné uživatelské rozhraní s legendou teplotní palety a dynamickými ukazateli teplotních extrémů snímků. Struktura aplikace byla navržena modulárně, díky čemu ji lze jednoduše rozšířit o další funkce, či případně použít některé části při vývoji vlastního projektu. Zejména vykreslovací

část dosahuje vysokého grafického výpočetního výkonu a bylo by možné ji použít pro vykreslování obrazových dat s násobně vyšším rozlišením a snímkovou frekvencí než dosahovaly předložené kamerové moduly.

Seznam použité literatury

1. PILLMANN, Jan. *Použití FLIR Lepton modulu* [online]. [B.r.]. [cit. 2024-03-24]. Dostupné z: <https://dspace5.zcu.cz/bitstream/11025/48951/1/zaverecna-prace.pdf>.
2. TEXAS INSTRUMENTS. *Radiation Handbook for Electronics* [online]. [B.r.]. [cit. 2024-03-05]. Dostupné z: <https://www.ti.com/seclit/eb/sgzy002a/sgzy002a.pdf>.
3. OMNIVISION. *OV2640 Camera Module Software Application Notes* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: <https://hobbylad.files.wordpress.com/2020/02/ov2640-camera-module-software-application-notes.pdf>.
4. OMNIVISION. *OV5642 Camera Module Software Application Notes* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://www.uctronics.com/download/Image_Sensor/OV5642_camera_module_software_application_notes.pdf.
5. OMNIVISION. *OV2640 Advanced Information Preliminary Datasheet* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://www.uctronics.com/download/cam_module/OV2640DS.pdf.
6. MIPI ALLIANCE. *Current Specifications* [online]. [B.r.]. [cit. 2024-02-26]. Dostupné z: <https://www.mipi.org/current-specifications>.
7. FUTURE TECHNOLOGY DEVICES INTERNATIONAL LIMITED (FTDI). *What is the Camera Parallel Interface?* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://www.ftdichip.com/old2020/Support/Documents/TechnicalNotes/TN_158_What_Is_The_Camera_Parallel_Interface.pdf.
8. OMNIVISION. *OV5642 datasheet PRELIMINARY INFORMATION* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://www.uctronics.com/download/cam_module/OV5642DS.pdf.
9. TEXAS INSTRUMENTS. *Understanding the I2C Bus* [online]. [B.r.]. [cit. 2024-02-27]. Dostupné z: <https://www.ti.com/lit/an/slva704/slva704.pdf>.
10. OMNIVISION. *OmniVision Serial Camera Control Bus (SCCB) Functional Specification* [online]. [B.r.]. [cit. 2024-02-26]. Dostupné z: https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/f2021/jfw225_aei23_dsb298/jfw225_aei23_dsb298/SCCBSpec_AN.pdf.
11. ARDUCAM. *ArduCAM-Mini-5MP-Plus OV5642 Camera Module 5MP SPI Camera Hardware Application Note* [online]. [B.r.]. [cit. 2024-02-29]. Dostupné z: https://arducam.com/downloads/shields/ArduCAM_Mini_5MP_Plus_OV5642_Camera_Module_Hardware_Application_Note.pdf.
12. ARDUCAM. *ArduCAM Library Introduction* [online]. [B.r.]. [cit. 2024-03-19]. Dostupné z: <https://github.com/ArduCAM/Arduino>.

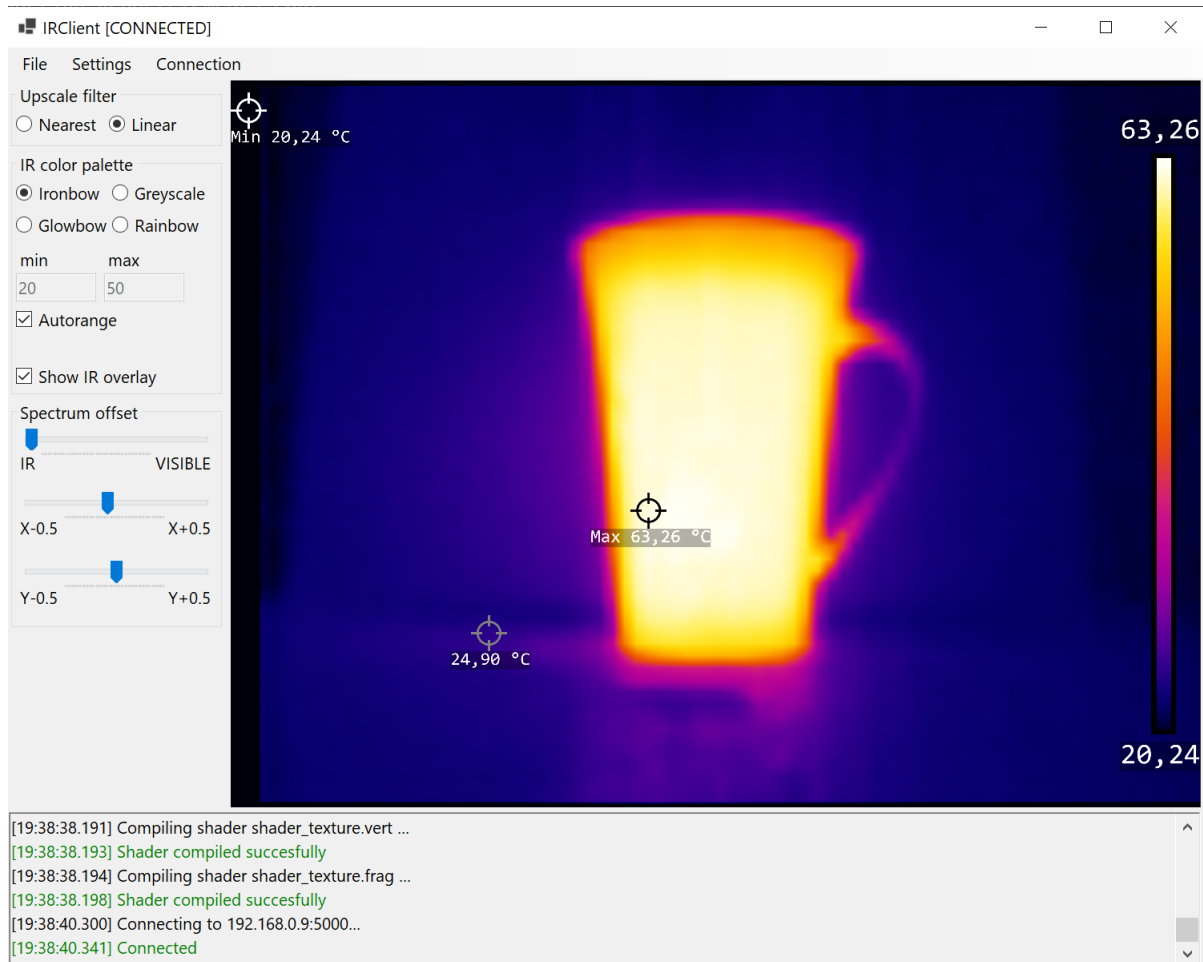
13. ARDUCAM. *ArduCAM-M-2MP Camera Shield 2MP SPI Camera User Guide* [online]. [B.r.]. [cit. 2024-03-22]. Dostupné z: https://www.uctronics.com/download/Amazon/ArduCAM_Mini_2MP_Camera_Shield_DS.pdf.
14. ARDUCAM. *Arducam Shield Mini 5MP Plus* [online]. [B.r.]. [cit. 2024-03-22]. Dostupné z: <https://docs.arducam.com/Arduino-SPI-camera/Legacy-SPI-camera/Hardware/Arducam-Shield-Mini-5MP-Plus/>.
15. FLIR. *FLIR LEPTON® Engineering Datasheet* [online]. [B.r.]. [cit. 2024-03-24]. Dostupné z: <https://www.flir.com/globalassets/imported-assets/document/flir-lepton-engineering-datasheet.pdf>.
16. PANASONIC INDUSTRY. *Infrared Array Sensor Grid-EYE AMG88xx (High performance type)* [online]. [B.r.]. [cit. 2024-03-03]. Dostupné z: <https://mediap.industry.panasonic.eu/assets/imported/industrial.panasonic.com/cdbs/www-data/pdf/ADI8000/ADI8000C66.pdf>.
17. PANASONIC CORPORATION. *I2C interface* [online]. [B.r.]. [cit. 2024-03-03]. Dostupné z: <https://mediap.industry.panasonic.eu/assets/imported/industrial.panasonic.com/cdbs/www-data/pdf/ADI8000/ast-ind-177617.pdf>.
18. .NET NANOFRAMEWORK. *AMG8833/AMG8834/AMG8853/AMG8854 Infrared Array Sensor Family* [online]. [B.r.]. [cit. 2024-03-03]. Dostupné z: <https://docs.nanoframework.net/devicesdetails/Amg88xx/README.html>.
19. MELEXIS. *MLX90640 32x24 IR array* [online]. [B.r.]. [cit. 2024-03-22]. Dostupné z: <https://www.melexis.com/en/documents/documentation/datasheets/datasheet-mlx90640>.
20. MELEXIS. *MLX90641 16x12 IR array* [online]. [B.r.]. [cit. 2024-03-22]. Dostupné z: <https://www.melexis.com/en/documents/documentation/datasheets/datasheet-mlx90641>.
21. MELEXIS. *mlx90640-library* [online]. [B.r.]. [cit. 2024-03-22]. Dostupné z: <https://github.com/melexis/mlx90640-library>.
22. MELEXIS. *mlx90641-library* [online]. [B.r.]. [cit. 2024-03-22]. Dostupné z: <https://github.com/melexis/mlx90641-library>.
23. ISO/IEC 7498-1:1994. *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model* [online]. [B.r.]. [cit. 2024-03-05]. Dostupné z: <https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html>.
24. IBM REDBOOKS. *TCP/IP Tutorial and Technical Overview* [online]. [B.r.]. [cit. 2024-03-07]. Dostupné z: <https://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf>.
25. IEEE STD 802.3-2002 (REVISION OF IEEE STD 802.3, 2000 edn). *IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications* [online]. [B.r.]. [cit. 2024-03-07]. Dostupné z: <https://ieeexplore.ieee.org/document/988967>.

26. STMICROELECTRONICS. *RM0090 Reference manual* [online]. [B.r.]. [cit. 2024-03-07]. Dostupné z: https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.
27. MICROCHIP TECHNOLOGY. *LAN8742A/LAN8742Ai Small Footprint RMII 10/100 Ethernet Transceiver with HP Auto-MDIX and flexPWR® Technology* [online]. [B.r.]. [cit. 2024-03-07]. Dostupné z: <https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/8742a.pdf>.
28. SAVANNAH. *lwIP 2.1.0 Lightweight IP stack* [online]. [B.r.]. [cit. 2024-03-18]. Dostupné z: https://www.nongnu.org/lwip/2_1_x/index.html.
29. MELEXIS. *MLX90640 32x24 IR array Driver* [online]. [B.r.]. [cit. 2024-04-11]. Dostupné z: <https://github.com/melexis/mlx90640-library/blob/master/MLX90640%20driver.pdf>.
30. OPENTK. *OpenTK* [online]. [B.r.]. [cit. 2024-04-09]. Dostupné z: <https://opentk.net/>.
31. MOURIS, Dimitris; GOUERT, Charles; GUPTA, Nikhil; TSOUTSOS, Nektarios. Peak your Frequency: Advanced Search of 3D CAD Files in the Fourier Domain. *IEEE Access* [online]. 2020, roč. 8, s. 1–1 [cit. 2024-04-03]. Dostupné z DOI: [10.1109/ACCESS.2020.3013284](https://doi.org/10.1109/ACCESS.2020.3013284).
32. JOEY DE VRIES, DYLAN PERKS. *LearnOpenTK* [online]. [B.r.]. [cit. 2024-04-01]. Dostupné z: <https://opentk.net/learn/>.
33. OPENGL WIKI. *Data Type (GLSL)* [online]. [B.r.]. [cit. 2024-04-11]. Dostupné z: [https://www.khronos.org/opengl/wiki/Data_Type_\(GLSL\)](https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)).
34. OPENGL WIKI. *Fragment Shader* [online]. [B.r.]. [cit. 2024-04-09]. Dostupné z: https://www.khronos.org/opengl/wiki/Fragment_Shader.
35. MARK SEGAL, Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 4.5 (Core Profile) - June 29, 2017)* [online]. [B.r.]. [cit. 2024-04-05]. Dostupné z: <https://registry.khronos.org/OpenGL/specs/gl/glspec45.core.pdf>.
36. OPENGL WIKI. *Image Format* [online]. [B.r.]. [cit. 2024-04-09]. Dostupné z: https://www.khronos.org/opengl/wiki/Image_Format.
37. WANG, Yizhi; LIAN, Zhouhui. DeepVecFont: Synthesizing High-quality Vector Fonts via Dual-modality Learning. *ACM Transactions on Graphics* [online]. 2021 [cit. 2024-04-04]. Dostupné z DOI: [10.1145/3478513.3480488](https://doi.org/10.1145/3478513.3480488).
38. FLOREZ, Horacio; INTESA, R.M.M. Automatic unstructured mesh generation around two-dimensional domains described by B-spline curves. In: [online]. 2001, sv. 1, s. 395–400 [cit. 2024-04-04].
39. LEMBERG, Werner. *The FreeType Project* [online]. [B.r.]. [cit. 2024-04-04]. Dostupné z: <https://freetype.org/>.
40. OPENGL WIKI. *Shader Storage Buffer Object* [online]. [B.r.]. [cit. 2024-04-09]. Dostupné z: https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object.
41. LIBJPEG-TURBO. *libjpeg-turbo* [online]. [B.r.]. [cit. 2024-04-13]. Dostupné z: <https://libjpeg-turbo.org/>.

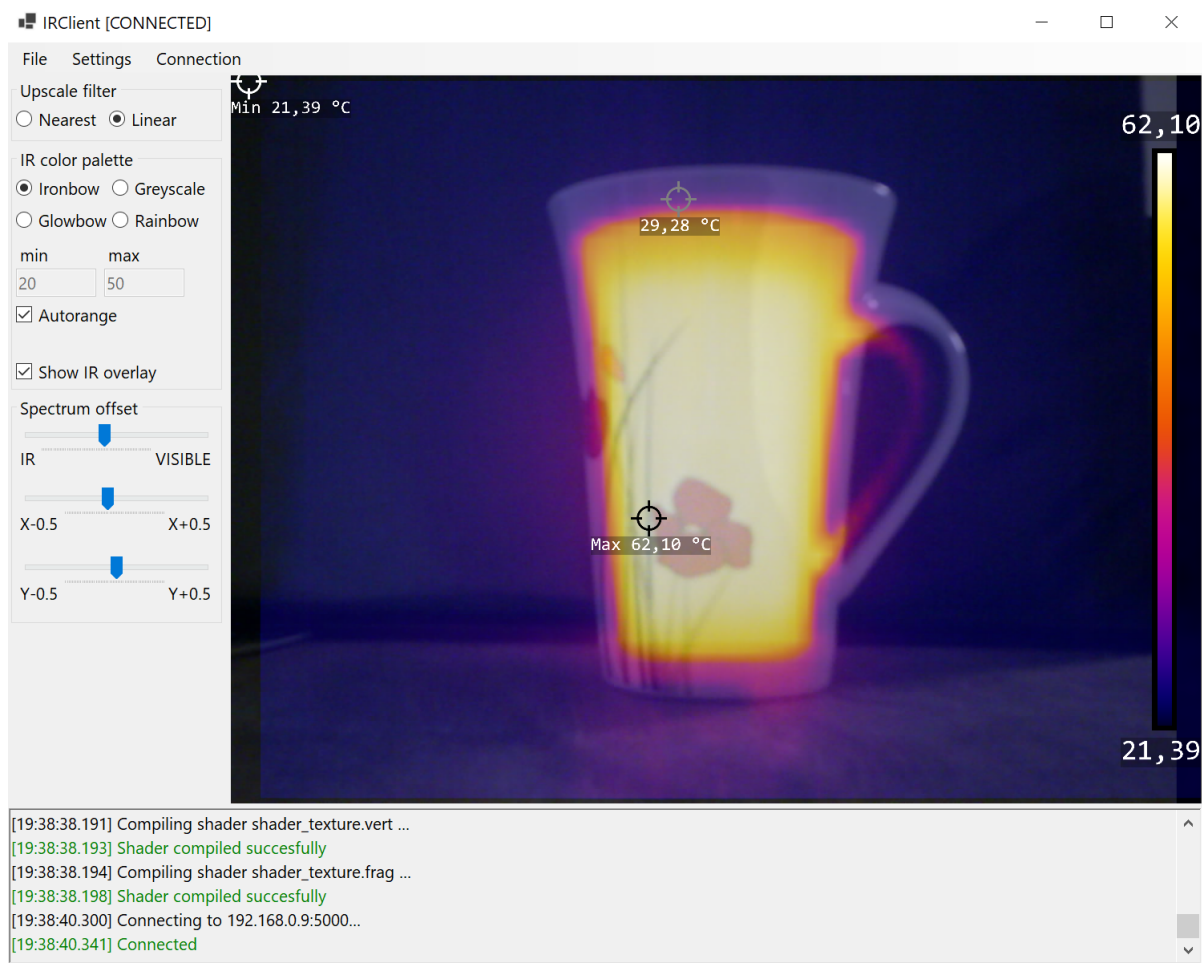
42. SERGEYTER. *libjpeg-turbo-net* [online]. [B.r.]. [cit. 2024-04-13]. Dostupné z: <https://www.nuget.org/packages/libjpeg-turbo-net>.
43. TATTERSALL, Glenn. *Choosing the appropriate colour palette for thermal imaging in animals* [online]. [B.r.]. [cit. 2024-04-12]. Dostupné z: <https://ramphastosramblings.blogspot.com/2014/03/choosing-appropriate-colour-palette-for.html>.

Příloha A

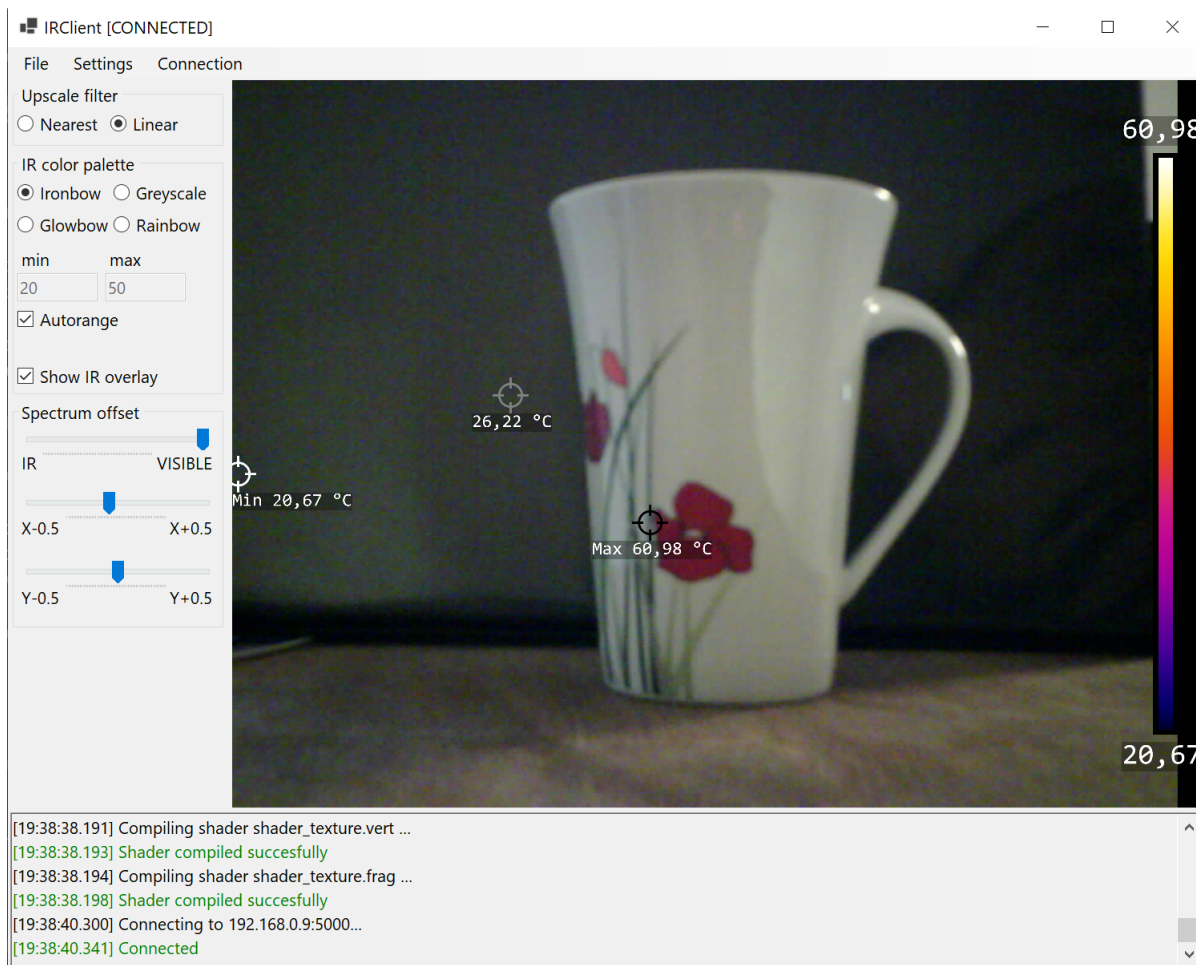
6.1 Ukázka grafického výstupu aplikace pro zpracování dat



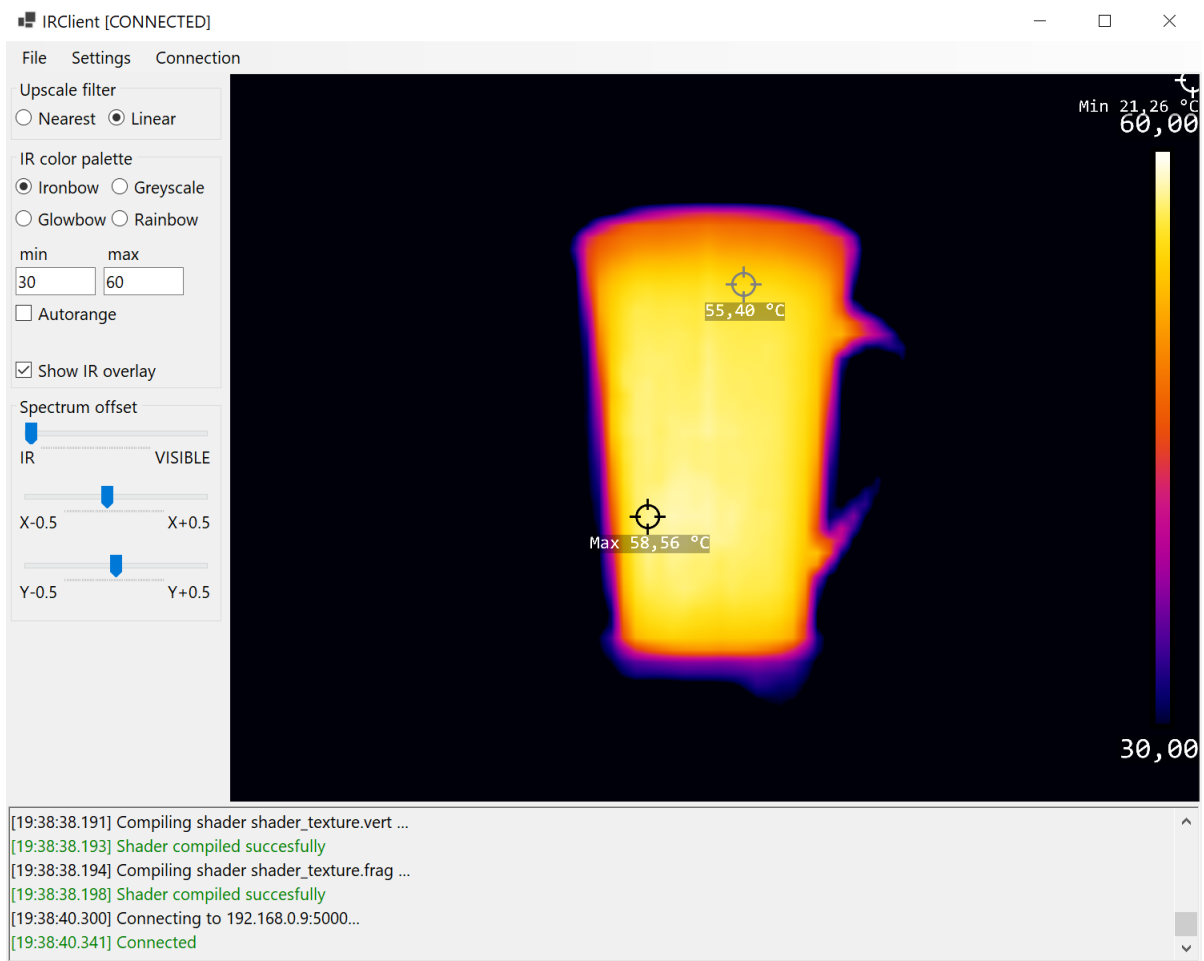
Obrázek 43: Ironbow paleta barev se zobrazením pouze infračerveného spektra



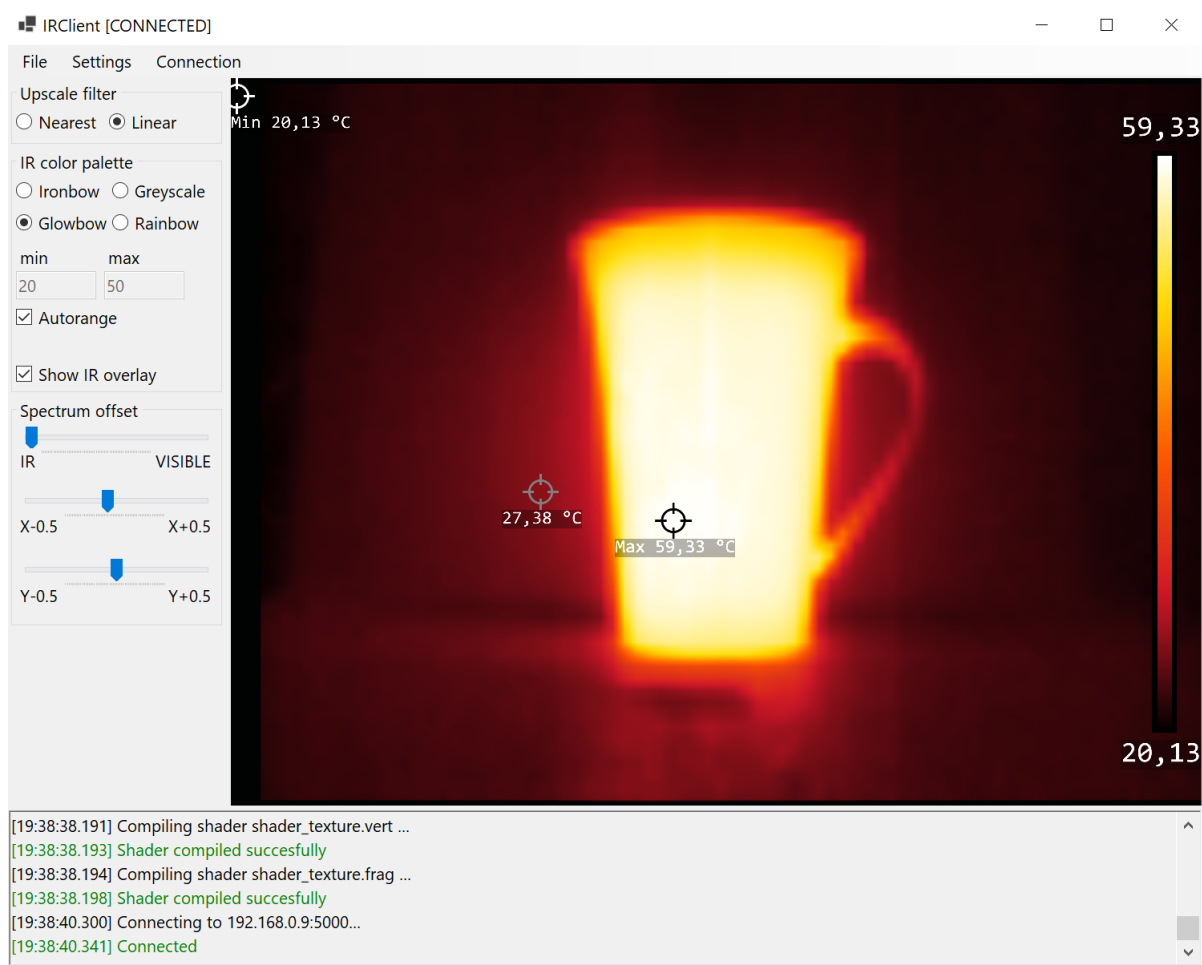
Obrázek 44: Ironbow paleta barev s prolnutím infračerveného spektra s viditelným spektrem



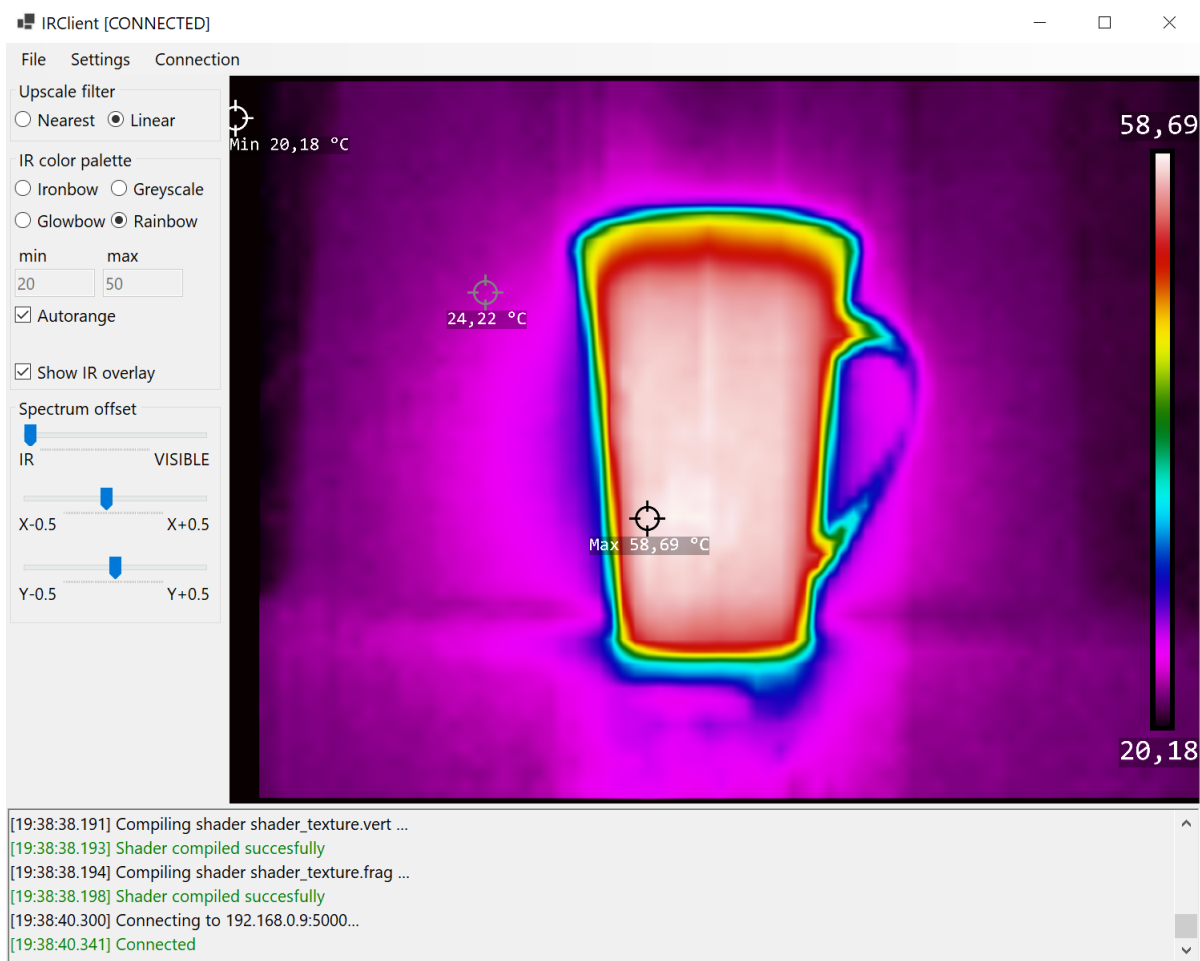
Obrázek 45: Zobrazení pouze viditelného spektra



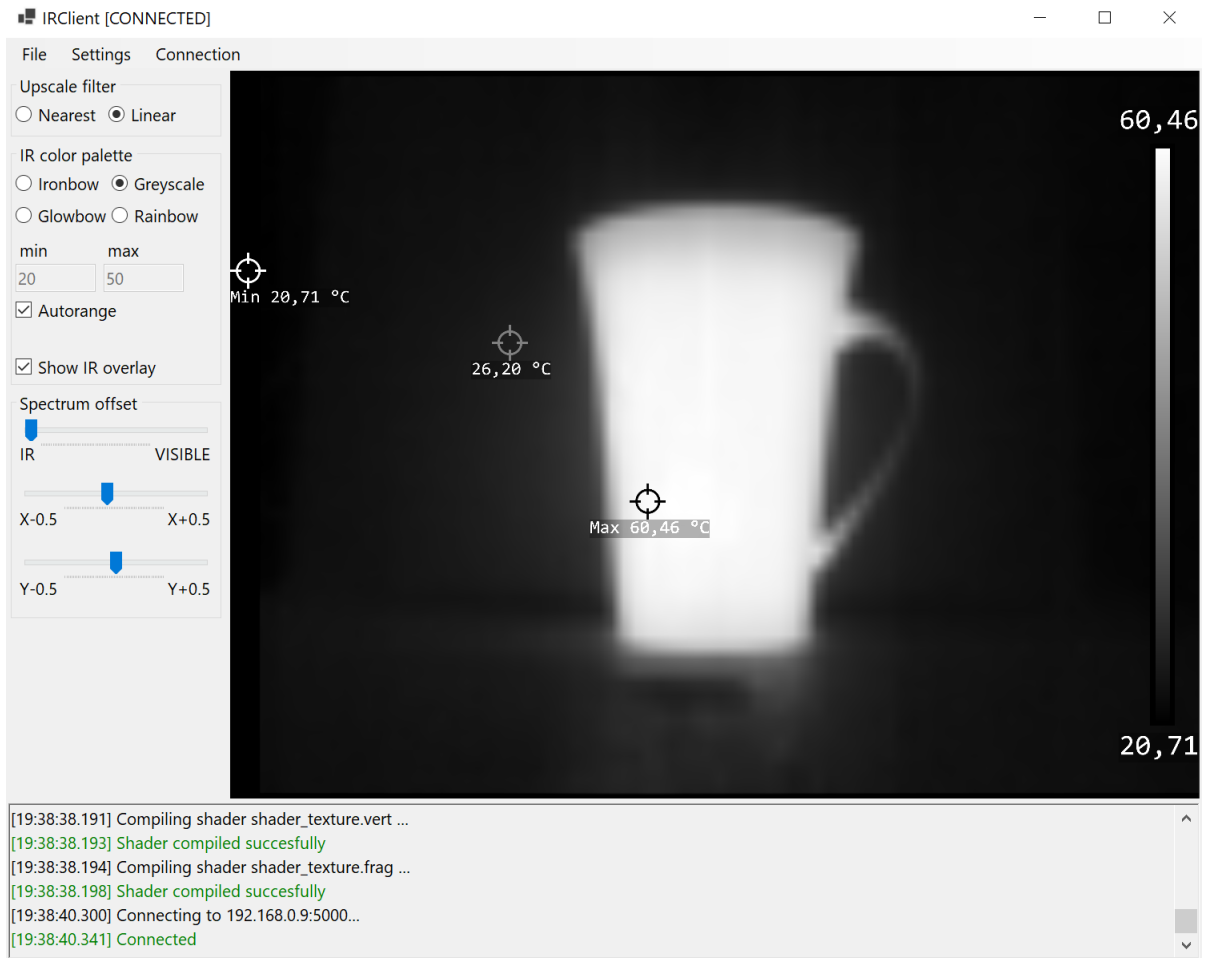
Obrázek 46: Ironbow paleta barev s pevně nastaveným mapováním na teplotní rozsah



Obrázek 47: Glowbow paleta barev



Obrázek 48: Rainbow paleta barev



Obrázek 49: Greyscale paleta barev