FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Inteligentní monitorovací systém
Student:	Michal Žůrek
Vedoucí:	Ing. Miroslav Skrbek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Seznamte se s moderními IOT technologiemi na bázi Azure Sphere a cloudovými službami Azure pro umělou inteligenci. Navrhněte a implementujete prototyp inteligentního kamerového monitorovacího systému, který bude zpracovávat snímky pořízené koncovým zařízením pokročilými technikami umělé inteligence a detekovat na nich požadované objekty. Obrazová data mohou být doplněna o audio signál a senzorická data. Uvažujte aplikační oblasti jako je například sledování parkovacích míst nebo zabezpečení objektů. Koncové zařízení by mělo být flexibilní, levné a výpočetně náročné funkce by měly být delegovány na cloud. Zvažte možnosti redukce spotřeby zařízení s ohledem na možnost solárního napájení cílového zařízení. Zařízení by mělo zajišťovat automatické připojení ke cloudovým službám, bezpečnou komunikaci s nimi a bezpečný upgrade firmwaru. Rozsah práce stanovte po dohodě s vedoucím práce.

Elektronicky schválil/a doc. Ing. Hana Kubátová, CSc. dne 11. prosince 2020 v Praze.

Bakalářská práce

INTELIGENTNÍ MONITOROVACÍ SYSTÉM

Michal Žůrek

Fakulta informačních technologií ČVUT v Praze
Katedra číslicového návrhu
Vedoucí: Ing. Miroslav Skrbek, Ph.D.
13. května 2021

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Michal Žůrek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technické v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bez uplatněných zákonných licencí nad rámec oprávnění uvedených v Prohlášení je nezbytný souhlas autora.

Odkaz na tuto práci: Michal Žůrek. *Inteligentní monitorovací systém.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Obsah

Poděkování ix

Prohlášení x

Abstrakt xi

Shrnutí xii

Seznam zkratek xiii

1 Úvod 1

2 Analýza 3
- 2.1 Monitorovací systém obrazových dat . . . 3
- 2.2 Existující řešení . . . 3
  - 2.2.1 Bezpečnost existujících systémů . . . 4
- 2.3 Platforma pro implementaci zařízení monitorovacího systému . . . 5
  - 2.3.1 Mikrokontroléry . . . 6
  - 2.3.2 Jednodeskové počítače . . . 8
  - 2.3.3 Hradlová pole . . . 9
  - 2.3.4 Prolínaní konceptů . . . 11
  - 2.3.5 Platforma Xilinx Zynq . . . 11
  - 2.3.6 Azure Sphere . . . 12
  - 2.3.7 Vývojové desky a moduly s platformou Azure Sphere . . . 14
- 2.4 Kamera . . . 16
  - 2.4.1 Datový tok obrazových dat . . . 16
  - 2.4.2 Sériové sběrnice . . . 16
  - 2.4.3 Paralelní sběrnice . . . 17
  - 2.4.4 Předpoklady obsluhy paralelní datové sběrnice . . . 17
  - 2.4.5 Formáty výstupního obrazu . . . 18
- 2.5 Využití cloudu . . . 19
- 2.6 Analýza obrazu . . . 20

3 Návrh systému 23
- 3.1 Části řešení . . . 23
- 3.2 Struktura řešení zařízení . . . 24
- 3.3 Logické propojení periferií zařízení . . . 24
  - 3.3.1 Revize vývojové desky Azure Sphere . . . 24
  - 3.3.2 Připojení GPIO expandéru . . . 25
  - 3.3.3 Připojení kamery . . . 26
  - 3.3.4 Generování hodin pro kameru . . . 26
  - 3.3.5 Zapojení datové sběrnice kamery . . . 27
  - 3.3.6 Připojení synchronizačních signálů kamery . . . 30
  - 3.3.7 Připojení mikrofonu . . . 30

3.4 Návrh desky plošného spoje . . . 31
3.5 Obvod pro snímání okolního zvuku . . . 31
3.6 Firmware zařízení . . . 32
3.6.1 Snímání obrázku z kamery . . . 33
3.6.2 Ukládání snímku do paměti . . . 34
3.6.3 Komunikace mezi dílčími aplikacemi . . . 36
3.6.4 Sdílení paměti mezi jádry . . . 37
3.6.5 Komunikační protokoly mezi aplikacemi . . . 41
3.6.6 Připojení zařízení do cloudu . . . 44
3.6.7 Zápis do cloudového uložiště . . . 47
3.7 Návrh cloudových aplikací . . . 47
3.7.1 Univerzální část aplikace . . . 47
3.7.2 Aplikačně specifická část aplikace . . . 50
3.7.3 Monitorovací systém obsazení parkoviště . . . 50
3.7.4 Zprávy zasílané mezi zařízením a cloudem . . . 51

4 Implementace řešení 53

4.1 Firmware zařízení . . . 53
4.1.1 Použité knihovny . . . 54
4.1.2 Sdílení kódů . . . 59
4.1.3 Ladění aplikací . . . 60
4.1.4 Čtení paměti zařízení do souboru pomocí OpenOCD . . . 61
4.1.5 Ladící UART . . . 64
4.1.6 Ladění pomocí výstupního portu GPIO . . . 65
4.1.7 Konfigurace registrů kamery . . . 65
4.1.8 Optimalizace částí aplikace kritické na časování . . . 66
4.1.9 Optimalizace doby běhu přerušení . . . 66
4.1.10 Optimalizace latence přerušení . . . 67
4.1.11 Optimalizace zápisů do paměti v jiných jádrech . . . 67
4.1.12 Implementace memory poolu . . . 67
4.1.13 Ovládání GPIO expandéru . . . 69
4.1.14 Detekce anomálií v okolí zařízení . . . 70
4.1.15 Zpracování senzorických dat . . . 70
4.1.16 Zpracování audio signálu . . . 70
4.1.17 Řízení snímání obrázku a jeho přenosu do cloudu . . . 70
4.1.18 Optimalizace spotřeby energie zařízení . . . 71
4.2 Cloudové aplikace . . . 71
4.2.1 Konfigurace IoT Hubu . . . 72
4.2.2 Konfigurace DPS . . . 73
4.2.3 Konfigurace uložiště . . . 73
4.2.4 Konfigurace front zpráv . . . 73
4.2.5 Aplikace běžící v bezstavové funkci univerzální části aplikace . . . 73
4.2.6 Emulátor zařízení . . . 74
4.2.7 Konfigurace databáze . . . 74
4.2.8 Konfigurace služby rozpoznávání obrazu . . . 75
4.2.9 Aplikace běžící v bezstavové funkci pro zpracování a analýzu obrázků . . . 76
4.2.10 Webová aplikace prezentující výsledky . . . 76
4.2.11 Konfigurace autentizace pomocí Azure AD . . . 78

5 Testování **79**
5.1 Testování splnění časových požadavků sběrnice . . . 79
5.2 Testování desky plošného spoje . . . 79
5.3 Testování možnosti vzdálené aktualizace firmware . . . 81

6 Závěr **83**

A Návod na zprovoznění cloudových aplikací **85**
A.1 Nasazení univerzální části cloudové aplikace . . . 85
A.2 Povolení emulátoru zařízení . . . 86
A.3 Nasazení aplikace monitorující parkoviště . . . 86
A.3.1 Cosmos DB . . . 87
A.3.2 Computer Vision . . . 87
A.3.3 Bezstavová analytická funkce . . . 87
A.3.4 Webová administrace . . . 87
A.4 Nasazení firmware zařízení . . . 88

B Schéma desky plošného spoje **89**

Obsah přiloženého média **95**

Seznam obrázků

2.1 Obecné schéma monitorovacího systému 4
2.2 Vývojová deska s mikrokontrolérem 6
2.3 Vývojová deska s mikrokontrolérem PSoC 62 a čipem CYW43012 7
2.4 Jednodeskový počítač Raspberry Pi 3B 8
2.5 Schéma možného řešení pomocí FPGA 10
2.6 Schéma možného řešení pomocí FPGA a čipu s podporou obsluhy síťového stacku 10
2.7 Schéma možného řešení pomocí FPGA a SoftCore 10
2.8 Vývojová deska Basys 3 11
2.9 Platforma Xilinx Zynq 12
2.10 Vývojová deska s čipem Xilinx Zynq 12
2.11 Platforma Azure Sphere 13
2.12 Struktura vývojové desky Azure Sphere 15
2.13 Vývojová deska Avnet Azure Sphere Starter Kit 15
2.14 Modul s kamerou OV7670 17
2.15 Potenciální řešení navržené podle principu IaaS 20
2.16 Potenciální řešení navržené podle principu PaaS 21

3.1 Obecné schéma topologie aplikace a škálovatelnosti 24
3.2 Obecné schéma návrhu a zapojení hardwarové části řešení 25
3.3 Přehled signálů vedených mezi vývojovou deskou a kamerou 26
3.4 Zapojení hlavního hodinového signálu kamery 27
3.5 Zapojení datové sběrnice kamery 27
3.6 Potenciální možné mapování GPIO portů na datovou sběrnici kamery pro nejefektivnější možné čtení dat ze sběrnice 28
3.7 Mapování GPIO portů na datovou sběrnici kamery optimalizováno s ohledem na množství křížení vodičů na desce plošného spoje pro platofrmu revize 1 29
3.8 Mapování GPIO portů na datovou sběrnici kamery optimalizováno s ohledem na množství křížení vodičů na desce plošného spoje pro platformu revize 2 29
3.9 Zapojení synchronizačních vodičů kamery 30
3.10 Zapojení mikrofonu 31
3.11 Zapojení mikrofonu na desce plošného spoje 32
3.12 Principiální struktura rozdělení úloh na procesorová jádra 33
3.13 Časování kamerové sběrnice 34
3.14 Sekvence událostí při čtení obrázku ze sběrnice do paměti 35
3.15 Dostupnost paměti na jednotlivých jádrech 35
3.16 Aplikace běžící na zařízení se zohledněním potřeby sdílet paměť 36
3.17 Schéma periferií používaných při komunikaci mezi jádry 37
3.18 Schéma datových toků při jednoduché implementaci přenášení paměti mezi jádry 38
3.19 Schéma datových toků při přenášení obrazových dat do pamětí v jiných jádrech . 40
3.20 Schéma komunikace mezi vysokoúrovňovou aplikací a aplikací ovládající kameru 41
3.21 Datagram zprávy pro přenos požadavku a odpovědi 42
3.22 Komunikační kanál mezi mikrokontrolérovými aplikacemi 43

3.23 Podpisový řetězec certifikátů, které používají zařízení Azure Sphere k připojení do Cloudu . . . 45
3.24 Zařízení, kterým IoT Hub důvěřuje při registraci certifikátů zařízení . . . 45
3.25 Zařízení, kterým IoT Hub důvěřuje při registraci certifikátu certifikační autority 46
3.26 Využité služby a tok informací při připojování zařízení ke cloudu . . . 47
3.27 Rozdělení cloudových aplikací . . . 48
3.28 Rozdělení univerzální a aplikační aplikace v cloudu . . . 48
3.29 Propojení cloudových služeb tvořících aplikaci monitorující parkoviště . . . 51
3.30 Komunikace mezi cloudem a zařízením za účelem získání krátkodobého přístupového tokenu k uložišti . . . 51
3.31 Komunikace mezi cloudem a zařízením za účelem notifikace o dostupnosti nového obrázku . . . 52

4.1 Mapování mezijaderného komunikačního protokolu na ISO/OSI model . . . 56
4.2 Zobrazení příkazu, kterým Visual Studio spouští OpenOCD ve Správci úloh . . . 62
4.3 Zobrazení adres program counteru a bufferu ve Visual Studiu . . . 63
4.4 Příkazy utility OpenOCD pro uložení obsahu paměti čipu do souboru. . . . 64
4.5 Operace Memory poolu . . . 68
4.6 Propojení GPIO expandéru . . . 69
4.7 Emulátor zařízení . . . 75
4.8 Webová aplikace zobrazující výstupy systému monitorujícího počet obsazených míst parkoviště z pohledu běžného uživatele . . . 76
4.9 Webová aplikace zobrazující výstupy systému monitorujícího počet obsazených míst parkoviště z pohledu administrátora . . . 77
4.10 Webová aplikace zobrazující možnost změnit monitorovaný objekt . . . 77

5.1 Test splnění časových požadavků datové sběrnice kamery . . . 80
5.2 Test mechanických kolizí součástek . . . 80
5.3 Výstup osciloskopu při zachytávání mluveného slova . . . 81

6.1 Vyrobené zařízení . . . 83

Seznam tabulek

3.1 Význam polí příkazů . . . 42
3.2 Přehled příkazů . . . 43
3.3 Přehled kódů odpovědí . . . 43

Seznam výpisů kódu

3.1 Zpráva ze zařízení obsahující požadavek o krátkodobý přístupový token k uložišti 52

3.2 Zpráva z cloudu obsahující odpověď na požadavek o krátkodobý přístupový token k uložišti . . . 52
3.3 Zpráva ze zařízení obsahující požadavek o krátkodobý přístupový token k uložišti 52
4.1 Funkce EINT_ConfigurePin [48] . . . 55
4.2 Struktura mapující zápisový a čtecí ukazatel ve sdílené paměti [51] . . . 57
4.3 Funkce EnqueueData [52] . . . 57
4.4 Funkce DequeueData [52] . . . 58
4.5 Atribut pro vygenerování spustitelného kódu do jiné sekce . . . 67

Mé poděkování patří Ing. Miroslavu Skrbkovi, Ph.D., vedoucímu práce, za vedení práce a cenné informace poskytované při jejím řešení. Děkuji také své rodině za velkou podporu a trpělivost při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 10. května 2020 . . . .

Abstrakt

Cílem práce bylo vyvinout bezpečný monitorovací systém, který snímá prostředí kamerou a obrazová data zasílá přes internet do cloudu, kde dochází k jejich zpracování. Práce se orientuje na bezpečnostní aspekty řešení.

V práci jsou popsány možnosti vývoje monitorovacího systému na různých platformách a vlastnosti, které z výběru platformy vyplývají. Je zde popsán návrh IoT zařízení založeného na platformě Azure Sphere připojeného ke cloudu Microsoft Azure, cloudová aplikace založená na službách cloudu Azure vyvinutá podle konceptu PaaS a uživatelské aplikace zobrazující výstupy z monitorovacího systému uživateli.

V práci jsou také popsány problémy spojené s vývojem na platformě Azure Sphere a možnosti jejich řešení.

Klíčová slova Internet of Things, monitorovací systém, kybernetická bezpečnost, cloudové zpracování obrazu, monitorování prostředí, pořízení snímku, analýza snímku, strojové vidění, Azure Sphere

Abstract

The goal of this thesis is to develop monitoring system which monitor environment using camera and transfers data of picture to cloud for further processing. Thesis targets security aspects of solution.

The thesis describes possible opportunities to develop monitoring system using different approaches and describes their properties that results from selection. Thesis describes development of IoT device based on Azure Sphere platform which is connected to Microsoft Azure cloud, cloud application based on Azure services developed according to concept PaaS and user application presenting outputs from monitoring system to user.

The thesis also describes some issues associated with development on the Azure Sphere platform and their possible solutions.

Keywords Internet of Things, monitoring system, cabersecurity, cloud image processing, evironment monitoring, image capturing, image analysis, computer vision, Azure Sphere

Shrnutí

Motivace

Motivací pro vznik této práce pro mě bylo zejména velké množství zpráv o prolomitelných IoT zařízeních v odborných médiích, které se objevují v posledních letech a velmi minimalistický přístup ke kybernetickému zabezpčení současných IoT zařízení ze strany některých výrobců. Motivací také bylo vyzkoušení si práce s moderní platformou, která implementuje řadu bezpečnostních konceptů nutných pro implementaci dlouhodobě bezpečného zařízení s využitím cloudových služeb ve veřejném cloudu.

Cíl práce

Cílem práce je zejména vyvinout moderní inteligentní monitorovací systém s vzdáleným zpracováním obrazu (v cloudu) s pomocí platformy, která nativně umožňuje bezpečně připojit dané IoT zařízení k internetu a zejména cloudu. Cílem praktické části práce je i navrhnout desku plošného spoje, umožňující sestavení výsledného zařízení a jeho zprovoznění. Cílem práce je popsat platoformu, její bezpečnostní vlastnosti, srovnat s alternativními možnostmi návrhu a implementace monitorovacího systému, popsat možnosti vzdáleného zpracování obrazu, srovnat s možnostmi lokálního zpracování obrazu a popsat návrh serverové (cloudové) strany monitorovacího systému. Cílem je také vyvinout ukázkovou aplikaci využívající vyinutý systém. Cílem práce je navrhnout systém dostatečně univerzálně a zároveň poskytnout konkrétní aplikaci ukazující možné použití takového monitorovacího systému.

Postup

V první fázi bude provedena analýza existujících řešení a platforem, s jejichž pomocí je monitorovací systém možno realizovat. Bude vyvinut firmware pro dané zařízení a cloudová aplikace, která bude provádět analýzu snímků zaslaných ze zařízení. Dále bude vyvinuta webová aplikace prezentující koncovému uživateli výstupy ze systému. V závěrečné fázi proběhne návrh a výroba desky plošného spoje, která umožní jednoduché a robustní připojení periferií k platformě Azure Sphere. Následně proběne jeho výroba, jeho osazení a zprovoznění.

Výsledky práce

Výsledkem práce je vyrobené robustní zařízení, umožňující monitorovat prostředí, snímky a data zasílat do cloudu, kde dochází k jejich zpracování s pomocí moderních cloudových služeb a dochází zde ke generování výstupů. Výsledkem práce je také vyrobená a otestovaná deska plošného spoje, která robustním a spolehlivým způsobem propojuje periferie zařízení k vývojové desce. Vyvinutý systém je velmi univerzální a výsledkem práce je i jedna ukázková aplikace, využívající tento monitorovací systém pro účel monitorování obsazenosti parkoviště.

Závěr

Práci a všechny její dílčí části se podařilo realizovat a otestovat jejich funkčnost. Podařil se navrhnout dostatečně robustní firmware zařízení, který zohledňuje aspekty vývojové platformy a efektivně využívá její dostupné prostředky. Podařilo se vyvinout cloudovou aplikaci, demonstrující možnosti monitorovacího systému a webovou aplikaci pro zobrazování výstupů systému. Také se podařilo navrhnout, osadit a otestovat desku plošného spoje, umožňující snadné a spolehlivé připojení periferií k vývojové platformě.

Seznam zkratek

AD	Active Directory
ADC	Analog to Digital Converter
API	Application Programming Interface
AzSphere	Azure Sphere
CSI	Camera Serial Interface
DoS	Denial of Service
DPS	Device Provisioning Service
DSI	Display Serial Interface
EF	Entity Framework
EINT	External Interrupt
FW	Firmware
GPIO	General Purpose Input Output
GPT	General Purpose Timer
HL App	High Level Appliaction
HS	Horizontal Sync
HSYNC	Horizontal Sync
HTTP	Hyper Text Transfer Protocol
HW	Hardware
I2C	Inter-Integrated Circuit
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
IoT	Internet of Things
IR	InfraRed
JSON	JavaScript Object Notation
MCLK	Main Clock
MIPI	Mobile Industry Processor Interface
MVC	Model View Controller
ORM	Object-Relational Mapping
PaaS	Platfrom as a Service
PCLK	Pixel Clock
PDM	Pulse Density Modulation
PWM	Pulse Width Modulation
RDP	Remote Desktop Protocol
REST	Representational state transfer
RGB	Red Green Blue
RT App	Realtime Appliaction
SAS	Shared Access Signature
SCCB	Serial Cameral Control Bus
SMD	Surface Mount Device
SoC	System on Chip
SSH	Secure Shell
SW	Software
TCL	Tool Command Language
TCM	Tightly Coupled Memory
VM	Virtual Machine
VS	Vertical Sync
VSYNC	Vertical Sync
WPF	Windows Presentation Foundation

Kapitola 1
Úvod

Monitorovací systémy se dnes používají na řadě míst k monitorování různých objektů a předmětů různými způsoby. Během let vznikly spousty různých monitorovacích systémů například ke sledování různých veličin nebo zabezpečení objektů. Takové systémy jsou již běžné a komerčně dostupné. Ne vždy však také dobře zabezpečené. V této práci bude popsán univerzální systém vytvořený jako IoT (Internet of Things) zařízení sloužící k pořizování obrazových dat a přenos těchto obrazových dat do cloudu. Tam dojde k jejich následnému vzdálenému zpracování. V práci lze vidět použití moderní vývojové platformy Azure Sphere, která umožňuje zajistit velmi bezpečné připojení koncového zařízení jako je monitorovací systém k internetu a umí zajistit důvěryhodné spojení s ostatními cloudovými službami ve veřejném cloudu. V rámci praktické části práce vznikne i deska plošného spoje, která umožní jednoduché a robustní připojení periferií k platformě.

Celou prací se bude prolínat rozdělení na čtyři samostatné dílčí části řešení. Jedná se o řešení HW, který je rozdělen na návrh logického propojení HW periferií a návrh desky plošného spoje. Další významnou části práce je návrh a vývoj firmware pro platformu Azure Sphere. Poslední částí práce je aplikace v cloudu a uživatelská aplikace sloužící k prohlížení výstupů z monitorovacího systému.

Práce je rozdělena do čtyř kapitol. První kapitola obsahuje analýzu problematiky a popis již existujících monitorovacích systémů, možnosti implementace zařízení monitorovacího systému pomocí různých moderních technologií a další. Druhá kapitola práce se zabývá popisem návrhu samotného řešení. V této kapitole se diskutují možnosti implementace konkrétních detailů systému a zejména se srovnává s alternativními možnostmi návrhu řešení. Třetí kapitola práce pak popisuje implementaci samotného řešení a diskutuje konkrétní problémy, možnosti jejich řešení a jak byly v rámci práce vyřešeny. V poslední části práce je popisováno testování řešení.

Práce je rozsahově zaměřená spíše směrem k hardware a méně k software běžícímu v cloudu. Velký důraz v práci bude kladen na kybernetickou bezpečnost řešení.

Kapitola 2
Analýza

První kapitola této práce se věnuje samotné definici pojmu monitorovací systém a několika aspektům, které je dobré prozkoumat ještě před samotným řešením. Jedná se například o analýzu dostupných existujících řešení podobných problémů a jejich vlastností. Dále pak budou popsány platformy s jejichž pomocí je možné monitorovací systém implementovat. Velký důraz je kladen na bezpečnostní aspekty a jednotlivé platformy umožňující implementovat monitorovací systém.

2.1 Monitorovací systém obrazových dat

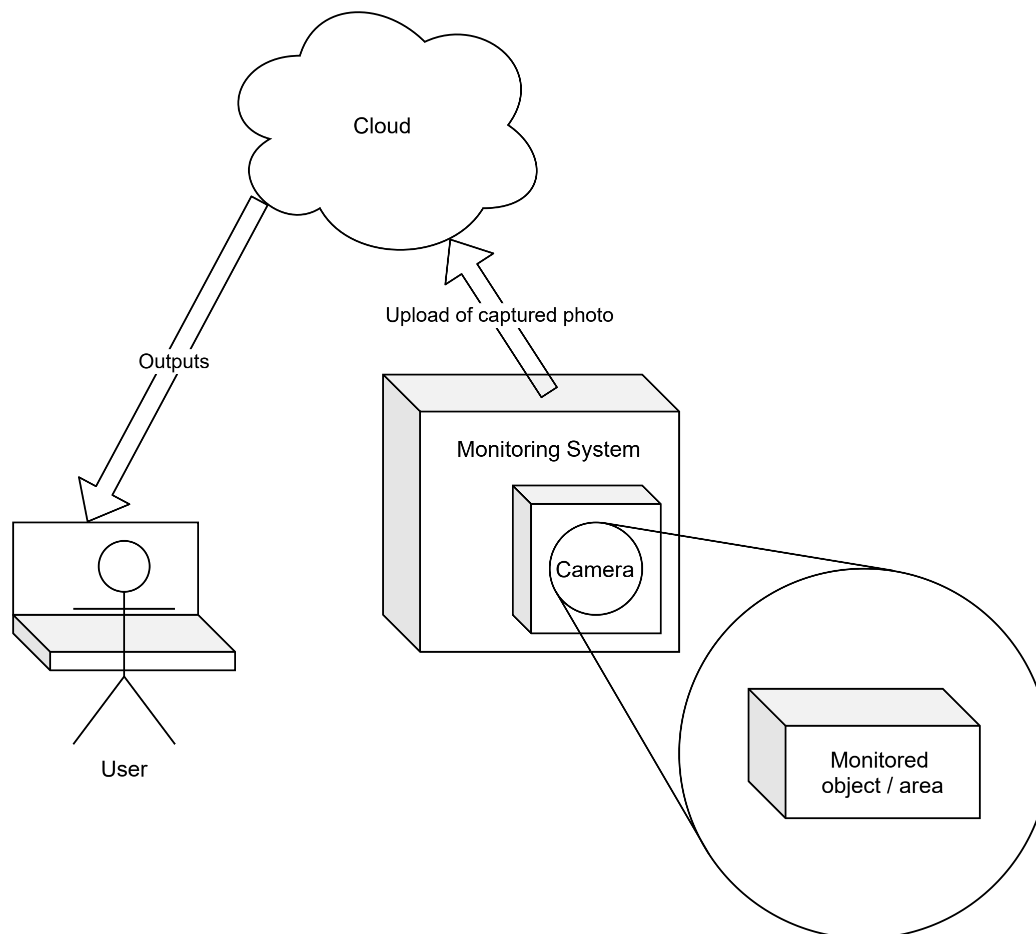
Monitorovací systém, kterým se bude tato práce zabývat monitoruje určitou část reálného světa s pomocí kamery a případně dalších senzorů. Primární jsou však obrazové informace. Tento monitorovací systém nezpracovává data lokálně, ale zpracovává je vzdáleně v cloudu. Případně tento systém může část dat zpracovávat lokálně, ale takové zpracování by nemělo být náročné. Zařízení zpracovává vzdáleně zejména obrazová data, která jsou poměrně náročná na analýzu. Ta zasílá pomocí sítě internet do cloudu k jejich dalšímu zpracování. Samotné zařízení monitorovacího systému neposkytuje uživateli žádnou zpětnou vazbu ani výstup. Schéma monitorovacího systému popisovaného v této práci vizualizuje následující obrázek 2.1.

2.2 Existující řešení

Existuje řada monitorovacích systémů monitorující různé vlastnosti za různými účely. I přestože se tato práce věnuje zejména systémům monitorujícím pomocí obrazových dat, tak to není nutný předpoklad pro klasifikaci daného řešení jako monitorovací systém. Je možné najít například systém [1], který *„měří, zaznamenává a vyhodnocuje fyzikální a chemické hodnoty“*.

Nalezneme také monitorovací systém parkoviště [2], který se velmi blíží monitorovacímu systému popisovanému a implementovanému dále v této práci. Systém s pomocí kamery a technologií umělé inteligence umí analyzovat a vyhodnocovat například počet neobsazených parkovacích míst a zobrazit toto číslo na informační tabuli, měřit délku stání a kontrolovat, zda řidiči nepřekračují maximální povolenou dobu parkování. Mimo jiné *„Systém dokáže upozornit na nevhodné parkování osobních automobilů na místech pro autobusy a nákladní soupravy.“*

Dalším velmi významným a rozšířeným typem monitorovacího systému jsou bezpečnostní systémy. Jedná se většinou o systém složený z kamer schopných zasílat obraz po protokolu IP a zařízení schopné záznamy z těchto kamer dlouhodobě uchovávat. Většinou se však nejedná o příliš komplikované a *inteligentní* systémy, nicméně se jedná o ověřené a dlouhodobě využívané systémy.



Obrázek 2.1 Obecné schéma monitorovacího systému

Většina těchto systémů je vyvíjena pro konkrétní specifický úkol, viz. první dva zmíněné systémy. I proto je cílem této práce navrhnout dostatečně univerzální zařízení, aby se samotná konkrétní aplikace a použití daného systému dalo v budoucnu změnit nebo rozšířit.

2.2.1 Bezpečnost existujících systémů

Každý systém připojený k internetu je nějakým způsobem náchylný na pontenciální kybernetické útoky prováděné vzdáleně přes internet. Stejně tak všechny monitorovací systémy připojené k internetu čelí těmto útokům. S rozmachem IoT zařízení se tento problém stupňuje. Řada těchto zařízení a zejména těch starších trpí řadou bezpečnostních chyb. Jedná se jak o zranitelnosti v software, které mohou umožnit útočníkům například převzít kontrolu nad zařízením a provádět na něm libovolné příkazy, nebo se velmi často jedná o chyby, které si uživatel zařízení (obvykle administrátor) zaviní sám. Jedná se například o nevhodně volené (slabé) heslo pro přístup do administrace zařízení, díky čemuž mohou útočníci převzít kontrolu nad zařízením prolomením (nebo častěji rychlým uhodnutím) slabého hesla. Jde nejen o chyby, které vznikly při vývoji zařízení (programování), ale také o chyby, při kterých má vývojář velmi omezené možnosti, jak je eliminovat. Skupiny chyb (zaviněné vývojářem × zaviněné uživatelem) však nejsou disjunktní.

Příkladem chyby (historie ukazuje že časté) je nevhodně nastavené výchozí heslo, které nastavuje vývojář a možnost eliminovat tuto chybu má uživatel, který to však často neudělá.

Vzhledem k množství potenciálně zranitelných zařízení, roste také zájem útočníků masivně ovládnout tato zařízení a přirozeně roste i potřeba tato zařízení bránit. V posledních letech tak vzniklo nepřeberné množství analýz takových IoT zařízení, jako jsou například [3], která popisuje i další metody, kterými se útočí na IoT zařízení nebo třeba práce [4].

Kompromitace zařízení tvořící monitorovací systém může mít na systém následky, jako je nedostupnost monitorovaných dat nebo jejich únik. Nedostupnost dat může působit velké problémy bezpečnostním systémům od nichž se očekává, že monitorují prostředí nepřetržitě za každých okolností. Útočník je může vyřadit z provozu a potenciálně provést „útok“ na původně monitorovaný objekt v reálném světě. Útokům vyřazujícím zařízení z provozu se říká DoS (Denial of Service), a přestože se většinou týkají webových aplikací, tak je není dobré podceňovat ani u jiných systémů včetně monitorovacích systémů. Podobně může být velkým problémem klasifikován únik citlivých informací. Příkladem může být útok rekonfigurující zařízení, aby monitorovaná data (např. snímek z monitorovacího systému) systém zaslal jinému nežádoucímu příjemci.

Při hodnocení bezpečnosti zařízení je třeba brát v potaz také údržbu v budoucnu. Přístup, že zařízení bude vyvinuto a dál se o něj nikdo nebude starat je z pohledu bezpečnosti silně nevhodný. Vývojář zařízení by měl být schopen reagovat na objevené zranitelnosti v firmware jeho zařízení, a také by ideálně měl být schopen distribuovat aktualizace těmto zařízením. Realita však ukazuje, že řada zařízení od výroby nikdy nedostane aktualizaci. Tento problém se netýká jen přímo IoT zařízení, ale i některých mobilních telefonů, které výrobci od určitého okamžiku přestanou podporovat a vydávat pro ně záplaty [5].

Proto je cílem této práce navrhnout dostatečně bezpečný systém na platformě, která umožňuje řešit velké množství bezpečnostních problémů bez nutnosti spolupráce programátora vyvíjejícího daná zařízení a tedy značně snižuje riziko vzniku bezpečnostní chyby. Podobně je nutné řešení navrhovat s ohledem na redukci možnosti degradace úrovně bezpečnosti uživatelem. Například navrhnout řešení využívající vzdálenou konfiguraci, místo vyžadování konfigurace hesla k lokálně dostupné administraci. To jsou přesně ta místa, kde jsou současná řešení mnohdy velmi zranitelná.

2.3 Platforma pro implementaci zařízení monitorovacího systému

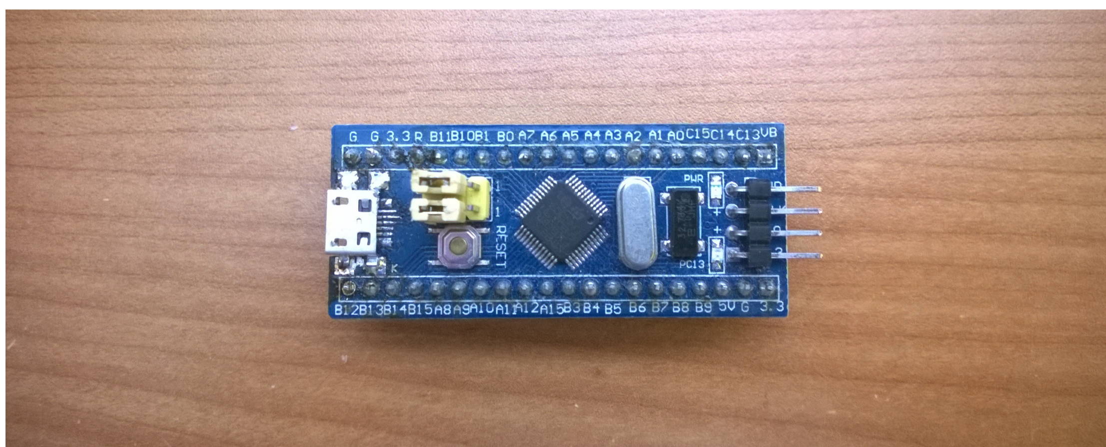
Hardware monitorovacího zařízení, kterým se bude tato práce zabývat, je složen z kamery, výpočetní jednotky a externího zařízení nebo systému na zpracování a analyzování dat z této kamery a zařízení. Hardware takového zařízení lze navrhnout různými způsoby a výsledné řešení bude mít různé vlastnosti. Při výběru HW je potřeba zohlednit řadu aspektů jako je jeho výkon, dostupná paměť, možnost připojení k internetu a bezpečnost. Dalším aspektem pro výběr vhodné platformy může být požadavek na zpracování obrazových dat v reálném času. Pokud by byl stanoven požadavek na vysokorychlostní zpracování obrazu s rychlostí v řádu několika desítek snímků za sekundu, je potřeba se omezit buďto na různé velmi výkonné platformy, jako jsou počítače s procesory architektury x86, případně úlohy vyřešit s pomocí hradlových polí. V této práci se však budu věnovat monitorovacím systémům, které pořizují snímek sporadicky, například jednou za minutu. Dalším parametrem vhodným k zohlednění může být podpora pro hardwarovou akceleraci pro ovládání rozhraní kamery. Volba hardware řešení také poměrně výrazně ovlivňuje bezpečnost z pohledu software. IoT zařízení by mělo mít možnost vzdálené aktualizace firmware, pro případ objevení bezpečnostní chyby, ať už v kódu samotné aplikace, tak například v kódu knihovny (např. kryptografické knihovny), kterou aplikace využívá. Na některých platformách se toto zajišťuje poměrně obtížně.

V následujících sekcích bude popsáno několik možností, na jakých platformách lze monitorovací systém vyvíjet a popíšu výše popsané aspekty.

2.3.1 Mikrokontroléry

Jedno z řešení, jakým by mohl být hardware monitorovacího zařízení realizován, je pomocí mikrokontroléru. Toto řešení by mělo hodně (často zásadně) omezujících podmínek na volbu mikrokontroléru, nicméně za určitých okolností může mít velmi zajímavé vlastnosti. Řešení implementované pomocí mikrokontroléru bude velmi flexibilní pro vývojáře, který si může vše navrhnout a naimplementovat dle libosti. Nevýhodou je, že to udělat musí a má jen omezené možnosti využití již existujícího softwaru, který je například velmi snadno dostupný na jiné platformě. Například si bude vývojář muset implementovat vlastní IP stack nebo integrovat existující IP stack jako je lwIP, což je součást, kterou by na industriálním počítači s procesorem architektury x86 a plným operačním systémem jako je Linux určitě řešit nemusel. Při výběru řešení tohoto typu je potřeba mít na mysli některé nutné podmínky, které je potřeba klást na vybíraný mikrokontrolér. Jednou z podmínek je velikost operační paměti. Ta musí být schopna uchovat celý obrázek, ať už se jedná o bitmapu nebo nějakým způsobem komprimovaný obrázek, pokud to související kamera podporuje. Dále také operační paměť musí být schopna uchovat všechny potřebné struktury pro funkčnost obsluhy internetových protokolů, což může v závislosti na komplexnosti zvoleného IP stacku působit taktéž značné problémy. Naopak výhodou tohoto řešení může být poměrně vysoká úroveň determinističnosti aplikace a schopnost provést poměrně detailní analýzu dokazující korektnost takto vyvinuté aplikace a ověření, že veškeré části kritické na časování (v případě tohoto monitorovacího systému se jedná zejména o obsluhu poměrně rychlé sběrnice mezi mikrokontrolérem a kamerou) jsou za každých okolností splněny. S rostoucí komplexitou platforem se tato analýza stává složitější. Jako velmi výhodné se nabízí použití 32bitových mikrokontrolérů, které dokáží ve srovnání s populárními 8bitovými mikrokontroléry zpracovat 4× víc dat během jedné instrukce. V dnešní době běžně dostupné 32bitové mikrokontroléry dokáží být navíc taktovány na podstatně vyšší frekvence, díky čemuž dosahují mnohem vyššího výkonu ve srovnání s 8bitovými mikrokontroléry.

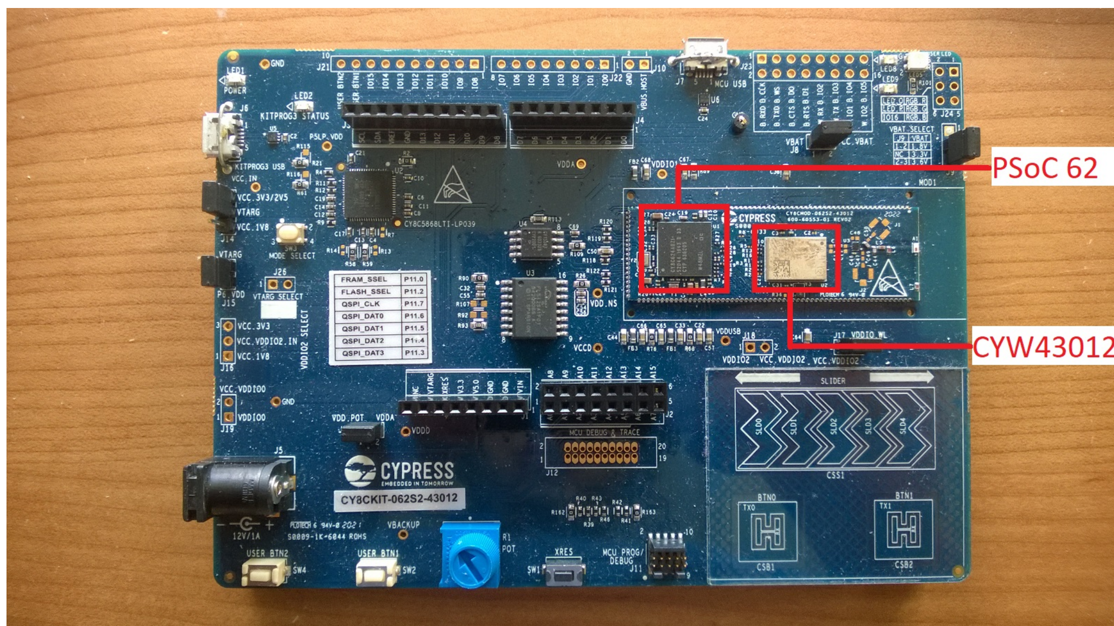
Následující obrázek 2.2 ukazuje malou a jednoduchou vývojovou desku s 32bitovým mikrokontrolérem STM32F103C8T6.



Obrázek 2.2 Vývojová deska s mikrokontrolérem

Některé mikrokontroléry jako např. mikrokontroléry řady RX72N dokumentované v [6] od firmy Renesas nebo řada STM32F4 [7] od ST navíc mají periferii pro obsluhu sběrnice, která se používá pro komunikaci s některými kamerami, díky čemuž dokážou vyřešit poměrně zásadní problém s vysokorychlostní sběrnicí bez nutnosti její obsluhy v software a značně tak zjednodušují návrh firmware takového zařízení. Pokud zvolený mikrokontrolér danou periferii nemá, je potřeba počítat s velkou výkonovou rezervou vyžadovanou pro softwarovou implementaci a ovládání potřebné sběrnice.

Další volbou pro výběr mikrokontrolérů může být podpora konektivity. Řada dostupných mikrokontrolérů obsahuje řadič pro sběrnici Ethernet. Ethernet podporuje většina výrobců high-end mikrokontrolérů, ale pouze obvykle v nejvyšších modelových řadách. V předchozím odstavci zmíněná řada mikrokontrolerů RX72N [6] od firmy Renesas je příkladem platformy, která má vhodné kamerové rozhraní i podporu pro sběrnici Ethernet. Velmi podobně jsou na tom například i vyšší modely mikrokontrolérů v řadě STM32F4 [7] od ST. Existují i mikrokontroléry s podporou Wi-Fi konektivity, která může být pro monitorovací systém tohoto typu vhodnější. Tento typ konektivity již podporuje výrazně méně výrobců mikrokontrolérů. Například zmíněná řada RX72N ani STM32F4 [7] Wi-Fi konektivitu nepodporují. Příkladem mikrokontroléru s podporou Wi-Fi může být například ESP32 [8] od firmy Espressif. Využití mikrokontroléru bez podpory Wi-Fi konektivity však nemusí být nutně velký problém, protože tento typ komunikace lze velmi snadno zajistit pomocí externího modulu. Někteří výrobci takové řešení přímo nabízí. Například Cypress nabízí vývojovou desku PSoC® 62S2 Wi-Fi BT Pioneer Kit (CY8CKIT-062S2-43012) [9], kde je takový princip využit. MCU je zajištěno mikrokontrolérem řady PSoC 62 a Wi-Fi konektivita pomocí externího čipu CYW43012, které jsou vzájemně propojený sběrnicí SDIO. Nicméně použitý mikrokontrolér nemá rozhraní pro HW akcelerované čtení dat z kamery do paměti. Vývojovou desku s mikrokontrolérem PSoC 62 a čipem CYW43012 s podporu Wi-Fi konektivity ukazuje následující obrázek 2.3.



Obrázek 2.3 Vývojová deska s mikrokontrolérem PSoC 62 a čipem CYW43012

Podobně jako navrhnul řešení Cypress, si implementátor může navrhnout svoje vlastní zařízení například s mikrokontrolérem z řady STM32F4 a Wi-Fi modulem s čipsetem CYW43012.

V závislosti na komplexnosti mikrokontroléru může být poměrně obtížné vyřešit některé bezpečnostní problémy. Většina mikrokontrolérů umožňuje uzamknout firmware proti čtení ze zařízení a některé mikrokontroléry umožňují i uzamknout flash paměť proti zápisu, aby se zabránilo přepisu firmware škodlivým firmware. Některé mikrokontroléry navíc umožňují Secure Boot, což v mikrokontrolérovém podání znamená, že mikrokontrolér před spuštěním firmware zkontroluje obsah flash paměti a digitální podpis. Mnohem obtížnější je však na mikrokontroléru řešit vzdálenou aktualizaci firmware a pro vývojáře to většinou znamená, implementovat si danou funkcionalitu celou ručně. Existují bootloadery pro mikrokontroléry, ty ale umožňují lokální přehrání firmware např. přes USB nebo UART. Obvykle nepodporují přeprogramování

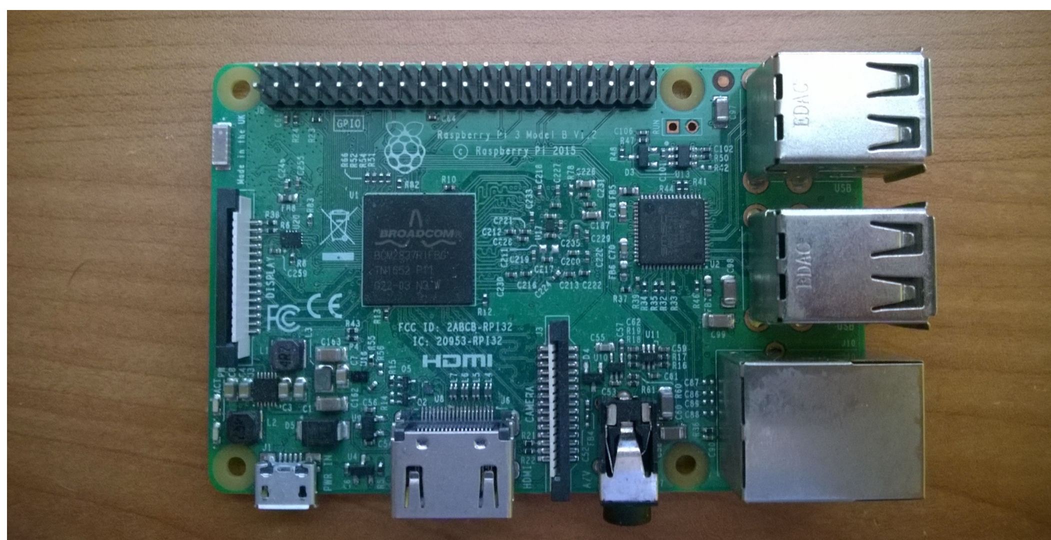
flash paměti čipu vzdáleně přes internet.

Narozdíl od všech dále popsaných možností řešení, toto řešení exceluje v možnostech optimalizace spotřeby energie zařízením. Mikrokontroléry samotné mívají poměrně nízkou spotřebu a obsahují efektivní mechanizmy, jak spotřebu redukovat pomocí režimů spánku, kdy část čipu po přechodu do takového režimu je zcela vypnutá a spotřeba dané části čipu je značně redukována. Mikrokontroléry také umí poměrně dobře jednotlivé periferie systému libovolně vypínat a zapínat.

2.3.2 Jednodeskové počítače

Další z možných řešení, které se nabízí je využití jednodeskového počítače jako hlavní platformy. Takovým počítačem může být například Raspberry Pi 4, Odroid-N2 nebo třeba Nvidia Jetson Nano. Řešení pak lze postavit vyvinutím standardní spustitelné aplikace, která poběží nad operačním systémem. Jednodeskové počítače narozdíl od mikrokontrolérů mají dostatečný výkon na provoz takového operačního systému i samotné aplikace. Poměrně vysoký výkon těchto zařízení také umožňuje připojení mnohem pokročilejších kamer nebo také kamer, připojených k systému přes rozhraní jako je USB. Připojení kamery přes USB je možné i na některých mikrokontrolérech, ale znamená to implementace několikaúrovňového protokolu nad USB a poměrně komplikované zpracování dat z těchto protokolů. Navíc pro přenos obrazových dat je potřeba USB ve variantě High Speed (480 Mib/s), ale většina mikrokontrolérů implementuje pouze USB ve variantě Full Speed (12 Mib/s). U jednodeskových počítačů toto není problém. Některé jednodeskové počítače dokonce obsahují i řadič USB ve variantě Super Speed, který umožňuje přenášet mnohem víc dat než je potřeba pro přenos obrazových dat z kamery. USB ve variantě High Speed podporují takřka všechny jednodeskové počítače. Na jednodeskovém počítači by takové řešení mohlo mít význam i z toho důvodu, že nízkoúrovňové problémy sběrnice jsou již jsou vyřešeny v rámci operačního systému a existujících aplikací.

Následující obrázek 2.4 ukazuje jednodeskový počítač Raspberry Pi 3B, který by mohl být použit k řešení úlohy.



Obrázek 2.4 Jednodeskový počítač Raspberry Pi 3B

Jako výhodné se nabízí použití jednodeskových počítačů s různými akcelerátory. Příkladem mohou být jednodeskové počítače Nvidia Jetson Nano, které v sobě vedle procesoru architektury Arm obsahují grafický akcelerátor pro akceleraci grafických a AI operací. Tato vlastnost může být velmi výhodná v případě koncepce lokálního zpracování obrazu, protože takové zařízení

má dostatek výkonu na kompletní zpracování obrazových dat i s velkou snímkovací frekvencí v reálném čase.

Nevýhodou jednodeskových počítačů (a zejména těch výkonných) je spotřeba a malé možnosti její optimalizace. Procesory těchto zařízení obvykle umožňují dynamicky nastavovat pracovní frekvenci, ale vzhledem k jejich koncepci a koncepci operačního systému prakticky není narozdíl od mikrokontrolérů možnost je pozastavit úplně. Se spotřebou a výkonem souvisí nutnost zařízení chladit. Většinu jednodeskových počítačů stačí chladit pasivně, ale stejně je potřeba zajistit alespoň minimální průtok vzduchu. U mikrokontrolérů tento problém není.

Řešení pomocí jednodeskového počítače má mnoho výhod v oblasti bezpečnosti. Podstatná část softwarových záležitostí jako je síťový stack je implementovaná v operačním systému. Běžně používané operační systémy jsou navíc poměrně rozšířené a lze předpokládat, že prošly důkladnými bezpečnostními analýzami. Lze očekávat že například síťový stack v Linuxu je bezpečnější než například IP stack lwIP zmíněný v sekci o mikrokontrolérech. Aktualizace softwaru, který běží na operačním systému je také mnohem jednodušší a lze dělat vzdáleně pomocí protokolu SSH (Secure Shell). Podobně je podpora vytváření šifrovaných spojení a ověřování potřebných certifikátů vyřešena již v operačním systému. Některé bezpečnostní požadavky se však zajišťují naopak hůře. Příkladem může být zabránění úniku firmware. Při řešení pomocí mikrokontrolérů je řešení tohoto problému obvykle poměrně jednoduché, protože většina mikrokontroléru umožňuje zablokovat rozhraní pro programování a ladění čipu. Případně umožňuje blokovat čtení z pamětí daného zařízení přes toto rozhraní. U jednodeskových počítačů je toto však obtížné a někdy i nemožné. Některé platformy umožňují šifrování disku, ale toto řešení přináší nutnost zadávat klíč k dešifrování obsahu při každém bootu zařízení, což je pro použití v embeded systému velmi omezující. Podobně je poměrně obtížné zajistit verifikaci integrity při bootování. Podpora technologie obvykle zvané Secure Boot je výsada platforem s procesory architektury x86 používající UEFI BIOS, ale například jednodeskové počítače s architekturou Arm toto většinou neumí. Prosazování této technologie na platformách Arm probíhá velmi pozvolna a lze pozorovat spíše pouze u mikrokontrolérů založených nad jádry Cortex-M, kde si to výrobci implementují proprietárně a sami. Část těchto požadavků lze částečně vyřešit pomocí vlastních kontrol v software a mechanickými prvky omezujícími fyzický přístup k zařízení.

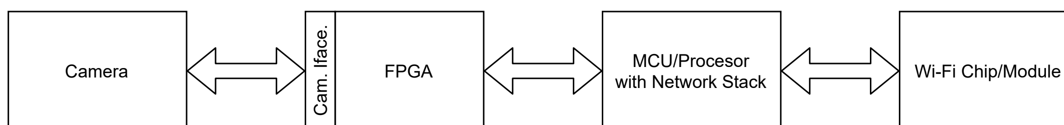
2.3.3 Hradlová pole

Poslední významný koncept, který je v této kapitole popsán je přístup, pomocí kterého by byl monitorovací systém implementován pomocí programovatelného hradlového pole (FPGA). Řešení s využitím hradlových polí by umožnilo implementovat takřka libovolný řadič pro rozhraní ke kameře. Omezením by byla rychlost paměti RAM, na níž jsou kladeny nároky zejména na rychlost (aby se dalo stihnout uložit proud dat z kamery do paměti) a kapacity (aby se tam obrázek vešel), nicméně u současných FPGA by splnění těchto požadavků nemělo být problém. Na FPGA se dají implementovat i kompresní algoritmy, které dokážou přijímaný proud takřka v reálném čase komprimovat a snížit tak nároky na paměť RAM.

Existují i výzkumy v této oblasti jako např [10] a [11], které zmiňují zejména velký výkon řešení postaveného nad FPGA.

Nevýhodou řešení pomocí hradlových polí je absence jakékoliv možnosti komunikace s internetem. Typicky zde není žádný procesor, na kterém by mohl běžet kód schopný komunikovat přes internet a obsluhovat síťový stack. Nejjednodušší možný návrh tak počítá s přidáním doplňujícího čipu, kterým je mikrokontrolér nebo nějaký jiný procesor a ten by zajišťoval komunikaci s internetem a cloudem. Obecně struktura takového řešení pomocí FPGA by mohla vypadat jako na následujícím diagramu 2.5.

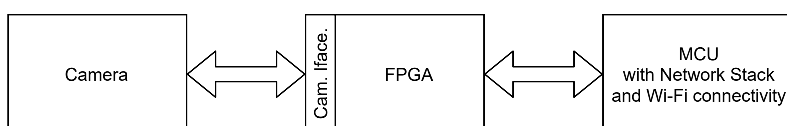
Řešení tak vyžaduje volbu dalších externích čipů a procesorů. Volbu Wi-Fi čipu lze udělat velmi podobně jako bylo popsáno v kapitole Mikrokontroléry. Jako mikrokontrolér nebo procesor obsluhující komunikaci s Wi-Fi lze volit takřka libovolný mikrokontrolér s dostatečným množstvím perzistentní paměti pro uložení kódu síťového stacku a dostatečným množstvím paměti



Obrázek 2.5 Schéma možného řešení pomocí FPGA

RAM pro uložení všech potřebných stavových informací. Také je vhodné, aby vybraný procesor/-mikrokontrolér obsahoval periferii pro (ideálně rychlou) komunikaci s Wi-Fi modulem a nějaké sériové rozhraní pro komunikaci s FPGA.

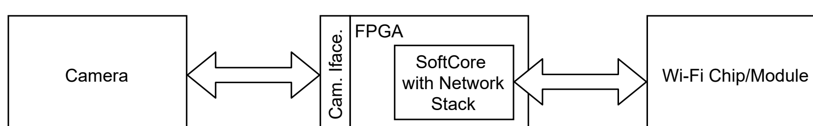
Řešení pomocí FPGA však přináší možnosti výběru čipů, které obsahují jednotlivé části sloučené do jednoho čipu. Nabízí se možnost sloučení mikrokontroléru obsluhující síťový stack a Wi-Fi modulu. Řešení by vypadalo jako na následujícím diagramu 2.6.



Obrázek 2.6 Schéma možného řešení pomocí FPGA a čipu s podporou obsluhy síťového stacku

Příkladem čipu, který umožňuje spustit kód síťového stacku a zároveň podporuje Wi-Fi konektivitu je ESP32 od firmy Espressif [8]. Důležitým aspektem řešení je možnost aktualizace firmware tohoto doplňujícího čipu pro případ objevení bezpečnostní zranitelnosti v kódu, který na tomto čipu běží. Důvodem je, že existují i Wi-Fi moduly které neumožňují uživatelsky aktualizovat firmware. To ale není případ zmíněného ESP32, které je tak vhodným kandidátem pro zajištění konektivity k FPGA.

FPGA umožňují i další možnost kombinace. Tím že uvnitř FPGA jde navrhnout takřka libovolný logický obvod, tak jde uvnitř FPGA navrhnout samozřejmě i procesor. Procesory implementovány uvnitř FPGA existují a nesou speciální označení SoftCores. Řešení by vypadalo jako na následujícím schématu 2.7.



Obrázek 2.7 Schéma možného řešení pomocí FPGA a SoftCore

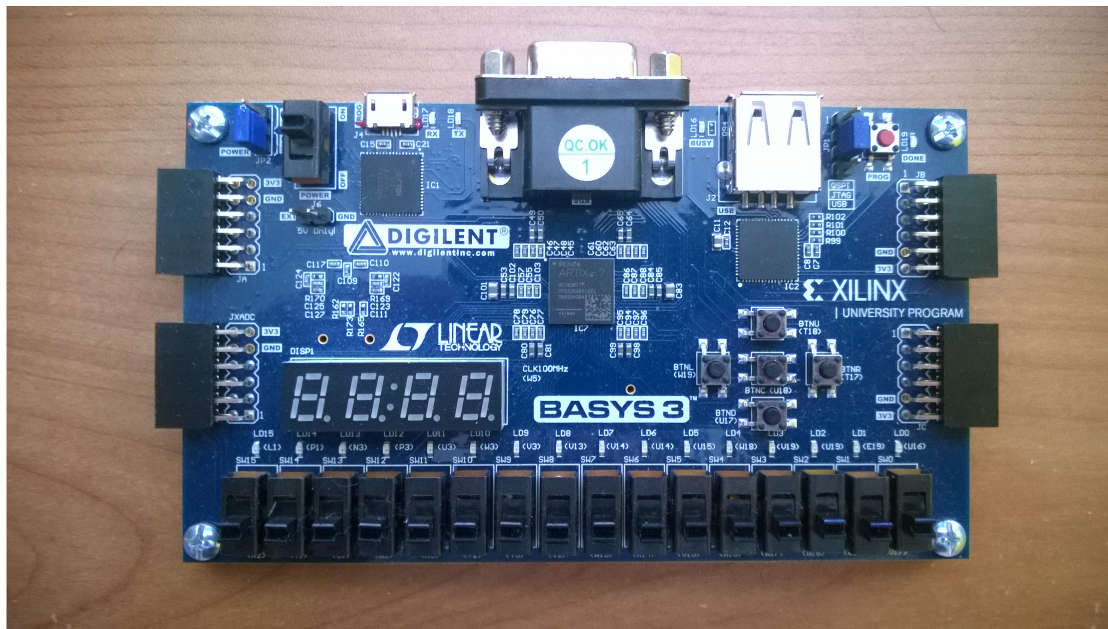
Existují komerční SoftCore jako jsou jádra Arm Cortex-M1 a M3 [12], které implementují instrukční sadu Arm (verze 6 v případě jádra Cortex-M1 a verze 7 v případě jádra Cortex-M3) a lze je integrovat v FPGA aplikacích v rámci omezené licence. Existuje komerčně dostupné jádro S8051XC3 od Xilinx [13], které implementuje instrukční sadu původních procesorů 8051, ale v rychlejší podobě. Existují také komerční i nekomerční jádra implementující otevřenou instrukční sadu RISC-V. Příkladem takového SoftCore může být jádro The Bonfire Processor [14]. Všechny tyto jádra dokážou obsluhovat síťový stack, který by na nich v tomto případě běžel. Při návrhu je potřeba počítat s tím, že tato jádra vyžadují část prostoru FPGA.

Při řešení pomocí FPGA je potřeba brát v potaz bezpečnostní aspekty. Je potřeba zajistit podporu pro vzdálenou aktualizaci firmware pro případ objevení bezpečnostní chyby v něm, což by ale vzhledem k existenci procesorového jádra v návrhu nemusel být problém. Podobně

je potřeba zajistit integritu firmware všech částí aplikace, ale to by mělo být taktéž poměrně snadno realizovatelné, protože FPGA, procesorové jádro (ať už MCU nebo SoftCore) i Wi-Fi modul většinou obsahují mechanizmy, jak zabránit přečtení firmware ze zařízení a zabránit jeho kompromitaci nahráním škodlivého firmware.

Řešení s FPGA narozdíl od řešení čistě s pomocí MCU má podobně jako řešení využívající jednodeskový počítač omezené možnosti redukce spotřeby zařízení, ale redukce je možná. Existuje možnost redukce hodin dynamicky podle potřeb aplikace a popisuje to například práce [15].

Příkladem hradlových polí, s pomocí kterých by bylo možné monitorovací systém implementovat, jsou například hradlová pole od společnosti Xilinx jako je Xilinx Artix-7 XC7A35T-1CPG236C na vývojové desce Basys 3, která je ukázána na následujícím obrázku 2.8.



Obrázek 2.8 Vývojová deska Basys 3

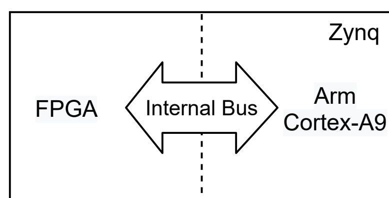
2.3.4 Prolínaní konceptů

Existují i platformy kombinující některé z výše uvedencýh konceptů. Jedná se například o platformy, které v jednom čipu kombinují aplikační procesor s hradlovým polem a platformu kombinující procesorová a mikrokontrolérová jádra. Samotný problém k některému z těchto řešení vybízí, protože řešení vyžaduje připojení ke kameře a zároveň k internetu, což jsou odlišné problémy vyžadující také odlišný přístup řešení. Pro čtení dat z kamery se více hodí hradlová pole nebo rychlé mikrokontroléry (případně mikrokontroléry s periferií pro danou sběrnici), ale pro přístup k internetu se zase hodí spíše procesorová řešení s operačním systémem. Platformy kombinující koncepty umožňují řešit tyto „kombinované problémy“ efektivněji, ve srovnání s jednoduchými koncepty, kde část problému bude řešena méně efektivně ve srovnání s jinou části problému.

2.3.5 Platforma Xilinx Zynq

První popsaný koncept, který kombinuje hradlové pole s procesorovým jádrem je platforma Xilinx Zynq [16]. Koncept vizualizuje následující obrázek 2.9.

Potenciální řešení na této platformě by bylo navrženo způsobem, že komunikaci s kamerou by zajišťovala část implementovaná na hradlovém poli a na procesorovém jádru by běžel operační

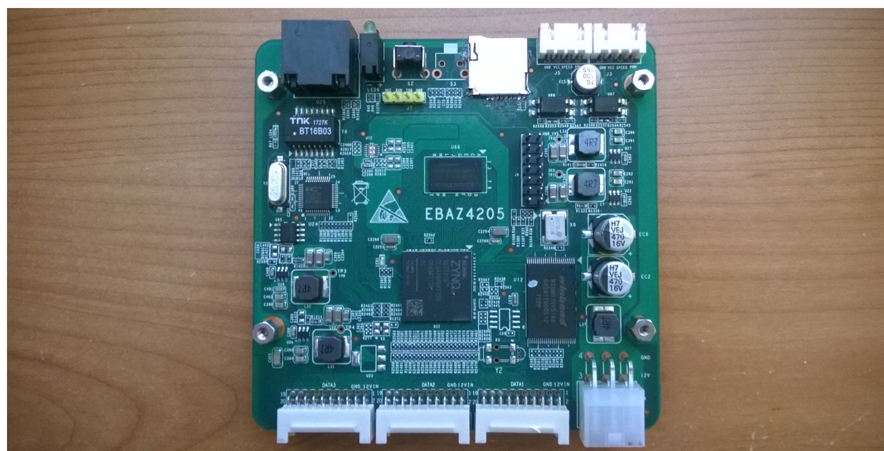


■ **Obrázek 2.9** Platforma Xilinx Zynq

systém (nejspíše Linux), na kterém by běžela aplikace, která by s pomocí ovladače pro operační systém (ten by bylo také nutné vyvinout) ovládala periferii na FPGA a získávala zachycený obrázek z FPGA.

Tento přístup by měl zajímavé vlastnosti a efektivně využíval jak část hradlového pole, tak procesorovou část systému. Koncept dědí výhody z obou přístupů, které kombinuje. Stejně tak koncept dědí i některé nevýhody. Z pohledu bezpečnosti je velmi výhodné, že komunikaci s internetem zajišťuje plnohodnotný operační systém s ověřeným síťovým stackem, možností snadné aktualizace software pro případ záplatování v budoucnu odhalených bezpečnostních děr a také má skvělé možnosti co se týče podpory kamer, protože na FPGA jde naimplementovat řadič i pro velmi rychlá kamerová rozhraní. Koncept také dědí některé nevýhody, jako je velmi omezená ochrana proti přečtení firmware nebo malé možnosti redukce spotřeby. Například firmware (což je v tomto případě některá z distribucí Linuxu) je uložen vždy v externí paměti, ze které jde snadno přečíst v případě získání fyzického přístupu k zařízení, který ale v případě monitorovacího systému nelze vyloučit. Nezáleží, jestli bude operační systém uložen na SD kartě, v QSPI, NOR nebo NAND paměti, protože všechny dokáže případný útočník přečíst.

Příkladem vývojové desky s čipem založeným na platformě Zynq je například vývojová deska Digilent Zybo nebo deska EBAZ4205, která je zobrazena na následujícím obrázku 2.10.

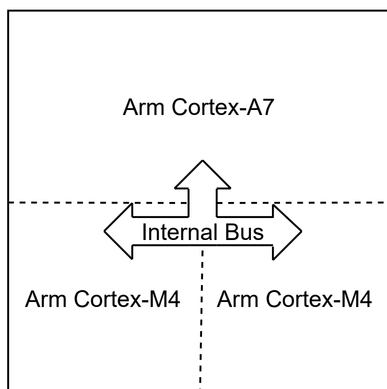


■ **Obrázek 2.10** Vývojová deska s čipem Xilinx Zynq

2.3.6 Azure Sphere

Druhým příkladem prolínaní konceptů je platforma Azure Sphere [17]. Tato platforma obsahuje jedno aplikační jádro (Arm Cortex-A7) a dvě nezávislá mikrokontrolérová jádra (Arm Cortex-M4). Aplikační jádro je taktováno na 500 MHz a mikrokontrolérová jádra lze taktovat na maximálně 200 MHz (reálně 197.6 MHz). Na aplikačním procesoru běží bezpečný operační systém

Azure Sphere OS a na mikrokontrolérových jádrech běží aplikace dodaná vývojářem. Pro aplikační jádro může vývojář dodat i tzv. vysokoúrovňovou aplikaci (HL app – High Level App), která poběží v omezeném prostředí v kontejneru na hlavním procesorovém jádře. Zjednodušenou strukturu jader platformy vizualizuje následující obrázek 2.11



Obrázek 2.11 Platforma Azure Sphere

Praktická část této práce bude řešena na této platformě. Řešení bude navrženo přesně způsobem, ke kterému platforma vybízí. Vysokoúrovňová aplikace běžící v kontejneru na Linuxu zajistí komunikaci s cloudem a aplikační jádra se postarají o zpracování dat z kamery. Detailnější popis návrhu řešení bude popsán v kapitole Návrh systému.

Platformu Azure Sphere vyvíjí společnost Microsoft ve spolupráci s výrobci čipů. Microsoft specifikuje požadavky na čipy, na kterých může Azure Sphere běžet. V době psaní této práce (duben 2021) je komerčně dostupný pouze jeden certifikovaný čip. Jedná se o čip MT3620 od MediaTeku. Zároveň ve stejné době se začínají objevovat zmínky o řadě čipů i.MX 8ULP-CS od NXP, které by v budoucnu měly také být certifikované a dostupné. První zmínky k těmto čipům se ve skutečnosti objevovaly již dříve (jako např. v oznámení pro investory [18] z roku 2019), ale vždy bez zásadních detailů. Výše popsaná struktura, popisována pro platformu Azure Sphere (Cortex-A7 a 2× Cortex-M4) tak ve skutečnosti platí pouze pro využití Azure Sphere s čipem MT3620. Nové čipy od NXP, dle doposud zveřejněných informací budou mít jako hlavní procesorové jádro místo Arm Cortex-A7 novější Arm Cortex-A35 a místo dvou mikrokontrolérových jader Arm Cortex-M4 bude k dispozici jedno mikrokontrolérové jádro novější architektury Arm Cortex-M33. Zároveň přibude Cadence® Tensillica® Fusion DSP pro zpracovávání signálů ze senzorů. V současnosti, ale není příliš zřejmé jakým způsobem bude zajištěna kompatibilita mezi různými čipy, protože je k dispozici jen jeden. Lze předpokládat, že vysokoúrovňové aplikace půjdou migrovat bez problémů (protože běží nad operačním systémem, který bude na všech čipech stejný), ale mikrokontrolérové aplikace budou pravděpodobně nekompatibilní a platformě závislé.

Platforma Azure Sphere nabízí některé velmi užitečné bezpečnostní mechanizmy orientované na tvorbu aplikace komunikující s cloudovými službami [19]. Hlavní vlastnosti platformy je, že každý certifikovaný čip obsahuje bezpečnostní jednotku Microsoft Pluton, která umožňuje jedinečnou identifikaci zařízení a uchování hardwarových klíčů nedostupných ze software. Tuto jednotku může využívat pouze operační systém. Aplikace ji nemohou využít přímo, ale mohou prostřednictvím operačního systému využívat jejich výhod. Jednotka obsahuje unikátní identifikátor zařízení a podporu pro ukládaní a práci s šifrovacími klíči. Operační systém pak jednotku používá pro tvorbu digitálních certifikátů, které platforma Azure Sphere umí digitálně podepsat a na základě šifrované komunikace využívající podepsaný certifikát lze bezpečně komunikovat s vzdálenými službami, které těmto certifikátům mohou důvěřovat.

Z pohledu bezpečnosti je velmi důležité, že tato funkcionalita je implementována v hardware. Stejnou funkcionalitu samozřejmě jde naimplementovat i v software na takřka libovolné platformě. *Unikátní* identifikátor lze vygenerovat při výrobě zařízení a nahrát ho do flash paměti daného alternativního systému. Stejně tak lze pracovat s šifrovacími klíči v software. Softwarové řešení by bylo funkční a dělalo co má, ale je problematické v okamžiku kompromitace zařízení, například přes bezpečnostní chybu typu buffer overflow, kdy útočník získá možnost spouštět na zařízení libovolný kód [20]. Tento *libovolný kód* může z paměti zařízení vyčíst všechny šifrovací klíče, může si jimi cokoliv podepsat, může si původně tajné klíče zaslat „k sobě domů“ a umožnit útočníkovi je zneužít kdykoliv později. Také může útočník vytvořit falešné zařízení, vydávající se za původní. Tím, že jsou klíče na platformě Azure Sphere uloženy v hardware, který jej nikdy nevydá, ale pouze umožňuje operace s nimi, tak kompromitace zařízení (například přes buffer overflow) znamená, že útočník se k samotnému klíči nikdy nedostane. Nemůže si například jimi podepisovat vlastní obsah. Protože však v daném okamžiku na platformě běží jeho kód, může je použít, stejně jako je mohla použít legitimní aplikace. Například v případě monitorovacího systému může otevřít spojení s cloudovým uložištěm a uložit tam podvrhnutý obrázek stejně jako tam legitimní aplikace mohla uložit správný obrázek. Tím, že nemůže dojít ke kompromitaci tohoto klíče, není potřeba klíče ani v případě kompromitace aplikace revokovat. Stačí vyměnit aplikaci za novou aplikaci s opravenou bezpečnostní chybou a případně odstranit škody, které útočník mohl napáchat svým přístupem k vzdáleným zdrojům v době, kdy k nim měl přístup.

Platforma Azure Sphere implementuje i řadu dalších bezpečnostních mechanizmů, které značně omezují možnosti útočníka v případě, že se mu podaří kompromitovat aplikaci. Příkladem může být nutnost, že každá aplikace specifikuje seznam povolených domén, se kterými chce aplikace komunikovat. To v případě kompromitace aplikace zamezí, aby si útočník mohl obrázek poslat „k sobě domů“, protože operační systém nedovolí kompromitované aplikaci otevřít spojení na adresu nepovolenou v manifestu (který není součásti kompromitované binárky a nejde za běhu aplikace změnit).

Nemožnost přečíst firmware platformy zajišťuje platforma přirozeně. Paměť v čipu MT3620 je pouze interní a není ji možné ani číst ani zapisovat. Není tak možné přečíst ani data operačního systému, ani aplikace, které tento operační systém spouští. Platforma samozřejmě umožňuje omezit ladící rozhraní v produkčním nasazení.

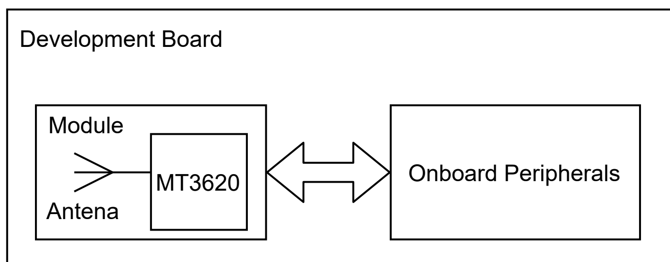
Další bezpečnostně relevantní aspekt je podpora vzdálené aktualizace bez nutnosti připojení zařízení k počítači. Aktualizace jde provést vzdáleně (většinou se to označuje OTA - Over The Air Update) bez jakékoliv nutnosti interakce zásahu uživatele. Operační systém pravidelně kontroluje aktuálnost jak sám sebe, tak aplikací, které spouští. Společnost Microsoft plánuje operační systém na čipu MT3620 podporovat nejméně do června 2031 [21] a platba za tuto podporu je zahrnutá v ceně čipu. Z toho vyplývá ještě jeden bezpečnostní aspekt. O aktuálnost a bezpečnost operačního systému se stará Microsoft a nikoli vývojář zařízení postaveného nad touto platformou. V případě vývoje zařízení s použitím jiného konceptu popsaného výše (mikrokontrolér, jednodeskový počítač, FPGA nebo platformě Zynq) si aktuálnost operačního systému musí zajistit vývojář sám a musí (respektive měl by) své zařízení po celou dobu jeho životnosti podporovat a udržovat, jak aplikaci, tak operační systém aktuální. Na platformě Azure Sphere se to redukuje na podporu pouze vlastní aplikace a nutnost starání se o operační systém odpadá. V závislosti na tom, jak moc se vývojář plánuje své aplikaci věnovat v průběhu jejího životního cyklu, se může jednat o značnou nebo dokonce i úplnou redukci nákladů na údržbu.

Všechny tyto bezpečnostní aspekty umožňují vytvořit velmi bezpečné IoT systémy a proto je praktická část této práce založená právě na této platformě.

2.3.7 Vývojové desky a moduly s platformou Azure Sphere

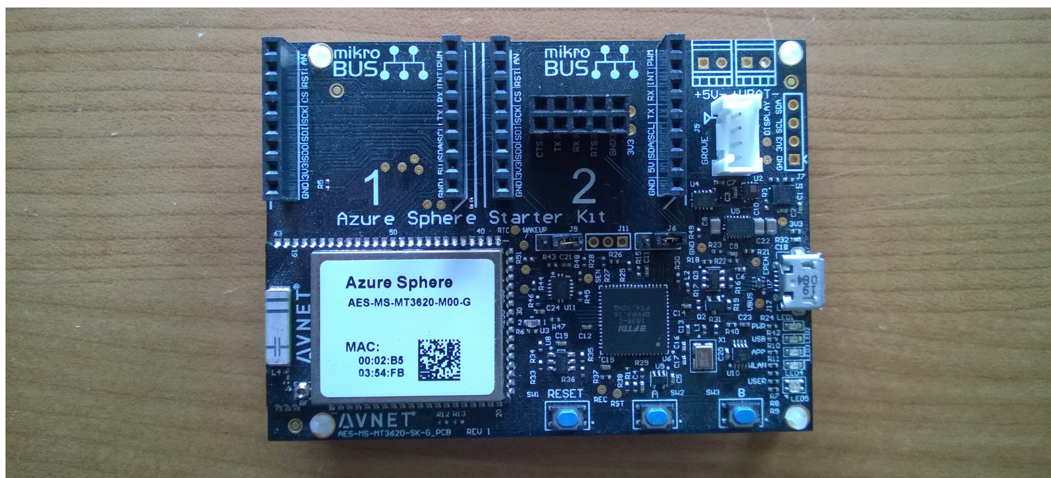
Platforma Azure Sphere je součásti čipu MT3620, ten ale v době psaní této práce (duben 2021) nelze koupit přímo, ale je možné jej zakoupit v rámci modulu. Modul má tu výhodu, že integruje kromě čipu i některé základní komponenty, jako jsou krystalové oscilátory a anténa. Zejména

integrace antény je velmi výhodná. Zapojení antény je poměrně komplexní problém na jehož výsledku závisí nejen kvalita signálu, ale i možnost a případně neúspěch u certifikací jako jsou certifikace shody s legislativou v oblasti EMI (Electromagnetic interference). Moduly jsou navrhnuty a optimalizovány, aby zjednodušily možnost certifikovat zařízení využívající tyto moduly a odbouraly nutnost návrhu poměrně citlivého zapojení antény k čipu, její umístění na desce plošného spoje, vedení signálu mezi čipem a anténou atd. Většina modulů s čipem MT3620 má vývody vyvedené na stranách a umožňuje připájení na desku plošného spoje, protože však taková práce je zejména ve vývojové fázi poměrně pracná, existují vývojové desky na kterých je již některý z modulů připájen. Vývojové desky obvykle obsahují nejenom modul, ale i další podpůrné obvody jako jsou čipy pro napájení a ladění, a také obsahují některé běžně používané periferie jako jsou tlačítka, LEDky a případně i různé senzory. Vývojové desky obsahují konektory s vyvedenými GPIO porty. Strukturu vývojové desky s modulem vizualizuje následující obrázek 2.12. Vývojová deska, na které bude řešena praktická část této práce, má přesně tuto strukturu.



Obrázek 2.12 Struktura vývojové desky Azure Sphere

Příkladem vývojové desky s modulem je například poměrně minimalistická deska MT3620 Mini Dev Board [22] od Seeed Technology integrující modul WF-M620-RSA1 od AI-Link. Existuje i vývojová deska Seeed MT3620 Dev Board [23], která integruje přímo čip MT3620 bez použití modulu (anténa je přímo na desce). Praktická část této práce bude založena na vývojové desce AES-MS-MT3620-SK-G většinou označované jako Avnet Azure Sphere MT3620 Starter Kit [24], která integruje modul AES-MS-MT3620-M-G (rozdíl mezi názvem vývojové desky a modulu je v chybějícím „SK“ u modulu). Vývojová deska Avnet Azure Sphere MT3620 Starter Kit je zobrazena na následujícím obrázku 2.13.



Obrázek 2.13 Vývojová deska Avnet Azure Sphere Starter Kit

2.4 Kamera

Při implementaci monitorovacího systému je potřeba věnovat pozornost nejen výběru platformy, ale i kamery. Dostupné kamery se liší v řadě parametrů a jde sehnat kamery od velmi jednoduchých a levných kamerek, po profesionální velmi drahé kamery s velkou snímkovací frekvencí a velkým rozlišením. Podstatnými parametry však nejsou jen parametry výstupního obrazu, ale i rozhraní, kterým se kamera připojí k platformě. V této sekci budou dále popsány dvě rozhraní, která kamery běžně využívají a jejich omezující parametry. Dále pak obrazové parametry kamer a detailněji bude popsána i kamera, která byla vybrána pro řešení praktické části práce.

2.4.1 Datový tok obrazových dat

Přenos obrazu je datově velmi náročný. Sběrnice pro přenos obrazu musí být dostatečně rychlá a mít dostatečnou kapacitu pro přenos velkého množství dat. Následujícím vztahem 2.1 lze spočítat velikost datového toku potřebného k přenosu n snímků za sekundu, kde každý snímek má rozlišení $p*q$ pixelů a každý pixel je kódován m bit. Předpokládá se přenos v nekomprimované podobě, ale existují i kamery, které umí snímky komprimovat.

$$bitrate = n * p * q * m \quad [\text{bit/s}] \tag{2.1}$$

Například pro přenos s frekvencí 30 snímků za sekundu s rozlišením $1024 * 768$ pixelů kódovaných formátem 8-bit RGB (celkem tedy 24 bitů na pixel) se jedná o datový tok 540 Mibit/s. Výpočet ukazuje následující rovnice 2.2.

$$biterate = 30 * 1024 * 768 * 24 = 566\,231\,040\,\text{bit/s} = 540\,\text{Mibit/s} \tag{2.2}$$

2.4.2 Sériové sběrnice

V současnosti primárním typem sběrnic, které se používají pro přenos obrazových dat jsou sériové sběrnice. S nástupem a rozšířením sériových rozhraní u kamer přišla také standardizace těchto rozhraní a dnes se výhradně používá rozhraní označené jako CSI (Camera Serial Interface). Rozhraní definuje asociace Mobile Industry Processor Interface (MIPI) Alliance jejíž členové navrhují jednotné standardy jako jsou mimo jiné zmíněný CSI a také „obrácený“ standard DSI (Display Serial Interface), který se používá pro přenos obrazových dat opačným směrem z procesoru do zařízení (displaye) [25]. Často jde sběrnici dohledat pod názvem MIPI CSI.

Na sběrnici MIPI CSI se obrazová data přenášení po 1–4 diferenciálních párech, takzvaných linkách. Princip linek se podobá principu linek na sběrnici PCIe. Protože se jedná o sériovou sběrnici, tak frekvence hodin řídící sběrnici musí být poměrně velká. Frekvence závisí na rozlišení a počtu přenášených snímků za sekundu. Například pro přenos 60 snímků s rozlišením $1920*1080$ pixelů za sekundu vyžaduje taktovat hodiny sběrnice CSI na frekvenci 148.5 MHz [26].

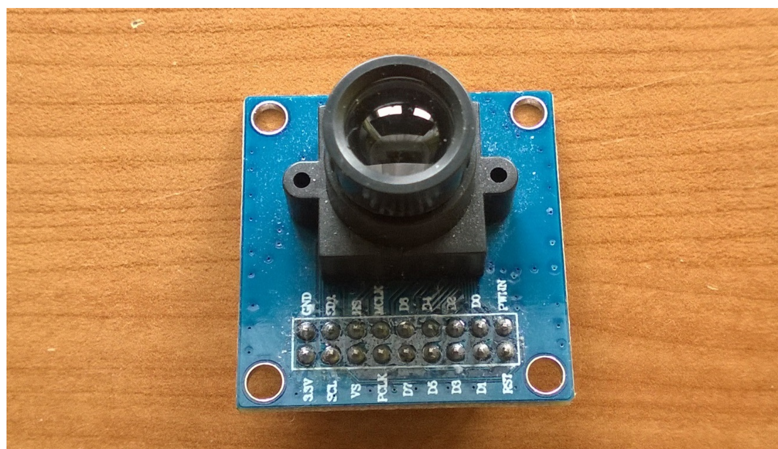
Příkladem kamery, kterou lze připojit je OV2775 od OmniVision [27]. OV2775 je samotný snímací čip, který je možno využít na vlastní desce plošného spoje s patřičnou optickou nadstavbou. Mnohem jednoduší je ale využít některý z již existujících modulů. Existuje například modul LI-OV2775-GMSL2-xxxH [28] nicméně z poměrně strohé dokumentace není jasné, jestli se k modulu jde připojit přímo k CSI rozhraní. V dokumentaci jsou pouze zmínky o proprietárním kabelu a desce umožňující připojit kameru přes USB 3.0 k PC. Další kamerou s CSI rozhraním je například IMX327 od Sony [29], ke kterému je i komerčně dostupný modul [30]. S rozhraním CSI je i poměrně známá kamera Raspberry Pi High Quality Camera [31] určena k použití s jednodeskovými počítači Raspberry Pi. Tato kamera má čip IMX477 od Sony.

2.4.3 Paralelní sběrnice

Alternativou k sériovým sběrnicím jsou paralelní sběrnice umožňující přenášet najednou více bitů paralelně po více vodičích. Pro účely práce s paralelní sběrnicí nás však příliš nebude zajímat datový tok v bitech za sekundu, ale bajtech za sekundu. Paralelní sběrnice přenáší několik (obvykle 8 nebo násobek 8) bitů paralelně na více vodičích najednou. Přestože paralelní sběrnice mohou být i s jiným počtem vodičů než násobkem osmi, tak se tato práce omezuje pouze na násobky osmi, respektive celé bajty. Jiné šířky sběrnic by komplikovaly následující výpočty a zaměřování se na ně nedává příliš smysl, protože všechny zde popisované kamery s paralelními sběrnicemi používají sběrnice o šířce 8 nebo násobku 8 vodičů.

Díky přenosu více dat najednou paralelní sběrnice vyžadují pomalejší hodinový signál než sériové sběrnice. Paralelní sběrnice mají však i nevýhody, jako je nutnost vést více vodičů nebo skutečnost, že od určité rychlosti hodin je potřeba hlídat, aby některý z vodičů sběrnice nebyl delší než ostatní vodiče, protože doba šíření signálu závisí na jeho délce a u rychlých signálů by se mohlo stat, že některé z bitů v důsledku delšího vodiče „přijdou později“, v době, kdy již data nejsou platná. Přestože jsou paralelní rozhraní na ústupu a většina dnešních kamer je se sériovým rozhraním, tak kamery s paralelním rozhraním existují. Zejména se však jedná o starší, ale stále pro spoustu použití použitelné kamery.

Příkladem kamery s paralelním rozhraním je kamera OV7670 [32], která je využita v praktické části práce. Tato kamera je k dispozici na poměrně levných modulech [33], obsahuje paralelní sběrnici a sériové konfigurační rozhraní. Jedná se o poněkud starší kameru s podporou rozlišení do $640 * 480$ pixelů, což však pro účely monitorovacího systému, popisovaného v této práci, je dostatečné. Modul s kamerou OV7670 je zobrazen na následujícím obrázku 2.14



Obrázek 2.14 Modul s kamerou OV7670

2.4.4 Předpoklady obsluhy paralelní datové sběrnice

Z velikosti datového toku vyplývají omezení, které kladou nároky na výkon procesoru systému, který má danou sběrnici číst a ukládat z ní obrazová data do paměti. Například stanovuje nutnou podmínku pro zpracovatelnost dat ze sběrnice v reálném čase daným procesorem při využití paralelního přenosu bajtů obrazových dat. Následující vztah 2.3 ukazuje, že velikost datového toku nesmí přesáhnout množství dat, které je schopen jednorázový procesor zpracovat za sekundu. F_{CPU} je frekvence, na které je procesorové jádro taktováno a N_{CPU} je hodnota určující kolik bajtů dokáže daný procesor zpracovat maximálně v jedné instrukci. N_{CPU} závisí na architektuře procesoru. Na 8bitových procesorech se jedná o 1 bajt, ale například 32bitové

procesory dokáží jednou instrukcí zpracovat 4 bajty (32 bitů). Podobně je možné použít konstantu N_{CPU} pro zohlednění případného použití vícejádrového procesoru.

$$byterate \leq F_{CPU} * N_{CPU} \tag{2.3}$$

Pokud je požadovaný datový tok 54 MiB/s a chtěli bychom tento datový tok přenášený 8bitovou paralelní sběrnicí zpracovávat na 8bitovém mikrokontroléru ATmega4808 [34], který může být taktován maximálně na 20 MHz, tak po dosazení dostáváme nerovnici 2.4, která zřejmě neplatí a tento datový tok není nikdy možné zpracovávat v reálném čase tímto mikrokontrolérem.

$$\begin{aligned} 54\,\text{MiB/s} &\not\leq 20\,\text{MHz} * 1 \\ 56\,623\,104 &\not\leq 20\,000\,000 \end{aligned} \tag{2.4}$$

Tento vztah ve skutečnosti není příliš vypovídající. Je to nutná, ale nikoli postačující podmínka k tomu, aby procesor dokázal zpracovat data z kamery v reálném čase. Předpokládá totiž zpracování dat ze sběrnice, jejíž šířka přesně odpovídá velikosti dat, které procesor zpracovává (např. pro 32bitový mikrokontrolér se jedná o paralelní sběrnici s 32 vodiči) a předpokládá, že čtení proběhne jedinou instrukcí, která v jednom hodinovém taktu tuto sběrnici přečte, data uloží do paměti a inkrementuje adresační ukazatel. Také předpokládá, že oba systémy na koncích sběrnice používají totožné hodiny. V podstatě předpokládá ideální sběrnici a ideální procesor, synchronizované stejnými hodinami a navíc předpokládá i ideální program. Tyto požadavky ale nelze reálně zajistit. Existují kamery a procesory, které dokážou zajistit některé z těchto požadavků, ale ne všechny. Proto se jedná jen o nutnou a nikoli postačující podmínku.

Podstatná skutečnost k zamyšlení z přechozí úvahy také je, zdali vůbec potřebujeme zpracovávat data v reálném čase? Monitorovací systém popisován v této práci dělá vždy jeden snímek v pravidelném intervalu (například jednou za minutu). V tomto případě tedy stačí, když procesor bude ukládat data ze sběrnice pouze v době kdy probíhá snímání daného jednoho snímku a nepotřebuje pracovat v době, kdy probíhá přenos předchozího nebo následujících snímků. Může tak například sejmout jeden snímek a nějaké jeho přípravy pro odeslání do cloudu udělat až v okamžiku, kdy už probíhá přenos dalšího snímku. Stále je však nutné zpracovávat poměrně rychlou paralelní sběrnici v reálném čase po určitou krátkou dobu, kdy probíhá přenos dat, která potřebujeme zachytit. Za určitých okolností si však můžeme dovolit nějakou část jejich zpracování „odložit“ na čas, kdy bude probíhat přenos snímku, který nepotřebujeme zachytit.

Na platformě Azure Sphere s čipem MT3620 není problém na mikrokontrolérovém jádře taktovaném na 197.6 MHz obsloužit paralelní sběrnici kamery OV7670, ale vyžaduje to rekonfiguraci děličky hodinového signálu. Kamera OV7670 dělení signálu sběrnice podporuje. Nutná podmínka, popsaná výše pro platformu Azure Sphere s předpokladem využití jednoho mikrokontrolérového jádra pro zachytávání dat z kamery a hodinovou frekvenci předělenou na frekvenci okolo 1 MHz, je splněna.

2.4.5 Formáty výstupního obrazu

Většina kamer umožňuje konfigurovat formát výstupního obrazu. Běžně se používá formát RGB, který přenáší složku červené, zelené a modré barvy zakódovanou jako číslo. Existuje více variant formátu RGB, které se liší v počtu bitů, kterými kódují jednotlivé složky. Obvykle se používá formát RGB888, který kóduje každou složku barvy osmi bity (jedním bajtem). Některé kamery však takový barevný rozsah nepodporují a nabízí například formát RGB565 nebo RGB444, které kódují jednotlivé složky jinými počty bitů. Krom formátu RGB, řada kamer umí ještě exportovat ve formátu YUV a jeho variantách.

Existují i kamery, obsahující akcelerátor komprese. Takové kamery nezasílají obrazová data v surové podobě, ale zasílají již komprimovaný obrázek. Nejčastější formát podporovaný kamerami je formát JPEG.

Kamera OV7670, s jejíž pomocí bude řešena praktická část této práce podporuje formát YUV a několik formátů RGB až do formátu RGB565, který bude v rámci řešení praktické části práce využit.

2.5 Využití cloudu

Dalším aspektem k zvážení je využití cloudu a případně jakého cloudu. Tato práce popisuje systém, který analýzy provádí v cloudu, což ale není jediná možnost, jak daný problém řešit. Přestože se jedná o moderní způsob, tak řada současných monitorovacích systémů stále spoléhá spíše na lokální zpracování dat na zařízení.

Využití cloudu může mít řadu výhod, jako je například možnost mnohem náročnějšího zpracování ve srovnání se zpracováním na zařízení s omezeným výkonem. Umožňuje to například redukovat spotřebu zařízení. Podobným benefitem může být využití zdrojů, které zařízení nemá, jako je takřka nekonečné diskové uložiště. Zařízení, pomocí kterého bude praktická část práce řešena, má paměť na uchování jednoho snímku v paměti RAM a několik snímků v perzistentní paměti (kolik, přesně záleží na množství dostupné paměti, kterou může operační systém poskytnout a s aktualizacemi operačního systému se může v čase měnit). Namísto toho v cloudu můžeme mít klidně uloženy statisíce obrázků z desetitisíců zařízení jak dlouho chceme (a hlavně po jakou dobu budeme ochotni za toto uložiště platit).

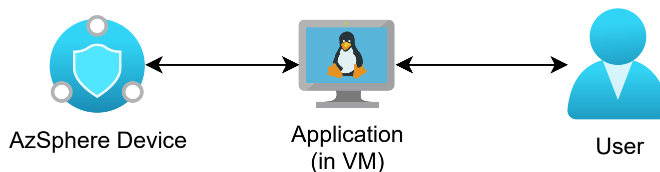
Při výběru se nabízí možnost využití jak privátního, tak veřejného cloudu. Neexistuje žádné omezení, které by znemožňovalo implementaci monitorovacího systému na některé z těchto platforem. Oba přístupy mají svoje výhody a nevýhody, které v některých situacích mohou být klíčové. Privátní cloud (tzn. cloud provozovaný jedincem nebo firmou na vlastní infrastruktuře) má řadu nevýhod jako jsou poměrně vysoké náklady na pořízení HW pro jeho provoz a nutnost administrovat tento cloud. Výhody však detailní kontrola nad daty, které jsou uloženy na vlastních discích. Ve většině případů příliš nedává smysl nasazovat privátní cloud jenom kvůli monitorovacímu systému, ale pokud již firma (nebo jedinec) provozuje privátní cloud, tak je samozřejmě vhodné ho využít i pro účel monitorovacího systému. Veřejný cloud má naopak velmi dobré vlastnosti co se týče prvotních nákladů, kdy není nutné kupovat žádný HW. Také náklady na administraci jsou nižší. Nevýhodou je, že potenciálně citlivá data se nachází na vzdálených serverech.

V případě volby veřejného cloudu se nabízí několik možností, jak zvolit poskytovatele cloudových služeb. V době psaní této práce (duben 2021) existují dva velké veřejné cloudy a jsou jimi (v abecedním pořadí) Amazon Web Services (AWS) a Microsoft Azure. Oba nabízí řadu obvykle srovnatelných služeb, které umožňují vytvořit nejen monitorovací systém. Oba veřejné cloudy také umožňují bezpečně připojit zařízení jako je Azure Sphere do svých cloudů pomocí dedikované služby. U AWS se jedná o službu IoT Core a u Azure je to služba IoT Hub. Krom velkých cloudových služeb existují rozsahově trochu menší poskytovatelé, kteří se snaží svými službami dorovnat a překonat tyto zaběhlé poskytovatele. Jedná se například o Google Cloud Platform nebo IBM cloud. Obě platformy nabízí všechny potřebné služby pro implementaci monitorovacího systému a je možné systém implementovat i s využitím jejich služeb. Krom velkých poskytovatelů existují i menší (zejména lokální) poskytovatelé jako je v České Republice Forpsi, Wedos, Zoner a další, kteří však většinou nabízí pouze pronájem virtuálních serverů. Nabízí tedy pouze koncept IaaS (IaaS bude popsán dále), což sice může dostačovat, ale koncept PaaS může monitorovacímu systému přinést lepší možnosti škálování i cenu za provoz. Přestože platforma Azure Sphere vybízí k použití služeb Azure, nejedná se o nutnou podmínku nebo omezení. Platformu Azure Sphere lze připojit k jakémukoliv cloudu u jakéhokoliv poskytovatele nebo i k privátnímu cloudu. Autentizace probíhá pomocí podepsaného certifikátu a platforma umožňuje pomocí knihovny CURL, která je na platformě Azure Sphere k dispozici otevřít TLS spojení s vzdáleným serverem (např. běžícím v AWS) s autentizací pomocí důvěryhodného certifikátu podobně jako se autentizuje zařízení při navazování spojení se službou IoT Hub. Výhodou připojování k službám Azure je, že operační systém integruje (a aplikacím nabízí) knihovnu pro připojení k IoT Hubu v Azure.

Pro připojení k jiné službě u jiného poskytovatele si vývojář musí integrovat knihovnu (jako je například [35]) sám.

Principiálně je možné navrhnout aplikaci v cloudu s využitím dvou odlišných konceptů popisovaných jako IaaS (Infrastructure as a Service) nebo PaaS (Platform as a Service). Jedná se o odlišné přístupy k řešení problémů s pomocí cloudových služeb a výběr ovlivňuje vlastnosti výsledného řešení.

Řešení přístupem IaaS by znamenalo, že v cloudu poběží pronajatý VM (Virtual Machine), ke kterému budeme mít k dispozici plný přístup. Obvykle cloudy poskytují přístup k SSH (Secure Shell) v případě Linuxového VM a RDP (Remote Desktop Protocol, v Česku známý také jako vzdálená plocha) v případě VM s Windows. V tomto VM budou spuštěny a implementovány potřebné aplikace (například webový server, databáze atd.) plně dle vlastní volby. V případě řešení tímto způsobem by se zařízení připojovalo přímo k tomuto VM, případně k load balanceru, který by vyvažoval zátěž mezi více VM. Přístup má velkou výhodu v možnostech volby použitého software. Nevýhodou tohoto přístupu jsou vyšší náklady při nižším zatížení serveru, protože jeho provoz je nutné platit po celou dobu, kdy má být aplikace dostupná (obvykle chceme, aby byla dostupná nepřetržitě, ale existují i přístupy, kdy se VM, na kterém je aplikace provozována vypíná v době, kdy je nevyužitý, například v noci). Druhou nevýhodou je, že se o celé prostředí, ve kterém naše aplikace (na daném VM) poběží musíme starat a spravovat jej. Musíme například udržovat operační systém aktuální. Řešení navržené podle principů IaaS schematicky vypadá poměrně jednoduše, protože všechna aplikační logika se koná ve virtuálním stroji. Schéma tohoto řešení z pohledu využití služeb v cloudu je vizualizováno následujícím obrázkem 2.15.



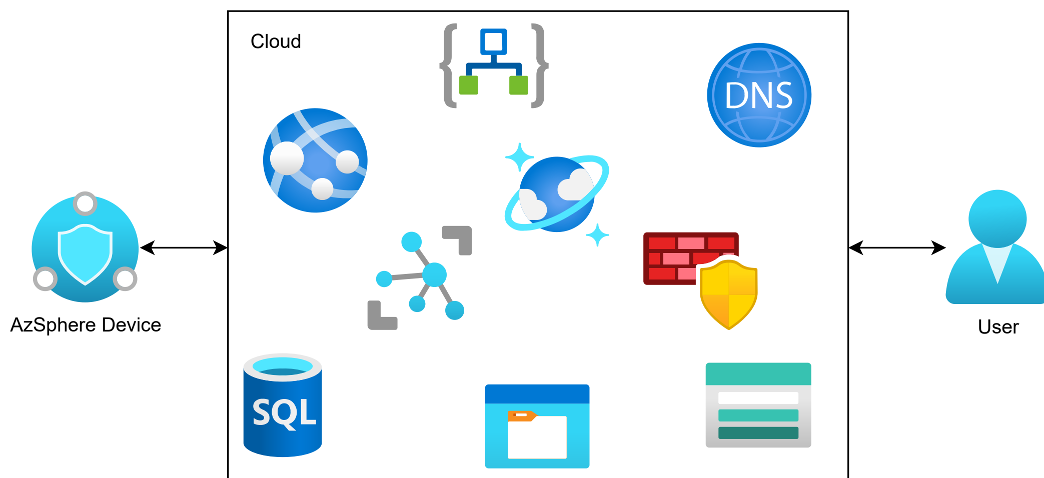
Obrázek 2.15 Potenciální řešení navržené podle principu IaaS

Druhým přístupem je přístup PaaS. Tento přístup využívá služby cloudu, které různými způsoby propojuje podle toho, co dané konkrétní služby umožňují. Takový přístup nevyžaduje provoz žádného VM. Využité služby se dle aplikace liší a jedná se například o službu objektového uložiště, databáze, bezstavové funkce atd. Výhodou řešení jsou náklady odpovídající přesné zátěži, protože jednotlivé služby jsou většinou účtovány podle počtu operací provedených proti dané službě a není tak nutné platit poplatek i v době, kdy se služba nevyužívá. Druhou výhodou je odpadnutí nutnosti starat se o dané služby, systémy a servery, na kterých dané služby běží, protože to je v režii poskytovatele cloudových služeb. V tomto ohledu přístup PaaS umožňuje vytvořit i bezpečnější řešení, protože není nutné se o bezpečnost serverů starat. Bezpečnost řešení však i v případě řešení PaaS lze významně ovlivnit nevhodným použitím daných služeb a jejich konfigurací. Příkladem nevhodné konfigurace může být nastavení zveřejnění dat uložiště (kde mohou být například snímky z monitorovacího systému) zcela veřejně bez vyžadování autentizace. Následující obrázek 2.16 vizualizuje řešení pomocí konceptu PaaS.

Monitorovací systém realizovaný v rámci praktické části této práce byl vyvinut na platformě Azure s použitím přístupu PaaS.

2.6 Analýza obrazu

Pro provádění analýzy obrázků a detekcí objektu na obrázku existují knihovny, které integrují algoritmy pro strojové učení, neuronové sítě a další potřebné algoritmy a matematické operace



Obrázek 2.16 Potenciální řešení navržené podle principu PaaS

pro tyto analýzy. Příkladem může být knihovna TensorFlow, jejíž použití je popsáno v knize [36]. Tyto knihovny jsou velmi dobrými kandidáty k řešení těchto úloh. Použití cloudu však přináší i další možnosti. Existují služby, které přes velmi jednoduché rozhraní umožňují dělat analýzy v cloudu. Při volbě metody provádění analýzy obrázků se tak dostáváme ke konceptům IaaS a PaaS. Buď si můžeme sami zprovoznit analýzu pomocí knihovny jako je TensorFlow (koncept IaaS) nebo můžeme využít cloudové služby (koncept PaaS). Oba přístupy mají svoje výhody a nevýhody. Analýza obrazu pomocí cloudové služby má výhodu, že model, nad kterým probíhá vyhodnocování snímků, může být postupem času vylepšován bez nutnosti přispění samotným vývojářem aplikace. Naopak vlastnoručně implementované zpracování s pomocí knihovny přináší vývojáři značně rozsáhlejší možnosti konfigurace a ladění.

Další vlastností, kterou je potřeba brát v potaz je místo, kde dochází ke zpracování obrázků. Mohou existovat monitorovací systémy, které nemají možnost zasílat snímky do jakéhokoliv externího systému. Může se jednat o systémy snímající nějaká citlivá místa. Takové systémy jsou pak omezeny na lokální zpracování, které musí provést pomocí vlastní implementace s pomocí knihovny pro strojové učení nebo jiných knihoven.

Tato práce se i v oblasti analýzy snímků bude držet konceptu PaaS a využije cloudovou službu pro detekci obejktů na obrázku.

Kapitola 3

Návrh systému

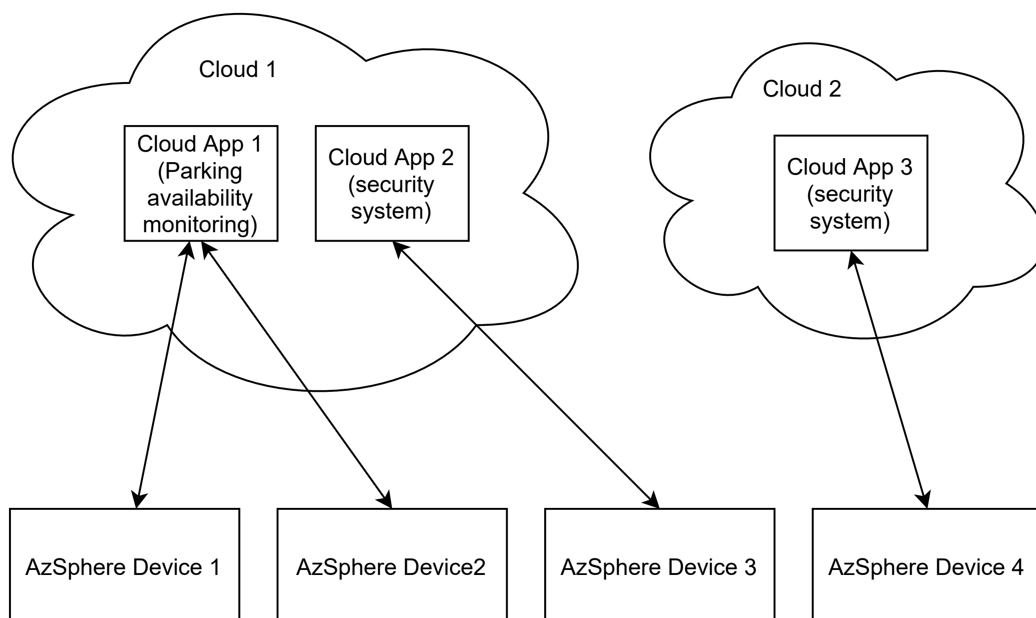
Následující kapitola se zabývá samotným návrhem zařízení a alternativními možnostmi řešení některých problémů. Samotný popis implementace a některých konkrétních problémů spojených s implementací navrženého řešení je popsán v následující kapitole Implementace řešení. Tato kapitola i kapitola Implementace řešení je rozdělena na 5 částí. První dvě se týkají hardwarové stránky řešení, konkrétně návrhu zapojení periferií k platformě a návrhu a výroby desky plošného spoje. Třetí část se týká návrhu a vývoje firmware pro platformu Azure Sphere. Čtvrtá a pátá část popisují návrh a implementaci cloudových služeb a aplikace prezentující výstupy z monitorovacího systému uživatelům.

3.1 Části řešení

Navržené řešení je rozděleno na dvě části. První část je část zařízení a druhá část řešení je aplikace běžící v cloudu. Část řešení implementována na zařízení musí zohledňovat HW aspekty jako je zapojení periferií zařízení, firmware daného zařízení a součásti této části řešení je návrh desky plošného spoje, která umožní jednoduché sestavení finálního zařízení a dostatečně robustní připojení periferií k vývojové desce. Účelem této části řešení je zejména obsluha kamery, analýza okolního zvuku a senzorických dat a předávání těchto informací do druhé části aplikace. Případně drobné zpracování některých těchto dat. Druhá část aplikace se nachází na cloudové straně. Na cloudové straně běží aplikace, které zpracovávají data ze zařízení, vyhodnocují je a vytváří výstupy. Obě části aplikace jsou nezávislé a škálovatelné. Samozřejmě lze vyrobit a nasadit zařízení ve vícero instancích. V cloudu může běžet více různých aplikací zpracovávajících data. Jedna cloudová aplikace dokáže zpracovávat data z více zařízení na jednou. Zobecněné schéma možné struktury a vazeb mezi částmi řešení zobrazuje následující obrázek 3.1.

Aplikace je navržena s ohledem na nezávislost zařízení na cloudu a cloudu na zařízení. Aplikace je navržena tak, aby se zařízení chovalo dostatečně univerzálně a samotnou aplikaci monitorovacího systému se dalo změnit pouze výměnou částí na cloudové straně. Cloudová strana aplikace je navrhována s ohledem na škálovatelnost a náklady na provoz. Jedna aplikace může být v cloudu spuštěna i vícekrát. S běžně dostupnými prostředky v cloudu jako je load-balancer lze zajistit poměrně pružné horizontální škálování monitorovacích aplikací.

Návrh umožňuje vyměnit běžící aplikaci (například vyměnit aplikaci sledující obsazení parkoviště za aplikaci hlídající jedno konkrétní místo parkoviště) v řádu sekund, bez nutnosti výměny firmware v zařízení. V případě alternativního návrhu aplikačně-specifického firmware zařízení je nutné při každé změně vyčkat na nasazení nového firmware. V případě využití platformy Azure Sphere, tuto výměnu firmware lze udělat vzdáleně, ale trvá to v řádu hodin, protože zařízení nekontroluje aktualizace na častější bázi.



Obrázek 3.1 Obecné schéma topologie aplikace a škálovatelnosti

Obě části (cloudová a HW) jsou dále složeny z dílčích aplikací. Na HW straně se jedná o aplikace, které běží na jednotlivých výpočetních jádrech platformy a na straně cloudu se jedná o zejména aplikace využívající funkce cloudu a konfigurace služeb, které zajišťují některou z dílčích části řešení.

V následujících sekcích se zaměřím na návrh jednotlivých části řešení (část implementovanou na zařízení a cloudovou část řešení).

3.2 Struktura řešení zařízení

Při návrhu HW části je potřeba navrhnout logické zapojení jednotlivých periferií a strukturu firmware zařízení. Oba návrhy spolu úzce souvisí a musí se navzájem reflektovat. S rostoucí komplexitou procesorů a mikrokontrolérů využívaných ve vestavných systémech, roste také komplexita a nároky na firmware. Azure Sphere je také příklad poměrně komplexního SoC (System On Chip) kombinující aplikační a mikrokontrolérová jádra. Proto je potřeba dbát i na vhodný návrh firmware zařízení a vhodné rozložení úloh mezi dostupná jádra.

3.3 Logické propojení periferií zařízení

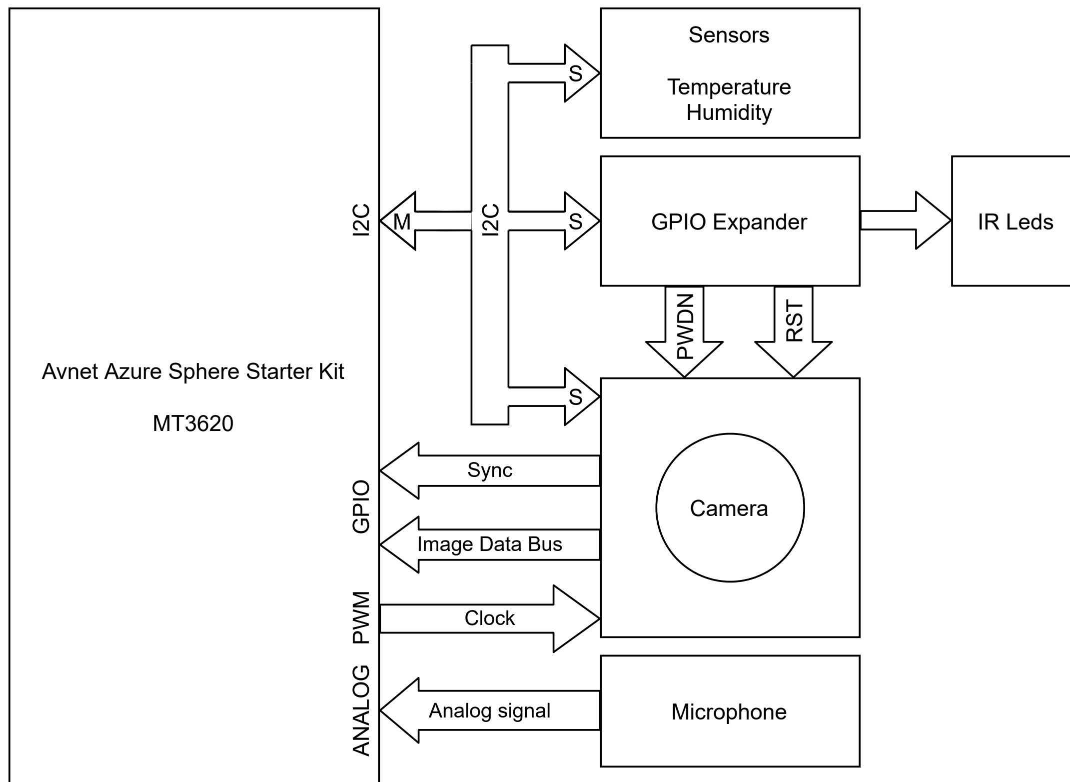
V následujících sekcích budou popsány aspekty návrhu samotného hardware zařízení. V prvních sekcích bude prostor věnován návrhu zapojení jednotlivých HW komponent a následně popsán i návrh desky plošného spoje umožňující jednoduché propojení jednotlivých periferií s platformou.

3.3.1 Revize vývojové desky Azure Sphere

Vývojová deska Avnet Azure Sphere Starter Kit, na které je řešení postaveno, se vyrábí ve dvou revizích. Konkrétně se jedná o revize označované jako revize 1 a 2. Jeden z nejvýznamnějších

rozdílů mezi revizemi je struktura vyvedených GPIO portů. Obě revize vyvádějí jiné GPIO porty na konektor pro rozšiřující desky. Sada vyvedených portů je podobná, ale porty jsou přeházené a existují porty, které jsou vyvedené jen na jedné z revizí. Toto značně komplikuje návrh zařízení, které má podporovat obě revize. V celé následující sekci tak bude vidět, že u většiny vývodů zařízení je zmíněn jiný GPIO port pro revizi 1 a jiný pro revizi 2. Pokud žádné odlišení zmíněno nebude, předpokládá se, že vyvedený GPIO port je na obou platformách stejný.

Následující blokové schéma na obrázku 3.2 ukazuje strukturu řešení HW části zařízení a jejich propojení.



Obrázek 3.2 Obecné schéma návrhu a zapojení hardwarové části řešení

3.3.2 Připojení GPIO expandéru

Blokové schéma reflektuje vlastnosti platformy Azure Sphere a omezení jako je množství GPIO portů, které platforma nabízí. Z toho důvodu je v návrhu použit GPIO expandér, který umožňuje řídit některé signály kamery bez nutnosti využít GPIO porty samotné platformy Azure Sphere. Vyžaduje samozřejmě připojení k portům sběrnice I2C, které jsou sdílené s kamerou a senzory. Při navrhování zařízení s GPIO expandérem je potřeba zohlednit omezení portů GPIO expandéru. Hlavním omezujícím kritériem je rychlost výstupního signálu, která nemůže přesáhnout rychlost sběrnice I2C. Lze odvodit následující vztah 3.1 pro určení maximální frekvence, kterou lze přepínat stavy výstupních portů GPIO expandéru.

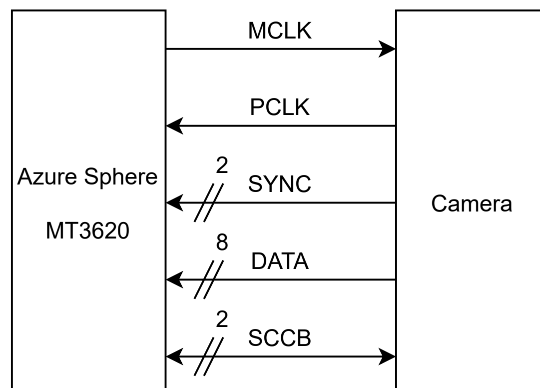
$$F_{GPIO_Switching} = \frac{F_{I2C_Bus}}{29} \tag{3.1}$$

Konstanta 29 vychází z nutnosti využít jeden hodinový takt pro generování START sekvence

na sběrnici I2C, dalších 7 cyklů je využito k přenosu adresy zařízení, osmý bit přenáší informaci o směru přenášených dat. Další bit je přenášený ACK/NACK bit z GPIO expandéru následovaný 8bitovou adresou registru, potvrzením bitem ACK/NACK, 8bitovou hodnotu registru, posledním potvrzením a poslední hodinový takt je využit na vygenerování STOP sekvence na sběrnici. Celkem tedy změna stavu portu GPIO expandéru trvá 1+7+1+1+8+1+8+1+1 = 29 hodinových cyklů sběrnice I2C. V případě, že hodinový signál sběrnice I2C je taktován na 400 kHz, tak maximální přepínací frekvence GPIO portu expandéru je přibližně 13.8 kHz. Řešení tak například nelze navrhnout způsobem, že by se hodinový signál pro kameru generoval bit-band metodou pomocí GPIO expandéru, čímž by se ušetřil další GPIO port zařízení. Volené porty PWDN a RST jsou příklady portů, které se mění jen zřídka a nenesou žádný dynamicky měnící se signál. Podobně to platí pro IR LED připojené ke stejnému expandéru, které slouží jako noční přisvícení.

3.3.3 Připojení kamery

Kamera je k zařízení připojena pomocí několika sběrnic a signálů. Jedná se o hlavní hodinový signál (MCLK, někdy označován jako XCLK), řídicí sběrnici SCCB (která je kompatibilní s I2C) a paralelní datovou sběrnici, jejíž součástí je 8 datových vodičů a její hodinový signál (PCLK). V následujících sekcích bude popsáno připojení jednotlivých signálů podrobněji. Obrázek 3.3 popisuje signály propojující vývojovou desku a kameru.



Obrázek 3.3 Přehled signálů vedených mezi vývojovou deskou a kamerou

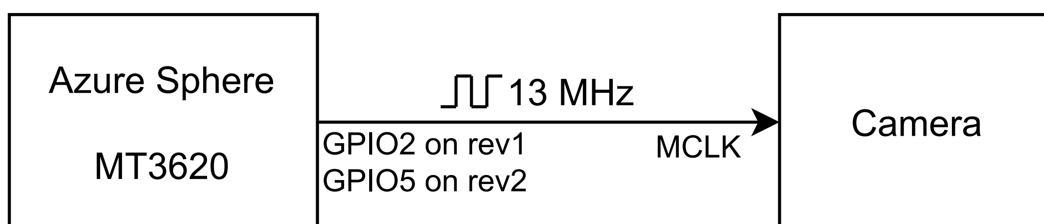
3.3.4 Generování hodin pro kameru

Navržené řešení nepoužívá pro generování hodin externí krystal, ani žádný jiný čip, ale pro co nejlepší využití zdrojů desky se používá výstupní port z čipu MT3620, který je nakonfigurován jako PWM výstup a generuje PWM signál se střídou 50 % a frekvencí 13 MHz, což je polovina frekvence krystalu, který taktuje samotný čip MT3620 na této vývojové desce (krystal je součástí modulu, který je na desce osazen). Toto řešení má výhodu v redukci množství potřebných součástek, a tedy i snížení nákladů na případnou výrobu zařízení. Nevýhodou je skutečnost, že na platformě Azure Sphere jsou proty s podporu PWM kanálů (je zde celkem 12 takových portů) připojeny k PWM kontrolérům po čtyřech portech a kdykoliv se kterýkoliv z těchto portů nastaví jako PWM, tak všechny ostatní porty ve skupině jsou ovládané PWM kontrolérem a nelze je použít jako běžný nezávislý GPIO port [37]. Na vývojové platformě [38] to konkrétně znamená, že při konfiguraci alespoň jednoho z portů GPIO0, GPIO1 a GPIO2 (které jsou vyvedeny na desce)

do režimu PWM nelze ostatní porty z této skupiny použít jako GPIO porty a jejich použití je tak omezené na režim PWM.

Alternativním řešením by mohlo být zmíněné použití externího krystalu, které by nevyžadovalo žádný GPIO port platformy, ale například by neumožňovalo vypnout generování hodinového signálu v době, kdy je kamera nečinná (nesnímá obrázek), a tím snížit její spotřebu energie. Tento aspekt by šel vyřešit použitím oscilátoru s vývodem označovaným ENABLE, který umožňuje generovat hodinový signál jen v případě, že je daný signál připojen k jedné z logických úrovní 0 nebo 1 v závislosti na daném oscilátoru. Takové řešení by umožňovalo zastavit generování hodin, ale vyžadovalo by opět připojení k GPIO portu platformy. Na druhou stranu by nevyžadovalo GPIO port s podporou PWM a šlo by pro tento signál využít i port z GPIO expandéru.

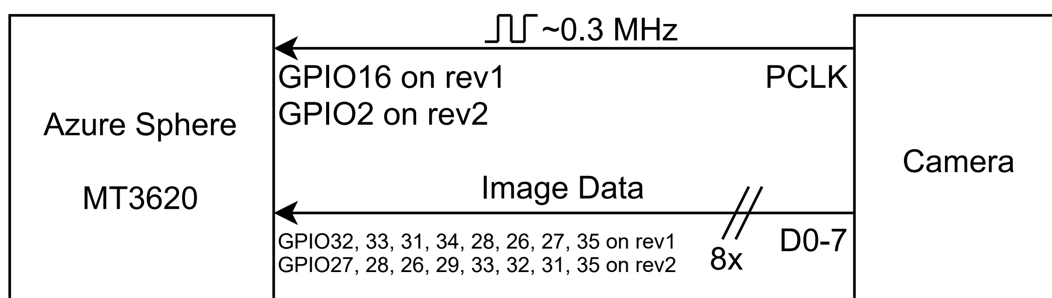
Dalším potenciálně zajímavým řešením, je využití výstupních hodin generovaných z mikrokontroléru/procesoru. Některé mikrokontroléry/procesory nabízí možnost generování výstupních hodin s frekvencí odvozenou nějakou předěličkou od některých z vnitřních hodin. Platforma Azure Sphere s čipem MT3620, ale nic takového neumožňuje [39] nebo to není uvedeno v žádné veřejné dokumentaci. Nicméně na některých platformách, které to umožňují, se jedná o preferované řešení a existují práce, které toho využívají. Příkladem je projekt [40].



Obrázek 3.4 Zapojení hlavního hodinového signálu kamery

3.3.5 Zapojení datové sběrnice kamery

Datová sběrnice kamery se skládá z 9 vodičů. 8 vodičů slouží jako paralelní sběrnice pro přenášení obrazových dat a zbývající vodič je hodinový signál, který určuje platnost dat na datových vodičích. Hodinový signál této sběrnice generuje kamera a je označen PCLK (pixel clock). Datové vodiče jsou připojeny na GPIO porty, jak je ukázáno na obrázku 3.5.



Obrázek 3.5 Zapojení datové sběrnice kamery

Při návrhu bylo potřeba vybrat ze dvou přístupů, jak volit GPIO porty propojující datovou sběrnici k jednotlivým datovým vodičům kamery. Na výběr byly následující dvě protichůdné

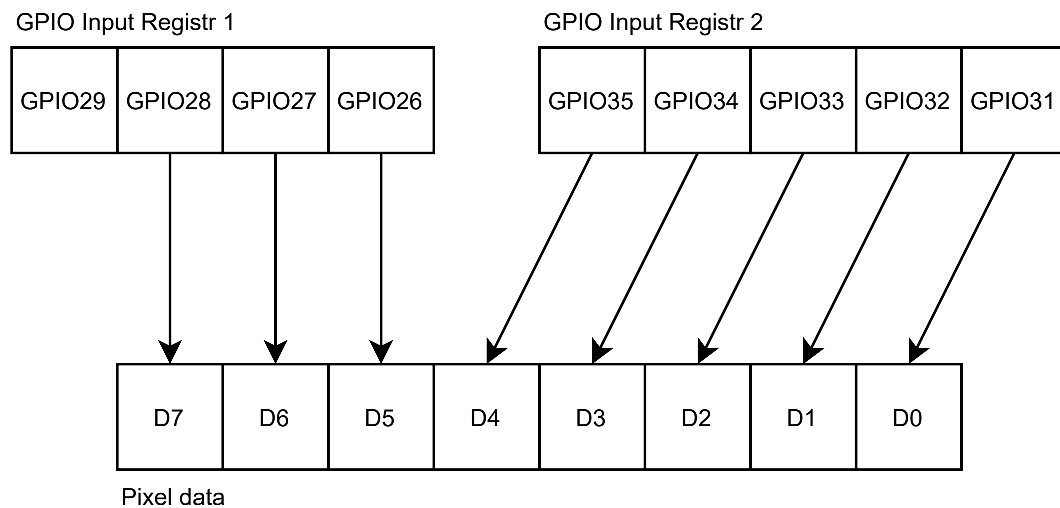
metody, jak toto zapojení provést.

1. s ohledem na počet procesorových instrukcí nutných k přečtení dat ze sběrnice

2. s ohledem na počet křížení vodičů na desce plošného spoje

První metoda respektuje fakt, že čtení kamery vyžaduje více instrukcí pro čtení stavu portů a kód také musí udělat mapování bitů z přečteného registru, aby získal pixelová data ve správném formátu. Platforma umožňuje číst porty po skupinách čtyř nebo pěti portů. Porty 26–29 jde přečíst jednou čtecí instrukcí a porty 31–35 lze přečíst také jednou instrukcí.

V případě ideálního zapojení (první zmíněná metoda) by mapování bitů z registrů GPIO portů do výsledných obrazových dat v paměti mohlo vypadat jako na následujícím obrázku 3.6.

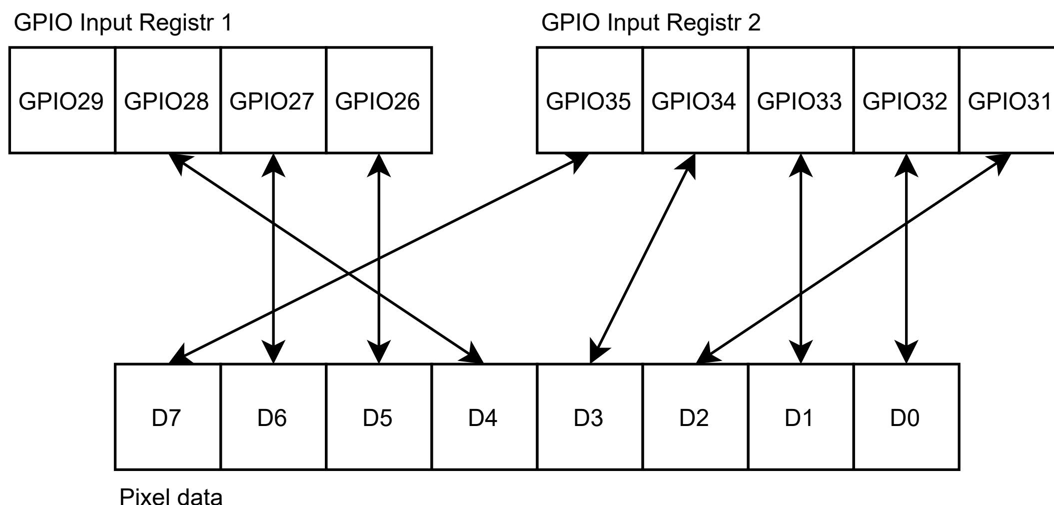


Obrázek 3.6 Potenciální možné mapování GPIO portů na datovou sběrnici kamery pro nejefektivnější možné čtení dat ze sběrnice

Řešení tak v podstatě přečte pouze oba registry, hodnotu z jednoho registru logicky posune a operací logický OR spojí hodnoty do jedné výsledné hodnoty. Toto řešení sice umožňuje číst data ze sběrnice s nejmenším možným počtem instrukcí, ale má však nevýhodu, že umisťování vodičů na desce plošného spoje (jehož návrh bude pospán dále) je extrémně komplikované. Komplikuje to i fakt, že vodiče musí být vedeny zvlášť pro platformu revize 1 a revize 2, protože na rozšiřujícím konektoru platformy, ke kterému je sběrnice připojena, jsou v obou revizích jiné GPIO porty. Efektivně to znamená, že vodiče nejenže se budou křížit navzájem v rámci propojení konektorů v rámci jedné revize, ale křížilo by se tam téměř 2× tolik vodičů.

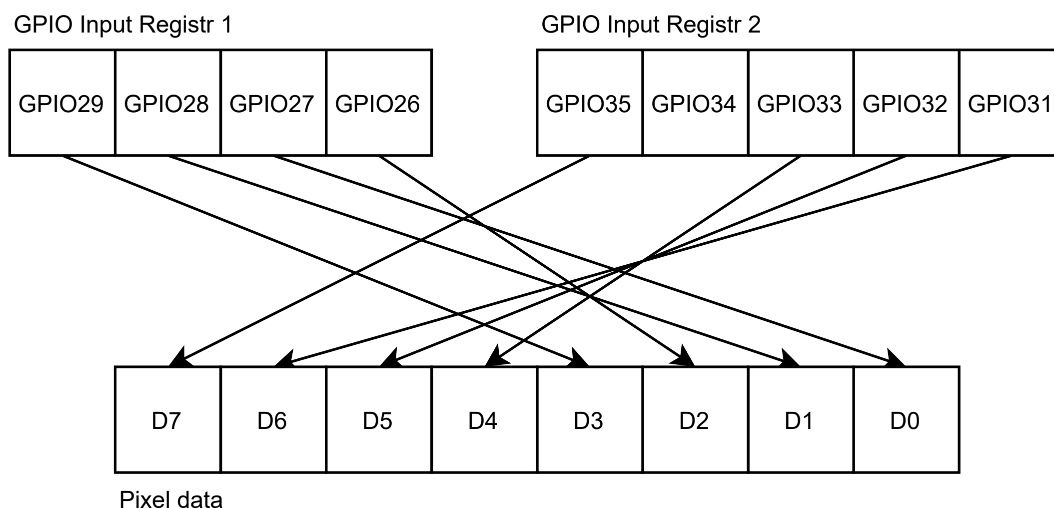
Zvolené řešení je tedy řešení metodou „s ohledem na počet křížení vodičů na desce plošného spoje“. Toto řešení sice mapuje datové bity komplikovanějším způsobem, který bude vyžadovat podstatně více operací pro sestavení správných dat, ale umožní vést vodiče na desce s malým množstvím křížení. Další vlastností tohoto řešení, vycházejícího z odlišně vyvedených portů na vývojových deskách revize 1 a 2 je odlišné mapování bitů GPIO portů na výsledná pixelová data v paměti zařízení. Firmware zařízení bude muset tuto skutečnost zohlednit. Pro první revizi je mapování bitů mezi registrem obsahující stav GPIO portů a proměnnou obsahující platná pixelová data ukázán na následujícím obrázku 3.7.

Jak je zde vidět, tak pro sestavení platných dat je zapotřebí nejen přečíst registry, ale také prohodit jednotlivé bity registrů tak, aby v paměti tvořily korektní pixelové data. Všechny tyto



Obrázek 3.7 Mapování GPIO portů na datovou sběrnici kamery optimalizováno s ohledem na množství křížení vodičů na desce plošného spoje pro platofrmu revize 1

prohozy znamenají vyšší množství instrukcí, které je budou realizovat. Efektivně to znamená instrukce pro maskování (logický AND), posuvy a spojování pomocí logického OR navíc. V druhé revizi mapování probíhá také. V důsledku vyvedení jiných GPIO portů na rozšiřujícím konektoru platformy vypadá jinak než u první revize a konkrétně vypadá způsobem ukázaným na následujícím obrázku 3.8



Obrázek 3.8 Mapování GPIO portů na datovou sběrnici kamery optimalizováno s ohledem na množství křížení vodičů na desce plošného spoje pro platformu revize 2

Množství instrukcí navíc lze odhadnout z množství křížení v předchozích obrázcích, ale přímá souvislost zde není. Obecně skupiny rovnoběžných šipek vedou k možnému zpracování těchto bitů najednou, a tedy k redukci množství potřebných instrukcí pro zpracování. Tím, že kód

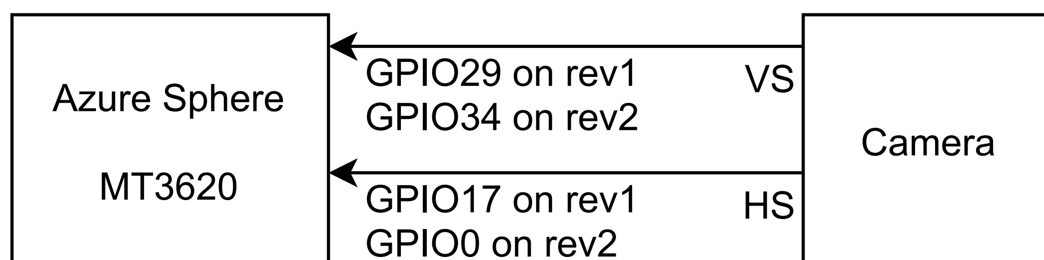
zpracovávající hodnoty, neobsahuje skokové instrukce ani instrukce které by s daty manipulovaly nad pamětí (až na počáteční čtení registrů a finální zápis do paměti), tak by mělo docházet k dobrému využití pipeline procesoru, a tedy poměrně efektivnímu a rychlému běhu daných instrukcí. Všechny operandy lze zpracovávat v registrech procesoru. Arm Cortex-M4 jich má dostatečné množství na potřebné zpracování. Je třeba ale poznamenat, že dosažení rychlého kódu také zaleží na kompilátoru, který musí být schopen využít všechny registry a využít je efektivně.

Posledním aspektem zohledněným při návrhu je hodinový vodič určující platnost dat na této sběrnici. Tento vodič je také připojen k GPIO portu platformy. Je však na něj vedeno omezení, že port ke kterému je připojen musí podporovat externí přerušení (tzn. součásti čipu musí být periferie schopná vygenerovat přerušení při náběžné hraně signálu na tomto vodiči). Toto přerušení podporují GPIO porty 0–23 a je třeba brát v potaz možnou kolizi s GPIO portem generujícím hlavní hodinový signál (MCLK), který k tomu využívá PWM kontrolér a který omezuje použití všech portů ve skupině na PWM výstup. Tedy pokud například GPIO2 slouží ke generování MCLK (což se tak na platformě revize 1 děje), tak porty GPIO0–3 nelze použít. V tomto případě se na desce revize 1 využívá port GPIO16. Podobně při využití GPIO5 jako PWM výstup (což se děje na platformě revize 2) nelze pro PCLK využít porty GPIO4–7. Využívá se port GPIO2 v tomto případě.

Při návrhu práce byla zvolena druhá možnost a tedy návrh s ohledem na množství křížení na desce plošného spoje. V potaz byla brána i skutečnost, že při zpracování obrázku lze bity nepřehazovat (stačí poze odfiltrovat nepoužité bity) a prohozy bitů provést nad sejmutým bufferem po skončení ukládání bufferu mimo časově kritickou sekci.

3.3.6 Připojení synchronizačních signálů kamery

Kamera obsahuje dva výstupní signály pro synchronizaci začátku obrazu a řádku obrazu. Jedná se o vodiče VS (vertical sync), jehož hrany určují začátek a konec snímku a HS (horizontal sync), jehož hrany určují začátek a konec řádku obrázku. Tyto vodiče je nutné připojit k GPIO portům, ale není na ně kladeno (narozdíl od hodinových signálů) nějaké významnější omezení. Volbu portu zobrazuje následující obrázek 3.9.



Obrázek 3.9 Zapojení synchronizačních vodičů kamery

3.3.7 Připojení mikrofonu

Řešení je navrženo s analogovým mikrofonem a potřebným obvodem. V úvahu přicházelo ještě řešení s digitálním mikrofonem připojeným na sběrnici I2S nebo mikrofon s modulací PDM. Přestože čip MT3620 obsahuje periferii pro obsluhu sběrnice I2S, tak žádný port s podporu této periferie není vyveden z čipu na modul, a tudíž ani nemůže být vyveden na samotnou vývojovou desku. Platforma má 2 porty s podporou analogově digitálního převodníku. Jedná se o porty

GPIO42 a GPIO43. Mikrofon je připojen k portu GPIO42. Bližší popis analogové části obvodu bude v sekci o návrhu desky plošného spoje dále v této kapitole. Zapojení také schematicky z pohledu platformy ukazuje diagram na následujícím obrázku 3.10.

Obrázek 3.10 Zapojení mikrofonu

3.4 Návrh desky plošného spoje

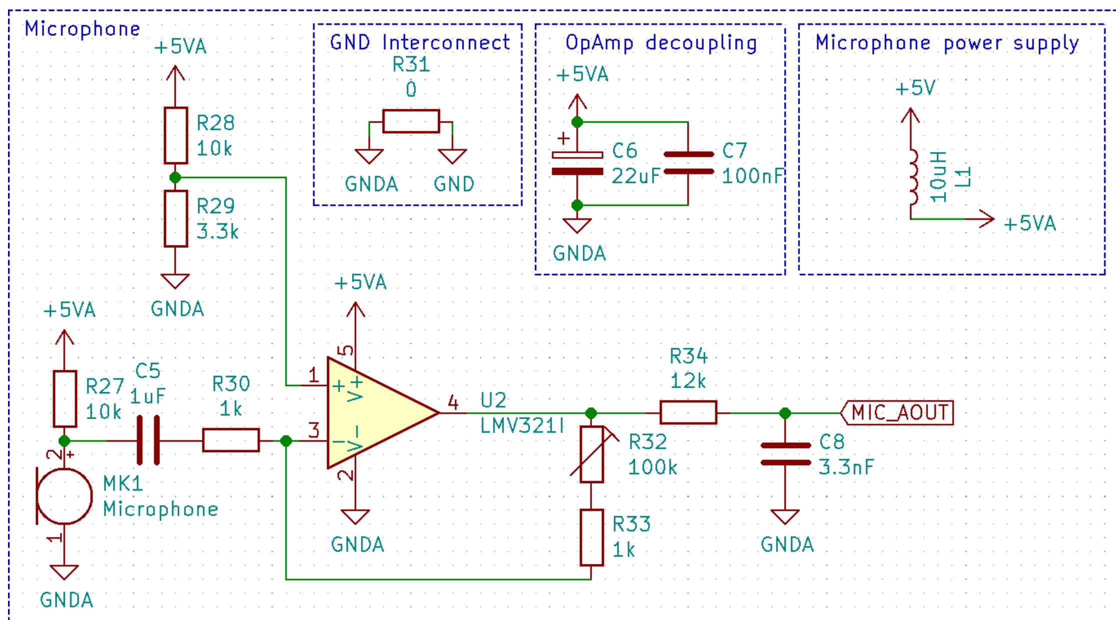
V rámci implementace praktické části práce se ukázalo prototypové zapojení komponent s využitím drátových vodičů a nepájivého kontaktního pole jako poměrně problematické. Propojení periferií čítá poměrně dost vodičů a podpora snímání audia také vyžaduje značné množství komponent citlivých na rušení, které se u prototypového řešení eliminuje velmi obtížně. Byla proto navržena deska plošného spoje, která umožní jednoduše a robustně připojit externí periferie k vývojové desce. Deska plošného spoje propojuje periferie z blokového schématu na obrázku 3.2 na straně 25. Výjimkou jsou senzory, které jsou k dispozici přímo na vývojové desce Avnet Azure Sphere Starter Kit. Deska plošného spoje tedy:

1. umožňuje připojení k vývojové desce pomocí rozšiřujících portů
2. obsahuje celý analogový obvod pro záchyt okolních zvuků
3. obsahuje konektor pro připojení modulu kamery
4. obsahuje infračervené LED pro noční přisvícení
5. obsahuje GPIO expandér pro připojení některých vodičů kamery a ovládaní IR LED

Návrh desky plošného spoje proběhl v nástroji Kicad. Návrh musel zohlednit navržené zapojení z kapitoly Logické propojení periferií zařízení a zejména podporovat obě revize vývojové desky, která má každá jinak vyvedené GPIO porty. Deska zároveň obsahuje obvod pro záchyt okolního zvuku složený z elektretového mikrofonu, operačního zesilovače a podpůrných pasivních součástek. Všechny volené komponenty s výjimkou konektorů a LED byly voleny jako SMD (Surface Mount Device) součástky, které se jednodušeji pájí při průmyslové strojové výrobě a zároveň jsou jejich pouzdra obvykle menší. Většina pasivních součástek byla volena v pouzdře 0603, které se jeví jako kompromis pro ruční pájení a velikost.

3.5 Obvod pro snímání okolního zvuku

Na desce plošného spoje se nachází podpůrný obvod pro záchyt okolního zvuku. Obvod byl navržen podle [41] a je tvořený mikrofonem, operačním zesilovačem, potenciometrem pro konfiguraci zesílení a dolní propustí. Obvod ukazuje výřez ze schématu zapojení desky na následujícím obrázku 3.11.



■ **Obrázek 3.11** Zapojení mikrofonu na desce plošného spoje

3.6 Firmware zařízení

V následující sekci bude popsán firmware zařízení a všechny jeho části. Tento firmware poběží na platformě Azure Sphere a bude ovládat výše popsaný HW a jednotlivé periferie. Firmware zařízení musí být navržen nejen s ohledem na použité komponenty, ale i s ohledem na zvolenou vývojovou desku pro její optimální použití.

Jako výhodné se tak jeví navrhnout aplikaci způsobem, že bude rozdělena na dvě části. Na hlavním jádru bude běžet část, která bude zajišťovat komunikaci s cloudem a na mikrokontrolérovém jádru poběží kód, který bude obsluhovat hardware. Aplikace na mikrokontrolérovém jádru zajistí zejména:

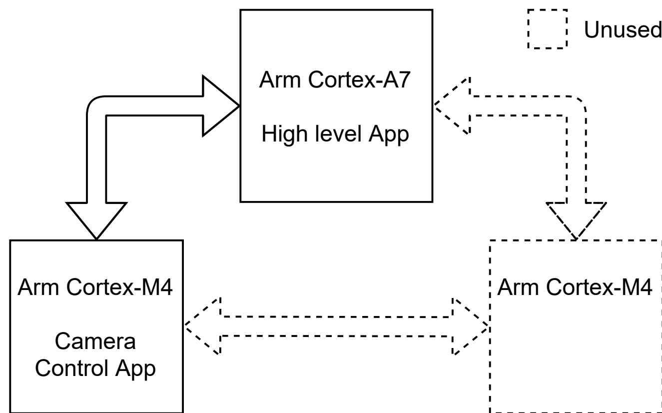
1. konfiguraci PWM pro generování hodin
2. konfiguraci registrů kamery
3. ovládaní GPIO expandéru
4. čtení a vyhodnocování hodnot senzorů
5. sejmutí snímku

Na mikrokontrolérovém jádru nepoběží žádná komunikace s internetem, ani konfigurace připojení k Wi-Fi síti, protože to platforma nijak z těchto jader neumožňuje. Na hlavním jádru poběží vysokoúrovňová aplikace, která bude zejména:

1. úkolovat mikrokontrolérovou aplikaci ovládající kameru
2. přijme sejmutý snímek
3. bezpečně se připojí ke službám cloudu
4. nahraje sejmuté snímky do cloudového úložiště

5. notifikuje cloudové služby informací o dostupnosti nových obrázků v uložišti.

Mezi aplikacemi bude provozován komunikační protokol a vysokoúrovňová aplikace bude mít možnost si od mikrokontrolérové aplikace vyčíst obrázek pomocí tohoto protokolu. Druhé mikrokontrolérové jádro se momentálně zdá nepoužité a jeho využití bude popsáno dále. Principiální schéma aplikace je ukázáno na následujícím obrázku 3.12.



Obrázek 3.12 Principiální struktura rozdělení úloh na procesorová jádra

3.6.1 Snímání obrázku z kamery

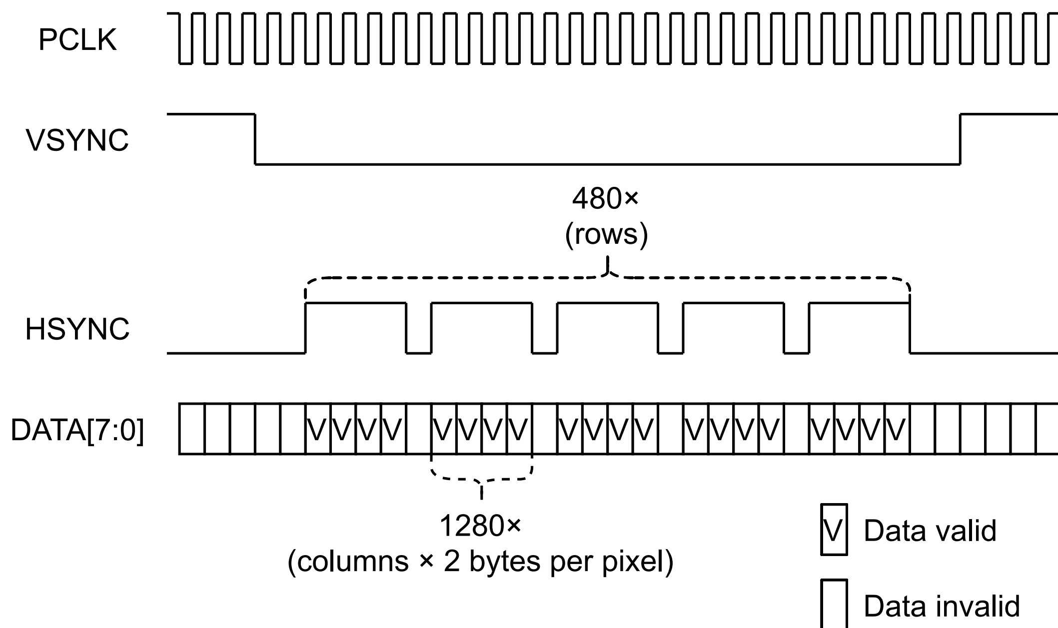
První popsaný aspekt návrhu řešení firmware bude samotné sejmutí obrázku z kamery. Jedná se o hlavní úlohu mikrokontrolérové aplikace. Předpokladem pro možnost sejmutí obrázku je:

1. hlavní hodiny kamery běží a jejich taktovací frekvence je v rozsahu 10–48 MHz [32]
2. byla provedena konfigurace registrů řadiče kamery pomocí sběrnice SCCB [32]

Aplikace musí zajistit splnění těchto předpokladů. Před čtením jakéhokoliv obrázku, musí nakonfigurovat periferii PWM pro generování 13MHz hodin (důvod proč je to zrovna takto byl popsán v předchozí sekci Generování hodin pro kameru) a musí s pomocí I2C kontroléru provést konfiguraci všech registrů přes rozhraní SCCB (které je s I2C kompatibilní). Jakmile jsou naplněny tyto předpoklady, tak kamera začne generovat hodiny PCLK (pixel clock), synchronizační signály VSYNC a HSYNC a data na paralelní sběrnici. Tato data pak generuje nepřetržitě dokud je splněn první uvedený předpoklad a nedošlo k jejímu vypnutí nebo resetování přes signál PWDN a RST. Strukturu a časování generovaných dat a synchronizačních signálů ukazuje následující obrázek 3.13.

Účel signálu VSYNC je informovat o začátku (sestupná hrana) a o konci (vzestupná hrana) celého jednoho snímku. Celý snímek je přenášen v době kdy signál VSYNC nabývá logické nuly. V době, kdy signál VSYNC nabývá logické jedničky se na sběrnici nenachází platná data a aplikace data z této sběrnice musí ignorovat. Signál HSYNC má obrácenou polaritu než signál VSYNC a indikuje začátek (vzestupná hrana) a konec (sestupná hrana) přenosu jednoho řádku snímku. Platná data obrázku se na sběrnici vyskytují v době, kdy HSYNC nabývá logické jedničky. Přenos pixelů snímku probíhá od shora dolů a zleva doprava. Aplikace, která čte data ze sběrnic, tak musí reflektovat toto schéma a číst data ze sběrnice pouze:

1. při náběžné hraně PCLK



Obrázek 3.13 Časování kamerové sběrnice

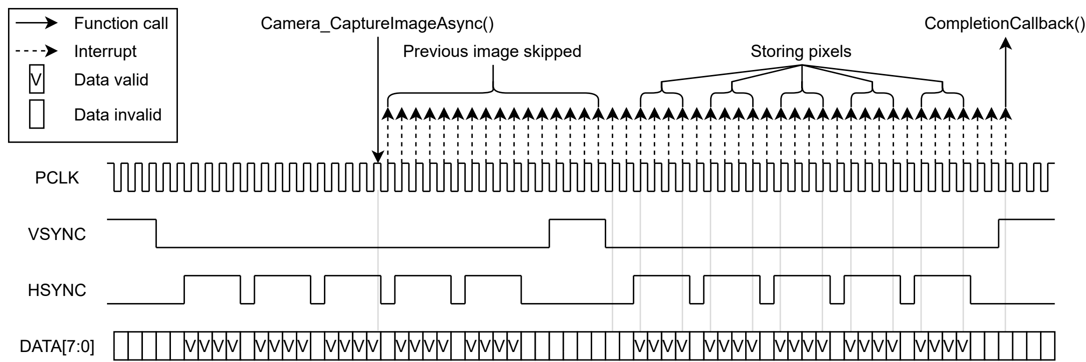
2. když signál VSYNC=0
3. když signál HSYNC=1

V návrhu aplikace je nutné zohlednit ještě jeden aspekt, a to že aplikace nejspíš začné sběrnici zpracovávat v průběhu přenosu snímku, nikoli na začátku jeho přenosu. Aplikace tak musí nejprve počkat, než doběhne vysílání tohoto snímku, jehož začátek „nestihla zachytit“ a dále číst ze sběrnice následující snímek, který má možnost sejmout celý.

Aplikace je navržená tak, že při zavolání funkce pro sejmutí obrázku `Camera_CaptureImageAsync` zapne generování přerušení při náběžné hraně na GPIO portu, ke kterému je připojen signál PCLK. Tato funkce je velmi jednoduchá a krom zapnutí zmíněného externího přerušení nastavuje jen výchozí hodnoty (globálních) stavových proměnných. V obsluze přerušení, které je voláno při každé náběžné hraně se navzorkují GPIO porty, ke kterým jsou připojeny synchronizační a datové porty sběrnice. Dojde k vyhodnocení stavu synchronizačních signálů a pokud byl kompletně přeskočen předchozí snímek a zároveň se na sběrnici nachází platná data (pozná se podle stavu synchronizačního signálu HSYNC), tak obsluha přerušení zapíše data do paměti (ve skutečnosti je nezapíše do paměti, ale pošle je přes FIFO do druhého procesorového jádra, nicméně to je podstatné z pohledu implementace a ne návrhu, který je popisován v této části) a inkrementuje zápisový pointer. Jakmile rutina dokončí čtení celého snímku (což pozná podle stavu synchronizačního signálu VSYNC), tak zakáže externí přerušení, aby se již dále nevolalo a zavolá callback, který byl registrován při volání funkce `Camera_CaptureImageAsync`. Obsluha přerušení zpracovávající náběžnou hranu PCLK musí být dostatečně rychlá a musí doběhnout dřív, než „přijde“ další náběžná hrana. Schéma a sekvence událostí při ukládání obrázku do paměti ukazuje následující obrázek 3.14.

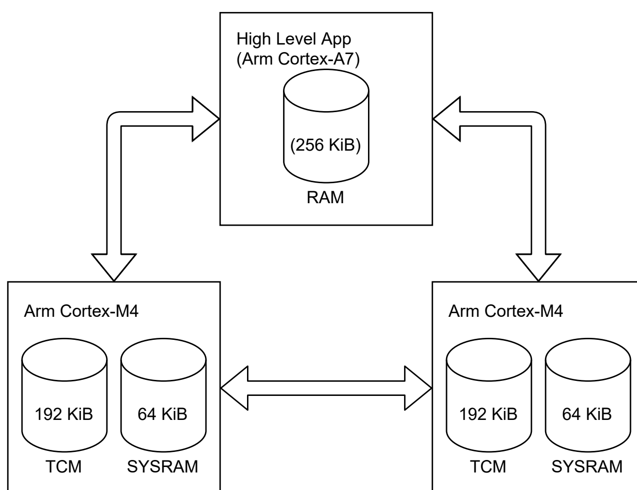
3.6.2 Ukládání snímku do paměti

Pro získání obrázku s rozlišením $640 * 480$ pixelů, kde každý pixel je kódovaný dvěma bajty kódováním RGB565 je zapotřebí mít k dispozici $640 \cdot 480 \cdot 2$ (614 400) bajtů operační paměti. Po



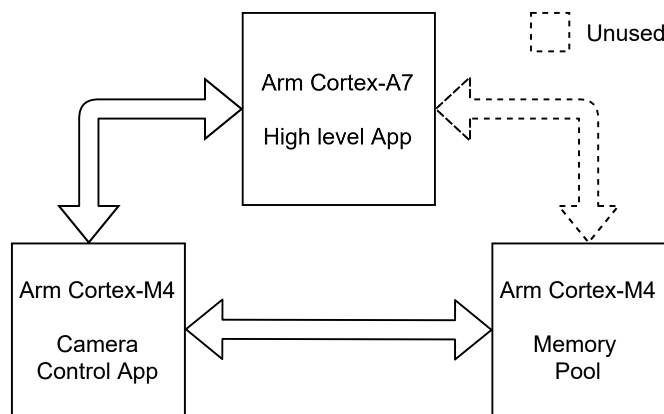
Obrázek 3.14 Sekvence událostí při čtení obrázku ze sběrnice do paměti

přepočtu na KiB se jedná o přesně 600 KiB. Tolik paměti, ale žádné jádro k dispozici nemá. Čip MT3620 obsahuje několik pamětí různých typů s různě omezenou dostupností. Vysokoúrovňová aplikace na aplikačním jádru si žádá o paměť od operačního systému, který ji přiděluje. Dokumentace [42] zmiňuje, že systém rezervuje pro vysokoúrovňové aplikace 256 KiB RAM, nicméně experimentálně se mi nikdy nepodařilo naalokovat tolik paměti. Patrně je do limitu zahrnutá i paměť kódového segmentu a další overhead spojený s během vysokoúrovňové aplikace nad operačním systémem Linux. Alokace o velikosti okolo 150 KiB se však jeví jako bezproblémové. Mikrokontrolérová jádra mají každé 192 KiB paměti označované TCM (Tightly Coupled Memory) a 64 KiB RAM označované jako SYSRAM. TCM i SYSRAM jsou paměti typu RAM. Mezi TCM a SYSRAM jsou rozdíly popsané v [43], ale z pohledu ukládání obrazových dat nejsou rozdíly příliš významné. Přestože systémy na čipu obsahující jádra Arm obvykle mají jeden adresní prostor, tak paměť TCM, i přestože je v obou jádrech mapována na stejnou adresu, je na obou jádrech odlišná a nezávislá (z definice pojmu TCM). To stejné na čipu MT3620 platí i pro paměť SYSRAM (u které to nevyplývá přímo z definice). Celkem tak zařízení obsahuje $(256) + 192 + 64 + 192 + 64 = 768$ KiB paměti RAM, která stačí na uložení obrázku o velikosti 600 KiB. Je ale potřeba obrázek distribuovat mezi (všemi třemi) jádry, protože žádné jádro nemá paměť k dispozici celou. Následující obrázek 3.15 vizualizuje dostupnou paměť na jednotlivých jádrech.



Obrázek 3.15 Dostupnost paměti na jednotlivých jádrech

V návrhu FW je tak potřeba zohlednit nutnost nejen přenosu servisních informací mezi aplikacemi (jako například příkaz zapni kameru, sejmi obrázek, předej data snímku atd), ale i obrazových dat směřujících do paměti na jiném jádru. Z potřeby využít i paměť třetího jádra, na kterém neběží žádná praktická aplikace vyplývá využití tohoto jádra a potřeba třetí aplikace, která nebyla ve schématu 3.12 na straně 33 vizualizována. Schéma aplikací běžících na zařízení Azure Sphere se zohledněním potřeby třetí aplikace pro zprostředkování paměti pro účely uchování snímku zobrazuje upravené schéma 3.16



Obrázek 3.16 Aplikace běžící na zařízení se zohledněním potřeby sdílet paměť

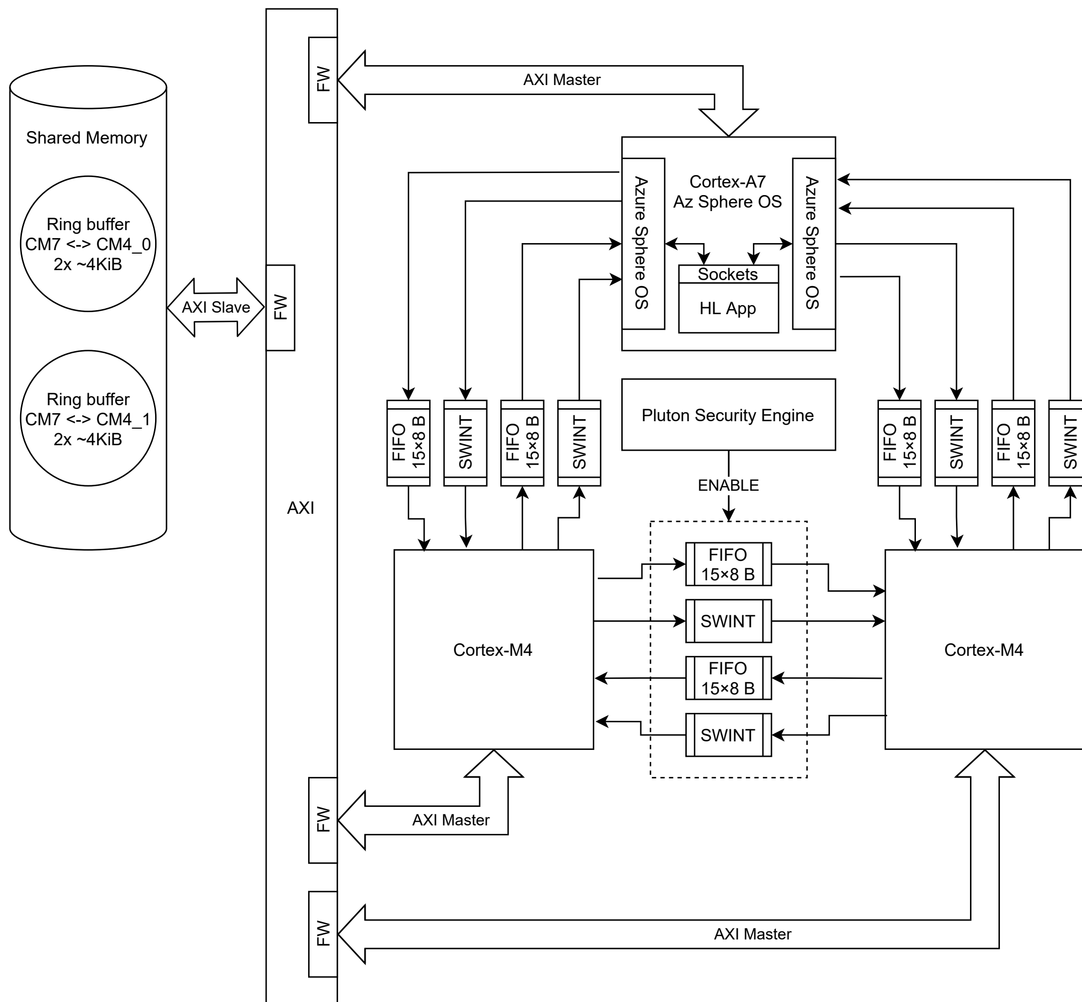
3.6.3 Komunikace mezi dílčími aplikacemi

Popis komunikačního protokolu sice není veřejně dokumentován, ale z existujících knihoven od výrobce čipu MT3620 [44], nezávislou knihovnou [45] a útržky kódu v rámci ukázek od Microsoftu [46] lze odvodit následující strukturu periferií, kterou je při návrhu potřeba brát v potaz. Je třeba poznamenat, že se nejedná o schéma sestavené na základě dokumentace, ale na základě určitého odhadu a některé zde popsané detaily nemusí být zcela přesné. Jedná se zejména o detaily propojení systémových sběrnic, které v rámci SoC mohou být implementovány jinak. Případné odlišnosti od skutečnosti však příliš nemění podstatu komunikačních protokolů a jejich funkci.

Mezi jádry existují komunikačni kanály, které se na HW úrovni skládají ze dvou FIFO bloků (jeden pro každý směr komunikace) a dvou bloku pro generování přerušení z software „na druhé straně“ tohoto komunikančího kanálu. Periferie pro generování tohoto přerušení je obvykle označována jako SWINT. SWINT je zde opět dva krát pro každý směr. Kanály jsou 3 (mezi každou dvojicí jader jeden). Pro komunikaci mezi hlavní aplikací se ještě používá sdílená paměť v adresním rozsahu, který není dokumentován ani v souhrnu dostupné paměti v dokumentaci na Microsoft Docs [47]. Mikrokontrolérová jádra získají pointer do této paměti pomocí FIFO daného kanálu. Mikrokontrolérová jádra mohou komunikační kanál využívat dle uvážení, ale pro komunikaci s vysokoúrovňovou aplikací (a tedy Azure Sphere OS) je vyžadováno dodržení speciálního protokolu a povolení komunikace v manifestu obou aplikací. Na straně vysokoúrovňové aplikace se pak rozhraní používá přes socket a patřičné ovládání periferie zajistí Azure Sphere OS. Komunikace mezi aplikacemi musí být navržena tak, aby dostatečně efektivně využila potřebné datové kanály. Komunikační kanál, který musí být návrhem FW reflektován vizualizuje následující schéma 3.17.

Po komunikačních kanálech je tedy potřeba přenést:

1. servisní informace v podobě příkazů a odpovědí mezi vysokoúrovňovou aplikací a aplikací ovládající kameru



■ **Obrázek 3.17** Schéma periferií používaných při komunikaci mezi jádry

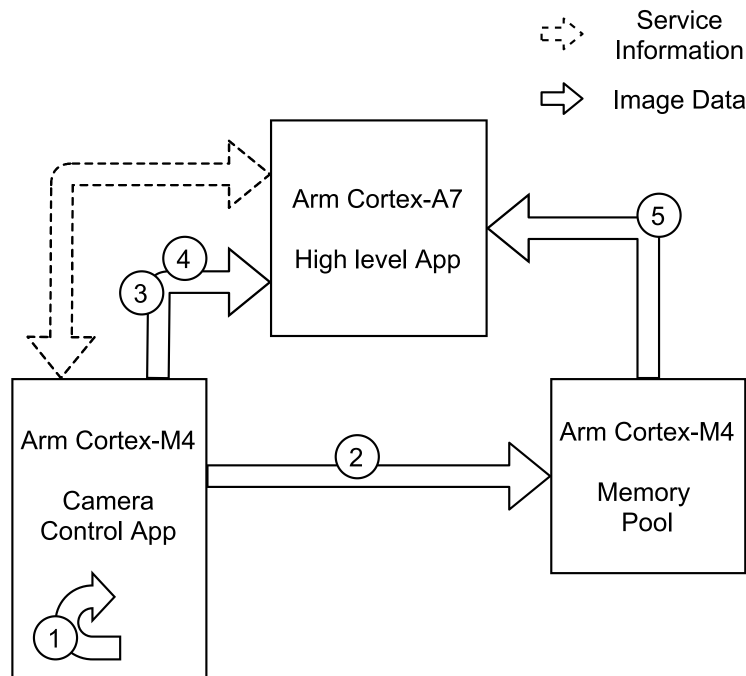
2. části obrázku přečtené aplikací ovládající kameru směřující do paměti ostatních aplikací

3.6.4 Sdílení paměti mezi jádry

Návrh přenosu servisních informací je poměrně jednoduchý a přímočarý. Aplikace si mezi sebou budou vyměňovat struktury, ve kterých budou kódované potřebné příkazy a odpovědi na příkazy. Mnohem zajímavější jsou však možnosti návrhu přenosu obrazových dat mezi jádry. Návrh těchto přenosů dramaticky ovlivňuje vlastnosti výsledného řešení a jeho realizovatelnost. Je potřeba brát totiž v potaz, že ukládání dat do paměti se děje v časově kritické části firmware (v obsluze přerušení), která musí doběhnout dřív než přijde následující hrana signálu PCLK. V rámci množství dostupného času není příliš problém stihnout vyhodnotit rozhodovací logiku na úrovni složitosti rozhodnutí, do kterého rozsahu (a tedy cílové) paměti aktuálně zapisovaný bajt (ve skutečnosti čtyřbajt, protože se jedná o 32bitový procesor a 32bitové operace jsou výhodnější) patří. Není však například čas na vytvoření a odeslání zprávy pomocí protokolu do vysokoúrovňové aplikace. To vyžaduje kopírování velkých bufferů, synchronizaci ve sdílené paměti a také čekání na obtížně predikovatelné potvrzení od operačního systému (který zprávu

předává přes socket vysokoúrovňové aplikaci). V rámci práce byly navrženy tři návrhy, z nichž první a třetí budou popsány. Druhý návrh se principiálně podobal finálnímu třetímu návrhu, ale měl řadu komplikujících vlastností, které podobně navržený třetí návrh odstraňuje. První a druhý návrh, které budou postupně popsány, byly v průběhu realizace praktické části práce realizovány a až v průběhu vývoje se první návrh ukázal jako nedostatečný.

První nejjednodušší návrh vychází z myšlenky přesouvat data do cílové paměti v obsluze přerušení. Aplikace v obsluze přerušení zhodnotí do jaké paměti ukládaný bajt patří (porovnáním čísla přijatého bajtu do virtuálních rozsahů) a tam ho přenese. Postup je vizualizován na obrázku 3.18.



Obrázek 3.18 Schéma datových toků při jednoduché implementaci přenášení paměti mezi jádry

První bajty obrázku tak uloží do svých lokálních bufferů (krok 1 na obrázku 3.18). Jamile místo v lokálních bufferech dojde, tak začne obrázky přenášet přes mezijaderný FIFO kanál do druhé mikrokontrolérové aplikace (krok 2 na obrázku 3.18) a jakmile dojde místo i tam, tak začne obrázek posílat do vysokoúrovňové aplikace (krok 3 na obrázku 3.18). Po skončení zachytávání obrázku pak vysokoúrovňová aplikace nahraje část, kterou získala, do cloudu. Tím si uvolní buffer, do kterého následně od mikrokontrolérové aplikace ovládající kameru obdrží přes stejný komunikační kanál (krok 4 na obrázku 3.18) část, která v době přenosu byla v kroku 1 uložena v paměti daného jádra. Tuto část následně nahraje do cloudu. Zbývá přenos části z druhé mikrokontrolérové aplikace. V podstatě jsou dvě možnosti, jak přenést část dat z memory poolu do hlavní aplikace. První možnost je využití kanálu mezi tímto jádrem a vysokoúrovňovou aplikací. Tuto variantu vizualizoval obrázek 3.18 krokem 5, ale existuje i možnost zasílat je zpět do mikrokontrolérové aplikace ovládající kameru, která následně provede přenos do vysokoúrovňové aplikace. To odstraní nutnost obsluhovat jeden komunikační kanál, ale zase zkomplikuje kód logiky aplikace.

Přístup popsaný v předchozím odstavci je však poměrně problematický, protože v obsluze přerušení, kde musí být zajištěno striktní dodržení časových požadavků, musí být ukládána obrázková data do lokálního bufferu (to není problém), poslána přes FIFO do memory poolu (to je

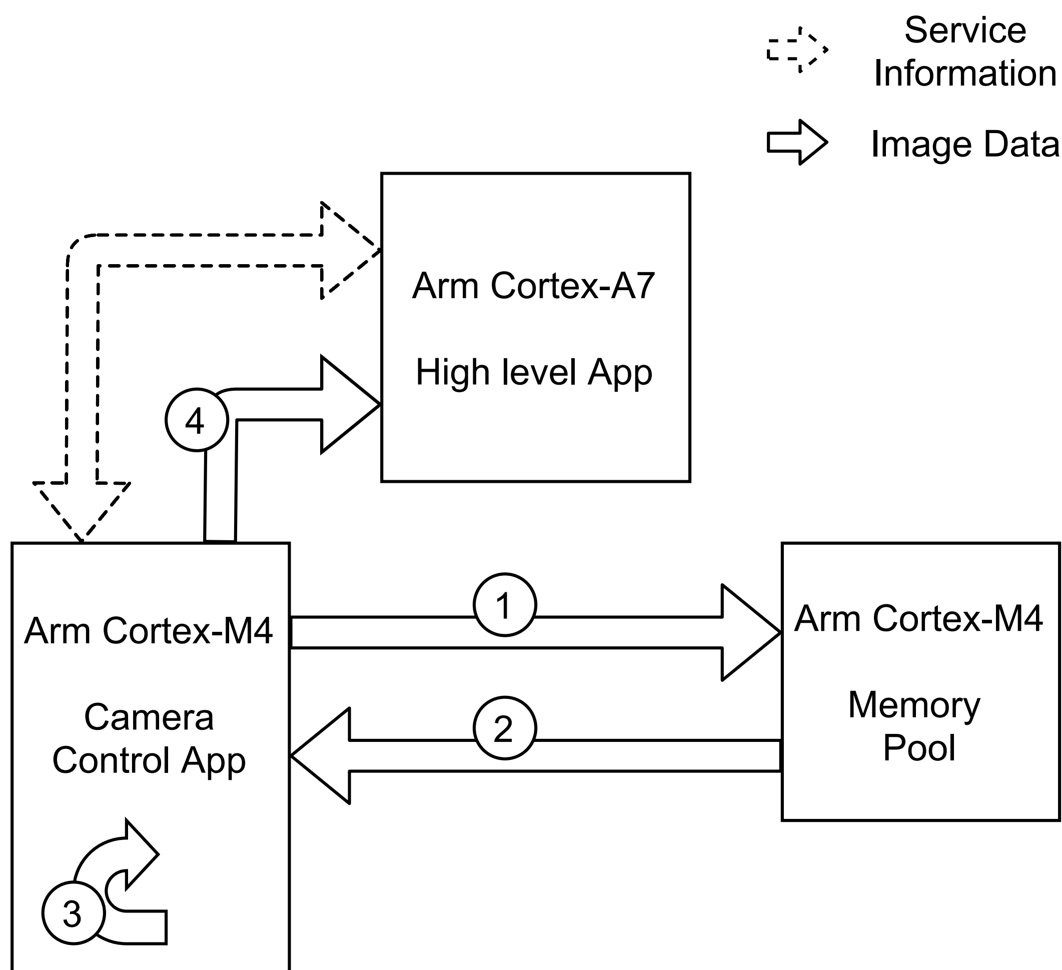
za normálních okolností čtení a zápis do dvou speciálních funkčních registrů, takže se také nejedná o problém), ale problém nastává v poslední části, kdy „zbytek“ obrázku musí být poslán do vysokoúrovňové aplikace. Odesílání dat do vysokoúrovňové aplikace je poměrně náročná operace. Bez optimalizací se jedná o:

1. sestavování hlavičky zprávy ve formátu požadovaném operačním systémem (obsahuje délku zprávy, GUID cílové aplikace a obsah zprávy)
2. načítání čtecího a zápisového pointeru ve sdílené paměti pro daný směr komunikace
3. vyhodnocení, zdali kruhový buffer ve sdílené paměti obsahuje dostatečné množství volného místa
4. zkopírování hlavičky a zprávy do sdílené paměti
5. inkrementování a upravení zápisového pointeru pro dodržení požadovaného zarovnání
6. vygenerování SW přerušení pro notifikaci operačnímu systému, že v kruhovém bufferu je nová zpráva

Při „inline“ implementaci bez použití knihovny, která by jinak tento proces vyřešila, lze proces redukovat o kopírování bufferu se zprávou (který se knihovně předává parametrem) do sdílené paměti. Data by bylo možné zapisovat rovnou do kruhového bufferu, i tak by ale zůstal poměrně komplexní proces s vyplňováním hlavičky a odesíláním zprávy, kdy by docházelo k porušení časových požadavků sběrnice. Nedává také smysl posílat každý bajt (efektivněji čtyřbajt) zvlášť, ale spíše je ukládat do bloku a poté celý blok odeslat najednou. Tímto se zrychlí zápis do bufferu, protože zápis prvních 511 bajtů (efektivněji 127 čtyřbajtů) se v podstatě vykoná jako zápis do normálního bufferu. Tímto se však problém nevyřeší, ale jen posune o 511 přenesených bajtů (efektivněji 127 čtyřbajtů) dál, protože k porušení časování dojde při ukládání 512. bajtu (efektivněji 128. čtyřbajtu).

Bylo tedy potřeba vymyslet přístup, který v obsluze přerušení nebude dělat přenos dat do vysokoúrovňové aplikace. Lepší přístupy (druhý a třetí) jsou přístupy, které na paměti nahlíží jako na velké FIFO a dělají některé přenosy paralelně. Třetí návrh je principiálně vizualizován následujícím obrázkem 3.19.

Oproti předchozímu popsanému návrhu je jednodušší v tom, že obsluha přerušení v aplikaci obsluhující kameru nemá žádnou podmínku rozdělující prvotní směr dat. Návrh všechna data směruje (krok 1 na obrázku 3.19) přes komunikační kanál do velkého FIFO do aplikace memory poolu. Zápis do tohoto kanálu je, jak bylo zmíněno, poměrně rychlý. Výhodou také je, že každý pixel obrázku se zpracovává stejně dlouho, což dělá aplikaci deterministictější a zjednodušuje kontrolu splnění časových požadavků. Ve stejné době, kdy aplikace plní v přerušení velké FIFO memory poolu, v hlavní smyčce aplikace běží kód, který bude z memory serveru číst data (krok 2 na obrázku 3.19) k sobě do pracovní paměti. V časově nekritické funkci v hlavní smyčce aplikace nad přijatými daty vyřeší jejich rozdělení, do které paměti data patří a tam je uloží nebo pošle. Fakticky tak hlavní smyčka aplikace získá z memory poolu úplně stejná data, která tam ta stejná aplikace před okamžikem poslala z přerušení. Výhodou je, že zatímco jejich zpracování při posílání bylo časově kritické, tak její zpracování při čtení už není. Hlavní smyčka aplikace nemusí data přečíst a uložit v rámci jednoho cyklu sběrnice kamery, ale musí to stihnout, než se zaplní velké FIFO v memory poolu. Deadline pro zpracování dat navíc není pevný, ale mění se. S každým zápisem se zkracuje doba (deadline), do kdy musí aplikace celý obsah velkého FIFO zpracovat, ale zároveň s každým zpracovaným čtyřbajtem se tato doba prodlužuje. Průběh zpracování přijatých dat vypadá následovně. Aplikace přečtená data nejprve směruje k sobě do lokálního bufferu (krok 3 na obrázku 3.19) a jakmile jej zaplní, tak data z velkého FIFO začne směrovat do vysokoúrovňové aplikace (krok 4 na obrázku 3.19). Jakmile se i ta zaplní, tak data z velkého FIFO přestane číst. Velké FIFO v memory poolu se tak začne zaplňovat poslední částí obrázku. Po dokončení přenosu vysokoúrovňová aplikace nahraje do cloudu část, kterou již má.



Obrázek 3.19 Schéma datových toků při přenášení obrazových dat do pamětí v jiných jádrech

Následně přes služební kanál požádá o další data z lokálního bufferu. Aplikace ovládající kameru ji zašle data stejným způsobem, jako to udělala v kroku 4. Nakonec si aplikace stejným způsobem požádá o poslední data z memory poolu. Aplikace ovládající kameru tato data přečte stejným způsobem jako četla data v kroku 2 na obrázku 3.19 a pošle je vysokoúrovňové aplikaci stejným způsobem, jako to dělala v kroku 4 na obrázku 3.19.

Druhý návrh se od třetího lišil tím, že přenášel logiku rozdělování paměti do druhé mikrokontrolérové aplikace. Aplikace ovládající kameru nejprve data zapisovala do svých lokálních bufferů a ostatní data zasílala vždy směrem k druhému mikrokontrolérovému jádru. Druhá aplikace si nejprve data zapisovala k sobě do lokálního bufferu a pak je začala posílat do vysokoúrovňové aplikace. Návrh tak vyžadoval zprovoznění komunikačního kanálu mezi vysokoúrovňovou a mikrokontrolérovou paměťovou aplikací a navíc část rozhodovací logiky stále byla součástí mikrokontrolérové aplikace ovládající kameru. Zároveň byl návrh poměrně komplikovaný pro vysokoúrovňovou aplikaci, protože ta musela přijímat data obrázku ze dvou mikrokontrolérových aplikací.

Pro celkové shrnutí výsledného (třetího) návrhu platí následující:

1. návrh nevyžaduje komunikační kanál mezi vysokoúrovňovou aplikací a paměťovou mikrokon-

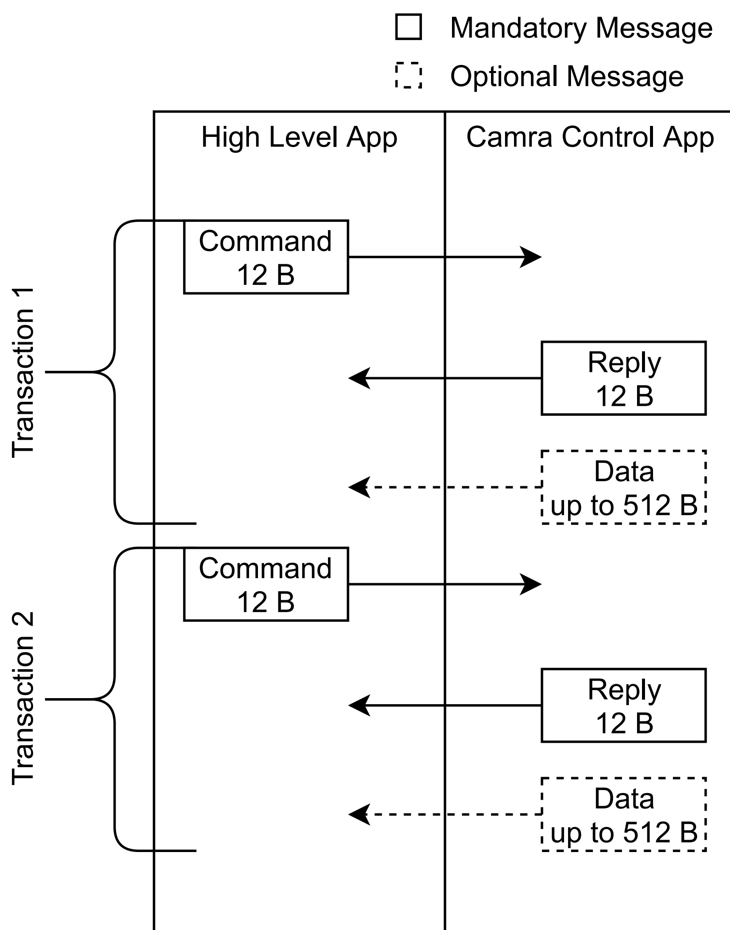
trolérovou aplikací

2. návrh nevyžaduje žádnou logiku memory poolu krom obsluhy FIFO
3. návrh nevyžaduje žádné zámky ani dočasné vypínání přerušení pro synchronizaci zápisových a čtecích pointerů v kterémkoliv přerušení
4. chyba mikrokontrolérové aplikace zpřístupňující paměť nezpůsobí selhání vysokoúrovňové aplikace (chyby této aplikace ošetřuje pouze aplikace obsluhující kameru)
5. je snadné ověřit splnění časových nároků sběrnice

3.6.5 Komunikační protokoly mezi aplikacemi

Na zařízení se využívají dva komunikační protokoly. Jeden mezi vysokoúrovňovou aplikací a mikrokontrolérovou aplikací ovládající kameru a druhý mezi oběma mikrokontrolérovými aplikacemi.

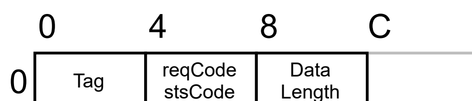
První zmíněny protokol je postaven nad protokolem požadovaným platformou Azure Sphere a spočívá v přenosu transakcí, které se skládají z požadavku, odpovědi a dat. Následující obrázek 3.20 ukazuje schéma komunikace.



Obrázek 3.20 Schéma komunikace mezi vysokoúrovňovou aplikací a aplikací ovládající kameru

Komunikaci může inicializovat pouze vysokoúrovňová aplikace. Nejedná se o obousměrný protokol, ale to vzhledem k jeho úloze ani není potřeba. Byl navržen s ohledem na jednoduchost a nikoli univerzálnost. V rámci transakce se vždy přenáší příkaz z vysokoúrovňové aplikace do mikrokontrolérové aplikace ovládající kameru. Na požadavek mikrokontrolérová aplikace ovládající kameru odpovídá odpovědí v podobném formátu jako byl přijatý požadavek. V závislosti na typu požadavku a odpovědi mohou z mikrokontrolérové aplikace následovat ještě obrazová data. To, zdali budou data odeslána nezávisí jen na typu požadavku, ale i na typu odpovědi. V případě chybové odpovědi nejsou žádná data zasílána.

Zpráva obsahující požadavek a odpověď má daný formát, který vizualizuje datagram na následujícím obrázku 3.21



Obrázek 3.21 Datagram zprávy pro přenos požadavku a odpovědi

Význam polí popisuje následující tabulka 3.1:

Offset	Datový typ	Název	Popis
0x0	uint32_t	Tag	Tag je náhodné číslo identifikující transakci. Vysokoúrovňová aplikace první Tag vygeneruje jako náhodné číslo a tagy dalších zpráv jsou vždy o jedno vyšší, než měla předchozí zpráva. Tag odpovědi na požadavek je shodný s tagem daného požadavku. Tag slouží k ujištění, že odpověď na požadavek odpovídá na očekávaný požadavek. V případě havárie některé z aplikací se může stát, že v meziprocesorovém komunikačním kanálu zůstane požadavek nebo odpověď z běhu před pádem a díky tagu vysokoúrovňová aplikace pozná že se jedná o odpověď na jiný požadavek než očekává a bude jej ignorovat.
0x4	uint32_t	ReqCode/StsCode	V případě požadavku se pole jmenuje ReqCode a v případě odpovědi StsCode. Seznam možných hodnot popisují tabulky 3.2 a 3.3.
0x8	uint32_t	Data Length	V případě požadavku je toto pole nastaveno na hodnotu 0. V případě odpovědi se jedná o délku volitelné datové části transakce v bytech. Pokud se jedná o odpověď na příkaz jehož součástí nejsou žádná data, pak pole nabývá hodnoty 0.

Tabulka 3.1 Význam polí příkazů

Podporované příkazy ReqCode uvádí následující tabulka 3.2.

Název příkazu	ReqCode	Přenos datové části
CCS_COMMAD_INIT_CAMERA_CONTROL_SERVER	0x01000000	ne
CCS_COMMAD_CAPTURE_IMAGE	0x04000000	výjimka
CCS_COMMAD_GET_CAPTURE_STATUS	0x05000000	ne
CCS_COMMAD_GET_DATA	0x06000000	ano

Tabulka 3.2 Přehled příkazů

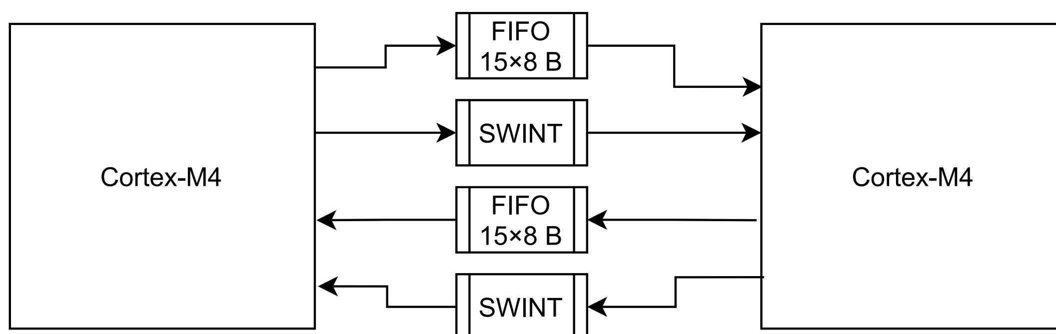
Podporované odpovědi StsCode uvádí následující tabulka 3.3.

Název odpovědi	StsCode	Přenos datové části
CCS_STATUS_INIT_COMPLETED	0x01000000	ne
CCS_STATUS_INIT_FAILED	0x01000001	ne
CCS_STATUS_CAPUTRE_IMAGE_COMPLETED	0x04000000	výjimka
CCS_STATUS_CAPUTRE_IMAGE_FAILED	0x04000001	ne
CCS_STATUS_CAPUTRE_STATUS_SUCCESS	0x05000000	ne
CCS_STATUS_CAPUTRE_STATUS_IN_PROGRESS	0x05000001	ne
CCS_STATUS_CAPUTRE_STATUS_BAD_RESOLUTION	0x05000002	ne
CCS_STATUS_GET_DATA_COMPLETED	0x06000000	ano
CCS_STATUS_GET_DATA_FAILED	0x06000001	ne

Tabulka 3.3 Přehled kódů odpovědí

V předchozích tabulkách byla uvedena výjimka u příkazů `CCS_COMMAD_CAPTURE_IMAGE` a příslušné odpovědi `CCS_STATUS_CAPUTRE_IMAGE_COMPLETED`. Tato výjimka vychází z nutnosti ukládat obrázková data v paměti vysokoúrovňové aplikace ještě v průběhu ukládání obrázku. Po vyvolání příkazu `CCS_COMMAD_CAPTURE_IMAGE` následuje odpověď dle popsaného protokolu, ale nenásleduje datová část dle tohoto protokolu (odpověď má nastavené pole Data Length na hodnotu 0). Nicméně i přesto dojde k přenosu dat po 512B blocích a navíc ne jen jednoho bloku, ale všech bloků, které se do paměti v této aplikaci musí zapsat. Toto je jediný přenos mezi vysokoúrovňovou aplikací a mikrokontrolérovou aplikací ovládající kameru, který neodpovídá popsanému protokolu, zejména z důvodu minimalizace množství logiky v mikrokontrolérové aplikaci.

Druhý komunikační kanál je komunikační kanál mezi mikrokontrolérovými aplikacemi a slouží k přenosu obrazových dat do velkého FIFO a čtení dat z něj. Na úvod je ptořeba si připomenout, jak tento mezijaderný kanál mezi jádry vypadá a funguje. Sturktura kanálu ovlivňuje možnosti návrhu takové komunikace. Kanál zobrazuje následující obrázek 3.22 a celé schéma se všemi kanály bylo k vidění na obrázku 3.17 na straně 37.



Obrázek 3.22 Komunikační kanál mezi mikrokontrolérovými aplikacemi

Mezi těmito jádry není žádná sdílená paměť, ale je zde poměrně rychlé FIFO, které má i tu

výhodu, že je možné ho v jednom cyklu z jedné strany plnit a z druhé číst. Není nutné udržovat žádné pointery tohoto meziprocesorového FIFO, protože vše je implementováno v HW. FIFO má kapacitu 15 položek a každá položka má kapacitu dvou čtyřbajtů. FIFO je navrženo pro přenos příkazu (jeden čtyřbajt) a parametru (druhý čtyřbajt). FIFO se čte a zapisuje pomocí registrů a obsahuje registr, ze kterého je možné vyčíst, kolik dat FIFO obsahuje. FIFO umí vygenerovat 4 typy přerušení. Dva z nich jsou hranové a vyvolají se při zápisu a čtení z FIFO a zbylé dva jsou úrovňové přerušení NE (Not Empty) a NF (Not Full). Krom FIFO komunikační kanál disponuje ještě možností vygenerování přerušení na protějším jádru kanálu.

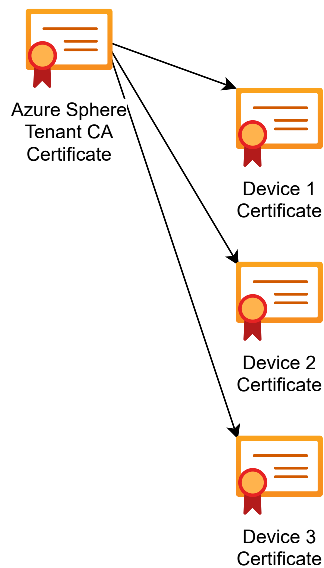
Jedna z možností návrhu je naimplementovat výměnu stejných nebo podobných zpráv jako probíhá mezi vysokoúrovňovou aplikací a aplikací ovládající kameru, takový návrh však je pro účely této komunikace poměrně komplikovaný a až příliš zatěžuje meziprocesorové FIFO servisními informacemi jako je tag, příkaz, délka dat atd. Byl proto zvolen návrh, že aplikace ovládající kameru do FIFO zapíše přímo obrazová data do obou čtyřbajtů najednou a memory pool si je z tohoto vstupního FIFO přečte a uloží do svého velkého bufferu. Pro ukládání dat do FIFO využívá přerušení NE, ve kterém všechna data z meziprocesorvého FIFO uloží do svého bufferu. Zároveň začne okamžitě plnit (z pohledu memory poolu výstupní) meziprocesorové FIFO. Tam v přerušení NF zaplňuje FIFO způsobem, aby toto FIFO vždy bylo ideálně plné. Memory pool tak s pomocí svého velkého bufferu zajistí prázdné vstupní FIFO, do kterého může aplikace ovládající kameru kdykoliv velmi rychle zapisovat a zároveň zajistí plné výstupní FIFO, ze kterého může logika v hlavní smyčce aplikace okamžitě číst a „přeposílat“ data do cílové aplikace. Výhoda tohoto řešení je, že na memory poolu nevyžaduje žádnou logiku v hlavní smyčce programu. Veškerá práce s daty (jejich zápis a čtení z velkého bufferu) se děje v obsluhách přerušení NE a NF. Díky tomu je implementace memory poolu v této podobě poměrně jednoduchá (a tudíž mimo jiné méně náchylná na chyby) a kamerová aplikace má k dispozici deterministický kanál ovládaný pouhým zápisem do registrů. Zároveň má možnost pomalu data z kanálu číst a vlastní takřka libovolně komplikovanou logikou buď data směrovat do cílové paměti (ve svém lokálním bufferu nebo do vysokoúrovňové aplikace) nebo data z memory poolu nečíst a využít tak paměti memory poolu. Návrh je omezen skutečností, že data, která zůstávají v memory poolu do něj musí přijít až po datech, které memory pool už v průběhu přenosu opouštějí. Reálně tak v memory poolu je vždy poslední část obrázku. Tato neflexibilita ale z pohledu implementace nepůsobí žádné problémy.

3.6.6 Připojení zařízení do cloudu

Další aspekt návrhu firmware, který se již týká vysokoúrovňové aplikace je její připojení ke cloudu. Tato část návrhu nepřináší příliš možností, protože platforma Azure Sphere přímo vybízí k použití doporučeného řešení. Platforma obsahuje přímo knihovnu libazureiot.so, kterou může vysokoúrovňová aplikace využít. Součástí knihovny jsou klienti pro služby Azure IoT Central a Azure IoT Hub. Klienti v této knihovně mají podobné rozhraní jako SDK těchto služeb pro jazyk C. Nejedná se, ale o zcela totožné SDK. Některé funkce byly z knihovny odstraněny kvůli redukci množství paměti zabrané knihovnou a zároveň byly přidány funkce specifické pro Azure Sphere, které umožňují například autentizaci pomocí certifikátu zprostředkovaným platformou.

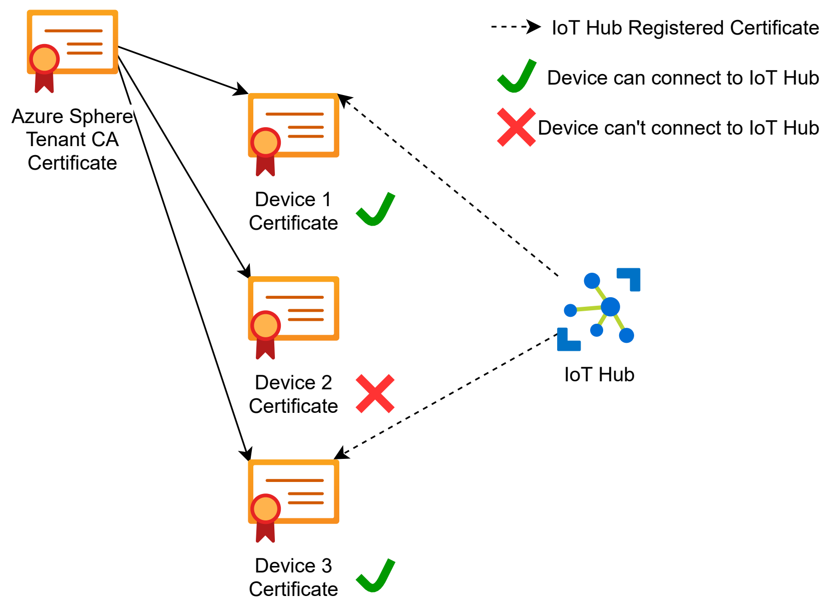
Komunikaci s dalšími cloudovými službami (jejichž návrh bude popsán dále v této práci) bude zajišťovat hlavně služba Azure IoT Hub, ke kterému se zařízení samo připojí. Služba vyžaduje autentizaci buďto předsdíleným klíčem, vlastním self-signed certifikátem nebo digitálně podepsaným certifikátem. Metoda předsdíleného klíče není příliš bezpečná, protože všechna zařízení mají stejný klíč a kompromitace jednoho zařízení znamená kompromitaci všech. Navíc předsdílený klíč je uložen ve firmware, takže klíč nemusí nutně uniknout z kompromitovaného zařízení, ale může například uniknout z počítače vývojáře, ať už v textové podobě ve zdrojových kódech nebo v binární podobě v zkompilovaném binárním souboru. V potaz tak připadá autentizace pomocí certifikátu. Platforma Azure Sphere poskytuje HW akcelerovanou bezpečnostní jednotku, která umí zprostředkovat digitálně podepsaný certifikát certifikační autoritou tenantu, do kterého je

zařízení registrované. Následující obrázek 3.23 vizualizuje podpisový řetězec certifikátů.



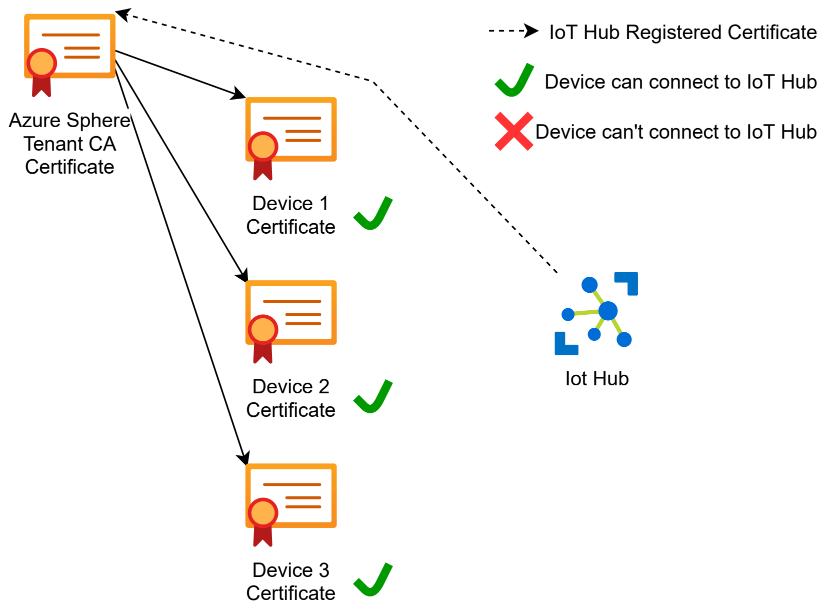
Obrázek 3.23 Podpisový řetězec certifikátů, které používají zařízení Azure Sphere k připojení do Cloudu

V IoT Hubu jde registrovat buďto certifikát konkrétního zařízení nebo certifikát certifikační autority. V případě použití certifikátu zařízení, se bude moct k danému IoT Hubu autentizovat pouze toto zařízení. Následující obrázek 3.24 ukazuje IoT Hub do kterého byl přidán certifikát prvního a třetího zařízení. IoT Hub, tak důvěřuje těmto dvěma zařízením, ale nedůvěřuje druhému zařízení i přestože je registrované do stejného tenantu.



Obrázek 3.24 Zařízení, kterým IoT Hub důvěřuje při registraci certifikátů zařízení

V případě registrování důvěryhodného certifikátu certifikační autority, která podepisuje jednotlivé certifikáty se budou moci k IoT Hubu připojit všechna zařízení, která mají podepsaný certifkát danou certifikační autoritou. V tomto případě mají k IoT Hubu přístup všechna zařízení, aniž by byl jejich certifikát přímo registrován v IoT Hubu. Ukázkový případ vizualizuje následující obrázek 3.25.



Obrázek 3.25 Zařízení, kterým IoT Hub důvěřuje při registraci certifikátu certifikační autority

To, že se zařízení umí platně autentizovat vůči IoT Hubu není jediný předpoklad povolení jeho využívaní. Zařízení ještě musí být v IoT Hubu registrováno. Registraci zařízení do IoT Hubu lze provést buďto ručně nebo pro každé zařízení nebo s využitím služby DPS (Device Provisioning Service). Služba DPS je provázaná s IoT Hubem a v případě, že si o její služby zařízení požádá, tak (zjednodušeně):

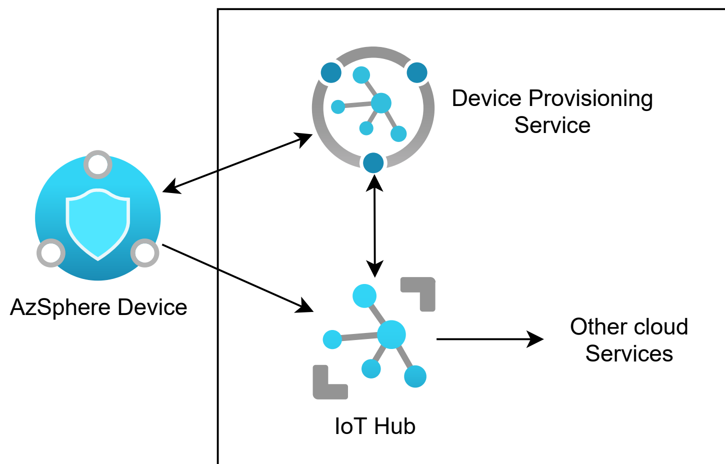
1. zaregistruje zařízení do IoT Hub
2. poskytne zařízení informaci, na jaké URL se IoT Hub nachází

Díky využití služby DPS tak odpadá:

1. nutnost registrovat zařízení ručně
2. nutnost mít URL adresu IoT Hubu součástí obrazu aplikace (nicméně vzniká povinnost mít součástí aplikace ID služby DPS)

Služba DPS je schopna registrovat všechna zařízení, kterým důvěřuje a ověřování důvěry vůči zařízení zde funguje naprosto totožně jako v případě IoT Hubu.

Navržené řešení bude využívat IoT Hubu a služby DPS. Zařízení se nejprve připojí k DPS, požádá o registraci a přístup k IoT Hubu a připojí se k IoT Hubu. Služby a toky informací při připojování zařízení ke cloudu vyzualizuje následující obrázek 3.26.



Obrázek 3.26 Využité služby a tok informací při připojování zařízení ke cloudu

3.6.7 Zápis do cloudového úložiště

S pomocí IoT Hubu má zařízení možnost zaslat do cloudu krátké zprávy a také přijmout z cloudu krátké zprávy. Nejedná se ale o vhodný protokol pro přenos obrázků do cloudu. Proto má IoT Hub SDK funkci `IoTHubDeviceClient_LL_UploadToBlob`, která prostřednictvím služby IoT Hub získá URL uložiště a název kontejneru, který je nakonfigurován ve službě IoT Hub a dočasný autorizační SAS token, který je potřeba k autentizaci zařízení vůči cloudovému uložišti. Problém je v tom, že funkce `IoTHubDeviceClient_LL_UploadToBlob` je jedna z funkcí, která byla z knihovny, dostupné na platformě Azure Sphere, vyřazena. Bylo tedy navrženo alternativní řešení, které v podstatě implementuje to, co implementuje `IoTHubDeviceClient_LL_UploadToBlob` ručně. Zařízení tedy odešle do IoT Hubu zprávu, že potřebuje přístup k uložišti, cloud vygeneruje krátkodobý autentizační token, který zařízení pošle. Co vše a jak přesně se stane v cloudu bude popsáno v jedné z následujících sekcí. Zařízení se pak samo připojí do cloudového uložiště a nahraje tam obrázek.

3.7 Návrh cloudových aplikací

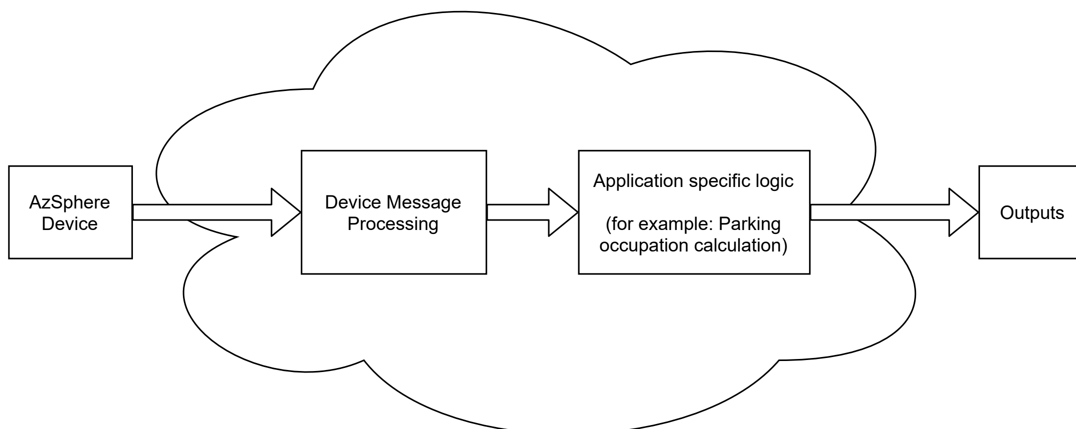
Druhá strana řešení je cloudová strana, která přijímá data ze zařízení a dále je zpracovává.

Cloudová aplikace bude rozdělena na dvě části. První část bude univerzální část, která zpracovává zprávy od zařízení a umožňuje zařízení uložit obrázek do uložiště. Druhá část pak bude aplikační částí, která implementuje konkrétní aplikaci jako je monitorování využití parkoviště. Univerzální část je stejná a společná pro všechny případné aplikace, nejen monitorovací systém parkoviště. Rozdělení aplikace a postup zpracování dat je vidět na následujícím obrázku 3.27.

3.7.1 Univerzální část aplikace

První části aplikace je univerzální aplikace, která je jednotná pro všechny typy aplikací. Účelem této aplikace je:

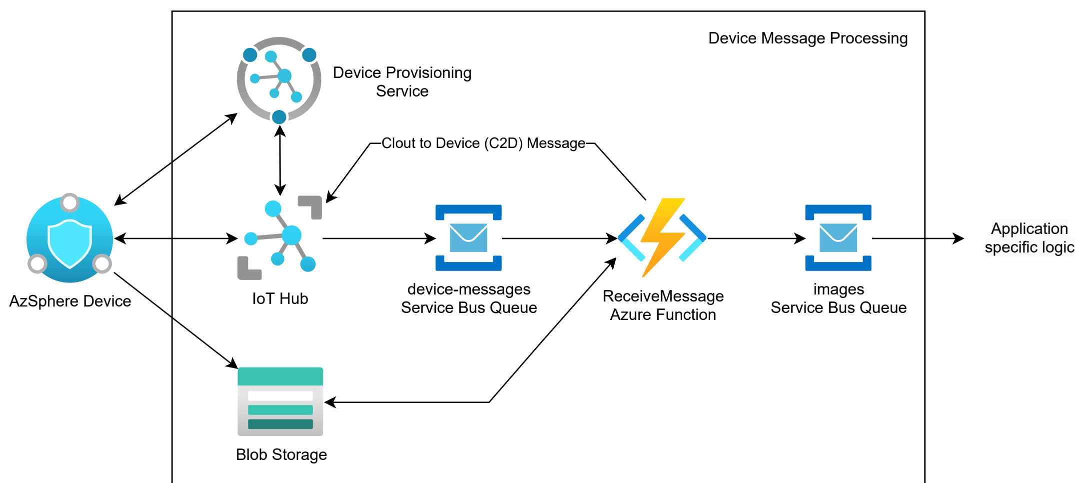
1. umožnit zařízení bezpečně se připojit ke cloudovým službám
2. umožnit zařízení získat přístup k nahrávání obrázků do uložiště



■ **Obrázek 3.27** Rozdělení cloudových aplikací

3. přijmout obrázek ze zařízení
4. převést obrázek ze zařízení z formátu RGB565 na formát JPEG
5. informovat aplikaci monitorovacího systému o novém obrázku

Aplikace je navržená podle konceptu PaaS a využívá tedy služeb cloudu Azure, které vhodně propojuje. Služby, které aplikace využívá, vizualizuje následující diagram na obrázku 3.28.



■ **Obrázek 3.28** Rozdělení univerzální a aplikační aplikace v cloudu

Na diagramu je dobré si všimnout směru šipek, které ukazují datové a informační toky. Na diagramu je také vidět, že zařízení komunikuje se třemi službami. První je služba Device Provisioning Service (DPS), která slouží k registraci zařízení do IoT Hubu. Proto jsou DPS a IoT Hub propojené. DPS je volitelná součást k IoT Hubu a umožňuje připojit zařízení k IoT Hub bez nutnosti manuálního zásahu (ať už ručně nebo skriptem) při zařazování zařízení do IoT Hubu. Jakmile je zařízení připojené k IoT Hubu, může do IoT Hubu zasílat zprávy. Zprávy zasílané mezi zařízením a IoT Hubem budou popsány dále v této sekci. IoT Hub směřuje všechny přijaté

zprávy přes frontu do bezstavové funkce, která jednotlivé zprávy od zařízení zpracovává. Funkce v závislosti na typu přijaté zprávy umí buďto poskytnout krátkodobý token umožňující zařízení dočasně zapisovat (ale jenom zapisovat) do uložiště nebo zpracovat obrázek, který zařízení do uložiště uložilo. Zařízení ukládá obrázky v surové podobě, tzn. jako bitmapu kódovanou formátem RGB565. Obrázek je ještě rozdělen na části, protože zařízení nemá dostatek prostředků, aby nahrálo obrázek najednou. Funkce bude při zpracování obrázku stahovat všechny části obrázku, spojí je, převede obrázek do formátu JPEG, uloží jej zpět do uložiště, smaže z uložiště všechny zpracované části a odešle do (z pohledu této aplikace výstupní) fronty zprávu, že obrázek je připraven k aplikačnímu zpracování. Protože tato část aplikace nedělá žádné analýzy zpracovávaných obrázků, tak se v diagramu ani nevyskytují žádné služby pro rozpoznávání obrazu. Podobně tato část aplikace nevyužívá žádnou databázi. Díky tomu, že aplikace neudržuje žádný stav, lze ji velmi dobře škálovat. Aplikace využívá následující služby cloudu k následujícím účelům:

1. Azure IoT Hub – Slouží k připojení zařízení ke cloudu a zprostředkovává přenos zpráv mezi zařízením a cloudem. Služba bude nakonfigurována, aby všechny zprávy ze zařízení směřovala do fronty služby Service Bus, která bude popsána dále. Služba udržuje spojení se zařízeními a umožňuje cloudovým službám odeslat zprávu do zařízení (což bude dělat bezstavová funkce popsaná dále).
2. Azure Device Provisioning Service – Slouží k automatizované registraci do IoT Hubu, aby nebylo nutné každé zařízení registrovat manuálně.
3. Azure Blob Storage – Blokové uložiště, do jehož kontejneru zařízení nahrává obrázky. Funkce zpracovávající zprávy ze zařízení je poté z tohoto uložiště čte. Uložiště je také použito pro ukládání překonvertovaných obrázků ve formátu JPEG a stavových informací některých dalších služeb v cloudu.
4. Azure Service Bus – Poskytuje fronty nezpracovaných zpráv pro předávání informací mezi dílčími službami. Aplikace využívá dvě fronty této služby. Jednu pro řazení zpráv přicházejících ze zařízení a druhou pro zprávy o zpracovaných snímcích, které jsou dostupné pro druhou část cloudové aplikace. Ve frontě jsou přímo zprávy ze zařízení ve formátu JSON. Součástí zpráv jsou také hlavičky, které přidává IoT Hub, umožňující identifikovat zařízení, které zprávu zaslalo a čas kdy ji zaslalo. Do první fronty zapisuje IoT Hub a čte z ní bezstavová funkce, která se stará o zpracování obrázků. Do druhé fronty jsou zapisovány názvy zpracovaných obrázků.
5. Azure Functions – Aplikace pro tuto službu bude zpracovávat zprávy ze zařízení a případně na ně odpovídat. V případě obdržení zprávy obsahující požadavek o dočasný přístupový token k uložišti, jej tato bezstavová funkce vygeneruje a zašle pomocí IoT Hubu do zařízení. V případě obdržení zprávy o nahrání nového obrázku, tak aplikace provede jeho zpracování.

Většina služeb popsaných v předchozí sekci jsou služby, kterým není potřeba dodávat libovolný kód a lze je nakonfigurovat buď ručně přes Azure Portal nebo s pomocí skriptu a utility Azure CLI nebo s pomocí volání REST API. Jediná část, která bude vyžadovat dodání aplikace je Azure Functions App, které musí vývojář dodat kód tvořící danou funkci. Funkce musí provést následující:

1. načte seznam všech částí obrázků dostupných v uložišti
2. stáhne části načtené v přechozím bodu z uložiště do paměti, aby vytvořila jeden souvislý buffer s obrazovými daty
3. zkontroluje velikost obrázku
4. převede obrázek z formátu RGB565 na JPEG

5. uloží obrázek ve formátu JPEG do uložiště
6. odstraní původní části obrázku ve formátu RGB565
7. zapíše do (z pohledu této aplikace výstupní) fronty cestu k obrázku ve formátu JPEG

Existuje i nespočet alternativních řešení a možností provázaní služeb. Tento je ale poměrně jednoduchý, levný na provoz a zároveň je dobře škálovatelný.

3.7.2 Aplikačně specifická část aplikace

Druhá část aplikace musí reagovat na zpracovaný nově příchozí obrázek vložený do fronty zpracovaných obrázků v první části aplikace. Součástí práce je implementovaná ukázková aplikace, která tohoto rozhraní využívá. Jedná se o systém monitorující obsazenost parkoviště. Struktura cloudové strany i aplikace zařízení byly navrženy univerzálně, takže nebude problém je využívat v případných dalších aplikacích. V případě implementace další aplikace je potřeba pouze zajistit, aby reagovala na obrázky, k nímž získá cestu do uložiště. Monitorovací aplikace získá z (z jejího pohledu vstupní) fronty cestu na obrázek ve formátu JPEG, který již byl předzpracován první části aplikace.

3.7.3 Monitorovací systém obsazení parkoviště

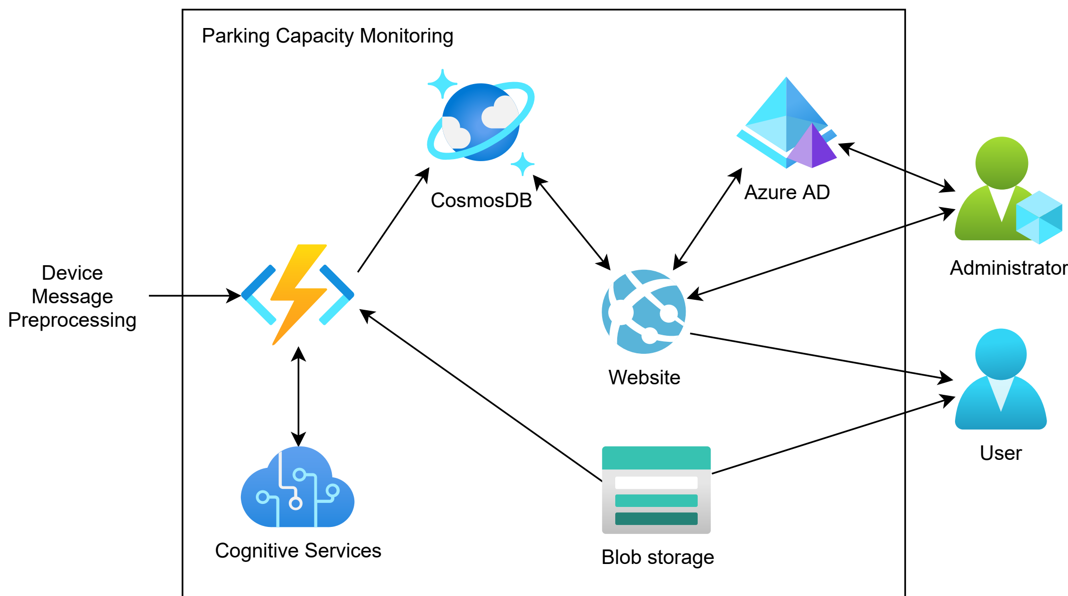
Cílem monitorovacího systému parkoviště je sledovat obsazenost parkoviště a vizualizovat ji ve webové aplikaci. Aplikace umožňuje přístup uživatelům a administrátorům. Uživatelé si mohou prohlédnout aktuální obsazenost parkovišť a administrátoři mohou tato parkoviště definovat a přiřazovat zařízení k jednotlivým parkovištím.

Systém je navržen s pomocí následujících cloudových služeb:

1. Azure Function App – Slouží ke zpracování obrázků přijatých z první části aplikace. Bezstavová funkce obrázek přečte a zavolá službu na rozpoznávání obrazu, která provede analýzu obrázku. Výsledek zapíše do databáze.
2. Azure Cosmos DB – Jedná se o velmi výkonnou a škálovatelnou databázi, kterou dílčí části aplikace využívají pro ukládání a čtení dat.
3. Azure App Service – Jedná se o hosting webových aplikací, na kterém poběží webová aplikace umožňující administrátorům konfigurovat parkoviště a přiřazovat k nim zařízení. Běžným uživatelům zobrazí aktuální stav obsazení parkoviště.
4. Azure Computer Vision – Slouží k detekci obejktů na obrázku. Bezstavová funkce tuto službu volá.
5. Azure Active Directory – Slouží k autentizaci administrátorů do webové administrace. Služba umožňuje zajistit moderní a bezpečnou autentizaci administrátorů do aplikace s pomoci infrastruktury Azure a nenutí tak aplikaci implementovat správu uživatelů ručně.

Provázání služeb vizualizuje následující obrázek 3.29.

I u této části aplikace se lze zamýšlet ohledně alternativních možností. Jednou z alternativních možností ke zvážení je například způsob autentizace uživatelů k administraci aplikace. Buď si to může aplikace zajistit sama (uživatelé a otisky jejich hesel budou v databázi) nebo aplikace může využít externích služeb jako je přihlášení pomocí účtu Microsoft, Google, Facebook, atd. Z pohledu bezpečnosti je druhé řešení výhodnější, protože minimalizuje dopady hrozeb jako je únik databáze v důsledku některé chyby. Zároveň takřka bezpracně umožňuje implementovat podpůrné bezpečnostní mechanizmy jako je dvoufaktorové ověření. Navržené řešení používá službu



Obrázek 3.29 Propojení cloudových služeb tvořících aplikaci monitorující parkoviště

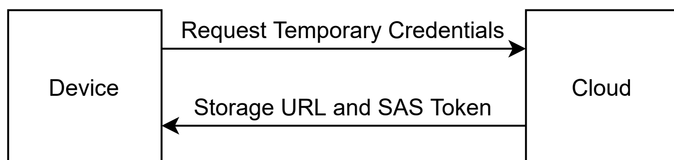
Azure Active Directory, která umožňuje přihlašování uživatelů, kteří jsou registrování v rámci specifikované Azure AD. Správa uživatelů probíhá v portálu Azure (nebo přes CLI a REST API) a nikoli v aplikaci.

3.7.4 Zprávy zasílané mezi zařízením a cloudem

Poslední částí návrhu je návrh formátu zpráv, které budou zasílány mezi zařízením a cloudem. Byl zvolen formát JSON pro jeho jednoduchost a dostupnost knihoven pro jeho zpracování. Na cloudové straně je podpora pro JSON integrována v základních knihovnách daných platforem.

Zařízení umí zasílat dva typy zpráv – požadavek o krátkodobý přístup k uložišti a notifikace dostupnosti nového obrázku. Každá zpráva ze zařízení má položku `Operation`, která určuje typ požadované operace. Položka `Operation` je i součástí odpovědi na požadavek o přístupový token.

Prvním typem zprávy je požadavek o krátkodobý SAS (Shared Access Signature) token pro přístup k uložišti. Průběh této komunikace je vizualizován na následujícím obrázku 3.30.



Obrázek 3.30 Komunikace mezi cloudem a zařízením za účelem získání krátkodobého přístupového tokenu k uložišti

Zpráva ze zařízení má strukturu ukázanou na násleudující ukázce 3.1.

Výpis kódu 3.1 Zpráva ze zařízení obsahující požadavek o krátkodobý přístupový token k uložišti

```
{
    "Operation":"GetShortTermSas"
}
```

Odpověď obsahuje informaci, že se jedná o zprávu odpovídající na akci `GetShortTermSas` a potřebné údaje. Strukturu odpovědi ukazuje následující ukázka 3.2.

Výpis kódu 3.2 Zpráva z cloudu obsahující odpověď na požadavek o krátkodobý přístupový token k uložišti

```
{
    "Operation":"GetShortTermSas",
    "StorageUrl":"...",
    "Sas":"..."
}
```

Druhý typ zprávy je notifikace o dostupnosti nového obrázku v uložišti po nahrání všech části obrázku do uložiště. Struktura komunikace je vizualizována následujícím obrázkem 3.31.

Obrázek 3.31 Komunikace mezi cloudem a zařízením za účelem notifikace o dostupnosti nového obrázku

Součástí požadavku je také identifikátor, podle kterého cloudová aplikace pozná, o jaký obrázek se jedná. Formát zprávy ukazuje následující ukázka 3.3.

Výpis kódu 3.3 Zpráva ze zařízení obsahující požadavek o krátkodobý přístupový token k uložišti

```
{
    "Operation":"ProcessImage",
    "Payload":"image prefix"
}
```

Kapitola 4

Implementace řešení

V následující části bude popsána implementace řešení tak, jak bylo navržené v části Návrh systému. V této kapitole bude popsáno, jakým způsobem byly naimplementovány všechny jednotlivé části systému. Budou také popsány některé problémy vývoje aplikací pro platformu Azure Sphere a jakým způsobem byly v rámci této práce řešeny.

4.1 Firmware zařízení

Firmware je implementován dle návrhu řešení a je rozdělen na tři aplikace běžící na jednotlivých jádrech. Úkolem jednotlivých aplikací je:

1. Vysokoúrovňová aplikace:
   a. připojit se k IoT Hubu pomocí DPS a udržovat spojení s IoT Hubem
   b. získat dočasný přístup k uložišti
   c. instruovat mikrokontrolérovou aplikace ovládající kameru k sejmutí snímku
   d. přijmout obrazová data v průběhu snímání
   e. odeslat přijatá data do cloudu, požádat o zbytek dat a také je odeslat do cloudu
   f. notifikovat služby v cloudu přes IoT Hub o dostupnosti obrázku
2. Mikrokontrolérová aplikace ovládající kameru:
   a. zpracovávat zprávy z vysokoúrovňové aplikace
   b. nastavit PWM pro generování hodin ke kameře
   c. konfigurovat registry kamery
   d. zahájit asynchronní snímání dat konfigurací externího přerušení při náběžné hraně PCLK
   e. v obsluze externího přerušení zasílat data do memory poolu
   f. přenášet data z memory poolu do správné paměti v průběhu snímání
   g. přenést sejmutá data do vysokoúrovňové aplikace
3. Mikrokontrolérová aplikace zprostředkovávající paměť (memory pool):
   a. přijmout data ze vstupního FIFO do velkého FIFO
   b. plnit data z velkého FIFO do výstupního meziprocesorového FIFO
   c. resetovat buffer při obdržení softwarového přerušení

4.1.1 Použité knihovny

Pro vývoj vysokoúrovňové aplikace nebylo potřeba do projektu zavádět žádné zásadní knihovny. Jediná knihovna zavedená do tohoto projektu je knihovna `pdjson` na parsování JSON, který se používá při komunikaci s cloudovými službami. Všechny ostatní knihovny jsou součástí SDK a operačního systému. Jedná se o následující knihovny:

1. standardní knihovna jazyka C – Využita pro práci s dynamicky alokovanou pamětí, textovými řetězci, formátováním řetězců, získávání času a uspávání programu.
2. standardní knihovny POSIX – Využita pro práci se sockety kvůli meziprocesorové komunikaci a generování náhodných čísel.
3. knihovny Azure Sphere SDK – Využita pro logování.
4. IoT Hub client – Pro přístup k cloudovým službám.
5. DPS client – Využita pro připojování k IoT Hubu pomocí DPS a autentizaci prostředky platformy Azure Sphere.
6. pthread – Využita pro udržování spojení s IoT Hubem pomocí vláken a synchronizaci pomocí mutexu.
7. CURL – Využita pro nahrávání snímků do cloudového uložiště.

Většina těchto knihoven jsou knihovny, které se v aplikacích běžících nad operačním systémem Linux používají běžně, případně se jedná o SDK pro jazyk C ke cloudovým službám. Řada těchto knihoven je však omezená, a ne všechny funkce jsou dostupné viz. popis návrhu odesílání obrázku v sekci Zápis do cloudového uložiště. Všechny tyto knihovny jsou dobře dokumentované a jejich použití bylo poměrně jednoduché a přímočaré.

Mnohem komplikovanější je situace s mikrokontrolérovými jádry. Mikrokontrolérová jádra a všechny periferie byly vyvinuty společností MediaTek a narozdíl od klasických mikrokontrolérů není jejich hlavní dokumentace příliš veřejná. Dokumentace je dostupná pouze pod NDA. V průběhu let, kdy byl čip MT3620 dostupný na trhu se objevily dvě knihovny, které periferie čipu MT3620 umí ovládat. Jedná se o:

1. knihovny M-HAL a OS-HAL [44] od samotného výrobce čipu MediaTeku
2. nezávislou knihovnu MT3620 M4 Drivers [45] od Codethink Labs

Knihovny jde do značné míry i kombinovat, ale vyžaduje to například úpravu tabulky vektorů přerušení, aby se volaly správné obsluhy přerušení správné knihovny. Knihovny jsou využity pro:

1. ovládaní GPIO portů
2. ovládaní PWM kontroléru
3. ovládaní I2C kontroléru
4. ovládání ladícího UARTu
5. ovládání ADC kontroléru
6. konfigurace periferie externího přerušení
7. konfiguraci a komunikaci meziprocesorového komunikačního kanálu

Kvalita knihoven je velmi proměnlivá. Obě knihovny mají řadu problémů jako je nepřesný význam konfiguračních možností, chybné zajištění zamykání zdrojů pro atomický přístup k zdrojům z hlavní smyčky a přerušení atd. I knihovna samotného výrobce čipu obsahuje poměrně dost chyb, zejména její část OS-HAL, která spoléhá na RTOS (Real Time Operating System). Při použití knihovny bez RTOS, knihovna vůbec nedělá zamýkání zdrojů sdílených mezi hlavní smyčkou aplikace a přerušením, případně jen velmi triviálně a nekorektně. Původně bylo řešení implementováno pomocí nezávislé knihovny MT3620 M4 Drivers, ale to se ukázalo problematické už při konfigurování externího přerušení. Na základě analýzy kódu obou knihoven jsem dospěl k poznatku, že kontrolér obsahuje funkci na ošetření zákmitů, pro použití externího přerušení pro detekci stisku tlačítka. Tuto funkci jde zapnout nebo vypnout, ale knihovna tuto funkci v metodě `EINT_ConfigurePin` v souboru `GPIO.c` na řádku 236, která je zobrazena v následující ukázce kódu 4.1, vždy zapíná a neumožňuje to nijak konfigurovat, narozdíl od ostatních vlastností této funkce.

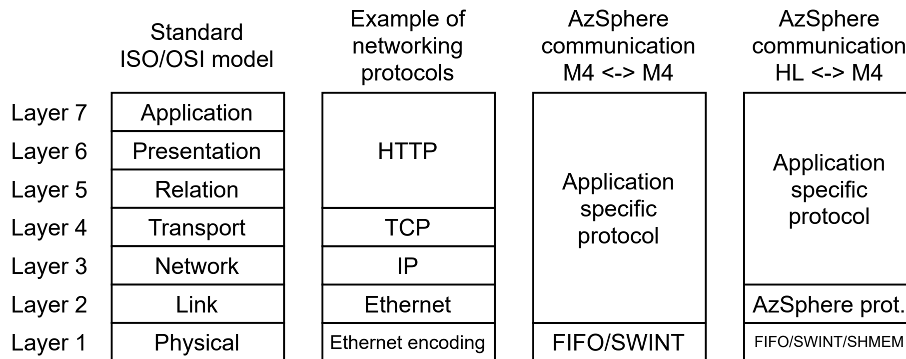
Výpis kódu 4.1 Funkce EINT_ConfigurePin [48]

```
222 int32_t EINT_ConfigurePin(
223     uint32_t           pin,
224     gpio_eint_attr_t  *attr)
225 {

        // zkráceno

236     MT3620_IRQ_DBNC_FIELD_WRITE(dbnc_con, en, pin, true);
237     MT3620_IRQ_DBNC_FIELD_WRITE(dbnc_con, pol, pin, attribute.positive);
238     MT3620_IRQ_DBNC_FIELD_WRITE(dbnc_con, dual, pin, attribute.dualEdge);
239     MT3620_IRQ_DBNC_FIELD_WRITE(dbnc_con, prescal, pin, attribute.freq);

        // zkráceno

245 }
```

Domnívám se, že autor si při psaní knihovny myslel, že příznak pojmenovaný `en` zapíná celé externí přerušení a ne jen jednu doplňkovou funkci této periferie. V knihovnách od výrobce čipu je toto implementováno správně a možnost vypnutí této funkcionality je z knihovny dostupné. Přestože nezávislá knihovna není úplně bezchybná, obsahuje řadu zajímavých a inspirativních informací, které není možné jinde získat, protože takřka veškerá dokumentace k periferiím je veřejně nedostupná. Při řešení bylo místo této knihovny využito knihovny výrobce čipu, byť není také zcela bezproblémová.

Druhým aspektem, kterým se knihovny velmi odlišují je obsluha meziprocesorových komunikačních kanálů. Komunikační kanál obvykle nazývají MBOX (Mail Box), což je obvyklý pojem pro meziprocesorovou komunikaci s využitím sdílené paměti. Na čipu MT3620 se MBOX kanálem rozumí meziproceosorové FIFO, SWINT a v případě komunikace s vysokoúrovňovou aplikací ještě oblast ve sdílené paměti RAM. Struktura komunikačního kanálu a jeho možnosti byla ukázána na obrázku 3.21 na straně 42. Pro komunikaci s vysokoúrovňovou aplikací je však zapotřebí implementovat protokol, kterému Azure Sphere OS (který běží na jádru, se kterým se komunikuje) rozumí. Azure Sphere OS obsluhuje tyto periferie a přenosy a data z nich dále předává do socketu vysokoúrovňové aplikaci. Strukturu protokolů jde přiblížit srovnáním s vrstvením síťovým protokolů podle modelu ISO/OSI [49]. Model ISO/OSI se používá k rozdělení nezávislých síťových protokolů do nezávislých (ale souvisejících) vrstev. Komunikační protokol, který používá platforma Azure Sphere ke komunikaci mezi mikrokontrolérovým jádrem a vysokoúrovňovou aplikací lze v podstatě namapovat na první dvě vrstvy ISO/OSI modelu. Mapování na protokoly standardního ISO/OSI model vizualizuje následující obrázek 4.1.

Fyzickou vrstvu tvoří periferie, které komunikaci fyzicky zajišťují. V případě síťových protokolů se na této vrstvě popisuje kódování nebo modulace použita pro fyzický přenos bitů. Na platformě Azure Sphere této vrstvě odpovídají periferie, které fyzicky přenáší data mezi jádry. Na



Obrázek 4.1 Mapování mezijaderného komunikačního protokolu na ISO/OSI model

druhé síťové vrstvě se nejčastěji používá protokol Ethernet, který obsahuje informace o zdrojové a cílové MAC adresy, délku paketu, obsah zprávy a kontrolní součty. Na platformě Azure Sphere se jedná o protokol vyžadovaný operačním systémem a má velmi podobnou strukturu. Posílá se jen cílová adresa, protože zdroj je znám na základě kanálu, po kterém zpráva přichází a adresa nemá formát MAC adresy, ale má formát identifikátoru komponenty, což je 128bitové číslo obvykle formátované jako GUID. Podobně obsahuje délku zprávy a obsah. Neobsahuje kontrolní součet, protože komunikační kanál je implementován uvnitř čipu a nemůže (s výjimkou vady obvodu v čipu, která je ale velmi nepravděpodobná) na něm dojít k chybám.

Obě knihovny obsahují nízkoúrovňový ovladač pro FIFO, který je možné použít pro nastavování meziprocesorvého FIFO, resetování jeho obsahu, zápisu do něj a čtení z něj. Podobně obě knihovny umí ovládat periferii SWINT pro generování přerušení „na druhé straně“ kanálu. Tato implementace je v obou knihovnách poměrně funkční a korektní. Obě knihovny tak umožňují komunikaci mezi mikrokontrolérovými jádry (jako je v případě monitorovacího systému popisovaného v této práci komunikace mezi mikrokontrolérovou aplikací ovládající kameru a memory poolem). Při pohledu na mapování na ISO/OSI model se jedná o první vrstvu, kterou tedy obě knihovny správně a funkčně implementují. Horší je to s implementací protokolu používaného na druhé vrstvě. Nezávislá knihovna od Codethink Labs obsluhu tohoto protokolu v rámci knihovny nepodporuje, ale implementuje tento protokol v jednom z ukázkových projektů v nezávislých souborech. Cílem vývojářů bylo, aby se API podobalo socketovému API, které je využito ve vysokoúrovňové aplikaci. Knihovna od výrobce čipu to sice implementuje v rámci knihovny, nicméně také v externím a nezávislém souboru, ve kterém se navíc nedodržují jmenné konvence dodržované ve zbytku knihovny. Kód je ale z několika důvodů při reálném použití nefunkční. Kód knihovny sice implementuje protokol, ale zapisuje špatné pointery na špatná místa sdílené paměti a navíc ignoruje skutečnost, že ke stejným pointerům přistupuje i operační systém z jiného procesorového jádra.

Knihovna MediaTeku pro udržování pointerů do sdílené paměti používá pointery pojmenované inbound a outbound. Jedná se o pointery do sdílené paměti RAM. Pointery knihovna vrátí uživateli při inicializaci a ten ji je při každém volání předává zpět. První zajímavou vlastností je, že volání `EnqueueData` a `DequeueData` (což jsou funkce pro zápis do komunikačního kanálu a čtení z komunikačního kanálu) vyžadují parametry inbound a outbound každá v obráceném pořadí, na což lze snadno zapomenout při kopírování a upravování kódu. Komunikační protokol specifikuje, že na začátku těchto bufferů je uložen zápisový a čtecí pointer [50]. Ve skutečnosti to není pointer, ale index od začátku bloku sdílené paměti. Tyto pointery a jejich lokaci definuje a vyžaduje protokol. Každá strana komunikace jeden pointer zapisuje a jeden čte. Například u bufferu určeného pro komunikaci směrem z vysokoúrovňové aplikace k mikrokontrolérové aplikaci zápisový pointer udržuje operační systém a čtecí pointer udržuje mikrokontrolérová aplikace. Knihovna proto na tuto paměťovou oblast mapuje strukturu definovanou v souboru `os_hal_mbox_shared_mem.h` na

řádcích 15–32. Strukturu ukazuje následující ukázka kódu 4.2:

Výpis kódu 4.2 Struktura mapující zápisový a čtecí ukazatel ve sdílené paměti [51]

```
typedef struct {
        /* <summary>
         * <para>Enqueue function uses this value to store the last position
         * written to by the real-time capable application.</para>
         * <para>Dequeue function uses this value to find the last position
         * written to by the high-level application.</summary>
         */
        u32 writePosition;
        /* <summary>
         * <para>Enqueue function uses this value to find the last position
         * read from by the high-level applicaton.</para>
         * <para>Dequeue function uses this value to store the last position
         * read from by the real-time application.</para>
         */
        u32 readPosition;
        /* <summary>Reserved for alignment.</summary> */
        u32 reserved[14];
} BufferHeader;
```

Přestože se jedná o strukturu mapovanou na paměť v přesně daném formátu, tak v kódu knihovny není u definice struktury uveden atribut `__attribute__((packed))`, který by kompilátoru bránil dělat optimalizace a zarovnávat pole této struktury. V rámci knihovny se jedná o poměrně malý problém. Navíc tím, že struktura obsahuje pouze pole 32bitových datových typů, tak se v tomto konkrétním případě nedá předpokládat, že by kompilátor dělal nějaké optimalizace, které by měnily strukturu dat v paměti, a tím způsobily nefunkčnost této struktury mapované na paměť.

Jak bylo zmíněno, buffery jsou dva. Jeden pro přenos dat směrem z vysokoúrovňové aplikace do mikrokontrolérové a druhý pro opačný směr. Funkce `EnqueueData` na ukázce kódu 4.3 by tak například měla pouze zapisovat do výstupního (`outbound`) bufferu.

Výpis kódu 4.3 Funkce EnqueueData [52]

```
125 int EnqueueData(BufferHeader *inbound, BufferHeader *outbound,
126                 u32 bufSize, const void *src, u32 dataSize)
127 {
128     u32 remoteReadPosition = inbound->readPosition;
129     u32 localWritePosition = outbound->writePosition;

        // zkráceno

192     outbound->writePosition = localWritePosition;

        // zkráceno

199     return 0;
200 }
```

Hned na začátku však funkce na řádcích 128 a 129 načítá čtecí a zápisové pointery z bufferů pro odlišný směr. Tyto pointery jsou samozřejmě irelevantní, na sobě nezávislé a nekompatibilní. Na základě těchto pointerů pak dělá knihovna výpočty volného místa v bufferu a další úsudky, které ale vycházejí z naprosto špatných ukazatelů, které mohou vést k nesprávným závěrům a chybným čtením nebo přepisům paměti na obou stranách kanálu. Na konci funkce se pak zapíše upravený ukazatel do sdílené paměti zpět. Buffer by mohl být upraven chybně podle stavu bufferu v opačném směru. Stejnou chybu knihovna dělá ve funkci `DequeueData`, jež jde

zkrácená vidět na následující ukázce 4.4. Ve funkci `DequeueData` je však ještě jedna chyba na řádku 289, kde aplikace upravený čtecí pointer nezapisuje do inbound bufferu, ze kterého četla, ale zapisuje jej do outbound bufferu, ze kterého knihovna v této funkci vůbec nic číst (a už vůbec ne zapisovat) nemá. Funkce tak přepíše ukazatel, který má ve sdílené paměti spravovat operační systém, nikoli ona.

Výpis kódu 4.4 Funkce DequeueData [52]

```
202 int DequeueData(BufferHeader *outbound, BufferHeader *inbound,
203                         u32 bufSize, void *dest, u32 *dataSize)
204 {
205     u32 remoteWritePosition = inbound->writePosition;
206     u32 localReadPosition = outbound->readPosition;

        // zkráceno

289     outbound->readPosition = localReadPosition;

        // zkráceno

297     return 0;
298 }
```

Poslední problém, který zde je, je ten, že podobně jako knihovna nespecifikovala atribut `__attribute__((packed))`, tak stejně tak, knihovna nenutí kompilátor generovat atomická čtení a zápis pointerů do sdílené paměti. Knihovna navíc nepoužívá ani datovou bariéru pro zajištění, aby došlo k fyzickému zápisu všech dat do paměti předtím, než se notifikuje softwarovým přerušením druhá strana kanálu. Nezávislá knihovna k těmto aspektům přistupuje mnohem obezřetněji a používá vestavěné funkce kompilátoru, jako jsou `__atomic_load`, `__atomic_store` a instrukci datové bariéry `dsb` před volání přerušení pro notifikování aplikace na druhé straně kanálu.

Velmi zajímavé je, že součástí knihovny je i ukázková aplikace, kde knihovna *úspěšně* přenáší data mezi aplikacemi, které vypisuje a aplikace *fungují*. Je to dáno tím, že aplikace je speciálně (ale pravděpodobně ne záměrně) navržena tak, že přenáší stejně dlouhé zprávy oběma směry a posílá stejné množství zpráv v obou směrech. Skutečnost, že chybně ovládá pointery ve sdílené paměti, které ji navíc nepatří, se tak překvapivě nikde neprojeví, protože operační systém posílající totožně dlouhou zprávu pointer upravil na úplně stejnou hodnotu, kterou pak aplikace zapíše v nesprávné funkci do nesprávného pointeru, ale tím, že jsou funkce volány za sebou, tak je stav kanálu upraven sice formálně špatně, ale přesto efektivně správně. Funkce sice nikde nemodifikuje čtecí pointer, ale zdá se, že operačnímu systému stačí potvrzení softwarovým přerušením, které aplikace generuje. Samozřejmě stačí, aby komunikace nebyla symetrická a dřív nebo později (záleží na okolnostech, ale při mých pokusech to nastalo okolo desáté přenesené zprávy) se v kanálu začnou objevovat chyby a kanál se velmi rychle stane nefunkční nebo bude produkovat chybné zprávy. Tím, že knihovna zapisuje pointer, který má zapisovat operační systém, tak se také velmi rychle stane, že pointer začne ukazovat místo na hlavičku zprávy, kterou vyžaduje protokol, do datové části nějaké zprávy a operační systém takovou zprávu nebude schopen zpracovat kvůli chybnému identifikátoru cílové komponenty. Operační systém v takovém případě vygeneruje chybu při čtení ze socketu, který má vysokoúrovňová aplikace otevřený.

Při zkoumání a zkoušení ukázek meziprocesorové komunikace jsem narazil ještě na ukázkové projekty [46] v repozitáři od Microsoftu. Tento repozitář obsahuje ukázky, jak některé periferie ovládat z vysokoúrovňové aplikace (což u některých periferií operační systém umí zajistit) a ve složce s mikrokontrolérovou aplikací je pouze odkaz na zmíněné knihovny od MetiaTeku a Codethink Labs. Meziprocesorová komunikace je zde ale výjimkou a je naimplementována Microsoftem včetně mikrokontrolérové aplikace. Součástí tohoto projektu je tak implementována podpora pro komunikaci mikrokontrolérové a vysokoúrovňové aplikace, která implementuje popsanou první a druhou vrstvu komunikačního protokolu. V této malé knihovně je však natvrdo zakompono-

vána podpora pouze pro komunikaci s vysokoúrovňovou aplikací a nejde ji použít pro komunikaci mezi mikrokontrolérovými jádry. Microsoft kromě tohoto ovladače pro jeden komunikační kanál nemá žádné ovladače periferií, s výjimkou velmi jednoduchého a omezeného ovladače UARTu a GPT (General Purpose Timer), které evidentně byly vytvořeny jen pro účely tohoto jednoho demonstračního projektu. Výhoda této malé knihovny je ta, že její kód je velmi stručný, korektní, dobře dokumentovaný (a funguje bez nutnosti oprav). Při řešení jsem část této malé knihovny integroval do projektu a využil ji pro komunikaci s vysokoúrovňovou aplikací. Pro komunikaci s memory poolem na druhém mikrokontrolérovém jádru jsem využil knihovnou MediaTeku. Tím, že pro komunikaci mezi mikrokontrolérovými jádry není potřeba dodržovat speciální protokol, se to jeví jako ideální řešení, které se osvědčilo. Jediná nevýhoda byla, že obě knihovny ve zdrojových kódech používají (totožnou) stejně pojmenovanou strukturu BufferHeader, kterou bylo nutné z jedné knihovny odstranit.

Nezávislá knihovna od Codethink Labs tak nakonec není použita ani v jedné mikrokontrolérové aplikaci a o veškerou meziprocesorovou komunikaci se starají jen knihovny OS-HAL od MediaTeku a malá knihovna z ukázkového projektu od Microsoftu. Pro ovládaní periferií Armového jádra (kontrolér přerušení NVIC a systémový timer SysTick) je do projektu zahrnuta i knihovna CMSIS, která je integrovaná jako součást v rámci knihoven M-HAL a OS-HAL.

Následující seznam shrnuje všechny použité knihovny ve všech aplikacích tvořících firmware:

1. Vysokoúrovňová aplikace:
   a. standardní knihovna jazyka C (libc.so)
   b. aplikační knihovny Azure Sphere (libapplibs.so)
   c. klienti IoT služeb Azure (libazureiot.so)
   d. klient služby DPS (libdps-custom-hsm.so)
   e. CURL (libcurl.so)
   f. pdjson
2. Mikrokontrolérová aplikace ovládající kameru:
   a. MediaTek OS-HAL
   b. knihovna pro komunikaci s vysokoúrovňovou aplikací z ukázkového projektu Microsoftu
   c. Arm CMSIS
3. Mikrokontrolérová aplikace zprostředkovávající paměť:
   a. MediaTek OS-HAL
   b. Arm CMSIS

4.1.2 Sdílení kódů

Část zdrojových kódů aplikace je mezi aplikacemi sdílená. Jedná se o kódy, které se používají v alespoň dvou aplikacích. Složka je v kořenové složce zdrojových kódů projektů (na stejné úrovni jako jednotlivé projekty) a jmenuje se `Shared`. Složka je zařazena v souborech `CMakeLists.txt` každého projektu do include paths. Ve stejném souboru jsou také zahrnuty do kompilace některé soubory z této složky. Ve složce se sdílenými zdrojovými kódy se nachází zejména:

1. hlavičkový soubor s konstantami používanými při komunikaci mezi vysokoúrovňovou aplikací a mikrokontrolérovou aplikací ovladající kameru
2. hlavičkový soubor s konstantami definující velikosti bufferů, které se používají pro ukládání obrazových dat napříč jádry

3. zdrojové kódy ovládající periferie, které jsou využity na obou mikrokontrolérových jádrech (ladící UART, SysTick, meziprocesorové FIFO a jednotka pro implementaci velkého FIFO složeného z více paměťových oblastí)

Ovladače periferií je možné zařazovat do kompilace pouze mikrokontrolérových aplikací. Závisí na externích knihovnách OS-HAL a CMSIS. Jejich zařazení do kompilace vysokoúrovňové aplikace povede k selhání při kompilaci.

Většina zdrojových kódů je rozdělelna standardním způsobem na hlavičkový (.h) a zdrojový (.c) soubor. Jedinou výjimku tvoří modul DebugUart, který je implementován se stejným rozhraním v obou typech (vysokoúrovňová a mikrokontrolérová) aplikací, ale na obou typech odlišnou implementací. Modul má jednotný hlavičkový soubor, který je k dispozici ve složce Shared. V této složce je také soubor DebugUart.c, který obsahuje implemetanci, kterou využívají obě mikrokontrolérové aplikace a využívá knihovny OS-HAL, přes kterou ovládá ladící UART, který platforma na mikrokontrolérových jádrech nabízí. Druhá implementace modulu umožňujícícho zapisovat ladící výstupy je pro vysokoúrovňovou aplikaci a nevyužívá UARTu, ale využívá prostředků, které nabízí apliakční knihovna. Výstupy pak lze vidět ve vývojovém prostředí Visual Studio v okně Outputs. V rámci řešení tak existují 2 soubory DebugUart.c, ale jen jeden hlavičkový soubor DebugUart.h, což vytváří unifikované rozhraní pro zápis ladících informací ve všech aplikacích. Následující výpis stromové struktury ukazuje v jakých složkách se nachází zdrojové sobury modulu DebugUart a jakým způsbem jsou sdílené.

```
AzureSphere/
├── HLMainApp/
│   ├── DebugUartHighLevelAppExtensions.h
│   └── DebugUart.c
├── RTCameraControlApp/
├── RTMemoryPoolApp/
└── Shared/
    ├── DebugUart.h
    └── DebugUart.c
```

4.1.3 Ladění aplikací

Při vývoji bylo využito několik mechanizmů pro ladění aplikací a ověřování její korektní funkčnosti. Jedná se o

1. krokování programu před GDB server zprostředkovaným operačním systémem Azure Sphere OS (při ladění vysokoúrovňové aplikace) a zprostředkovaným utilitou OpenOCD při ladění mikrokontrolérových aplikací
2. dumpování paměti přes terminál utility OpenOCD dostupný přes telnet
3. ladící výpisy vypsané na UART (nebo do okna Output ve Visual Studiu v případě vysokoúrovňové aplikace)
4. ovládání výstupních GPIO signálů a jeho připojení k logickému analyzátoru

Platforma umožňuje krokovat program a analyzovat jeho stav u obou typů aplikací. Pro nasazení do produkce pak jde rozhraní zakázat, nicméně se jedná o jednorázovou a nevratnou akci. Z uživatelského pohledu nejsou na první pohled znatelné rozdíly mezi laděním vysokoúrovňové a mikrokontrolérové aplikace, ale přesto lze pozorovat odlišnosti. První rozdíl je v rychlosti a odezvě na ladění. Přestože vysokoúrovňové aplikace běží na rychlejším jádru, jejich ladění je poměrně pomalé. Vysokoúrovňové aplikace se interně ladí připojením k GDB serveru běžícím jako samostatné aplikace v Azure Sphere OS. Spojení s tímto serverem je zajištěno přes protokol IP enkapsulovaný v USB. Symboly používané při ladění k mapování instrukcí na kód v jazyce C

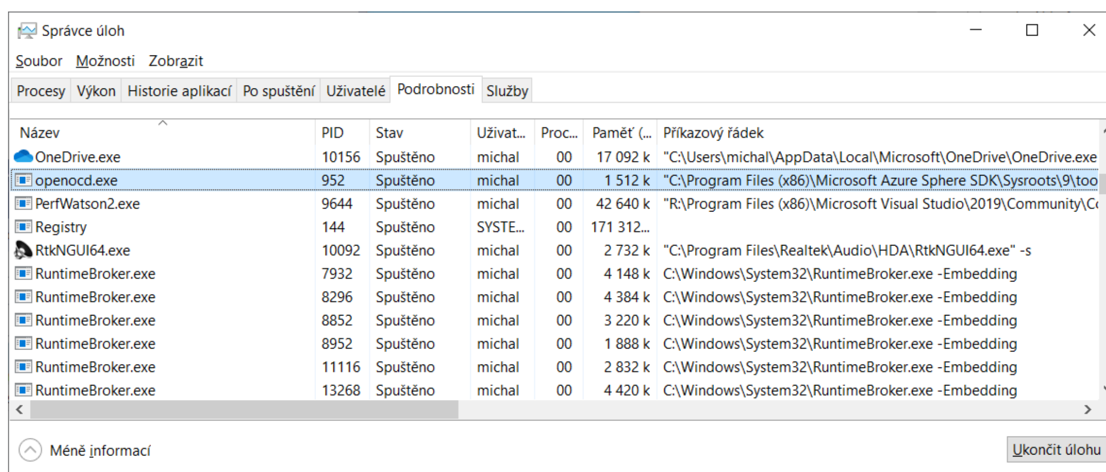
a názvy externích funkcí se navíc pro externí knihovny stahují ze serveru, díky čemuž je i inicializace debuggeru značně pomalá. Naopak mikrokontrolérové aplikace využívají při ladění HW podporu, která je součástí všech jader Arm Cortex-M4 (byť se návrháři čipu mohou rozhodnout, že dané vodiče z čipu nevyvedou) a umožňuje ladit program na jádře pomocí rozhraní SWD. Na vývojové desce je toto rozhraní připojeno k čipu FTDI4232HQ, který jej přes sběrnici USB zpřístupňuje operačnímu systému a aplikacím. Jedna z aplikací, která umí ovládat SWD přes čip FTDI je utilita Open On-Chip Debugger (známá častěji pod zkratkou OpenOCD). Tuto utilitu interně používá i Visual Studio. Utilita otevírá GDB server, ke kterému se vývojové prostředí Visual Studio připojuje. Přestože je utilita uživateli nedostupná, tak jde reference k ní najít v chybových hláškách z Visual Studia. Utilitu jde spusit i ručně. Vhodně upravené konfigurace utility bylo při řešení praktické části práce využito.

4.1.4 Čtení paměti zařízení do souboru pomocí OpenOCD

V průběhu vývoje bylo nutné ověřovat přijatá data z kamery (která příjímá mikrokontrolérová aplikace). Jedna z možností, jak to zajistit, je poslat data vysokoúrovňové aplikaci a nechat je touto aplikací nahrát do cloudu. Takové řešení je však jen velmi obtížně použitelné v době, kdy není k dispozici vyvinutý funkční komunikační protokol mezi mikrokontrolérovou a vysokoúrovňovou aplikací. Přesně to se při vývoji několikrát stalo v důsledku problémů s knihovnami a jejich laděním, popsaném v předchozí sekci Použité knihovny. Proto bylo vhodné vyvinout jiný způsob řešení jak přenést sejmutá data z paměti mikrokontrolérové aplikace do počítače. Nabízelo se například přenést data přes debugovací UART, v počítači je zachytit a rekonstruovat z nich binární data, která byla v paměti zařízení. Takový postup však vyžaduje psaní logiky jak v mikrokontroléru, tak v počítači příjímacím data. Implementace také omezuje použití ladícího UARTu pro jeho původní účel a také nutí vývojáře vyvinout počítačovou aplikaci, která bude data rekonstruovat. Bylo tedy zvoleno flexibilnější řešení. Utilita OpenOCD, která se používá k ladění mikrokontrolérových aplikací, umožňuje číst a zapisovat paměť zařízení v době, kdy je zastaveno laděné procesorové jádro. OpenOCD lze ovládat několika způsoby, ale uživatelsky nejjednodušší dosažitelné rozhraní je konzole TCL dostupná přes telnet. Využitý postup přečtení dat ze zařízení s pomocí utility OpenOCD je následující:

1. spuštění utility `openocd` s potřebnými inicializačními skripty jako parametry
2. připojení se k telnet serveru utility OpenOCD
3. nastavení breakpointu na vhodném místě, kde máme zájem o přečtení dat z paměti
4. vyčkání na zastavení programu v daném místě
5. přečtení paměti do souboru příkazem `dump_image`, který jako první parametr vyžaduje cestu k souboru, do kterého má zapsat přečtenou paměť, adresu ze které má paměť číst a velikost regionu, který má číst

První bod postupu je spuštění utility s cestami k inicializačním skriptům. Jako vzor těchto příkazů je nejjednodušší využít parametry příkazového řádku příkazu OpenOCD, který spouští Visual Studio. Parametry příkazového řádku je možné zjistit ve Windows ve Správci úloh na záložce Podrobnosti. Správce úloh ve výchozím stavu nezobrazuje sloupec Příkazový řádek, ale je jej možno zobrazit pomocí pravého kliknutí na záhlaví seznamu, volby Vybrat sloupce a zatržení položky Příkazový řádek. Příkaz příkazového řádku, kterým spouští Visual Studio OpenOCD lze vidět na následujícím obrázku 4.2.



Obrázek 4.2 Zobrazení příkazu, kterým Visual Studio spouští OpenOCD ve Správci úloh

Příkaz, který spouští Visual Studio je rozepsán celý na následujícím řádku:

```
"C:\Program Files (x86)\Microsoft Azure Sphere SDK\Sysroots\9\tools\openocd\
openocd.exe" -s "C:\Program Files (x86)\Microsoft Azure Sphere SDK
\Sysroots\9\tools\openocd" -f "mt3620-rdb-ftdi.cfg" -f "mt3620-io1.cfg" -c
"gdb_port" -c "targets"
```

Lze vidět, že Visual Studio spouští OpenOCD s pěti parametry:

1. parametr `-s "..."` nastavuje adresář, ve kterém se budou vyhledávat skripty
2. parametr `-f "mt3620-rdb-ftdi.cfg"` načte skript `mt3620-rdb-ftdi.cfg` a spustí jej. Tento skript obsahuje konfiguraci potřebnou pro ladění čipu MT3620 přes sběrnici SWD, která je připojena k čipu FTDI
3. parametr `-f "mt3620-io1.cfg"` načte a spustí skript, který konfiguruje OpenOCD pro ladění 1. mikrokontrolérového jádra (existuje ještě skritp pro ladění 0. mikrokontrolérového jádra)
4. parametr `-c "gdb_port"` spouští příkaz gdb_port, který vypíše na kterém TCP portu otevřelo OpenOCD GDB server (Visual Studio to z výstupu nejspíše čte)
5. paramtr `-c "targets"` spouští příkaz targets, který vypisuje všechna dostupná jádra, která lze ladit

Drobným problémem pro náš účel použití je, že skript obsahuje příkaz `telnet_port disabled`, který zakazuje (z bezpečnostních důvodů) připojení k OpenOCD pomocí telnetu. Protože Visual Studio telnetové spojení s OpenOCD nepoužívá, tak pro něj nemá smysl tento port otevírat. Protože my stejně potřebujeme spustit OpenOCD ručně, tak si můžeme udělat kopii tohoto skriptu, ve kterém bude tento řádek zakomentovaný. Takto to bylo vyřešeno při řešení praktické části práce.

Jakmile OpenOCD běží a jsme k němu připojeni přes telnet, tak můžeme spouštět příkazy. Pro nastavení breakpointu a kopírování paměti do PC však potřebujeme znát:

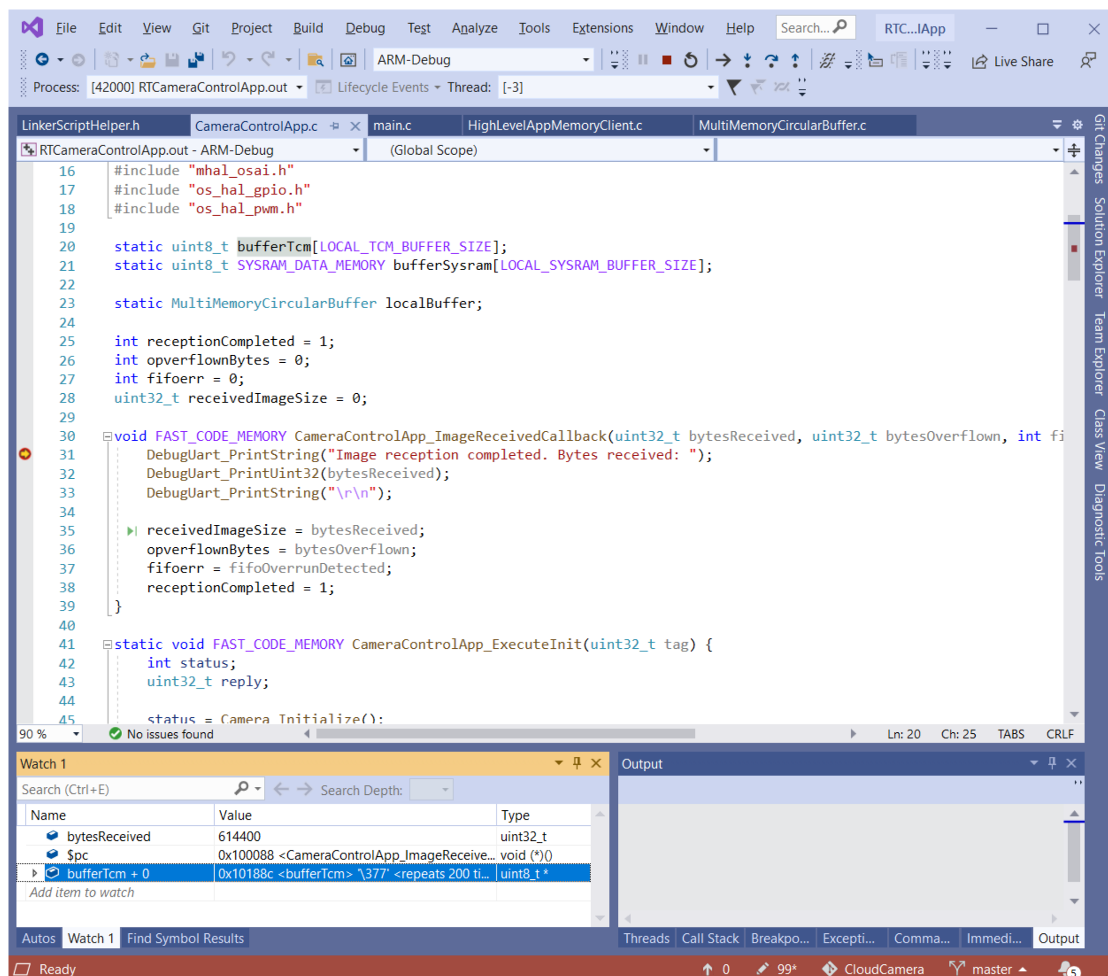
1. adresu instrukce, na které chceme program zastavit

2. adresu bufferu, který potřebujeme kopírovat
3. velikost bufferu, který potřebujeme kopírovat

Nejjednodušší cesta, jak zjistit tyto potřebné údaje (krom třetího, který je známý přímo ze zdrojového kódu) je zastavit si program ve Visual Studio a v okně Watch si nechat vyhodnotit následující výrazy:

1. `$pc` pro zjištění adresy instrukce na které je zastaven breakpoint Visual Studia (na stejné adrese budeme zastavovat i OpenOCD)
2. `buffer + 0` vypíše adresu bufferu. Visual Studio normálně vypisuje buffery intuitivním způsobem, výraz `+ 0` potlačí intuitivní výpis a vypíše pointer jako adresu.

Následující obrázek 4.3 ukazuje Visual Studio zastavené na breakpointu v callbacku, který se volá při dokončení snímání obrázku a ukazuje adresu v paměti, na které je k dispozici jeden z lokálních bufferů. Adresa breakpointu tak v tomoto případě je 0x100088, adresa bufferu je 0x10188c a velikost buffer je 172 KiB, tedy 176 128 B. Adresa se po provedení úprav může měnit, proto není garantováno, že stejné adresy kódu i bufferu budou po překompilování programu stále platit.



Obrázek 4.3 Zobrazení adres program counteru a bufferu ve Visual Studiu

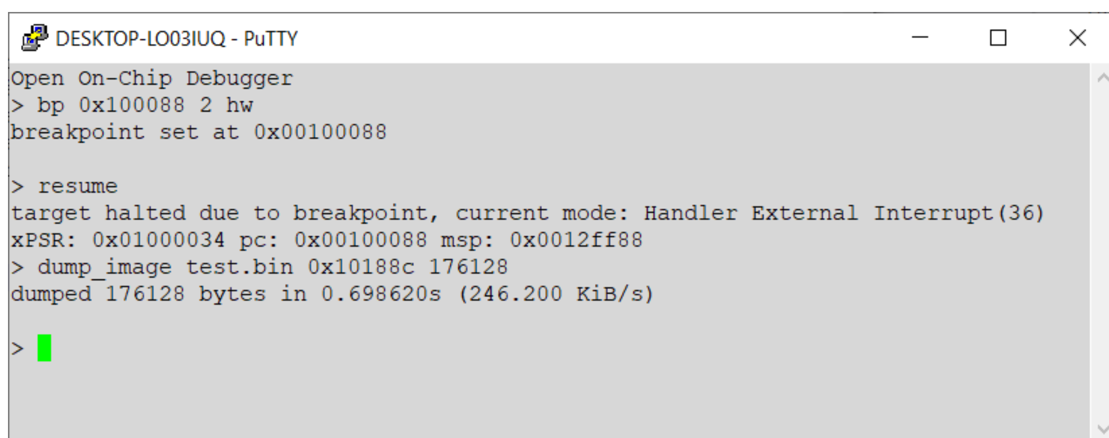
Tyto hodnoty je poté možné doplnit do následujícího příkazu, kde dvojka znamená velikost instrukce, na které se zastavujeme (OpenOCD vyžaduje tento parametr pro Arm Cortex-M4 vždy nastavený na 2) a parametr hw říká, že chceme použít hardwarový breakpoint.

```
bp 0x100088 2 hw
```

Jakmile se v terminálu objeví, že program byl zastaven na adrese 0x100088, tak můžeme udělat výpis paměti (do souboru `test.bin` v tomto případě) pomocí následujícího příkazu:

```
dump_image test.bin 0x10188c 176128
```

příklad vyexportování paměti ukazuje screenshot na obrázku 4.4.



Obrázek 4.4 Příkazy utility OpenOCD pro uložení obsahu paměti čipu do souboru.

Tento postup byl využíván při implementaci firmware. Byl využit zejména v době, kdy ještě nebyl k dispozici funkční komunikační kanál mezi aplikacemi, ani upload do cloudu. Umožnil tak jednoduché a neinvazivní získání obrázku bez nutnosti psát libovolný kód ani v aplikaci, ani v počítači.

4.1.5 Ladící UART

Pro ladění bylo také využito ladících výpisů přes UART. Jak bylo zmíněno v předchozí sekci Sdílení kódů, tak v obou typech aplikací bylo použito jednotné rozhraní s odlišnou implementací. Ladící výpisy byly využity zejména pro výpis selhání funkcí. Takřka všechny funkce jsou navrženy způsobem, že vracejí chybový kód. Každá funkce ošetřuje volání a návratové hodnoty všech funkcí, které volá a v případě jejich selhání vypíše název funkce, která selhala a jaký byl její návratový kód. Toto se děje na všech úrovních. Je tak možné čistě s těchto výpisů dohledat prvotní příčinu selhání.

Výhodné použití sdílené knihovny v mikrokontrolérových jádrech funguje hlavně díky tomu, že čip MT3620 nabízí mikrokontrolérovým jádrům mimo standardní moduly UART jeden omezený modul UART, který umí generovat pouze výstup a nepodporuje vstup ani signály řídící zahlcení. Tento modul platforma nabízí oběma jádrům a na obou jádrech je mapován na stejnou adresu. Díky tomu lze použít totožný kód (s totožnou adresou periferie) ve sdílené knihovně a v obou aplikacích bez jakékoliv nutnosti rozlišovat použitý modul.

Vysokoúrovňová aplikace používá stejné rozhraní pro zápis logovacích zpráv jako mikrokontrolérové aplikace. Původní plán byl, že implementace pro vysokoúrovňovou aplikaci bude umět odesílat ladící zprávy do cloudu. Z tohoto důvodu vznikl i pomocný soubor DebugUartHighLevelAppExtensions.h, který přidal doplňující funkci pro ladící zprávy, které by bylo vhodné vidět

ve vývojovém prostředí Visual Studio, ale zároveň by nebylo vhodné tyto zprávy posílat do cloudu. Například protože nemají moc velkou informační hodnotu a/nebo je jich hodně. Nakonec se však ukázal posílání ladích zpráv do cloudu jako poměrně zbytečné a navíc vyčerpávájící kapacitu zpráv, které dokáže IoT Hub zpracovat, takže i všechny ostatní funkce mimo soubor DebugUartHighLevelAppExtensions.h logují pouze lokálně.

4.1.6 Ladění pomocí výstupního portu GPIO

V průběhu vývoje bylo také velmi aktivně využíváno GPIO portů 42 a 43. Port 42 je také využit jako analogový vstup pro zvukový signál z mikrofonu, ale tato funkcionalita byla implementována až později. GPIO port byl konfigurován jako výstup a byl připojen k logickému analyzátoru, ke kterému byly připojeny i další signály sběrnice mezi vývojovou deskou a kamerou. Signály byly využity pro ladění a kontrolu, zda dochází ke splnění časových nároků paralelní sběrnice, ke které je připojena kamera. Pro zajištění co nejnižší zátěže je přepínání stavů GPIO portů řešeno makrem, které obsahuje přímý zápis do registru (adresa i zapisovaná hodnota jsou přímo součástí makra). Ladící makra jsou součástí kódu aplikace stále a jsou zakomentované pro případné budoucí ladění nebo verifikování splnění časových požadavků.

4.1.7 Konfigurace registrů kamery

Aplikace konfiguruje registry na hodnoty, které jsou součástí souboru `ov7670_regs.c`. Tento soubor je příkazem preprocesoru `#include`. Hodnoty v tomto souboru vychází z již existujících projektů jako je [40]. Většina dostupných existujících zdrojů používá velmi podobné hodnoty konfiguračních registrů. Konfigurace se obvykle liší jen v registrech, které jsou dokumentované. Při vývoji praktické části této práce docházelo k úpravám registrů pro snadnější ladění aplikace. Ve vývojové fázi, kdy ještě nebyl vyvinut a odladěn komunikační kanál pro sdílení paměti, bylo měněno výstupní rozlišení obrázků. Rozlišení bylo snižováno pro pohodlnějšího ladění a aby se obrázek vešel do jednoho bufferu na jednom mikrokontrolérovém jádře. Podobně byla měněna hodnota používaná k dělení frekvence výstupního hodinového signálu PCLK vůči vstupním MCLK.

Konfigurace probíhá přes sběrnici SCCB, která je však kompatibilní se sběrnicí I2C. Proto je pro konfiguraci využita periferie I2C čipu MT3620. Kamera používá pro konfiguraci podobné rozhraní jako například senzory nebo GPIO expandér, které jsou přítomny v řešení. Pro nastavení registru na hodnotu je potřeba na sběrnici provést transakci složenou z následujících kroků:

1. vygenerovat START sekvenci
2. zaslat 7bitovou adresu 0x21
3. zaslat bit indikující zápisovou transakci
4. zkontrolovat, že zařízení zápis potvrdilo ACK bitem
5. zaslat 8bitovou adresu registru, do kterého zapisujeme hodnotu
6. zkontrolovat, že zařízení zápis adresy potvrdilo ACK bitem
7. zaslat 8bitovou hodnotu zapisovanou do registru
8. zkontrolovat, že zařízení zápis hodnoty potvrdilo ACK bitem
9. vygenerovat STOP sekvenci

Knihovna OS-HAL provede všechny tyto operace (s pomocí HW periferie) sama a aplikace ji jen musí předat informaci o směru (to je rozlišeno názvem volané funkce) a buffer se dvěma bajty, ve kterém první bajt je adresa registru a druhý bajt je nová hodnota registru.

4.1.8 Optimalizace částí aplikace kritické na časování

Podstatnou částí implementace práce bylo ověřování splnění časových požadavků sběrnice pro přenášení obrazových dat z kamery do paměti a vývoj potřebných optimalizací, aby ke splnění požadavků došlo. Pro verifikaci požadavků byl použit logický analyzátor připojený k ladícímu signálu, jak bylo popsáno v předchozí sekci Ladění pomocí výstupního portu GPIO. Zpracování dat aplikace provádí v obsluze přerušení na vzestupné hraně signálu PCLK. Na začátku přerušení se ladící signál nastavoval na logickou jedničku a na konci přerušení na logickou 0. V logickém analyzátoru tak bylo možné sledovat:

1. latence přerušení (doba mezi náběžnou hranou a skutečným spuštěním přerušení)
2. dobu běhu přerušení

Krom optimalizací rychlosti běhu bylo také nutno provést ještě optimalizaci velikosti kódu, která zdánlivě nesouvisí s optimalizací dobou běhu, ale tato optimalizace značně ovlivnila dobu běhu kódu a vytvořila nutnost vzniku dalších výkonových optimalizací. Důvod pro nutnost optimalizace velikostí kódu vychází z toho, že v paměti RAM se sdílí prostor pro kód i velké buffery na ukládání obrázků. Cílem bylo, co nejvíce paměti RAM (respektive pamětí TCM a SYSRAM) použít jako buffer pro ukládání obrazových dat, a proto bylo nutné značně redukovat množství kódu, který běží z pamětí TCM a SYSRAM. Protože procesory architektury Arm obsahují jeden adresní prostor, tak lze spouštět kód nejen z paměti TCM a SYSRAM, ale i přímo z paměti FLASH, která je do adresního prostoru také mapována. Paměť FLASH je však pomalejší než paměti RAM a proto překonfigurování linker scriptu na mapování kódu do regionu paměti FLASH místo RAM došlo k výraznému poklesu výkonu. Navíc paměť FLASH je sdílena mezi všemi jádry (včetně operačního systému Azure Sphere OS), takže výkon paměti závisí i na činnostech ostatních aplikací a operačního systému. Pro eliminaci dopadů této optimalizace bylo nutné vyvinout řadu dalších optimalizací, které budou popsány dále.

V aplikaci byly provedeny následující optimalizace. Každá z nich bude popsána detailněji dále v této sekci.

1. optimalizace doby běhu přerušení
2. optimalizace latence přerušení
3. optimalizace zápisů do paměti v jiných jádrech

4.1.9 Optimalizace doby běhu přerušení

Přerušení, ve kterém se čtou obrazová data ze sběrnice, je nutné navrhnout a implementovat co nejrychleji je možné. Pro splnění tohoto požadavku byl kód optimalizován do velkých detailů. Jsou zde eliminovány takřka veškerá volání knihovních funkcí. Místo nich se používají přímé přístupy do registrů. Přímý přístup do registrů se zde používá pro:

1. čtení stavů GPIO portů (datové sběrnice a synchronizačních signálů)
2. ovládání diagnostických výstupů pro měření doby běhu přerušení
3. vložení dat do meziprocesorového komunikančího FIFO pro memory pool

Tyto optimalizace byly dostatečné do okamžiku, než došlo k přesunu kódu do paměti FLASH, ze které je vykonávání kódu pomalejší. Zpomalení způsobovala porušení časových požadavků a aplikace nebyla schopna zpracovat všechna obrázková data. Byla tak vyvinuta optimalizace v linker scriptu. Do linker scriptu, který mapovalo kód do paměti FLASH byla přidána sekce `text_enforce_tcm`, která je mapována vždy do paměti TCM narozdíl od standardní kódové sekce

`text`, která je v paměti FLASH. Toto samo o sobě nedonutí kompilátor generovat kód do sekce jiné než `text`, ale cílovou sekcí funkce jde u kompilátoru GCC (který se používá ke kompilaci všech částí firmware) změnit přidáním atributu z následující ukázky kódu 4.5 mezi název funkce a výstupní datový typ funkce:

Výpis kódu 4.5 Atribut pro vygenerování spustitelného kódu do jiné sekce

```
__attribute__((__section__(".text_enforce_tcm")))
```

Tento atribut tak byl přidán obsluze přerušení. Implementované řešení tak zajistí, že všechny funkce aplikace, s výjimkou jediné funkce, budou v sekci `text` v paměti FLASH a funkce realizující obsluhu přerušení bude jako jediná umístěna v paměti TCM. Vykonávání běžného kódu, tak bude sice pomalé, ale nebude zabírat místo v pamětech RAM a přerušení poběží z mnohem rychlejší (a determinističtější) paměti TCM, ve které ale bude zabírat část místa. Původně řešení počítalo s použitím jen v obsluze přerušení, ale nakonec je attribut přidán většímu množství funkcí. Důvod pro to bude uveden dále. Tím, že se ale jedná o malou část veškerého kódu (a nikoli například desítky funkcí z knihoven), tak spotřeba paměti TCM, ve které je kód, se pohybuje v řádů jednotek až nižších desítek KiB.

Pro snadnější zápis atributu je atribut definován makrem `FAST_CODE_MEMORY` v souboru `LinkerScriptHelper.h`, který je sdílen pro obě mikrokontrolérové aplikace.

4.1.10 Optimalizace latence přerušení

V průběhu vývoje se však ukázalo, že zatímco předchozí zmíněná optimalizace běh přerušení výrazně zrychlila, tak po přesunu kódu do paměti FLASH došlo i ke zvýšení latence spuštění obsluhy přerušení. Tento problém ale předchozí optimalizace nijak neovlivnila. Ukázalo se, že latence přerušení nezávisí jen na rychlosti zpracování instrukcí, které tvoří kód obsluhy přerušení, ale závisí i na kódu, který je přerušován. Patrně to souvisí s tím, že procesor před spuštěním přerušení dokončí celou prováděnou instrukci. Pokud vykonávání přerušované instrukce trvá, tak i latence vyvolání oblsuhy přerušení se zvyšuje.

Proto byla provedena analýza kódu, který běží v hlavní smyčce programu (včetně knihoven) a všech funkcí, které se volají a běží na procesoru v době, kdy dochází k snímání obrázku (a tedy volání přerušení). Všem těmto funkcím (včetně knihovních) byl přidán atribut, který je přesouvá do rychlé paměti TCM.

Tato optimalizace latenci vrátila na původní úroveň (před přesunutím kódu do FLASH).

4.1.11 Optimalizace zápisů do paměti v jiných jádrech

Firmware byl navržen tak, že obsluha přerušení vždy každý osmý bajt přenese po meziprocesorovém komunikačním kanálu do memory poolu. Aby došlo k co nejrychlejšímu zpracování, tak je na straně aplikace ovládající kameru využit již popsaný koncept přímého zápisu do registrů. Na straně mikrontrolérové aplikace tvořící memory pool, je pak krom tohoto výstupu využit princip popsaný v sekcích Optimalizace doby běhu přerušení a Optimalizace latence přerušení.

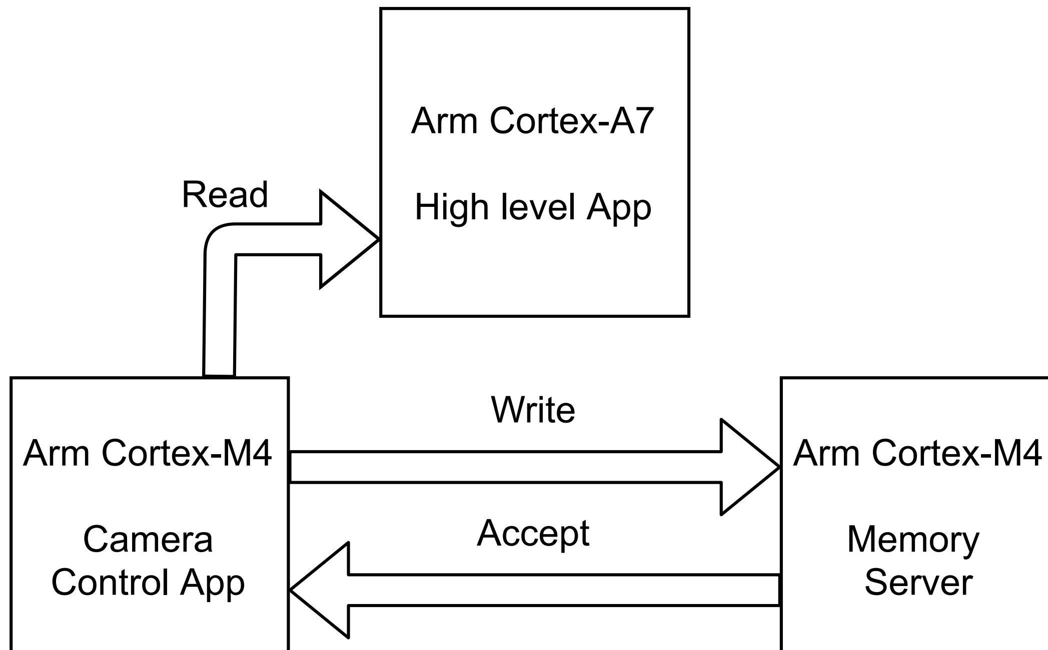
Pro zápis a čtení z kanálu byly vytvořeny makra definované v sdíleném souboru `FifoHelper.h`, která umožňují použít přímé zápisy do registrů s pomocí rozhraní připomínající volání funkcí.

4.1.12 Implementace memory poolu

Implementace Memory poolu je poměrně komplexní a skládá se z několika komponent napříč všemi aplikacemi.

Nejdůležitější součást je samotná komponenta zvaná `MemoryPool`, která běží na mikrokontrolérové aplikaci ovládající kameru a poskytuje jádru možnost chovat se k externím pamětem jako by se jednalo o jeden velký spojitý blok, který se chová jako FIFO. Tato jednotka je zodpovědná

za distribuci části obrázků do pamětí na jiných jádrech a jejich čtení z těchto pamětí. Jednotka s daty provádí tři operace pojmenované jako `Read`, `Accept` a `Write`, které přenášejí data z a do memory poolu. Tyto operace a směr dat, kterými se přenáší vizualizuje následující obrázek 4.5.



Obrázek 4.5 Operace Memory poolu

První operací, ke které dochází je operace Write, která by za normálních okolností byla volána z externího přerušení, ale vzhledem k optimalizacím je implementována „inline“ přímými zápisy do meziprocesorového FIFO bloku. Druhá operace je Accept, která se volá v hlavní smyčce aplikace a ta všechna data, která nemají zůstat na memory serveru získává zpět a přenáší je buď do svých lokálních bufferů nebo do vysokoúrovňové aplikace. Třetí operace je operace Read, která slouží k čtení dat vysokoúrovňovou aplikací. Vysokoúrovňové aplikaci jsou data předávána na vyžádání.

Významná součást Memory Poolu je Memory Server, který zprostředkovává paměť druhé mikrokontrolérové aplikaci a zároveň tvoří velký vyrovnávací buffer, který umožňuje pomalejší nebo blokové zpracování dat (v rámci operace Accept). Implementace serverové strany se nazývá `MemoryServer` a v aplikaci ovládající kameru se jedná o modul pojmenovaný `MemoryServerClient`. Implementace serverové strany je velmi jednoduchá a je složena z inicializační funkce a podpůrných funkcí. Memory Server je implementován pomocí incializační funkce, dvou přerušení a podpůrných funkcí. Memory Server využívá úrovňových přerušení NE (Not Empty) vstupního meziprocesorového FIFO a NF (Not Full) výstupního meziprocesorového FIFO. V obluze přerušení NE plní svůj interní velký buffer a v obsluze NF zase meziprocesorové výstupní FIFO. Pokud nejsou v bufferu žádná data, tak obsluhu přerušení NF zastavuje. Jamile přijdou nová data, tak obsluhu přerušení znovu obnoví, aby se začal plnit výstupní meziprocesorové FIFO daty (která již jsou k dispozici).

Zápis do lokálních bufferů si Memory Pool zajistí sám. Lokální buffery příjímá v inicializační metodě. Možnost zápisu do vysokoúrovňové aplikace, který je potřeba provést ještě v době příjímání snímku je na aplikaci ovládající kameru zprostředkována modulem `HighLevelAppMemoryClient`. Vysokoúrovňová aplikace tato data po zavolání (a obdržení potvrzu-

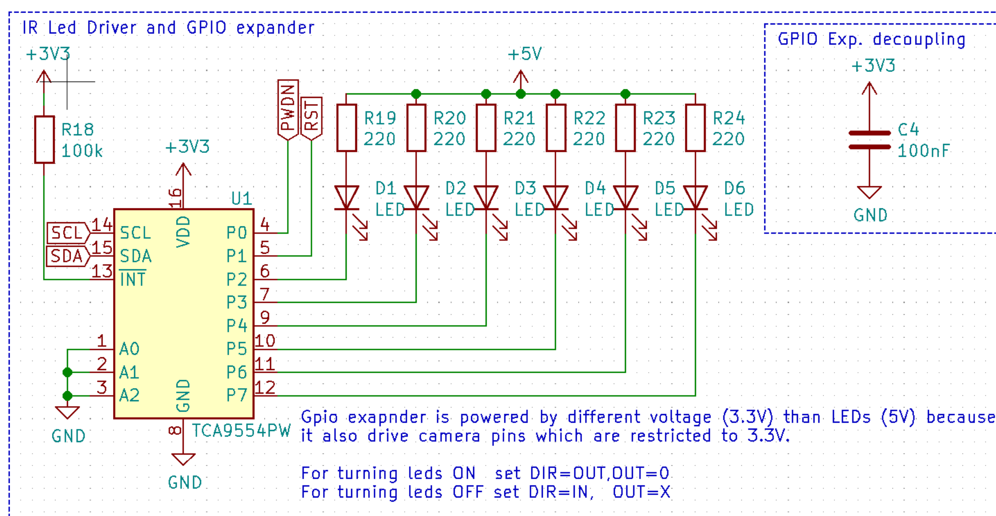
jící odpovědi) příkazu `CCS_COMMAD_CAPTURE_IMAGE`.

Memory Server i samotný Memory Pool se musí vypořádat s tím, že jejich lokální buffer se zkládá ze dvou pamětí (TCM a SYSRAM), proto vzniknul pomocný modul `MultiMemoryCircularBuffer`, který umí registrovat regiony dostupné paměti a vytvořit pro ně rozhraní, které se chová jako FIFO. Aplikace tak mohou se svými lokálními paměťmi zacházet jako s FIFO (což je vzhledem k ostatním přenosům ve firmware výhodné). Modul zjednodušuje logiku, která by musela rozlišovat, do kterého bufferu aktuální data patří. Logika je navíc sdílená mezi oběma mikrokontrolérovými aplikacemi.

4.1.13 Ovládání GPIO expandéru

Pro ovládání GPIO expandéru byl napsán ovladač v souboru `PCA9554.c`. PCA9554 je zařízení připojené ke stejné I2C sběrnici jako senzory a rozhraní SCCB kamery. GPIO expandér je součástí desky plošného spoje a jeho osazení není nutné. S tím počítá i firmware, který ignoruje chyby způsobené nepotvrzením přenosu dat (ovladač ale chyby reportuje správně). Ovladač umožňuje vše, co aplikace pro ovládání signálů RST a PWDN kamery a IR LED vyžaduje. Ovladač umí přepínat porty GPIO expandéru mezi vstupním a výstupním režimem, nastavovat logické hodnoty výstupu a nastavovat invertování polarity (to sice aplikace nepotřebuje, ale nevyžadovalo moc práce tuto funkcionalitu naimplementovat, takže ji ovladač obsahuje také). Ovladač neumí číst stavy vstupních portů, nicméně hlavičkový soubor obsahuje zakomentovanou definici funkce pro případnou budoucí implementaci.

Přepínání stavů portů PWDN a RST probíhá standardně nastavením portu jako výstupní a mění hodnoty logických stavů 0 a 1. Řízení LED však probíhá jiným způsobem. LED jsou totiž napájeny napětím 5 V zatímco expandér je napájen napětím 3.3 V. Ve výstupním režimu by při pokusu „vypnutí“ LED nastavením portu na logickou hodnotu jedna stále docházelo k úbytku napětí na LED a tedy toku malého množství proudu. Proto ovládaní LED probíhá tak, že výstupní logická hodnota je vždy nastavena na logickou 0 a „vypnutí“ LED se řídí přepnutím portu na vstupní port (tedy port v režimu vysoké impedance). Toto chování umožňuje řídit LED napájené jiným napětím než je napájen expandér a netrpí na pronikající proud v době, kdy jsou LED neaktivní. Toto použití explicitně zmiňuje (a povoluje) katalogový list čipu [53]. Na následujícím obrázku 4.6 lze vidět část schématu zapojení desky plošného spoje obsahující připojení GPIO expandéru.



Obrázek 4.6 Propojení GPIO expandéru

4.1.14 Detekce anomálií v okolí zařízení

V zařízení byl implementován mechanizmus, který detekuje anomálie v okolí zařízení monitorovacího systému. V případě takové detekce zařízení zvýší frekvenci, se kterou snímá snímky a odesílá je do cloudu. Detekce spočívá ve sledování náhlých změn teploty a hlasitých zvuků v okolí zařízení. Aplikace pro detekci:

1. čte teplotu ze senzoru LSM6DSO, který je součástí vývojové desky Avnet Azure Sphere Starter Kit
2. pomocí ADC snímkuje analogový vstup, ke kterému je připojen obvod mikrofonu na navržené desce plošného spoje.

4.1.15 Zpracování senzorických dat

V aplikaci je naimplementován jednoduchý ovladač pro senzor LSM6DSO, který podporuje čtení a konverzi teploty z tohoto senzoru. Čtení teploty je jediná funkce čipu, kterou ovladač podporuje, protože to pro účely navržené detekce stačí. Ovladač je jednoduchý a obsahuje 2 funkce. Jednu inicializační a jednu, která přečte teplotu z čipu a konvertuje ji na hodnotu float.

4.1.16 Zpracování audio signálu

Součástí detekce anomálií je i snímání okolního zvuku. To je zajištěno analogovým elektretovým mikrofonem, operačním zesilovačem a podpůrnými obvody složenými z pasivních součástek. Výstup tohoto obvodu je připojen k ADC (Analog to Digital Converter), který pravidelně tento port vzorkuje. Detekce probíhá srovnáním nejvyšší a nejnižší naměřené hodnoty v určitém časovém úseku. Spouštění pravidelného vzorkování je zajištěno v obsluze přerušení GPT (General Purpose Timer), který toto přerušení vyvolává s frekvencí přibližně 16 kHz.

4.1.17 Řízení snímání obrázku a jeho přenosu do cloudu

Přenosy obrázků jsou kritické místo vysokoúrovňové aplikace. Proto vznikl modul `CaptureToCloudProcessManager`, který řízení tohoto procesu prolínajícího přenosy jednotlivých částí obrázku do cloudu s jejich získáváním ze zařízení ovládá. Modul zveřejňuje pouze tři funkce. Jedna inicializační, jedna umožňující nastavit autentizační údaje přijaté zprávou z cloudu v jiném modulu a metodu, která inicializuje a řídí proces snímání obrázku, získávání jeho dat a přenos jednotlivých částí do cloudu. Nejpodstatnější je metoda `CaptureToCloudProcessManager_CaptureAndUploadImage`, která sekvenčně volá následující pomocné funkce, které zprostředkují celý proces snímání a přenos do cloudu.

1. `CaptureToCloudProcessManager_GenerateRandomName` – První volání je na funkci, která vygeneruje náhodný identifikátor pro obrázek, který bude zaslán do cloudu. Díky náhodnému identifikátoru nedochází při nahrávání obrázku ke kolizím v důsledku nahrávání obrázku více zařízeními najednou.
2. `CameraControlServerClient_TriggerCaptureImage` – Tato funkce (která jako jediná pochází z jiného modulu, protože se jedná o jednoduché volání bez komplikovanější logiky přípravy parametrů nebo zpracování návratové hodnoty) odešle mikrokontrolérové aplikaci příkaz, aby sejmula obrázek.
3. `CaputureToCloudProcessManager_ReceiveHighLevelAppPartOfImage` – Tato funkce navazuje na předchozí volání a příjímá data, která je potřeba zapsat do bufferu vysokoúrovňové aplikace v průběhu přenosu obrázku.

4. `CaptureToCloudProcessManager_WaitForReceptionComplete` – Protože data směřující do tohoto bufferu netvoří poslední část obrázku, tak i po skončení předchozí funkce stále ještě dochází k přenosu. Tato funkce pomocí komunikačního protokolu s mikrontrolérovou aplikací ovládající kameru čeká na skutečné dokončení přenosu a ověřuje stav přenosu. Přenos může selhat například z důvodu chyby na sběrnici s kamerou, zákmitu nebo jiných chyb, které většinou vedou k nesejmutí celého obrázku a chybějícím pixelům. Proto aplikace ve spolupráci s mikrokontrolérovou aplikací ovládající kameru kontroluje, že byl přijat očekávaný počet pixelů.

5. `CaptureToCloudProcessManager_RequestTemporarySas` – Pokud došlo k úspěšnému sejmutí snímku, tak funkce začne interagovat s cloudem a požádá cloud o dočasné přístupové údaje k uložišti, do kterého budou následující volání zapisovat části obrázku.

6. `CaptureToCloudProcessManager_UploadHighLevelAppPartOfImageToCloud` – Nejprve dojde k uložení části obrázku, která byla přenesena v rámci volání `CaputureToCloudProcessManager_ReceiveHighLevelAppPartOfImage`.

7. `CaputureToCloudProcessManager_RetrieveAndUploadBuffers` – Následně dojde k volání pomocné funkce, která interně zavolá další pomocné funkce, které:

   a. `CameraControlServerCleint_RetrieveCapturedImage` – přijme obsah lokálního bufferu z mikrokontrolérové aplikace ovládající kameru do bufferu, který byl původně využit pro ukládání části obrázku, která již byla nahrána do cloudu.

   b. `CaptureToCloudProcessManager_GeneratePartUrl` – Vygeneruje URL adresu do uložiště pro aktuálně zpracovávanou část dat.

   c. `AzureStorageCurlUpload_UploadBuffer` – Nahraje buffer do cloudu.

8. `CaputureToCloudProcessManager_RetrieveAndUploadBuffers` – Funkce zmíněna v předchozím bodě, se následně volá znovu a tentokrát přenáší část obrázku, která se nachází v Memory Serveru. Zprostředkování paměti se však děje přes operaci Read memory poolu, takže rozhraní pro zpracování bufferu z obou mikrokontrolérových jader je jednotné.

9. `CaptureToCloudProcessManager_NotifyIotHub` – Poslední akce, která tento proces obnáší je notifikování cloudových služeb o dostupnosti nového obrázku. Zpráva obsahuje identifikátor vygenerovaný v prvním kroku, podle kterého pozná o jaké objekty nahrané do cloudu v předchozích částech procesu se jedná.

4.1.18 Optimalizace spotřeby energie zařízení

Obě mikrokontrolérové aplikace pro redukci spotřeby elektrické energie využívají instrukci `WFI` (Wait for Interrupt), která jádro Arm Cortex-M4 uspí do doby než přijde (libovolné) přerušení. Aplikace také vypíná periferie, které v dané době nepotřebuje využívat. Proto například každý modul ovládající sběrnici I2C dekonfiguruje periferii sběrnice.

4.2 Cloudové aplikace

Jak bylo v kapitole Návrh systému navrženo, tak implementace implemetuje dvě části aplikace – univerzální a aplikačně specifickou. Jako ukázka aplikačně specifické aplikace byla vyvinuta aplikace monitorování parkoviště. V následujících sekcích budou popsány jednotlivé aplikace běžící na některé z cloudových služeb a konfigurace ostatních cloudových služeb. Bude také popsán emulátor zařízení, který vznikl pro usnadnění a zrychlení ladění cloudových aplikací. Popsány budou:

1. univerzální část aplikace
   a. konfigurace IoT Hubu
   b. konfigurace DPS
   c. konfigurace uložiště
   d. konfigurace front zpráv
   e. aplikace běžící v bezstavové funkci obsluhující příjem zpráv od zařízení a předzpracování obrázků
2. emulátor zařízení
3. monitorovací systém parkoviště
   a. konfigurace databáze
   b. konfigurace služby rozpoznávání obrazu
   c. aplikace běžící v bezstavové funkci pro zpracování a analýzu obrázků
   d. konfigurace autentizace pomocí Azure AD
   e. webová aplikace prezentující výsledky

4.2.1 Konfigurace IoT Hubu

Služba IoT Hub nevyžaduje žádný program, ale vyžaduje nakonfigurování následujících vlastností:

1. Endpoint, do kterého bude IoT Hub zasílat zprávy ze zařízení. Endpoint musí být nastaven na frontu, která bude popsána dále.
2. Route, která nebude nijak filtrovat zprávy a všechny bude směřovat do endpointu konfigurovaném v prvním kroku.
3. Certifikát certifikační autority tenantu Azure Sphere pro umožnění přístupu zařízením.

Konfiguraci služby lze provést buď ručně v Azure Portal nebo pomocí skriptu, jehož popis je v příloze této práce. První dvě potřebné konfigurace lze udělat snadno nastavením správných parametrů. Třetí konfigurace se skládá z následujících pěti kroků:

1. stažení certifikátu tenantu Azure Sphere
2. nahrání certifikátu do IoT Hubu
3. vygenerování verifikačního kódu v IoT Hubu
4. ověření verifikačního kódu pomocí služby Azure Sphere
5. nahrání podepsaného verifikačního certifikátu do IoT Hubu

Všechny tyto operace lze nejjednodušeji provést pomocí utilit `az` a `azsphere`. Tato operace je automatizována v rámci skriptu popsaného v příloze této práce, ale lze ji provést i ručně.

4.2.2 Konfigurace DPS

V službě DPS je potřeba nakonfigurovat následující:

1. propojit DPS s IoT Hubem, do kterého bude registrovat zařízení
2. certifikát certifikační autority tenantu Azure Sphere pro umožnění využití služby DPS zařízeními

Konfiguraci propojení s IoT Hubem lze zajistit přes Azure Portal nebo pomocí Azure CLI. Tato konfigurace je také součástí skriptu, popsaného v příloze této práce. Postup konfigurace a ověřování certifikátu tenantu Azure Sphere je totožný s konfigurací stejného certifikátu v IoT Hubu.

4.2.3 Konfigurace uložiště

Aplikace neklade na blokové uložiště žádná omezení. Lze volit libovolnou třídu replikace dat dle potřeb nasazované aplikace. Skript, který je popsán v příloze práce, nastavuje výchozí typ uložiště, což je uložiště s lokální replikací dat, která umožní plnohodnotný přístup v případě selhání první instance dat a zároveň data replikuje do párového datacentra. V případě selhání hlavního datacentra, tak uložiště zajistí záložní přístup k datům přes párové datacentrum. Nicméně v případě použití záložního uložiště je uložiště dostupné jen v read-only režimu. Pokud použití aplikace vyžaduje spolehlivější (a dražší) uložiště, tak si ho implementátor může nasadit ručně nebo upravením skriptu. Podobně lze použít i levnější uložiště, které nedělá replikaci do párového datacentra.

V uložišti musí být kontejner s názvem `images`.

4.2.4 Konfigurace front zpráv

Při konfiguraci fronty zpráv je nutné vytvořit dvě fronty pojmenované `device-messages` a `images`, které další části aplikace používají. Aplikace neklade žádné další nároky na konfiguraci této služby. Konfiguraci lze provést ručně přes Azure Portal nebo pomocí skriptu popsaného v příloze této práce.

4.2.5 Aplikace běžící v bezstavové funkci univerzální části aplikace

První aplikace vyžadující nějaký program je bezstavová Azure Functions App, která zpracovává zprávy ze zařízení a případně na ně odpovídá. Aplikace byla vyvinuta pomocí programovacího jazyka C# a je postavena nad technologií .NET Core. Aplikace využívá následující knihovny distribuované přes Nuget. Knihovny, které jsou nezbytné pro implementaci Azure Functions App a jsou součástí výchozího projektu, nejsou v tomto seznamu zahrnuty:

1. `Azure.Messaging.ServiceBus` – pro zasílání zpráv do výstupní fronty
2. `Azure.Storage.Blobs` – pro čtení, zápis a mazání částí obrázků v uložišti
3. `Magick.NET-Q16-AnyCPU` – pro převod obrázků z formátu RGB565 na JPEG
4. `Microsoft.Azure.Devices` – pro zasílání zpráv z cloudu do zařízení

Hlavní funkce je v souboru `ReceiveMessage.cs`, ve kterém se zpráva parsuje a rozhoduje se, která část aplikace bude dále zprávu zpracovávat. Pokud se jedná o zprávu požadující krátkodobý

přístup k uložišti, tak se zpráva zpracuje v souboru `ShortTermSasProvider.cs`, kde dojde k vygenerování tokenu a zaslání odpovědi do zařízení. V případě notifikace o novém obrázku, dojde ke zpracování požadavku ve třídě v souboru `ImageProcessor.cs`. V tomto souboru se obrázek přímo nezpracovává. To se děje ve třídě v souboru `ConvertSession.cs`, který udržuje stavové informace o průběhu všech částí této operace. V tomto souboru jednotlivé metody postupně zajistí:

1. `LoadOriginalImageBlobs` – načtení seznamu částí obrázků, které zařízení nahrálo do uložiště
2. `DownloadOriginalImage` – stažení částí a jejich spojení do jednoho souvislého bufferu
3. `CheckImageSize` – kontrolu, že velikost obrázku odpovídá požadované velikosti
4. `ConvertRawToJpeg` – konverzi z formátu RGB565 do formátu JPEG
5. `UploadConvertedImage` – nahrání konvertovaného obrázku do uložiště
6. `DeleteOriginalBlobs` – odstranění původních částí obrázku ve formátu RGB565

4.2.6 Emulátor zařízení

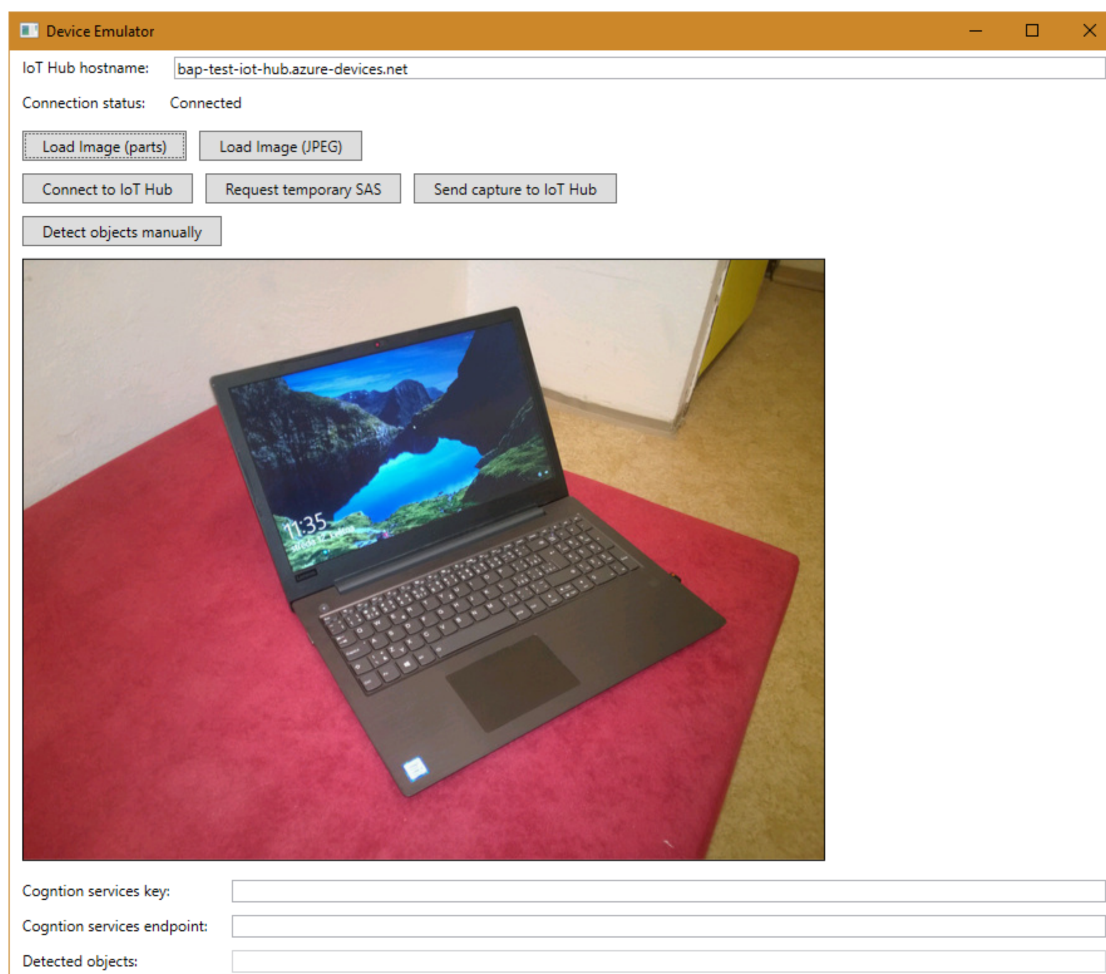
Pro zjednodušení vývoje a testování cloudových aplikací a zejména aplikační aplikace, která bude popsána dále, byl vyvinut emulátor zařízení, který bez větší prodlevy umí na vyžádání zaslat do cloudu vybraný obrázek z počítače ke zpracování. Emulátor byl vyvinut v programovacím jazyce C# jako okenní aplikace využívající technologi WPF (Windows Presentation Foundation). Emulátor je navržen tak, že generuje totožné zprávy včetně požadavku o krátkodobý přístupový token, jako to dělá skutečné zařízení. Stejným způsobem rozděluje obrázek na části a převádí jej na formát RGB565, i přestože má k dispozici obrázek ve formátu JPEG. Aplikace se musí do cloudu autentizovat pomocí certifikátu podepsaného certifikační autoritou. Součástí nasazovacích skriptů je skript, který umí vygenerovat a do IoT Hubu zaregistrovat potřebný certifikát. Postup je podobný jako byl uveden v sekci 4.2.1. Oproti autentizaci reálného zařízení je nutné vygenerovat certifikáty certifikační autority a emulátoru a jejich digitální podepsání. Na zařízení toto zajišťuje (bezpečněji) platforma Azure Sphere. Pro generování se používá utilita openssl. Na konci skriptu je klíč a certifikát převeden do formátu PFX, který musí být k dispozici v pracovním adresáři při spuštění emulátoru zařízení.

Okno aplikace emulátoru zařízení s vybraným testovacím obrázkem ukazuje následující obrázek 4.7.

V následujících sekcích bude popsána konfigurace cloudových služeb a programy realizující demonstrační aplikaci – monitorování obsazenosti parkoviště. Pro nasazení další části aplikace již však není k dispozici skript, protože většina služeb vyžaduje konfiguraci určité cenové a výkonové hladiny. Služby si musí uživatel vytvořit sám v cenových hladinách (a případně s dalšími konfiguracemi) dle vlastního uvážení a zohlednění požadavků nasazení. Dále také musí implementátor všechny služby propojit podle popisu v příloze této práce.

4.2.7 Konfigurace databáze

Pro řešení aplikace byla využita moderní databáze Azure Cosmos DB. Interně je to nerelační databáze, která šak poskytuje řadu rozhraní pro zajištění kompatibility s jinými databázemi včetně relačních. Databáze nabízí rozhranín SQL s omezenými funkcionalitami. Aplikace k této databázi však bude přistupovat pomocí ORM (Object-Relational Mapping) frameworku Entity Framework Core. Databázi je nejjednodušší vytvořit ručně přes Azure Portal, kde jsou přehledně prezentovány cenové možnosti. Aplikace vyžaduje typ rozhraní nastavený na `Core (SQL)`. Aplikace nepožaduje žádný konkrétní režim škálování. V průběhu implementace praktické části bylo využito režimu Serverless, který byl v dané době vývoje (duben 2021) ve fázi Preview. Funkce v režimu Preview jsou nové funkce Azure, které ještě neprošly dostatečným testováním a neplatí



Obrázek 4.7 Emulátor zařízení

pro ně žádné garance SLA ze strany Microsoftu. Nicméně i přesto s databází ve škálovacím režimu Serverless nebyl žádný problém. Režim Serverless má výhodu ve značné redukci ceny při malém zatížení databáze. Účtování probíhá podle množství operací, provedených proti databázi a neplatí se žádný konstantní poplatek za provoz infrastruktury. Použití databáze v tomto režimu, tak umožňuje značně redukovat cenu při velmi nízkém zatížení a zároveň umožňuje podle potřeb škálovat nebo například vykrýt nečekaně velkou zátěž.

Pro definici objektů v databázi byl vytvořen sdílený projekt mezi aplikací pro Azure Functions App a webovou administraci, protože obě mají přístup k databázi. V databázi se ukládají entity Parking a Device. Device je vytvářen v bezstavové funkci Azure Functions App při zpracování prvního obrázku z doposud neregistrovaného zařízení. Propojení zařízení s parkovišti může uživatel provádět přes webovou administraci. Pro přístup k databázi z obou aplikací byla využity knihovna s ovladačem pro Azure Cosmos DB pro Entity Frameowk Core. Jedná se o knihovnu `Microsoft.EntityFrameworkCore.Cosmos` dostupnout přes Nuget.

4.2.8 Konfigurace služby rozpoznávání obrazu

Služba nevyžaduje žádnou konfiguraci. Je nutné ji jen ručně vytvořit.

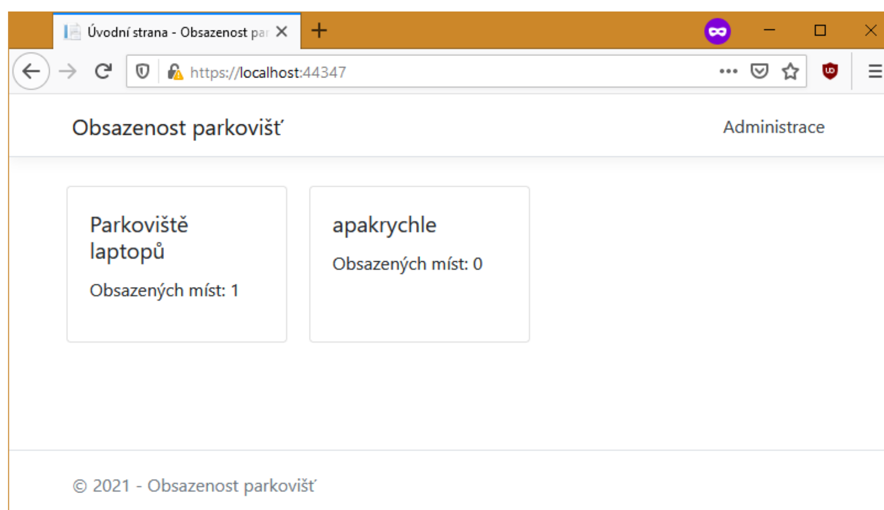
4.2.9 Aplikace běžící v bezstavové funkci pro zpracování a analýzu obrázků

První významnější aplikace je bezstavová funkce, která je vyvolávána při přijetí zprávy o dostupnosti obrázku ve formátu JPEG z první části aplikace, provede analýzu obrázku a zápis potřebných výstupů do databáze. Funkce je implementována jako Azure Functions App projekt podobně jako byl projekt v první části aplikace. Totožně je i tento projekt implementován v jazyce C# na platformě .NET Core. Funkce je přímočará a je implementována ve třídě v souboru `ParkingCapacityImageAnalysis`. Funkce provede:

1. parsování zprávy z fronty
2. načtení entity zařízení, které zaslalo snímek z databáze
3. stažení obrázku (již ve formátu JPEG) z úložiště
4. odeslání obrázku k analýze
5. spočítání, kolikrát se v odpovědi vyskytuje hledaný objekt
6. uložení výstupu do databáze

4.2.10 Webová aplikace prezentující výsledky

Další část cloudové aplikace je webová aplikace prezentující výstupy uživatelům a umožnění administrace systému administrátorům. Aplikace je založena na technologii ASP.NET Core a je programována v jazyce C#. Aplikace využívá konceptu MVC (Model View Controller). Úvodní stránka aplikace zobrazuje monitorovaná parkoviště a počet obsazených míst. Z pohledu běžného uživatele ji lze vidět na následujícím obrázku 4.8.



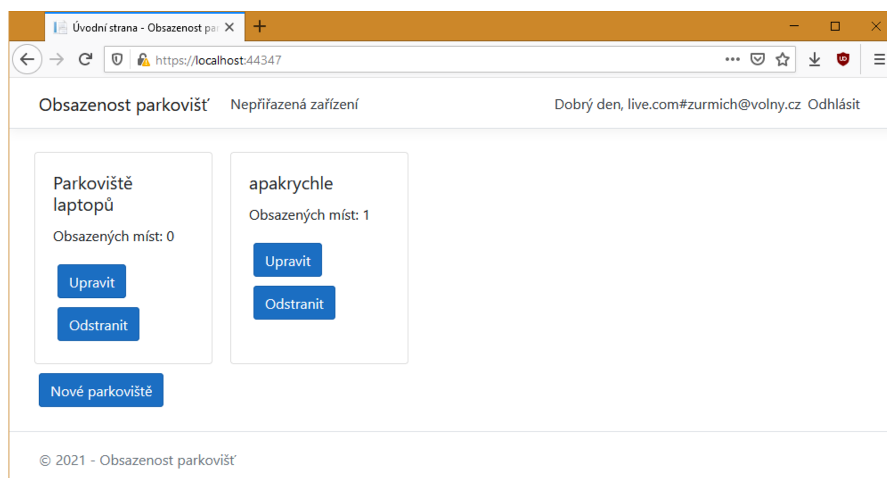
Obrázek 4.8 Webová aplikace zobrazující výstupy systému monitorujícího počet obsazených míst parkoviště z pohledu běžného uživatele

Administrátor má v aplikaci po přihlášení možnost:

1. přidávat parkoviště
2. měnit název parkoviště

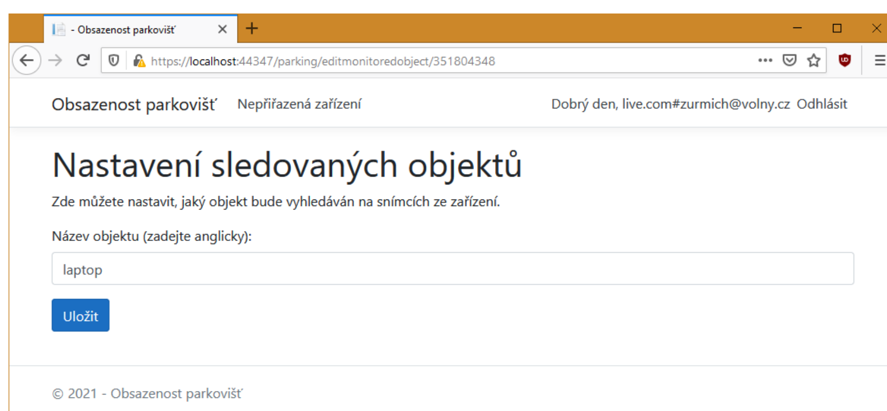
3. odstraňovat parkoviště
4. přiřazovat zařízení k parkovištím
5. odpojovat zařízení od parkoviště

Úvodní obrazovku po přihlášení k aplikaci zobrazuje následující snímek 4.9.



Obrázek 4.9 Webová aplikace zobrazující výstupy systému monitorujícího počet obsazených míst parkoviště z pohledu administrátora

Po zavedení nového zařízení do systému se zařízení ve webové aplikaci objeví po tom, co nahraje do cloudu první snímek. V administraci se objeví na záložce Nepřiřazená zařízení a je možné jej přiřadit k libovolnému parkovišti. U zařízení přiřazených k parkovišti lze nastavit název objektu, který se monitorovací systém snaží detekovat. Při registraci zařízení se tento řetězec nastaví na text „car“ a v administraci jde systém překonfigurovat, aby počítal výskyty jiných objektů. Následující obrázek 4.10 ukazuje možnost změny názvu sledovaného objektu ve webové administraci.



Obrázek 4.10 Webová aplikace zobrazující možnost změnit monitorovaný objekt

4.2.11 Konfigurace autentizace pomocí Azure AD

Poslední služba, která je součástí druhé části cloudové aplikace je Azure Active Directory, která umožňuje autentizovat administrátory k webové aplikaci. Výchozí AD je součástí většiny předplatných používaných v Azure. Většinou tak tuto službu již není potřeba vytvářet. Ve službě se musí zaregistrovat aplikace a povolit generování ID tokenů této aplikaci.

Kapitola 5
Testování

Všechny dílčí části aplikace byly otestovány. Byla otestována funkčnost logického zapojení zařízení, desky plošného spoje, firmware i cloudové aplikace. Byla otestována funkčnost zasílání obrázků ze zařízení do cloudu. Pomocí aplikace „Microsft Azure Storage Explorer“ bylo zkontrolováno, že nahrané části obrázku jsou korektní. Aplikační část cloudové aplikace byla otestována zejména s použitím emulátoru zařízení.

V následujících sekcích budou detailněji popsány některé specifické testovací procedúry.

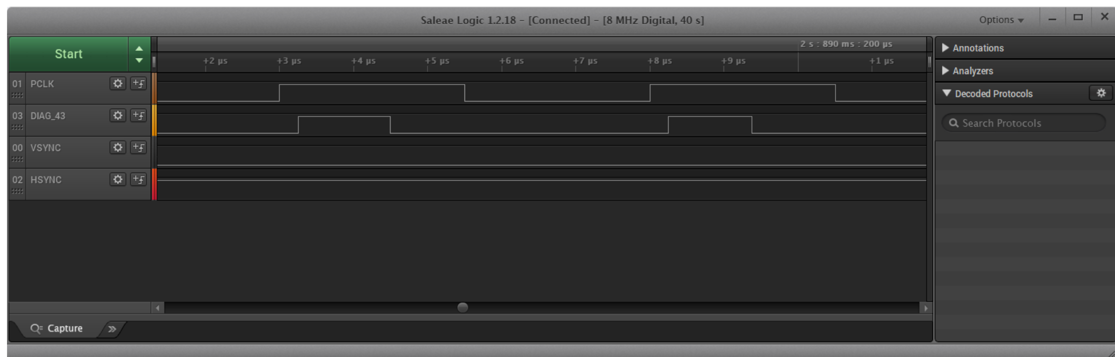
5.1 Testování splnění časových požadavků sběrnice

Jedním z významných testů aplikace je splnění striktních časových požadavků sběrnice mezi čipem MT3620 a kamerou. Aplikace musí všechna data sběrnice zpracovat dřív, než přijde další náběžná hrana hodin. Testování probíhala s pomocí logického analyzátoru, ke kterému byly připojeny signály PCLK, VSYNC, HSYNC a diagnostický signál DIAG_43. PCLK jsou hodiny, které určují platnost dat na sběrnici. Na jejich náběžné hraně musí aplikace začít zpracovávat data ze sběrnice. Pro účely testu byl do přerušení na začátek vložen kód, přepínající logický stav diagnostického signálu na logickou 1 a před každým příkazem `return` přidán kód nastavující hodnotu diagnostického signálu na hodnotu logická 0. Bylo potřeba zanalyzovat každý možný průběh časově kritické obsluhy přerušení. Přerušení musí být dostatečně rychlé nejen když se vzorkují data, ale i v době, kdy jsou na sběrnici neplatná data nebo probíhá přenos předchozího obrázku. Proto jsou k logickému analyzátoru připojeny i signály VSYNC a HSYNC, ze kterých lze odvodit průběh přerušení.

Následující snímek obrazovky 5.1 z logického analyzátoru ukazuje nejkritičtější situaci, kdy se na sběrnici nachází platná data (VSYNC=0 a HSYNC=1). Zároveň jde vidět, že latence přerušení je poměrně nízká a doba běhu přerušení nejenže nepřekračuje další náběžnou hranu, ale nepřekračuje ani sestupnou hranu. Obsluha přerušení tak má dostatečně velkou časovou rezervu. Všechny časové požadavky sběrnice jsou tak splněny.

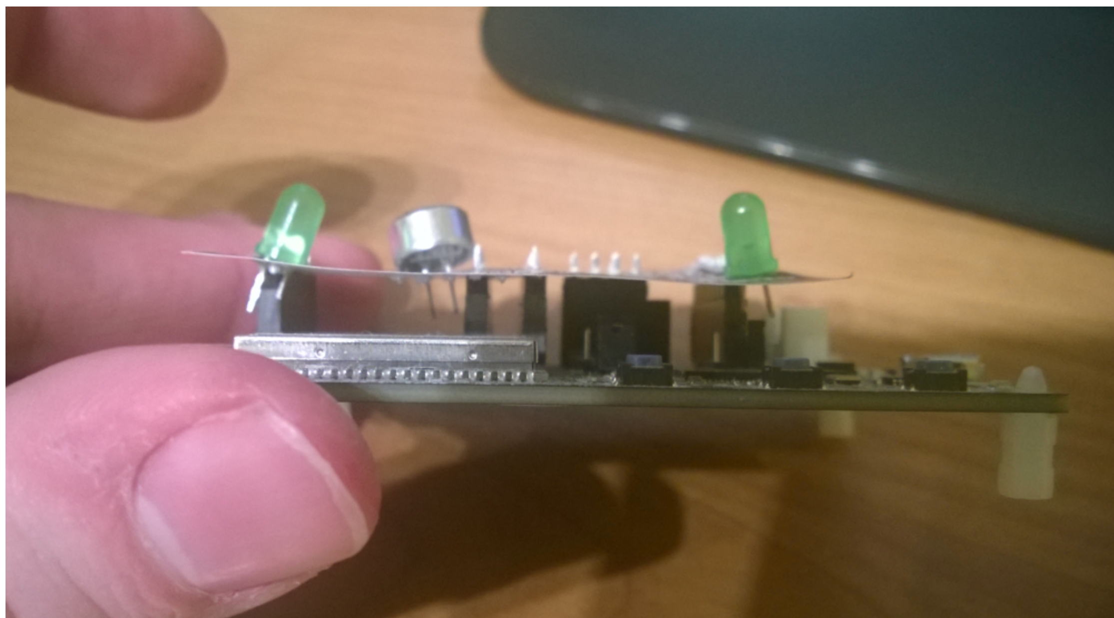
5.2 Testování desky plošného spoje

Ještě před zasláním desky plošného spoje proběhl test mechanických kolizí součástek. Test proběhl způsobem, že návrh desky byl vytištěn na papír a bylo otestováno, zdali se některé součástky nedotýkají nebo nekolidují. Potenciální kolize mohla například vzniknout s některými součástkami vývojové platformy jako jsou její jumpery. Na základě tohoto testu došlo k mírnému posunutí mikrofonu pro minimalizaci překryvu s blízkým konektorem a kondenzátorem. Následující fotografie na obrázku 5.2 ukazuje vytištěnou a osazenou desku plošného spoje, na které lze vidět,



Obrázek 5.1 Test splnění časových požadavků datové sběrnice kamery

že žádná ze součástek s vývojovou deskou nekoliduje.



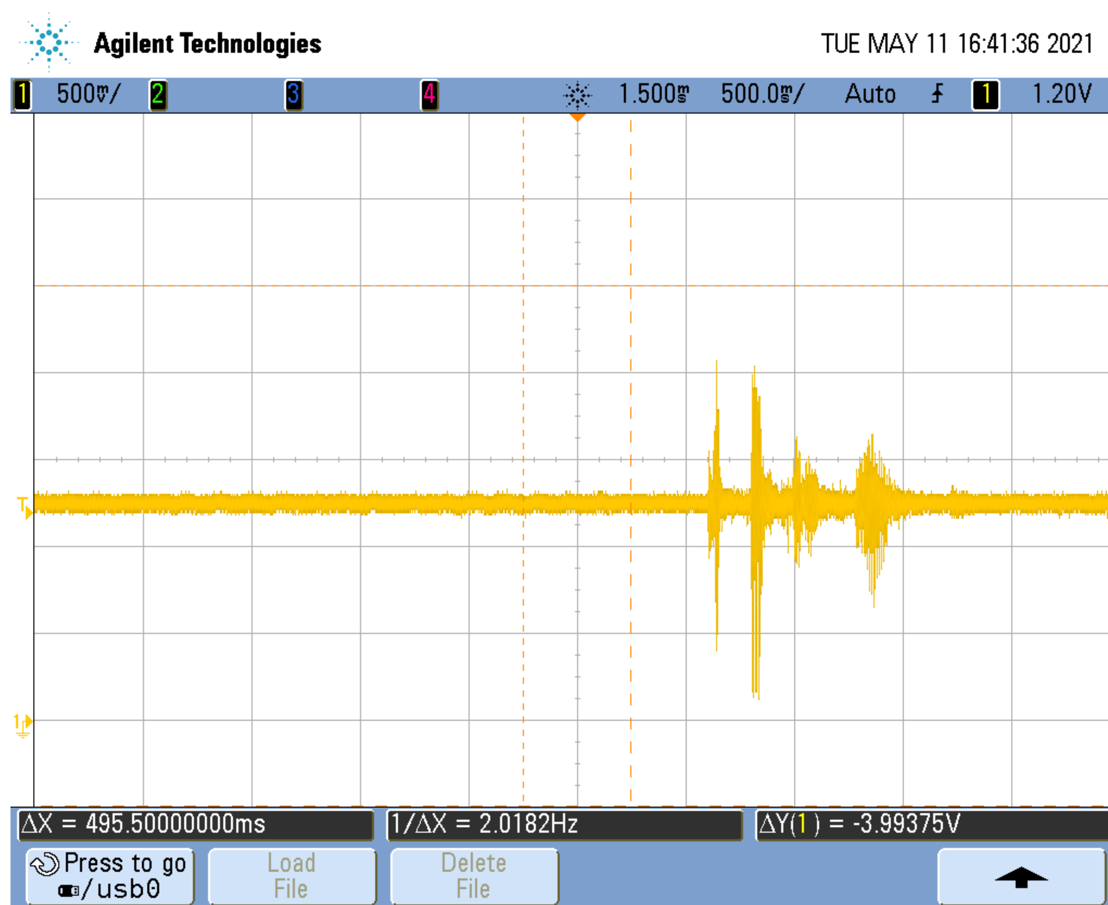
Obrázek 5.2 Test mechanických kolizí součástek

Po výrobě desky plošného spoje proběhlo jeho otestování. Proběhly následující testy:

1. test komunikace s kontrolním rozhraním kamery na sběrnici SCCB (I2C)
2. test přenosu obrazových dat na paralelní sběrnici
3. test komunikace s GPIO expandérem na sběrnici I2C
4. test možnosti ovládání portů GPIO expandérem (jak signálů RST a PWDN kamery, tak IR LED)

Všechny testy dopadly úspěšně a testování neukázalo žádný problém.

Další významný test desky plošného spoje byl test audio signálu z operačního zesilovače. Test proběhl připojením osciloskopu k výstupnímu signálu. Na následujícím obrázku 5.3 lze vidět zachycený signál osciloskopem při vyslovení slova „apakrychle“ v okolí zařízení.



Obrázek 5.3 Výstup osciloskopu při zachytávání mluveného slova

5.3 Testování možnosti vzdálené aktualizace firmware

Při vývoji zařízení byla také otestována možnost vzdálené aktualizace firmware z cloudu. Do cloudu byl nahrán nejprve obraz aplikace, která blikala zelenou LED a poté byla nahrána aktualizace měnící blikání na oranžovou LED. Zařízení aktualizaci během 24 hodin úspěšně stáhlo a firmware restartovalo. V průběhu práce také zařízení samo několikrát aktualizovalo svůj operační systém, což bylo pozorováno z výstupu příkazu `azsphere device show-os-version`.

Kapitola 6
Závěr

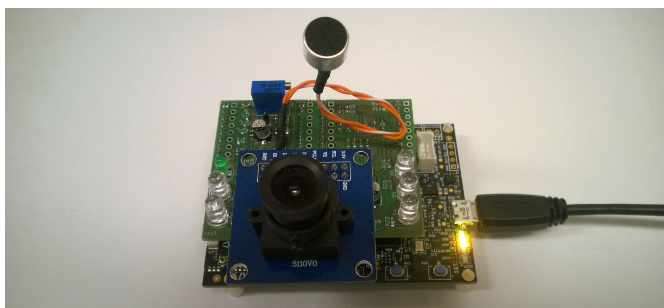
Výsledkem bakalářské práce je návrh a realizace univerzálního inteligentního monitorovacího systému se zařízením založeným na vývojové desce Avnet Azure Sphere Starter Kit, pro kterou byla navržena a vyrobena deska plošného spoje tvořící robustní a spolehlivé propojení kamery s vývojovou deskou. Deska plošného spoje také obsahuje mikrofon a potřebný obvod zesilovače pro zachytávání audio signálu. Celek tak tvoří kompaktní zařízení, které lze bezpečně připojit ke cloudu pomocí sítě Wi-Fi. Firmware zařízení je možné plně aktualizovat vzdáleně. Zařízení umí do cloudu zasílat zachycené snímky z kamery. Výsledkem práce je i ukázková cloudová aplikace, která dokáže monitorovat počet vozidel (nebo jiných předmětů) v zorném poli kamery. Aplikace vizualizuje výstupy ve webové aplikaci s administrací a pro svůj běh využívá moderní cloudové služby jako je databáze Cosmos DB.

Při návrhu řešení bylo maximálně využito možností vývojové desky. Byl implementován komplexní mechanizmus sdílení paměti mezi procesorovými jádry, který umožňuje využití celé paměti zařízení pro ukládání snímaného snímku. Periferie čipu jako je PWM kontrolér jsou využívány pro redukci množství externích součástek.

Byly provedené optimalizace kódu kritického na časování při zachytávání dat z kamery a bylo ověřeno splnění všech časových nároků této sběrnice.

Řešení bylo vyvinuto s ohledem na bezpečnost zařízení a cloudové aplikace. Zařízení využívá bezpečné připojení ke službám veřejného cloudu Microsft Azure pomocí šifrovaného spojení a autentizuje se vůči cloudu na základě digitálního certifikátu. Cloudová aplikace pro autentizaci uživatelů využívá přihlášení pomocí domény Azure Active Directory, díky čemuž dokáže využít všechny bezpečnostní funkce jako je dvoufaktorové ověření, které minimalizují možnost neoprávněného vniknutí do aplikace.

Bylo otestováno snímání snímků, zasílání snímků do cloudu i jejich zpracování. Vyrobené zařízení ukazuje následující fotografie 6.1.



Obrázek 6.1 Vyrobené zařízení

Příloha A

Návod na zprovoznění cloudových aplikací

Tato příloha popisuje postup nasazení aplikace běžící v cloudu a firmware do zařízení.

A.1 Nasazení univerzální části cloudové aplikace

Univerzální část aplikace lze nasadit pomocí bash skriptu `setup_cloud.sh`, který je k dispozici ve složce `CloudSide\DeploymentScripts` na přiloženém médiu. Před spuštěním je potřeba skript upravit a doplnit obsah proměnné `REGION` na název regionu cloudu Azure, ve kterém mají být služby založeny a parametr `RESOURCE_GROUP_NAME` na název pro Resource Group, která bude skriptem vytvořena. Všechny zdroje, které skript v cloudu vytvoří budou součástí této Resource group.

Skript má několik předpokladů pro úspěšné spuštění:

1. nutnost mít nainstalovaný nástroj Azure CLI
2. nutnost mít nainstalovaný nástroj Azure Sphere CLI
3. být přihlášen v utilitě Azure CLI (příkaz `az login`)
4. být přihlášen v utilitě Azure Sphere CLI (příkaz `azsphere login`)
5. mít zprovozněný alespoň jeden tenant služby Azure Sphere s alespoň jedním zařízením (příkaz `azsphere device claim`, jedná se o jednorázovou a nevratnou operaci!)
6. v případě že, přihlášený uživatel má k dispozici více předplatných, musí si vybrat předplatné, pod kterým chce zakládat zdroje pomocí příkazu
   `az account set --subscription <name or id>`
7. nainstalované rozšíření Azure CLI IoT Extension pomocí příkazu
   `az extension add --name azure-iot`
8. nainstalovanou utilitu jq (obvykle příkazem `apt install jq`)

Po spuštění skriptu je potřeba nasadit bezstavovou funkci, kterou skript do cloudu nenasazuje. Pro nasazení je potřeba provést následující:

1. Otevřít řešení `CloudSide\DeviceMessagesProcessing\DeviceMessagesProcessing.sln` ve Visual Studiu.

2. Kliknout v okně `Solution Explorer` pravým tlačítkem myši na projekt a vybrat možnost `Publish`.
3. V dialogovém okně vybraz možnost `Azure`.
4. V dialogovém okně vybrat možnost `Azure Function App (Windows)`.
5. Přidat novou Function App tlačítkem plus.
6. V dialogovém okně je potřeba nastavit jméno, region a `PlanType` na `App Service Plan`.
7. U položky `Hosting plan` vybrat `new`.
8. V novém dialogovém okně vybrat jméno, region a na jak výkonný server má být funkce nasazena. Volba záleží na odhadované zátěži. Funkce nemá žádné minimální požadavky.
9. Potvrdit všechna dialogová okna.
10. Stisknout tlačítko `Publish`.

Po publikování je potřeba přejít do administrace Azure Portal, vyhledat `Azure Function App`, otevřít detail nově založené `Function App`, přejít do sekce `Configuration` v menu a tlačítkem `New application settings` postupně zakládat následující konfigurační vlastnosti:

1. `DeviceMessageServiceBusConnectionString` – je potřeba přejít (například v druhé záložce prohlížeče) ke službě `Service Bus`, otevřít detail front, které založil skript, přejít na záložku `Queues` v menu, otevřít frontu `device-messages`, přejít na záložku `Shared sccess policies` v menu, tlačítkem `Add` vytvořit novou politiku s možností `Listen`, kliknout na nově vytvořenou položku a do vytvářené konfigurační hodnoty Function App zkopírovat hodnotu pole `Primary Connection String`.
2. `OutputImagesServiceBusConnectionString` – postup je podobný jako u předchozí položky. Politiku je potřeba vytvořit u fronty `images` a politika musí povolovat operaci `Send`.
3. `BlobStorageUrl` – URL adresa úložiště, které založil skript. Lze vidět v utilitě `Microsoft Azure Storage Explorer`.
4. `BlobStorageSas` – SAS token úložiště s možností `Read`, `Write`, `Create` a `Delete`.
5. `BlobStorageAccountName` – Název úložiště. Lze vyčíst z URL adresy úložiště.
6. `BlobStorageKey` – Klíč úložiště. Lze získat v utilitě `Microsoft Azure Storage Explorer`.
7. `BlobStorageContainerName` – Nastavit na řetězec `images`.
8. `IotHubConnectionString` – Lze nalézt u služby `IoT Hub`, kterou založil skript, na záložce `Shared access policies` v menu po rozkliknutí položky `iothubowner`.

A.2 Povolení emulátoru zařízení

Povolení emulátoru předpokládá spuštění skriptu z předchozí sekce a také vyžaduje nastavení proměnných `RESOURCE_GROUP_NAME` a `IOT_HUB_NAME`, které jsou na začátku skriptu. Je nutné je do skriptu před spuštěním doplnit.

A.3 Nasazení aplikace monitorující parkoviště

Nasazení aplikace pro monitorování parkoviště vyžaduje založení a provázání následujících služeb:

A.3.1 Cosmos DB

Službu lze najít po vyhledání `Cosmos DB` v Azure Portal

1. `API` při zakládání musí být nastaveno na `Core (SQL)`
2. vše ostatní může být nastaveno libovolně podle výkonových požadavků aplikace
3. po založení databáze je v jejím detailu potřeba přejít na záložku `Data Explorer` v menu
4. stisknout tlačítko `New Database`
5. nastavit název na řetězec `ParkingMonitoring`
6. potvrdit

A.3.2 Computer Vision

Službu lze vyhledat v sekci `Cognitive Services`.

1. v Marketplace je potřeba při zakládání vyhledat `Computer Vision` a založit jej

A.3.3 Bezstavová analytická funkce

1. Je potřeba otevřít řešení `CloudSide\ParkingMonitoring\ParkingMonitoring.sln` ve Visual Studiu.
2. Nasadit `ParkingMonitoring.ImageAnalysis` podobně jako bylo popsáno u nasazování funkce `DeviceMessagesProcessing` v předchozí sekci.
3. V portálu je potřeba nastavit aplikaci následující vlastnosti:
   a. `ServiceBusConnectionString` – Je potřeba vytvořit přístupovou politiku podobně jako bylo ukázáno u front funkce `DeviceMessagesProcessing`. Politika musí být pro frontu `images` a musí umožňovat `Listen`.
   b. `BlobStorageUrl` – Stejná hodnota jako při nastavování `DeviceMessagesProcessing`.
   c. `BlobStorageSas` – Stejná hodnota jako při nastavování `DeviceMessagesProcessing`.
   d. `CognitiveServiceEndpoint` – Lze najít na záložce `Keys and Endpoints` vytvořené služby `Computer Vision`.
   e. `CognitiveServiceKey` – Lze najít na záložce `Keys and Endpoints` vytvořené služby `Computer Vision`.
   f. `CosmosEndpoint` – Lze najít na záložce `Keys` vytvořené databáze `Cosmos DB`.
   g. `CosmosKey` – Lze najít na záložce `Keys` vytvořené databáze `Cosmos DB`.
   h. `CosmosDatabaseName` – Nastavit na řetězec `ParkingMonitoring`.

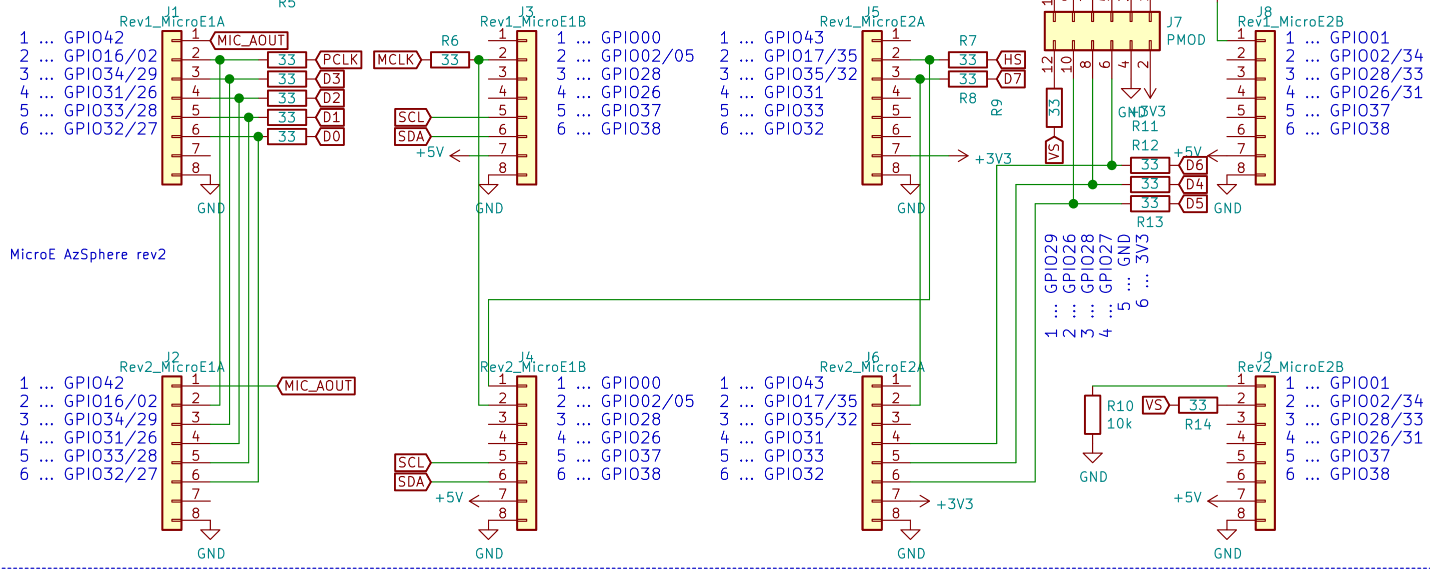
A.3.4 Webová administrace

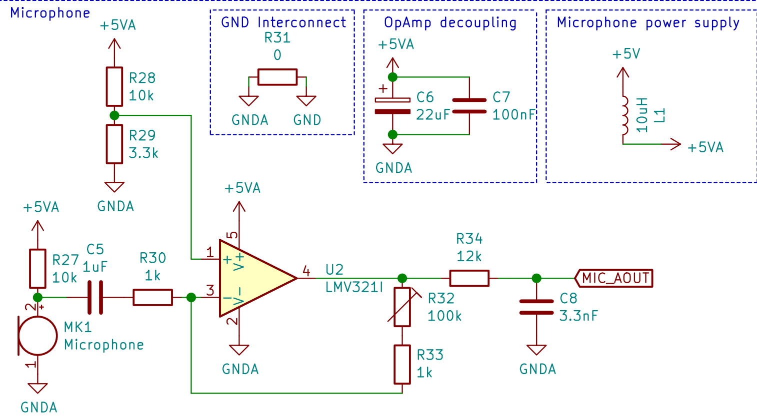
1. Nasazení webové aplikace do cloudu probíhá stejně jako bezstavové funkce. Je potřeba publikovat projekt z Visual Studia a projít průvodcem.
2. Při výběru `App Hosting Plan` je pro redukci nákladů dobré zvolit již existující plán a nezakládat nový.
3. Po nasazení je potřeba přes Azure Portal nastavit vlastnosti `CosmosEndpoint`, `CosmosEndpoint` a `CosmosEndpoint` na stejné hodnoty jako u funkce `ImageAnalysis`.

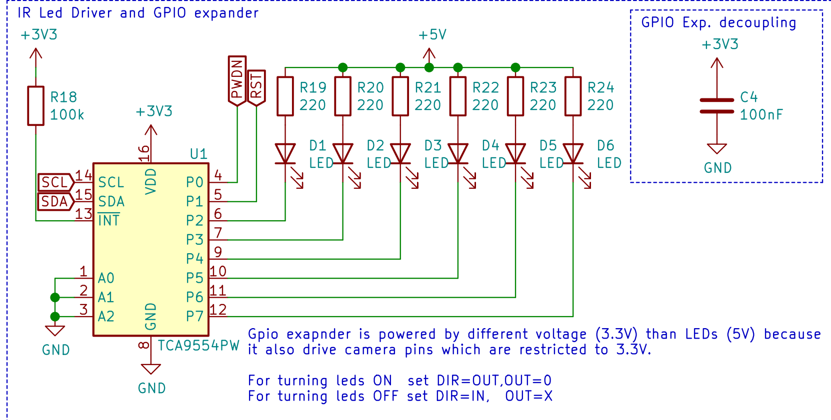
A.4 Nasazení firmware zařízení

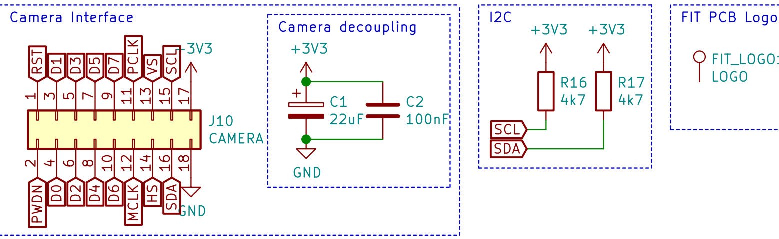
Pro nasazení firmware zařízení je potřeba do souboru `app_manifest.json` vysokoúrovňové aplikace doplnit správné URL adresy vytvořeného IoT Hubu, služby DPS a uložiště. Firmware lze nasadit pomocí Visual Studia nebo nasadit produkčně do cloudu pomocí postupu posaného na `https://docs.microsoft.com/en-us/azure-sphere/deployment/deployment-concepts`.

Příloha B
Schéma desky plošného spoje









I2C

FIT PCB Logo

Sheet: /
File: AzureSphereCameraBoard.sch

Title: Avnet Azure Sphere Camera Board

Size: A4	Date:	Rev: A
KiCad E.D.A. eeschema (5.1.9)-1		Id: 1/1

Bibliografie

1. O.K.SERVIS BIOPRO, S.R.O. *Monitorovací systém PROXIMOS* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://www.proximos.cz/`.
2. PARKINGDETECTION BY RCE SYSTEMS S.R.O. *ParkingDetection* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://www.parkingdetection.com/cs/domu/`.
3. ABDALLA, Peshraw Ahmed; VAROL, Cihan. Testing IoT Security: The Case Study of an IP Camera. In: *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*. 2020, s. 1–5. Dostupné z DOI: `10.1109/ISDFS49300.2020.9116392`.
4. ALHARBI, Rana; ASPINALL, David. An IoT analysis framework: An investigation of IoT smart cameras' vulnerabilities. In: *Living in the Internet of Things: Cybersecurity of the IoT - 2018*. 2018, s. 1–10. Dostupné z DOI: `10.1049/cp.2018.0047`.
5. VIDAS, Timothy; VOTIPKA, Daniel; CHRISTIN, Nicolas. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. *Woot*. 2011, roč. 11, s. 8–9.
6. RENESAS ELECTRONICS CORPORATION. *RX72N Group User's Manual: Hardware*. 2021. Č. R01UH0824EJ0111. Rev. 1.11.
7. STMICROELECTRONICS. *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm ® -based 32-bit MCUs Reference manual*. 2021. Č. RM0090. Rev. 19.
8. ESPRESSIF SYSTEMS. *ESP32 Series Datasheet*. 2021. Version 3.6.
9. CYPRESS SEMICONDUCTOR CORP. *PSoC® 62S2 Wi-Fi BT Pioneer Kit (CY8CKIT-062S2-43012)* [online]. 2021-03 [cit. 2021-04-25]. Dostupné z: `https://www.cypress.com/documentation/development-kitsboards/psoc-62s2-wi-fi-bt-pioneer-kit-cy8ckit-062s2-43012`.
10. STEVANOVIC, U.; CASELLE, M.; CHILINGARYAN, S.; HERTH, A.; KOPMANN, A.; VOGELGESANG, M.; BALZER, M.; WEBER, M. High-speed camera with embedded FPGA processing. In: *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*. 2012, s. 1–2.
11. LIU, Huizhong; TANG, Wei. Video Camera System Based on FPGA. In: *2014 Seventh International Symposium on Computational Intelligence and Design*. 2014, sv. 2, s. 249–252. Dostupné z DOI: `10.1109/ISCID.2014.152`.
12. ARM LTD. *DESIGNSTART FPGA* [online] [cit. 2021-04-25]. Dostupné z: `https://www.arm.com/resources/designstart/designstart-fpga`.
13. XILINX, INC. *S8051XC3: Super-Fast 8051 Microcontroller Core with Configurable Features and Peripherals* [online] [cit. 2021-04-25]. Dostupné z: `https://www.xilinx.com/products/intellectual-property/1-5ohxze.html`.

14. THE BONFIRE PROCESSOR PROJECT. *The Bonfire Processor* [online]. 2020 [cit. 2021-04-25]. Dostupné z: https://bonfirecpu.eu/.
15. BHAMBORE, Archana. S.; HARKARE, R. R. Dynamically reconfiguration of PLL using FPGA. In: *2010 International Conference on Computer and Communication Technology (ICCCT)*. 2010, s. 260–264. Dostupné z DOI: 10.1109/ICCCT.2010.5640519.
16. XILINX, INC. *Zynq-7000 SoC* [online] [cit. 2021-05-01]. Dostupné z: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.
17. MICROSOFT CORPORATION. *What is Azure Sphere?* [Online]. 2021-04 [cit. 2021-05-01]. Dostupné z: https://docs.microsoft.com/en-us/azure-sphere/product-overview/what-is-azure-sphere.
18. NXP USA, INC. *NXP and Microsoft Bring Microsoft Azure Sphere Security to the Intelligent Edge with a New Energy-Efficient Processor* [online]. 2019-06 [cit. 2021-05-01]. Dostupné z: https://media.nxp.com/news-releases/news-release-details/nxp-and-microsoft-bring-microsoft-azure-sphere-security.
19. HUNT, Galen; LETEY, George; NIGHTINGALE, Edmund B. *The Seven Properties of Highly Secured Devices* [online] [cit. 2021-05-01]. Dostupné z: https://aka.ms/7properties. 2nd Edition.
20. ING. TOMÁŠ EUR ING, Ph.D. Zahradnický; KOKEŠ, Ing. Josef. *Lecture notes in Computer Assisted Diagnosis*. České vysoké učení technické v Praze, Fakulta informačních technologií, Katedra informační bezpečnosti, 2021.
21. MICROSOFT CORPORATION. *Azure Sphere pricing* [online] [cit. 2021-05-01]. Dostupné z: https://azure.microsoft.com/en-us/pricing/details/azure-sphere/.
22. SEEED TECHNOLOGY CO.,LTD. *MT3620 Mini Dev Board* [online] [cit. 2021-05-03]. Dostupné z: https://www.seeedstudio.com/MT3620-Mini-Dev-Board-p-2919.html.
23. SEEED TECHNOLOGY CO.,LTD. *Azure Sphere MT3620 Development Kit_EU Version* [online] [cit. 2021-05-03]. Dostupné z: https://www.seeedstudio.com/Azure-Sphere-MT3620-Development-Kit-EU-Version-p-3134.html.
24. AVNET, INC. *AES-MS-MT3620-SK-G* [online] [cit. 2021-05-03]. Dostupné z: https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-ms-mt3620-sk-g-3074457345636825680.
25. MIPI ALLIANCE, INC. *mipi aliance* [online] [cit. 2021-05-01]. Dostupné z: https://www.mipi.org/.
26. LATTICE SEMICONDUCTOR CORP. *MIPI D-PHY Bandwidth Matrix Table*. 2018. Č. FPGA-UG-02041. Version 1.1.
27. OMNIVISION TECHNOLOGIES INC. *OV2775* [online] [cit. 2021-05-01]. Dostupné z: https://www.ovt.com/sensors/OV2775.
28. LEOPARD IMAGING INC. *LI-OV2775-GMSL2-xxxH Data Sheet*. 2018. Revision 1.2.
29. SONY CORPORATION. *IMX327LQR/LQR1 [Product information]*. 2018. Ver 1.3.
30. E-CON SYSTEMS™. *e-CAM220_CUMI327_MOD - Full HD Sony IMX327 Ultra-Lowlight Camera Module* [online] [cit. 2021-05-02]. Dostupné z: https://www.e-consystems.com/camera-modules/imx327-sony-starvis-full-hd-camera-module.asp#evaluation-kit.
31. RASPBERRY PI FOUNDATION. *Raspberry Pi High Quality Camera* [online] [cit. 2021-05-02]. Dostupné z: https://www.raspberrypi.org/products/raspberry-pi-high-quality-camera/.
32. OMNIVISION TECHNOLOGIES INC. *OV7670/OV7171 CMOS VGA (640X480) CAMERACHIP™ with OmniPixel® Technology*. 2005. Version 1.01.

33. LASKARDUINO.CZ. *CMOS VGA kamera OV7670 640x480 bez pamětia* [online] [cit. 2021-05-02]. Dostupné z: `https://www.laskarduino.cz/cmos-vga-kamera-ov7670-640x480--bez-pameti/`.

34. MICROCHIP TECHNOLOGY INC. *ATmega4808/4809 Data Sheet*. 2021. Č. DS40002173C. Rev. C.

35. NEO. *AWS IoT device SDK for Embedded C on Azure Sphere* [`"https://github.com/xiongyu0523/azure-sphere-aws-iot-device-sdk-embedded-c"`]. GitHub, 2020-12.

36. PLANCHE, Benjamin; ANDRES, Eliot. *Hands-On Computer Vision with TensorFlow 2: Leverage deep learning to create powerful image processing apps with TensorFlow 2.0 and Keras*. Packt Publishing, 2019. ISBN 978-1788830645.

37. MICROSOFT CORPORATION. *MT3620 Support Status* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/en-us/azure-sphere/hardware/mt3620-product-status`.

38. FENN, Peter. *Azure Sphere MT3620 Starter Kit Hardware User Guide*. Avnet, Inc., 2019. Č. DS40002173C. v1.5.

39. MEDIATEK INC. *MT3620 Product Brief*. MediaTek Inc., 2020. Č. PDFMT3620PB. v1.5.

40. PETR MACHALA. *Kamera OV7670 se STM periferií DCMI* [online]. Ústav radioelektroniky, Fakulta elektrotechniky a komunikačních technologií, VUT v Brně, 2015 [cit. 2021-04-25]. Dostupné z: `http://www.urel.feec.vutbr.cz/MPOA/2014/cam-ov7670`.

41. ATMEL CORPORATION. *AVR42777: Digital Sound Recorder using DAC with ATtiny817* [online]. 2016 [cit. 2021-05-13]. Č. AVR42777. Dostupné z: `http://ww1.microchip.com/downloads/en/appnotes/atmel-42777-digital-sound-recorder-using-dac-with-attiny817_applicationnote_avr42777.pdf`. Doc. Rev. 42777A.

42. MICROSOFT CORPORATION. *Memory use in high-level applications* [online]. 2021-02 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/en-us/azure-sphere/app-development/application-memory-usage`.

43. MICROSOFT CORPORATION. *Manage memory and latency considerations* [online]. 2019-11 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/en-us/azure-sphere/app-development/memory-latency`.

44. MEDIATEK LABS. *MediaTek MT3620 M4 Driver & Real-Time Application Sample Code* [`https://github.com/MediaTek-Labs/mt3620_m4_software`]. GitHub, 2019.

45. CODETHINK LABS. *MT3620 M4 Drivers* [`https://github.com/CodethinkLabs/mt3620-m4-drivers`]. GitHub, 2019.

46. MICROSOFT AZURE. *Azure Sphere Samples* [`https://github.com/Azure/azure-sphere-samples`]. GitHub, 2018.

47. MICROSOFT CORPORATION. *Memory available on Azure Sphere* [online]. 2019-07 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/cs-cz/azure-sphere/app-development/mt3620-memory-available`.

48. CODETHINK LABS. *GPIO.c* [online] [cit. 2021-05-09]. Dostupné z: `https://github.com/CodethinkLabs/mt3620-m4-drivers/blob/ff68ea510e55752ff24a55fe80f6562a4d7ed385/GPIO.c`.

49. FESL, Jan. *Počítačové sítě, Úvod do problematiky*. Katedra počítačových systémů, FIT ČVUT, 2021.

50. MICROSOFT CORPORATION. *Communicate with a high-level application* [online]. 2020-10 [cit. 2021-05-08]. Dostupné z: `https://docs.microsoft.com/en-us/azure-sphere/app-development/inter-app-communication`.

51. MEDIATEK LABS. *os_hal_mbox_shared_mem.h* [online] [cit. 2021-05-08]. Dostupné z: https://github.com/MediaTek-Labs/mt3620_m4_software/blob/406b600c70025b27307056f0d2fcc0ef35de1129/MT3620_M4_Sample_Code/OS_HAL/inc/os_hal_mbox_shared_mem.h.

52. MEDIATEK LABS. *os_hal_mbox_shared_mem.c* [online] [cit. 2021-05-08]. Dostupné z: https://github.com/MediaTek-Labs/mt3620_m4_software/blob/406b600c70025b27307056f0d2fcc0ef35de1129/MT3620_M4_Sample_Code/OS_HAL/src/os_hal_mbox_shared_mem.c.

53. TEXAS INSTRUMENTS INCORPORATED. *PCA9554 Remote 8-Bit I 2 C AND SMBus I/O Expander With Interrupt Output and Configuration Register* [online]. 2021 [cit. 2021-05-10]. Č. SCPS128D. Dostupné z: https://www.ti.com/lit/ds/symlink/pca9554.pdf?ts=1620667074865&ref_url=https%253A%252F%252Fwww.google.com%252F. Revision D.

Obsah přiloženého média

- `AzureSphere/` .... adresář s dílčími částmi firmware zařízení
  - `HLMainApp` .... adresář s vysokoúrovňovou aplikací
  - `RTCameraControlApp` .... adresář s mikrokontrolérovou aplikací ovládající kameru
  - `RTMemoryPoolApp` .... adresář s mikrokontrolérovou aplikací poskytující paměť
  - `Shared` .... sdílený kód mezi jednotlivými aplikacemi firmware
- `AzureSphereCameraBoard/` .... adresář s návrhy a výrobními daty desky plošného spoje
- `CloudSide` .... zdrojové kódy a části cloudové aplikace
  - `DeploymentScripts` .... adresář se skripty pro nasazení univerzální cloudové aplikace
  - `DeviceEmulator` .... emulátor zařízení pro snadnější ladění cloudových služeb
  - `DeviceMessagesProcessing` adresář s Azure Functions App projektem pro příjem dat ze zařízení
  - `ParkingMonitoring` .... adresář s ukázkovou aplikací pro sledování obsazenosti parkoviště
    - `ImageAnalysis` .... adresář s Azure Functions App projektem pro provádění analýzy obrázků
    - `Model` .... sdílený model s třídami pro EF
    - `WebInterface` . adresář s ASP.NET Core aplikací vizualizující výstupy a administrací pro monitorovací systém sledující obsazenost parkoviště
  - `DeploymentScripts` .... skripty pro nasazení základních cloudových služeb
- `Thesis` .... zdrojová forma práce ve formátu LaTeX
  - `zurekmi1.pdf` .... text práce ve formátu PDF
- `README.md` .... stručný popis bakalářské práce a obsahu média