České Vysoké Učení Technické v Praze
Fakulta Elektrotechnická

Diplomová práce

Zpracování videa pro mobilní roboty procesorem i.MX1

Aleš Zikmund 2009

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne:

podpis

Poděkování

Velmi rád bych poděkoval a vyslovil uznání všem, kteří mi pomáhali při vzniku této práce. Především Ing. Michalu Sojkovi, vedoucímu mé diplomové práce, za trpělivé vedení a množství praktických rad. Dále Ing. Pavlu Píšovi, který mi vždy ochotně pomohl cennými informacemi v oblasti programování ovladačů pro Linux.

V neposlední řadě bych chtěl poděkovat svým rodičům za vytrvalou podporu během studia.

Abstrakt

Práce se zabývá návrhem a realizací embedded systému snímání obrazu. Pro tento systém je použit procesor i.MX, který má hardwarovou podporu pro zachytávání obrazu. Práce obsahuje popis hardwaru i.MX periferie pro připojeni kamery a popis senzoru OV7660, který byl vybrán pro účel této práce. Na základě těchto dokumentací je v práci proveden návrh hardwaru pro propojení s deskou PiMX. V dalších kapitolách jsou vysvětlena jaderná programovací rozhraní potřebná pro implementaci driverů pod operačním systémem Linux. Těmi jsou V4L2, VIDEOBUF ale také DMA rozhraní, které umožňuje urychlit záznam dat. Po zdokumentování základních funkci driverů obsluhujících snímání obrazu je v práci uveden příklad aplikace zaznamenávající obraz. Při testování aplikace se zjistilo, že výsledný embedded systém nedosahuje, díky malému výpočetnímu výkonu, velkých vzorkovacích rychlostí.

Abstract

The thesis contains complete design and implementation of capture image embedded system. This system use processor i.MX that disposes of the hardware support of image capture. The thesis contains description of i.MX capture image peripheral and image sensor OV7660 description, that was selected for purpose in this thesis. The thesis next contains hardware design for connecting with PiMXboard. Next chapters explain the kernel programming interfaces that are required driver implementation on the operating system Linux. Thare are V4L2, VIDEOBUF and DMA interfaces. After principal function documentation of driver the thesis contains the image capture application example. At application testing it was taken that this complete embedded system don't reach fast frame rates because the i.MX has small computing power.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V poslední době přibývá aplikací, kde je třeba snímat obraz okolí. Užití zpracování obrazu umocňuje především u mobilních robotů, kde znalost obrazu prostředí, v němž se robot nachází, přináší značné výhody. Z obrazu lze například získat informace o poloze, směr jízdy, rychlost a další.

Získávání a zpracování obrazu přináší značné úskalí, protože obraz s kvalitním rozlišením, pro rozpoznání detailů ve snímku, obsahuje velké množství dat, které je nutné uchovat pro další výpočty z obrazových dat. To zapříčiňuje velké nároky na paměťový prostor pro data, ale také problém, jak přenést tak velké množství dat.

Hlavní centrum činnosti mobilních robotů většinou tvoří procesor, který řídí vlastní činnost robota na základě algoritmu (softwaru) a senzorů, jimiž je vybaven. Snímače slouží k vnímat vnějšího okolí a pokud je daným snímačem obrazový senzor, umožňuje robotovi vnímat okolí obdobně jako člověk.

Na trhu se vyskytuje velké množství procesorů dostačujících pro základní funkci autonomnosti. U mobilních robotů disponujících komplexnějším druhem řízení se vybírají procesory s nízkou spotřebou, vysokým výpočetním výkonem, možnostmi připojení periferií a komunikační moduly, které jsou implementované přímo na jednom čipu procesoru. Tyto parametry splňují procesory založené na architektuře ARM9[1].

Pro účel diplomové práce poskytl Ing. Pavel Píša desku PiMX, která byla vyvinuta firmou Pikron. Deska je založena na procesoru i.MX (ARM9328)od firmy FreeScale. Deska neobashuje pouze procesor, ale je také osazena dalšími komponenty pro možnosti programování procesoru a připojení periferií. Těmi jsou například USB, ethernet řadič, UART atd. Přesný popis desky je uveden v [1].

a)Horní pohled b)Spodní pohled

Obrázek 1.1 Deska PiMX

Další nespornou výhodou procesoru i.MX je přítomnost modulu pro zachytávání obrazových dat přímo ze senzoru snímající obraz, bez nutnosti připojení dalších podpůrných obvodů. Tím se značně zjednoduší návrh elektroniky pro zachytávání obrazu. Alternativním způsobem zachycení dat ze senzoru, je pomocí hradlového pole, který je dobré uvažovat, jedná-li se o aplikace, v kterých je nutné dosáhnout velké snímkovací rychlosti. S hradlovým polem však přibývá značné množství elektroniky vyžadující jeho zapojení.

[1]Advance RISC Machine

Modul procesoru i.MXpro připojení kamery se nazývá CMOS Sensor Interface zkráceně CSI (rozhraní pro připojení CMOS senzoru). Deska PiMX má vstupy a výstupy CSI modulu vyvedeny jako samostatný konektor (viz JP3 na obrázeku 1.1) pro snadnější připojení senzoru.

Úkolem a bylo tedy vybrat vhodný senzor, vyrobit vlastní desku s tímto senzorem a připojit ji k desce PiMX. Při výběru CMOS obrazového senzoru se však musí respektovat specifické parametry, které jsou charakteristické pro CSI modul na i.MX procesor. Například osmibitová datová sběrnice, synchronizační signály, napájení a další.

Jak už bylo zmíněno dříve procesor i.MX disponuje značným množstvím periferií a jak bývá zvykem, u takto komplexních procesorů se aplikace neprogramují přímo nad hardwarem, ale používá se operačního systému. K použití operačního systému přispívá přítomnost jednotky MMU (Memory Managment Unit), sloužící k virtualizaci paměti, nezbytnou pro práci operačního systému.

Procesory s jádrem ARM jsou podporovány jádrem unixového operačního systému Linux. Operační systém GNU/Linux na této architektuře je plnohodnotný operační systém s možnostmi spuštění více aplikací, správou paměti a další. Na desce PiMX aktuálně funguje Linux s jádro 2.6.24, na které musí být aplikováno několik patchů[2] (záplat) specifických pro tuto desku.

Operační systém Linux používá pro správu periferií nacházejících se v systému tzv. model ovladačů (driver model). Ten specifikuje každé zařízení tak, že musí mít v systému k sobě odpovídající ovladač (driver), který obsluhuje dané zařízení a poskytuje jeho funkcionalitu do celého systému. Přesný popis architektury ovladačů a jejich programování je uveden v [2]. Jelikož pro zařízení CSI není v jádru 2.6.24 podpora, bylo dalším úkolem diplomové práce implementovat odpovídající driver pro správný chod zařazení v operačním systému.

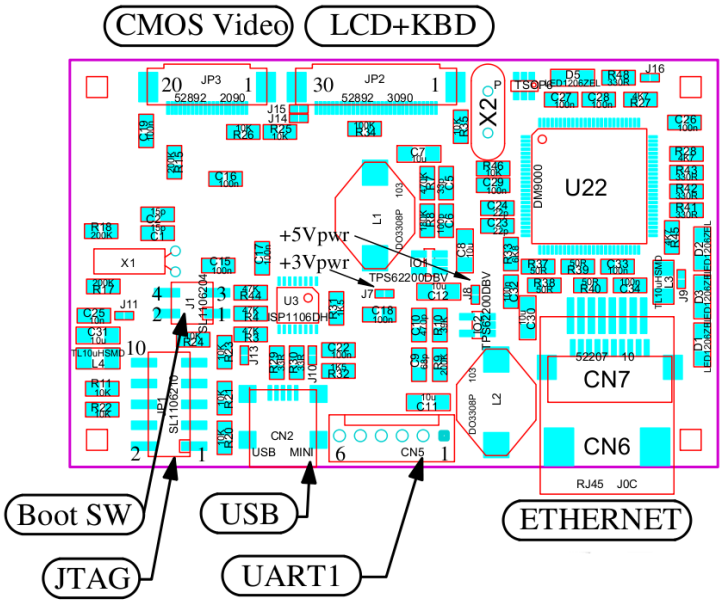
CSI modul představující video zařízení by měl být obsluhován driverem nabízejícím V4L2 API (Video for Linux 2), které je součástí jádra Linuxu a usnadňuje přistup k video zařízením a také jejich programování.

Zbytek práce je členěn následovně:

Kapitoly 2 a 3 popisují funkce hardwaru použitých pro vytvoření systému pro zachytávání dat. Následující kapitoly 4, 5 vysvětlují programovací prostředí pro implementaci driverů pro operační systém Linux.

Má vlastní práce zahrnující je uvedena v kapitolách 6 pro návrh hardwaru, 7 pro popis naprogramovaných driverů a 8 pro testovací aplikaci zachytávání dat.

[2] Patch (česky záplata) označuje (zpravidla) opravný kód, který pozměňuje původní zdrojový kód. Patche mohou obsahovat i více změn najednou, a to i pro různé soubory v rámci projektu. V patchy je přesně určeno, které řádky či části souboru má nahradit.



Obrázek 1.2 Popis desky PiMX

2 Rozhraní pro CMOS obrazový senzor

Tato kapitola obsahuje popis modulu pro připojení CMOS senzoru (CSI). Kapitola objasňuje architekturu, programovací model a inicializační sekvenci CSI modulu, který tvoří rozhraní umožňující k procesoru i.MX přímo připojit externí CMOS obrazový senzor.

2.1 Architektura CSI modulu

Hlavní rysy CSI modulu je možno do následujících bodů:

- Konfigurovatelná logika rozhraní pro podporu CMOS senzorů
- 8 bit datový port pro YCC, YUV nebo Bayer-RGB formát vstupních dat[3]
- 32 x 32 FIFO paměť pro ukládání obrazových data s podporou DMA[4] přenosů do systémové paměti.
- Maskovatelné zdroje přerušení : Start of Frame, FIFO full, and FIFO Overrun
- Konfigurovatelné master hodiny pro nastavení výstupní frekvence pro senzor
- Podporuje statistiku dat pro automatické nastavení expozice (AE) a automatické řízení vyvážení bílé (AWB).

Tomu odpovídá i architektura celého modulu zobrazená na obrázku 2.1. Význam důležitých signálů jsou popsány v tabulce 2.1

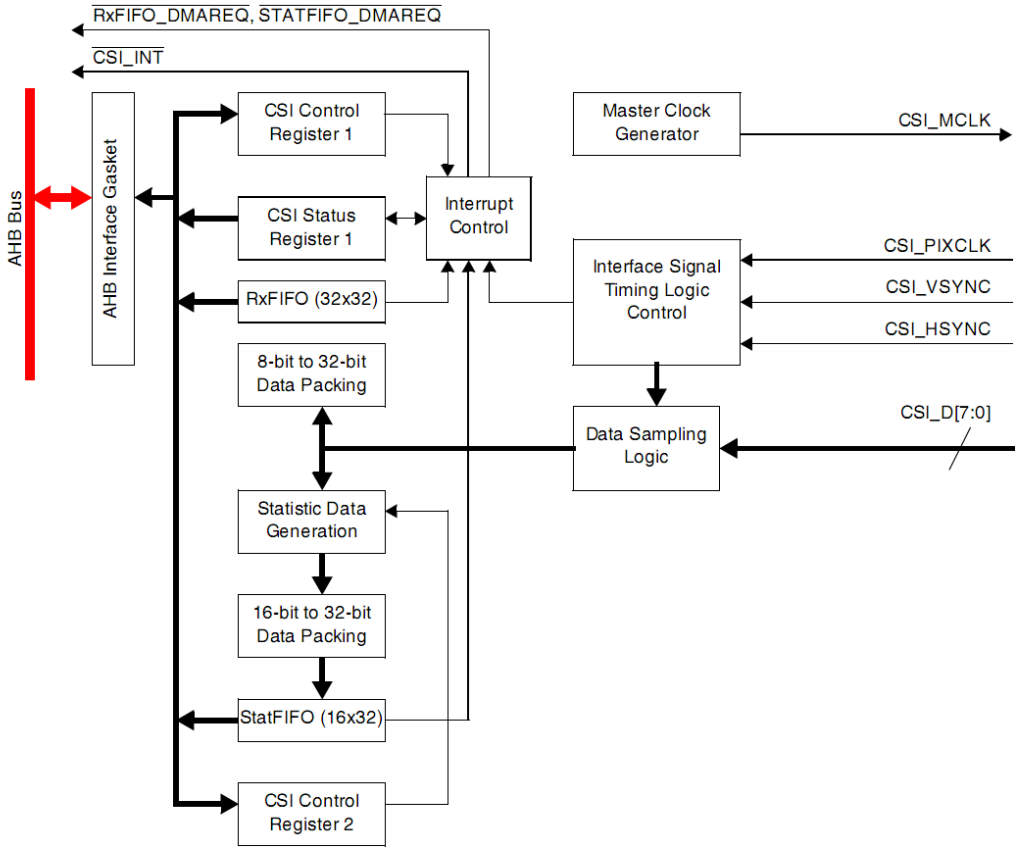
Architektura CSI modulu lze popsat následovně. Po nastavení registrů CSI modulu (viz kapitola 2.2 se začnou na výstupu CSI generovat hodinové pulsy na *MLCK*, ke kterým je připojen senzor a slouží jako hlavní taktovací kmitočet pro daný senzor. Tím začne obrazový senzor generovat potřebné signály odpovídající průběhům z obrázku 2.2. CSI modul zachytává data ze sběrnice *CSI_DATA* na základě signálu PIXCLK a ukládá je přímo do FIFO paměti. Vypočtená statická data se však ukládají do *STATFIFO*.

V závislosti na nastavení funkcí modulu lze k datům přistupovat dvěma způsoby. Pro oba platí, že se oběma FIFO pamětem nastaví úroveň zaplněnosti, která definuje množství dat pro přenos do systému. Jakmile je při plnění pamětí tato úroveň překročena generuje se

1. přerušení, po kterém lze data vyčíst nebo
2. modul požádá DMA řadič o přenesení dat do paměti.

Využití DMA přenosu je výhodnější v tom, že během přenosů se nezatěžuje procesor (viz kapitola 5).

[3]Typy formátů a jejich princip je vysvětlen v kapitole 3.4
[4]DMA-Direct Access Memory přímý přístup do paměti bez využití procesoru
[5]Start of Frame - začátek nového snímání obrazu



Obrázek 2.1 Blokové znázornění CSI modulu

Signál	Význam
CSI_VSYNC	Vertikální synchronizace nebo SOF[5]
CSI_HSYNC	Horizontální synchronizace dat
CSI_D[7:0]	8 bitová obrazová data
CSI_MCLK	Hlavní hodiny pro senzor
CSI_PCLK	Hodiny pro synchronizaci zachytávání dat

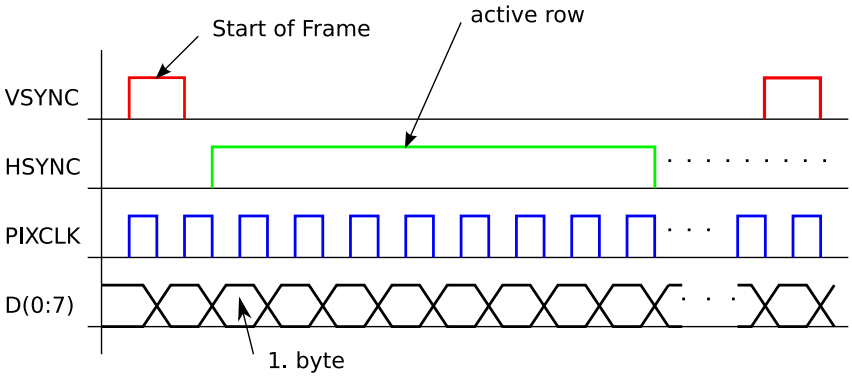
Tabulka 2.1 Význam důležitých signálu CSI modulu

Aby bylo zaručeno, že se všechna zachycená data přenesou do systému beze ztráty, lze zaregistrovat přerušení, které je vyvoláno při přetečení jednotlivé FIFO. Dalším důležitým přerušením je od signálu SOF označující začátek snímku.

Zachytávání dat ze senzoru probíhá dle obrázku 2.2.

Signály *VSYNC*[6] a *HSYNC*[7] určují v průběhu zachytávání obrazových dat platnost a pozici dat v obrazu. Každý snímek začíná pulsem na signálu *VSYNC*, který označuje začátek snímku. Od této chvíle začíná CSI zachytávat data ze senzoru na sběrnici *D(0:7)* podle hodinového signálu *PIXCLK*[8]. Data se však ukládají do paměti tehdy, je-li k aktivní řádkový synchronizační signál *HSYNC*, který generuje senzor na základě velikosti snímaného obrazu.

[6]VSYNC - Vertikální synchronizace
[7]HSYNC - Horizontální synchronizace
[8]PIXCLK - hodinový signál platnosti dat pixelů



Obrázek 2.2 Časování zachytávání dat CSI modulu

2.2 Programovací model CSI

Činnost CSI modulu je řízena pěti uživatelsky přístupnými registry, které jsou 32-bitové. Jejich pojmenování a uspořádání v paměti je následující:

Register	Adresa	Popis
CSICR1	0x00224000	Řídicí register 1
CSICR2	0x00224004	Řídicí register 2
CSISR	0x00224008	Stavový register
CSISTATR	0x0022400C	Register FIFO paměti pro statická data
CSIRXR	0x00224010	Register RXFIFO pro obrazová data

Tabulka 2.2 Registry CSI modulu

CSI Control registr 1

Tento 32-bitový registr poskytuje nastavení CSI modulu z hlediska časování a přerušení. Význam jednotlivých bitů je v tabulce 2.3.

Bity, které nejsou uvedeny jsou rezervovány a tudíž zápisem do nich se nic nezmění a při čtení dostaneme vždy logickou nulu. Pokud chceme pracovat s CSI modulem a zapisovat do dalších registrů CSI, musí být nejprve nastaven bit EN v registru CSICR1, jinak je celý modul neaktivní.

CSI Control register 2

Tento 32-bitový registr poskytuje nastavení bloku pro statistická data. Význam jednotlivých bitů je v tabulce 2.4.

CSI Status registr 1

Tento 32-bitový registr poskytuje nastavení bloku pro statistická data. Význam jednotlivých bitů je v tabulce 2.5.

Název (bit)	Popis
SF_OR_INTEN (25)	STATFIFO Overrun Interrupt Enable – Povolení/Zákaz Přerušení pro přetečení STATFIFO
RF_OR_INTEN (24)	RxFIFO Overrun Interrupt Enable – Povolení/Zákaz RxFIFO overrun přerušení
STATFF_LEVEL (23-22)	STATFIFO Full Level – definuje STATFIFO plnící úroveň, když je množství dat v STATFIFO stejný jako tato úroveň, FIFO full interrupt (je-li povolen) a DMA požadavek jsou generovány.
STATFF_INTEN (21)	STATFIFO Full Interrupt Enable— Povoleno/Zakázáno STATFIFO full přerušení
RXFF_LEVEL (20-19)	RxFIFO Full Level – definuje RxFIFO úroveň plnosti. Když je množství dat v RxFIFO shodné s touto úrovní, FIFO full interrupt (je-li povolen) a DMA požadavek jsou generovány.
RXFF_INTEN (18)	RxFIFO Full Interrupt Enable - Povoleno/Zakázáno RxFIFO full přerušení.
SOF_POL (17)	Start Of Frame Interrupt Polarity - Výběr hrany (náběžná nebo sestupná), která generuje SOF přerušení.
SOF_INTEN (16)	Start Of Frame Interrupt Enable - Povoleno/Zakázáno přerušení od signálu SOF
MCLKDIV (15–12)	MCLK Divider - Obsahuje děličku na systémových hodin pro generování MCLK signálu pro externí CMOS senzor.
MCLKEN (9)	MCLK Enable — Povolení/Zakázání MCLK signálu k externímu senzoru.
FCC (8)	FIFO Clear Control - Řídí mazání RXFIFO a STATFIFO, Synchronně po každém příchodu SOF. Asynchronně při zápisu 1 na bit CLR_RXFIFO nebo CLR_STATFIFO
BIG_ENDIAN (7)	Big Endian Enable - Konfigurace formátu dat pro RXFIFO
CLR_STATFIFO (6)	Clear STATFIFO - STATFIFO je smazána po zápisu 1 na tento bit
CLR_RXFIFO (5)	Clear RxFIFO - RxFIFO je vymazána po zápisu 1 na tento bitu
GCLK_MODE (4)	Gated Clock MODE - Řídí, jak způsob zachytávání dat. Když je bit nastaven zachytávají se data na vybranou hranu pixel hodin zároveň s aktivním signálem CSI_HSYNC. Při vynulování bitu jsou data zachycena na hranu vybranou hodinami pixelu bez ohledu CSI_HSYNC.
INV_DATA (3)	Invert Data Input - Invertuje CSI_D[7:0] data
INV_PCLK (2)	Invert PIXCLK Input - invertuje CSI_PIXCLK signál.
REDGE (1)	Rising Edge - Určuje hranu CSI_PIXCLK signálu, na kterou se zachytávají data
EN (0)	Enable - Povolení/Zakázání CSI

Tabulka 2.3 Význam jednotlivých bitů registru *CSICR1*

CSI Statistic FIFO a CSI RXFIFO registr

Oba 32-bitové registry slouží k vyčítání dat z přijímacích FIFO pamětí. Ať už z přímých dat obrazu pro RXFIFO nebo statistických dat pro STATFIFO. Zápisem do registrů nemá žádný efekt. Protože jsou registry 32-bitové a obrazová data jsou 8-bitová, jsou příchozí data spojeny na šířku 32 bitů dle sekvence dané formátem obrazu.

2.3 Generování statistických dat

Statistický blok generuje množinu normalizovaných sum pro červené, modré a zelené pixely podle obsazenosti v obrazu. Software může tyto hodnoty použít pro řízení osvitu (AE) a automatické naladění rovnováhy bílé barvy kamery. Blok dokáže také počítat sumu rozdílů

Název (bit)	Popis
DRM (26)	Double Resolution Mode - Indikuje, zda-li se vyhodnocuje pole 8x6 nebo 8x12 hodnot pro statistická data.
AFS (25-24)	Auto Focus Spread - Určuje , jak je skupina zelených pixelů porovnána pro výpočet absolutního rozdílu pro automatické řízení.
SCE (23)	Skip Count Enable - Povoluje/Zakazuje počítání přeskoků.
BTS (20-19)	Bayer Tile Start - Určuje vzor pro Bayer data
RLVRM (18-16)	Live View Resolution Mode - Určuje velikost obrazu (od 288x212 do 512x384 pixelů) pro generování statistických dat.
VSC (15-8)	Vertical Skip Count - Indikuje počet řádků k přeskočení během vypočtu statistických dat pro obrazovou velikost větší než 512x384 pixelů.
HSC (7-0)	Horizontal Skip Count - Indikuje počet horizontálních pixelů k přeskočení během vypočtu statistických dat pro obrazovou velikost větší jak 512x384 pixelů.

Tabulka 2.4 Význam jednotlivých bitů registru *CSICR2*

Název (bit)	Popis
SFF_OR_INT (25)	STATFIFO Overrun Interrupt - Indikuj přetečení STATFIFO. Bit je vymazán zapsáním 1.
RFF_OR_INT (24)	RxFIFO Overrun Interrupt - Indikuje přetečení RXFIFO. Bit je vymazán zapsáním 1.
STATFF_INT (21)	STATFIFO Full Interrupt - Indikuje, zda je STATFIFO plná. Bit je vymazán automaticky po provedeni vyčtení STATFIFO.
RXFF_INT (18)	RxFIFO Interrupt - Indikuje, zda RxFIFO je plná. Bit je vymazán automaticky po provedeni vyčteni RxFIFO.
SOF_INT (16)	Start Of Frame Interrupt - Indikuji příchod signálu SOF, který startuje nový frame. Bit je vymazán zapsáním 1.
DRDY (0)	Data Ready - Indikuje , že jsou data připravena k vyčteni z RxFiFO. Když je nastaven, je nejméně jedno datové slovo připraveno v RxFIFO

Tabulka 2.5 Význam jednotlivých bitů *CSISR1* registru

zelených pixelů pro určitý úsek obrazu, které lze použít k určení relativního zaostření obrazu. Důležitou vlastností je, že tato část modulu pracuje správně pouze tehdy, jsou-li senzorem posílána data ve Bayer formátu (viz kapitola 3.4).

2.3.1 Automatický osvit a rovnováha barev

Aby bylo možné počítat osvit a rovnováhu barev, snímá se paralelně k zachytávání obrazu další obraz, který má však maximální rozlišení 512x384 pixelů. Takto paralelně nasnímaný obraz se rozdělí do čtvercových bloků, z nichž dojde k vypočtení hodnot pro každou barvu. Suma každé z barev v bloku jsou dočasně uloženy v registru a v závislosti na *DRM* bitu registru *CSICR2*, je hodnota buď dělena 4 (*DRM*=0) nebo 2 (*DRM*=1), dříve než je uložena do *STATFIFO*. Z hodnot uložených v *STATFIFO* je možno dopočítat rovnováhu barev a osvit.

Suma diference barev je generována blokem SOAD a dočasně uložena v registru. SOAD hardware běží paralelně s hardwarem, který provádí sumu barev v obraze, ale generuje pouze jednu hodnotu na blok rozdělující obraz a to ze zelené barvy. Rozdíl je pak počítán na každém páru zelených pixelů v řádku bez překrývání.

Data vygenerovaná tímto modulem jsou pak ukládány do *STATFIFO* paměti v pořadí popsaném v tabulce 2.6.

bity 0-15	bity 16-31	číslo bloku
Červená	Zelená	1
Modrá	Zaostření	1
Červená	Zelená	2
Červená	Zaostření	2
⋮	⋮	⋮

Tabulka 2.6 Uspořádání dat v *STATFIFO*

Všechny ostatní údaje týkající se CSI modulu jsou uvedeny v [3]

3 CMOS senzor OV7660

V této kapitole je uveden popis senzoru obrazu OV7660, který byl vybrán s ohledem na CSI rozhraní procesoru i.MX. Je zmíněna vlastní architektura senzoru a také hlavní principy nastavení senzoru přes SCCB sběrnici včetně důležitých konfiguračních registrů registrů.

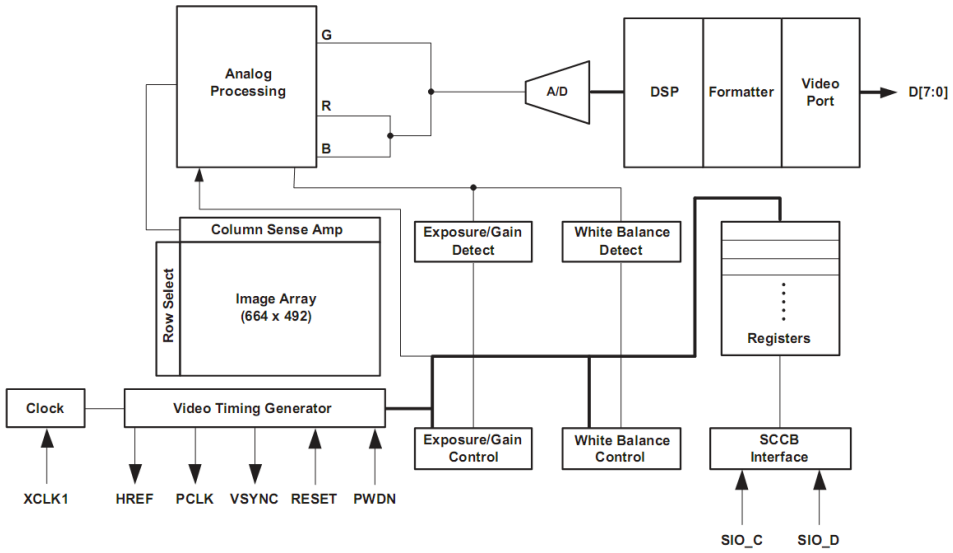
3.1 Popis senzoru OV7660

Čip OV7660 je nízkonapěťový CMOS obrazový senzor od firmy OmniVision navržený na velmi malou spotřebu řadově desítky mW a je zapouzdřen v malém BGA pouzdře. Velikost senzoru je 1/5 palce, což je přibližně 6 mm. Senzor poskytuje plnou funkci VGA kamery a obrazový procesor pro předzpracování dat. Nabízí jak plné VGA rozlišení, tak rozlišení v menších formátech.

Senzor má obrazové pole schopné operovat až 30 snímků (frame) za sekundu pro VGA rozlišení. Všechny požadované obrazové funkce jako například řízení osvitu, rovnováhy bílé, řízení saturace a zbarvení a další jsou programovatelné. Navíc tento senzor používá technologií k vylepšení kvality obrazu založenou na redukci eliminaci běžných rušivých zdrojích. Eliminace používá filtru, jehož konstanty lze nastavovat v registrech senzoru.

Řízení senzoru se provádí přes SCCB rozhraní (Serial Camera Control Bus), které je kompatibilní s I2C sběrnicí vysvětlené v kapitole 3.2.

Blokové schéma je na obrázku 3.1, z jehož uspořádání jsou viditelné důležité hlavní bloky celého senzoru.



Obrázek 3.1 Blokové schéma senzoru OV7660

Image sensor array

Aktivní obrazové pole, které má rozměr 640 sloupců x 480 řádků.

Timming generator

Časová základna, která se stará především časování synchronizačních signálů, časování odebírání obrazových dat a další časovací záležitosti na základě vstupních hodin *XCLK1*.

Analog signal processing

Tento blok se stará o obrazové funkce jako je automatické řízené zesílení a automatickou rovnováhu bílé.

A/D converter

Po analogovém zpracování obrazových dat, jdou data po jednotlivých pixelech dle Bayer vzoru do 10-bitového A/D převodníku, který převádí analogové hodnoty na digitální.

Digital signal processor

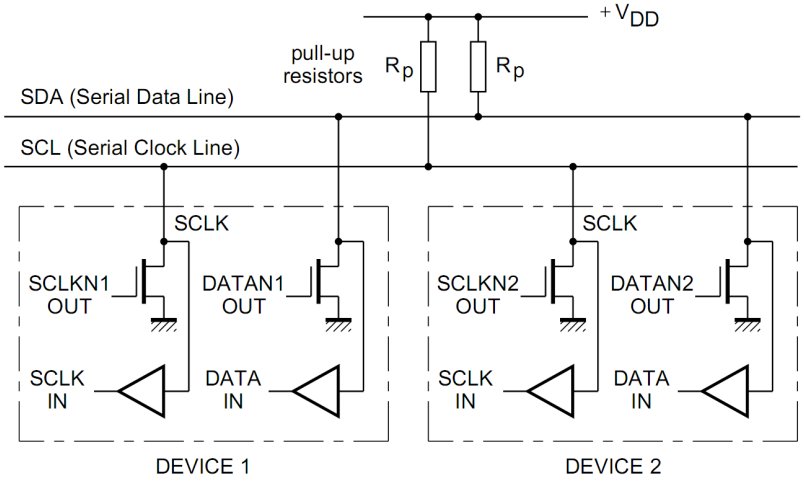
Signálový procesor převádí řádková data do odpovídajícího formátu, počítá saturaci a zbarvení a také převádí 10-bitová data na 8-bitová.

SCCB interface

Jak už bylo zmíněno, jde o sběrnici, přes kterou lze nastavit parametry senzoru. Její vysvětlení je v kapitole 3.2

3.2 SCCB sběrnice

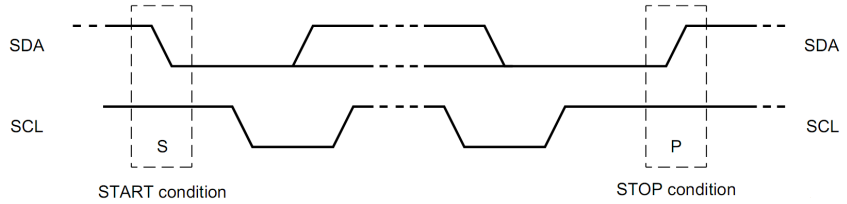
Tato sériová sběrnice je odvozena od sběrnice I2C a umožňuje taktovací rychlost až 160kHz. Skládá ze dvou obousměrných linek, které jsou buzeny otevřeným kolektorem. Linky se jmenují *SDA* (data) a *SCL* (hodiny). Aby fungovala sběrnice správně musí být vybavena pull-up odpory ke kladnému napájecímu napětí, které určují napěťové úrovně na sběrnicí. Zapojení sběrnice je na obrázku 3.2.



Obrázek 3.2 Zapojení I2C sběrnice

V základním stavu je na obou linkách vysoká úroveň a časování sběrnice se skládá ze tří základních sekvencí.

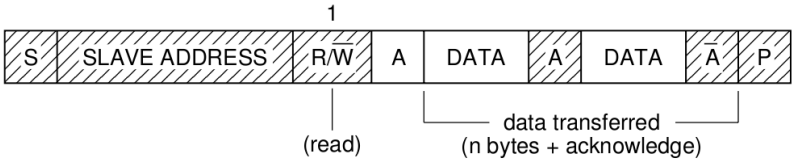
- *SDA* spádová hrana *SCL* vysoká úroveň - Start sekvence
- *SDA* platný bit *SCL* náběžná hrana - Záznam platného bitu
- *SDA* náběžná hrana *SCL* vysoká úroveň - Stop sekvence



Obrázek 3.3 Start a stop podmínka sběrnice I2C

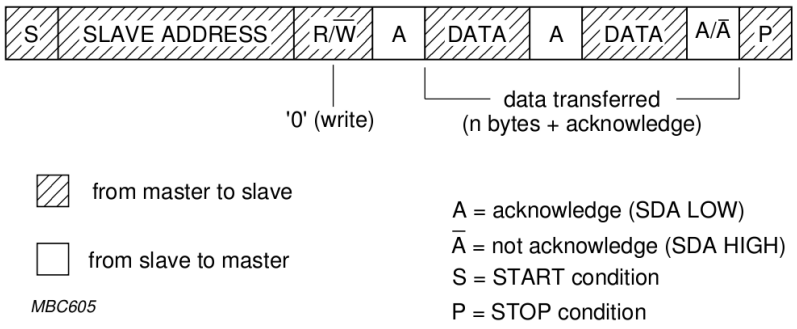
Data na I2C sběrnici jsou přenášena vždy po 9 bitech, kde devátý bit je potvrzovací přijímacím zařízením. Protože jsou všechna zařízení připojena k téže sběrnici, přistupuje master k jednotlivým slave zařízením pomocí adresy přenášené po *SDA* vodiči stejně jako data. V adresním bytu je však oproti datovému jako osmý bit posílá příznak, zda se jedná o čtení (vysoká úroveň) nebo o zápis (nízká úroveň). Tím je však omezeno adresování na 7 bitů (128 zařízení).

Přenos dat vypadá následně. Všechny přenosy začíná vysláním *START* podmínky mastera následující vysíláním 7 bitová adresy příjemce a jeden bit *R/W*, který indikuje požadovanou operaci (čtení/zápis). Potom slave odpoví bitem (*ACK*) pro potvrzení či nepotvrzení, že akceptoval požadavek (*ACK*=H vysoká úroveň) respektive neakceptoval pro (*ACK*=L nízká úroveň). Dále jsou přenášena data ve směru určeném předchozím bitem *R/W*. Každý byte je následován jedním potvrzovacím bitem *ACK* určující úspěch přenosu. Po ukončení přenosu je vyslána podmínka *STOP*.



Obrázek 3.4 Čtení dat na I2C sběrnici

Pro řízení komunikace se na I2C používá metoda s detekcí kolize. Každá ze stanic může zahájit vysílání, je-li předtím sběrnice v klidovém stavu. Během vysílání musí neustále porovnávat vysílané bity se skutečným stavem *SDA*. Je-li zjištěn rozdíl mezi očekávaným a skutečným stavem linky *SDA*, je to indikace kolize mezi několika stanicemi. Vzhledem k charakteru sběrnice (otevřené kolektory) může k této situaci dojít, pokud určitá stanice vysílá úroveň H, zatímco jiná stanice vysílá úroveň L. Stanice, která na lince zjistí úroveň L, zatímco sama vysílá H, musí vysílání okamžitě ukončit. K řízení komunikace většinou dochází během vyslání několika prvních bitů, kdy je vysílána adresa přijímací stanice. Pokud by se např. dvě stanice současně pokusily o zápis do stejného obvodu, nastane



Obrázek 3.5 Zápis dat na I2C sběrnici

kolize až při přenosu vlastních zapisovaných dat. V krajním případě, kdy několik stanic současně zapisuje stejná data na stejnou adresu, nemusí být kolize vůbec detekována.

U některých systémů je podpora tzv. *repeated start* (opakovaný start), kdy se nevysílá *STOP* podmínka, ale vyšle se znovu *START* podmínka, po které následuje ihned další přenos se stejným slavem. Tím je zaručeno, že daný master neztratí přístup k sběrnici, chce-li na ni znovu přistoupit k slavu. Obrazový senzor OV7660, však repeated start nepodporuje, což muselo být zohledněno při programování driveru pro komunikaci s tímto senzorem v kapitole 7.4.

3.3 Řídící registry senzoru OV7660

Důležitým parametrem pro nastavení registrů senzoru je slave adresa zařízení na sběrnici. Pro čip OV7660 jsou to adresy 0x42 pro zápis a 0x43 pro čtení. Tento senzor obsahuje kolem sta osmibitových registrů, proto budou vysvětleny významy registrů nutných pro změnu formátu, časování a identifikaci čipu.

Další registry senzoru OV7660 včetně jejich nastavení jsou uvedeny v [4] a vysvětlení výstupních formátů dat je uvedeno v následující kapitole.

3.4 Obrazové formáty

Tato kapitola značně souvisí jak s vlastností hardwaru senzoru, tak s později uvedenou kapitolou 4.1 týkající se V4L2 programovacího rozhraní.

Pro plně barevný popis obrázku je zapotřebí snímat celý prostor barev. V zásadě se používají dva druhy prostorů, kterými lze vyjádřit všechny barvy

1. RGB R-červená, G-zelená, B-modrá nebo

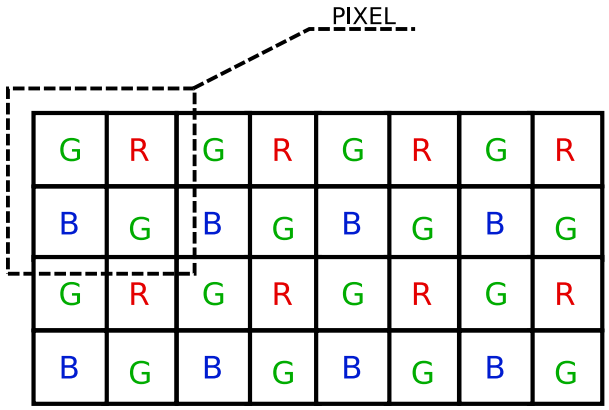
2. YUV/YCC Y-jas, U(Cr) - kontrast červené , V(Cb) - kontrast modré.

Mezi těmito dvěma vyjádřeními barev existuje jednoznačný přepočet a obrazové senzory většinou ovládají oba druhy. Výběrem jednoho z těchto záleží pak pouze na dané aplikaci. Snímací matice senzorů však bývá většinou vyrobena pro snímání RGB dat a na YUV formát se data přepočítávají.

Adresa (hex)	Jméno registru	Popis bitů
04	COM1	bit 6 - CCIR656 formát bit 5 - QQVGA or QQCIF formát
09	COM2	bit 4 - Softwarový sleep mode bi 1:0 - Výstupní rychlost
0A	PID	Identifikační číslo MSB
0B	VER	Identifikační číslo LSB
11	CLKRC	bit 6 - užití vnitřní děličky bit [5:0] - dělící poměr
12	COM7	bit 7 - Reset všech registrů bit 5 - Výstupní formát CIF bit 4 - Výstupní formát QVGA bit 3 - Výstupní formát QCIF bit 2 - Výstupní formát RGB bit 0 - Výstupní formát řádkový RGB
17	HSTART	Start horizont. synchr. sloupec
18	HSTOPT	Konec horizont. synchr. sloupce
19	VSTART	Start vertik. synchr. řádek
1A	VSTOPT	Konec vertik. synchr. řádek
1C	MIDH	Výrobní číslo (horní byte)
1C	MIDL	Výrobní číslo (dolní byte)
40	COM15	bity 7:6 - rozsah výstupních hodnot bit 5:4 - RGB 555/565 nastavení

Tabulka 3.1 Vybrané registry senzoru OV7660

Protože fyzicky nelze vyrobit senzor, který by snímal celý barevný prostor v jednom pixelu, vytváří se matice pixelů, kde každý pixel snímá jinou barvu. Uspořádáním snímacích pixelů v obrazu pak lze dosáhnout správného barevného rozlišení. Typické uspořádání matice je na obrázku 3.6, kde zelená barva je obsažena vícekrát, protože lidské oko je citlivější právě na zelenou barvu oproti modré a červené. Pak pro získání jednoho pixelu se všemi barvami musíme použít 4 subpixely.



Obrázek 3.6 Rozmístění obrazové matice na senzoru

Hodnoty pixelů jsou tedy vyjadřovány n-ticemi, které obyčejně obsahují obsahují RGB nebo YUV hodnoty. Pro organizaci těchto n-tic do obrazu existují dvě nejčastěji používané metody:

- Packed (spojené, sloučené) formáty ukládají všechny hodnoty jednoho pixelu v paměti dohromady a
- Planar (planární, ploché) formáty rozdělují všechny části do samostatných polí. Planar YUV formát tedy bude mít všechny hodnoty Y uloženy postupně v jednom poli, hodnoty U v dalším a hodnoty V ve třetím. Pole jsou obvykle uloženy postupně v jediném bufferu, ale není to pravidlo.

U obou typů formátů záleží na tom, jak se data posílají po sběrnici.

Kódy fourcc

Barevné formáty jsou v V4L2 API popsány pomocí mechanismu *fourcc* kódů. Tyto kódy jsou 32-bitové hodnoty generované ze čtyř ASCII znaků. Lze s nimi tedy lehce manipulovat a jsou snadno čitelné. Když například vidíte kód barevného formátu RGB4, není nutné vyhledávat význam v tabulkách. *Fourcc* však označuje pouze mechanismus a neříká nic o tom, jaké kódy jsou vlastně používány.

RGB formáty

V popisech formátů RGB (viz níže) jsou byty vždy řazeny podle umístění v paměti - dle little-endien jsou nejméně významné byty jako první. Nejvýznamnější bit je vyznačen světlejší barvou.

V4L2 název	fourcc	byte 0	byte 1	byte 2	byte 3
V4L2_PIX_FORMAT_RGB332	RGB1				
V4L2_PIX_FORMAT_RGB444	R444				
V4L2_PIX_FORMAT_RGB555	RGB0				
V4L2_PIX_FORMAT_RGB565	RGBP				
V4L2_PIX_FORMAT_RGB555X	RGBQ				
V4L2_PIX_FORMAT_RGB565X	RGBR				
V4L2_PIX_FORMAT_RGB24	RGB3				
V4L2_PIX_FORMAT_SBGGR8	BA81				

Tabulka 3.2 Vybrané RGB formáty

Při použití formátu *BA81* jsou hodnoty čteny přímo ze senzorové matice bez aproximace. Takovémuto formátu se říká Bayer.

YUV formáty

Příklady packed formátů v YUV jsou v tabulce 3.3. Šedé pole značí intenzitu Y, modrá jas Cb(U) a červená jas Cr(V).

V4L2 název	fourcc	byte 0	byte 1	byte 2	byte 3
V4L2_PIX_FORMAT_GREY	GREY				
V4L2_PIX_FORMAT_YUYV	YUYV				
V4L2_PIX_FORMAT_UYVY	UYVY				
V4L2_PIX_FORMAT_Y41P	Y41P				

Tabulka 3.3 Vybrané YUV formáty

Nezanedbatelným parametrem obrazu při jeho zachytávání je rozlišení obrazu. Rozlišení popisuje míru detailů v obraze. Nejčastěji se používá k vyjádření rozlišení počet sloupců obrazu (horizontální počet pixelů) a počet řádků obrazu (vertikální počet pixelů). Těmito dvěma údaji je také určen poměr stran (počet sloupců/počet řádků).

Z důvodu uspořádání snímací matice na obrázku 3.6 nelze tvrdit, že snímač má ve všech barvách stejné rozlišení jako je počet pixelů v obraze, protože jednotlivé pixely nenesou informaci o všech barvách (R, G, B).

Většinou se používají rozlišení daná standardem, aby byla kompatibilní s ostatními zařízeními. Typické standardy pro malé kamerky jsou v tabulce 3.4.

Standard	Rozlišení	Poměr stran	počet pixelů
QQVGA	160x120	4:3	19k
QCIF	176x144	11:9	25k
CGA	320x200	16:10	64k
QVGA	320x240	4:3	77k
CIF	352x288	11:9	101k
VGA	640x480	4:3	307k

Tabulka 3.4 Standardizované rozlišení

Parametry formátů jsou důležité při tvorbě driverů pro obrazový senzor (viz kapitola 7), kdy je nutné znát nastavené nejen rozlišení, ale také formát posílání dat, aby se dali správně interpretovat obrazová data přicházející ze senzoru.

4 API pro podporu video zařízení v jádře OS Linux

Tato kapitola obsahuje popis API přítomných v jádře operačního systému Linux, které podporují funkce video zařízení. Popisuje především hlavní principy V4L2 API ale také jeho struktura a nutné vlastnosti pro psaní driverů. Kapitola dále zahrnuje popis jaderného API sloužící pro podporu bufferů na zachycená data.

4.1 Video for Linux 2 API

Video for Linux Two (dále jen V4L2) je název pro programovací rozhraní (API) obsažené v jádře operačního systému Linux, které slouží k práci s video zařízeními. Druhá verze tohoto rozhraní je náhradou za předchozí první verzi a poprvé se objevila v linuxové jádře ve verzi 2.5.46. V4L2 rozhraní podporuje následující typy rozhraní:

- Rozhraní pro zachytávání videa (video capture) - bere video data z tuneru nebo kamery.
- Rozhraní pro výstup videa - aplikacím umožňuje ovládat zařízení, která umí poskytovat video obrázky například formou televizního signálu.
- Overlay video zařízení - je variací zachytávacího rozhraní, které se stará o přímé zobrazení video dat ze zachytávacího zařízení. Video data jdou přímo ze zachytávacího zařízení na displej, aniž by procházela přes procesor systému.
- VBI rozhraní -poskytují přístup k datům přeneseným během *blanking* intervalu. Jedná se o interval, kdy se nevysílají obrazová data, ale je možné stále zaznamenávat. Takto bývá přenášen například *teletext*.
- Rádiové rozhraní - poskytuje přístup k audio proudům z AM a FM tunerů.

a pro každé zařízení je možné implementovat tyto vlastnosti:

- Otevření a přístup k zařízení z uživatelské aplikace.
- Změna vlastností zařízení, výběr vstupů respektive výstupů atd.
- Nastavení formátu.
- Nastavení vstupně-výstupní metody pro čtení resp. zápis dat.
- Zavření a ukončení činnosti zařízení z uživatelské aplikace.

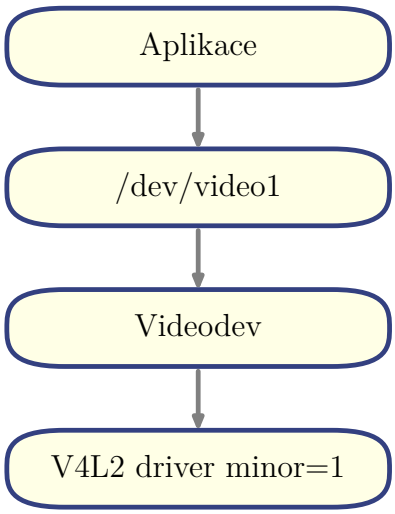
Ovšem v závislosti na typu zařízení nemusí být implementované všechny tyto vlastnosti.

Toto programovací rozhraní se používá především k obsluze video zařízení v aplikacích a driver zařízení musí být přizpůsoben tak, aby vyhovoval jeho struktuře.

4.1.1 Struktura V4L2 rozhraní

V4L2 je dvouvrstvý systém, který slouží pro psaní aplikací k obsluze video zařízení. Horní vrstvou je *videodev* modul, který zajišťuje při své inicializaci registraci všech znakových zařízení s major číslem 81 a jednotlivým zařízením přiřadí odpovídající funkce definované ve vrstvě V4L2 driver. Tím se stanou všechny V4L2 drivery klienty *videodev* modulu. Ten následně volá klientské drivery přes nadefinované V4L2 funkce driveru.

Když se V4L2 driver inicializuje, zaregistruje se každé zařízení k obsluhujícímu *videodev* modulu předáním *videodev* struktury, ve které jsou uvedeny V4L2 metody driveru. Ke každému V4L2 zařízení je pak přiděleno minor číslo, dle kterého vrstva *videodev* rozpozná o jaké zařízení se jedná. *Videodev* modul podle minor čísla volá odpovídající metody driveru. Struktura V4L2 lze jednoduše vyjádřit schématem na obrázku 4.1.



Obrázek 4.1 Struktura programovacího rozhraní V4L2

V4L2 metody driveru jsou velmi podobné standardní metodám linuxových znakových driverů, ale mají rozdílné parametry speciální pro V4L2 driver. Když aplikace provádí volání driveru, řízení předá nejprve do *videodev* modulu, kde se přeloží pointer na strukturu souboru na odpovídající pointer V4L2 struktury a volá metodu V4L2 driveru. Tak se *videodev* modul chová jako tenká vrstva kolem V4L2 driveru. *Videodev* je implementován jako kernel modul a všechny V4L2 drivery se implementují také jako kernel moduly.

4.1.2 Videodev modul

Videodev modul taky zahrnuje množinu pomocných funkcí, které mohou být užitečné pro toho, kdo píše V4L2 driver. Například, jednoduché frontové řízení, přetypování struktury *file** na V4L2 strukturu, pomocné funkce pro mapování paměti atd. Je to tedy rozhraní, které se stará o přenos dat mezi uživatelským prostorem a jádrem, a odesílá ovladači jednotlivá *ioctl, open, close* a další volání.

4.2 Registrace driveru a jeho metod

Když se driver inicializuje, detekuje se zařízení v systému, musí se pro něj vyplnit *video_device* struktura a následně se předá pointer na tuto strukturu do *v4l2_register_device* funkce. Typicky driver bude vnitřně definovat mnohem delší strukturu zařízení, která začne strukturou *struct video_device*, nasledovaná všemi stavovými informacemi o driveru a zařízení související s konkrétním zařízením. Pointer na tuto dlouhou strukturu muže byt použit (s odpovídajícím obsazením struktury *v4l2_device* na prvním místě) všude, kde je volána struktura *video_device*

Před voláním *v4l2_register_device* driver musí vyplnit *name, type a minor* pole a *fops* metody. Další metody a pole jsou nepovinné, ale jejich vyplnění je však dobrým pravidlem. *V4l2_register_device()* funkce si ověří, že dané minor číslo je dostupné a provede se inicializace. Jestliže se skončí inicializace úspěšně, pak je registrace kompletní. *Videodev* uloží pointer ve vnitřní tabulce, a tím je zajištěno zpřístupnění metod V4L2 driver jak *videodev* modulu tak i nadřízené aplikaci. Když se driver odpojuje ze systému, musí se zavolat metoda *v4l2_unregistr_device()* s pointrem na zařízení v argumentu, která zaručí bezpečné odebrání zařízení ze systému.

Všechny funkce provádějící se nad video zařízením z aplikace, se vyplňují do struktury *struct file_operations*, jejíž ukazatel se následně předá do struktury *video_device*. Protože se pro nastavování jednotlivých parametrů, kterých je u video zařízení velké množství (formáty, počet vstupu atd.), využívá V4L2 rozhraní *IOCTL*.

Struktura *video_device* obsažená v hlavičkovém souboru *v4l2_dev.h* je nadefinována:

```
struct video_device
{
  /* souborové operace se zařízením */
  const struct file_operations *fops;
  /* sysfs */
  struct device class_dev;        /* v4l device */
  struct device *dev;             /* device parent */
  /* info o zařízení */
  char name[32];
  int type;       /* v4l1 */
  int type2;      /* v4l2 */
  int minor;
  int debug;      /* ladící úroveň */
  /* Video standard proměnné */
  v4l2_std_id tvnorms;            /* Supported tv norms */
  v4l2_std_id current_norm;       /* Current tvnorm */
  /* zpětné volání při odebrání driveru */
  void (*release)(struct video_device *vfd);
  /* ioctl zpětné volání */
  /* VIDIOC_QUERYCAP handler */
  int (*vidioc_querycap)(struct file *file, void *fh, struct v4l2_capability *cap);
  /* další ioctl funkce*/
  int users;                    /* počet uživatelů na zařízení */
  struct mutex lock;            /* pomocný zámek */
}
```

4.3 V4L2 volaní fukncí driveru

Metoda *open*

```
int open(struct inode *inode, struct file *file)
```

Funkce je volána, při otevření filedescriptoru nad souborem zařízení (*/dev/videoX*). Funkce alokuje filehandle[9] zařízení obsahující pointer na vnitřní strukturu driveru, která obsahuje všechny stavové informace, včetně pointeru na zařízení použitého při registraci. Tento filehandle se dále používá pro přístup k zařízení přes strukturu *file* a její člen *private_data*. Protože V4L2 zařízení muže být otevřeno z více aplikací souběžně nebo z jedné aplikace vícekrát, je nutné v driveru držet informaci o počtu otevření a pro každé otevření tento počet zvýšit. Navratová hodnota je 0 pro úspěšné otevřeni nebo záporné číslo pro chybový kód. Navratová hodnota je vrácena aplikaci zpět.

Metoda *close*

```
int close(struct inode *inode, struct file *file)
```

Tato metoda se volá, když je zavírán filedescriptor zařízení. Metoda přijímá ukazatel na strukturu *file*, která obsahuje zmíněný filehandle, který byl aplikaci přidělen při předchozím volání metody *open*. Metoda zastaví činnost zařízení a uvolní odpovídající vnitřní struktury. Driver v této metodě také dekrementuje počet uživatelů, které zařízení používají.

Metoda *read*

```
ssize_t read(struct file *file, char __user *data, size_t count, loff_t *ppos)
```

Videodev modul volá tuto metodu driveru tehdy, když aplikace volá *read*, tedy čtení dat aplikací ze zařízení. Opět je předána struktura *file* nesoucí informaci o otevřeném zařízení a dále buffer **data*, délka bufferu *count* a offset **ppof*. Navratová hodnota je počet přečtených dat nebo záporné číslo pro chybový kód. Návratová hodnota je videodev modulem vrácena aplikaci zpět.

Metoda *write*

```
ssize_t write(struct file *file, char __user *data, size_t count, loff_t *ppos)
```

Je duální metodou k funkci *read*, ale slouží k zápisu dat z aplikace do driveru zařízení. Pro video zařízení zachytávající obraz (capture device) nemá význam tuto metodu implementovat, a proto se ve struktuře ponechává nevyplněna.

Metoda *mmap*

```
int mmap(struct file *file, struct vm_area_struct * vma)
```

[9]filehandle - struktura obsahující stavové informace oo otevřeném souboru

Aby se dalo snadno manipulovat s velkým objemem dat, což u video zařízení je základní předpoklad bývá také implementována metoda *mmap*. Parametr *struct file* představuje dané otevřené zařízení stejně jako v metodách *read,ioctl,..* a druhý parametr ukazatel na strukturu popisující oblast virtuální paměti.

Při použití *mmap* se namapují paměťové prostory z prostoru jádra do uživatelského prostoru. To způsobí možnost výměny dat bez nutnosti jejich kopírování, a tak není nutné provádět velké a časté přenosy dat mezi těmito dvěma prostory. Tato metoda značně souvisí s kapitolou4.4. Metoda vrací nulu při úspěšném namapování nebo zápornou hodnotu jako chybu.

Metoda *ioctl*

```
int ioctl (struct inode *inode, struct file *file, unsigned int command, unsigned long arg);
```

Jelikož dřívější video ovladače tradičně obsahovali velmi dlouhé *ioctl* funkce (switch příkaz), tak V4L2 API bylo v jádře 2.6.18 změněno, dlouhá funkce *ioctl* byla nahrazena velkou sadou zpětných volání, která implementují jednotlivé funkce *ioctl*. Tyto zpětné volání obstarává již zmíněná vrstva *videodev*. V jádře 2.6.19 jich je 79, ale většina ovladačů nemusí všechny implementovat. Odpovídající *ioctl* volání se vyplní do struktury *video_device* uvedené a aby je bylo možné využívat, musí ovladač jako svou *ioctl* metodu ve struktuře *video_device* používat *video_ioctl2*.

4.3.1 Volání *ioctl*

Protože V4L2 obsahuje velmi mnoho *ioctl* volání, jsou vysvětleny jen základní *ioctl* zpětné volání, použité pro video zařízení zachytávající obraz. Společné parametry pro všechna *ioctl* volání jsou *fd*, což je filedescriptor vrácený metodou *open*, a *request* určující volání.

ioctl *VIDIOC_QUERYCAP*

```
int ioctl(int fd, int request, struct v4l2_capability *argp);
```

Toto volání se používá k identifikování zařízení, předání informací o driveru a jeho hardwarových schopnostech. Informace jsou obsaženy ve struktuře *V4L2_CAPABILITY* popsané v [5].

ioctl *VIDIOC_QUERYCTRL*

```
int ioctl(int fd, int request, struct v4l2_queryctrl *argp);
```

Z tohoto volání se aplikace dozví o řídících informacích driveru pro dané *id* specifikované ve struktuře *v4l2_queryctrl* z uživatelského prostoru. Driver doplní zbytek struktury a vrátí ji zpět aplikaci.

ioctl *VIDIOC_G_CTRL*, *VIDEOC_S_CTRL*

```
int ioctl(int fd, int request, struct v4l2_control *argp);
```

G[10] - Driver předá prostřednictvím struktury *v4l2_control* aktuální hodnoty řízení dané *id* v této struktuře, kde *Id* specifikuje o jaký druh řídícího parametru se jedná. S[11] - Driver nastaví pomocí stejné struktury řídící hodnoty driveru.

ioctl *VIDIOC_G_PARM*, *VIDEOC_S_PARM*

```
int ioctl(int fd, int request, struct v4l2_streamparm *argp);
```

G - Driver předá prostřednictvím struktury *v4l2_streamparm* aplikace aktuálně nastavení hodnoty streamování (počet snímků za sekundu, atd.) S - Driver nastaví parametry streamování podle aplikací vyplněné struktury.

ioctl *VIDIOC_ENUM_FMT*

```
int ioctl(int fd, int request, struct v4l2_fmtdesc *argp);
```

Metoda zjistí, zda-li je obrazový formát dostupný podle *type* a *index* pole ve struktuře *v4l2_fmtdesc*. Pokud driver nalezne shodný formát vyplní zbytek struktury a předá ji zpět. Jestliže však nenajde stejný formát vrací chybový kód.

ioctl *VIDIOC_G_FMT*, *VIDIOC_S_FMT*, *VIDIOC_TRY_FMT*

```
int ioctl(int fd, int request, struct v4l2_format *argp);
```

Tyto ioctl volání slouží k pokusu nastavení (*TRY*), nastavení(*S*), respektive zjištění (*G*) aktuálního formátu dat, která jsou čtena z driveru. Driver vyplní nebo přijme strukturu *v4l2_format*, v níž jsou uvedeny atributy daného formát. A podle druhu *ioctl* příkazu buď nastaví formát dat nebo odpoví aplikaci aktuálně nastaveným formátem.

ioctl *VIODIOC_G_INPUT*, *VIODIOC_S_INPUT*

```
int ioctl(int fd, int request, int *argp);
```

(Dotaz na aktuální video vstup) G-Metoda uloží do argumentu **arpg* počet vstupů. (Výběr video vstupu) S-Při tomto volání nastaví driver požadovaný vstup dle čísla předaného v argumentu **arpg*.

ioctl *VIODIOC_ENUM_INPUT*

```
int ioctl(int fd, int request, struct v4l2_fmtdesc *argp);
```

Driver zjistí při tomto volání přítomnost vstupu specifikovaného ve struktuře *v4l2_input* v poli *index*. Pokud najde stejný vstup doplní driver zbytek struktury, v které jsou uvedeny informace o daném video vstupu daném jako například (typ, použitý standard, jméno atd.).

[10]Písmeno G je zkrátka pro *GET*, *ioctl* s tímto písmenem provádí získání hodnoty ze zařízení
[11]Písmeno S je zkrátka pro *SET*, *ioctl* s tímto písmenem provádí nastavení hodnoty do zařízení

ioctl *VIODIOC_QUERYBUF*

```
int ioctl(int fd, int request, struct v4l2_buffer *argp);
```

Toto *ioctl* volání je součástí metody pro mapování paměti. A může být použita k dotazování na stav jednotlivých bufferů, jestliže byl již buffer alokován pomocí *VIDIOC_REQBUFS*

ioctl *VIODIOC_QBUF*, *VIDIOC_DQBUF*

```
int ioctl(int fd, int request, struct v4l2_buffer *argp);
```

Aplikace volá *VIDIOC_QBUF*, aby driver zařadil prázdné nebo plné buffery do vstupní fronty a naopak volá *VIDIOC_DQBUF* pro vyřazení plných a zobrazených bufferů z výstupních front driveru. Používá se pro mapované buffery.

ioctl *VIODIOC_REQBUFS*

```
int ioctl(int fd, int request, struct v4l2_requstbuffers *argp);
```

(Initial Memory Mapping) Toto *ioctl* volání je používáno pro přípravu na mapování paměti bufferů. Mapované buffery jsou umístěny v paměti jádra a musí být alokovány právě tímto voláním, aby mohly být namapovány do aplikačního adresního prostoru.

Pro alokování bufferů v driveru aplikace inicializuje tři pole struktury *v4l2_requestbuffers*. Nastaví pole *type* znamenající typ buffer, *count* pole pro určení počtu bufferů k alokaci a pole *memory* je nastaveno na *V4L2_MEMORY_MMAP*. Pokud však není možné alokovat potřebné množství bufferů, například v důsledku nedostatku paměti, vrací ve stejném poli *count* počet úspěšných alokovaných. Aplikace může volat *VIDIOC_REQBUF* opětovně pro změnu počtu bufferů, nicméně musí být zaručeno, že všechny předchozí jsou již odmapovány.[12]

ioctl *VIDIOC_STREAMON*, *VIDIOC_STREAMOFF*

```
int ioctl(int fd, int request, const int *argp);
```

Volání startuje (*ON*) či zastavuje (*OFF*) proces zachytávání obrazu během *I/O* streamování. Streamování je možné spustit pouze tehdy, je-li nějaký buffer zařazen ve frontě tzv. aktivní, jinak se zachytávání přeruší.

Přesný popis všech *ioctl* příkazů V4L2 programovacího rozhraní je zdokumentováno v [5] včetně popisu předávaných struktur.

[12]Velikost bufferu jsou typicky závislé na nastaveni formátu dat (obrazu). Pokud by se nejprve pokusil změnit formát zatímco buffer je mapován, vyskytla by se chyba. Proto formát muže byt změněn až po té, co všechny buffery závisející na formátu byly odmapovány. Ale musí být opět provedeno *ioctl* volání *VIDIOC_REQBUFS*.

4.3.2 Přístup k datům v V4L2

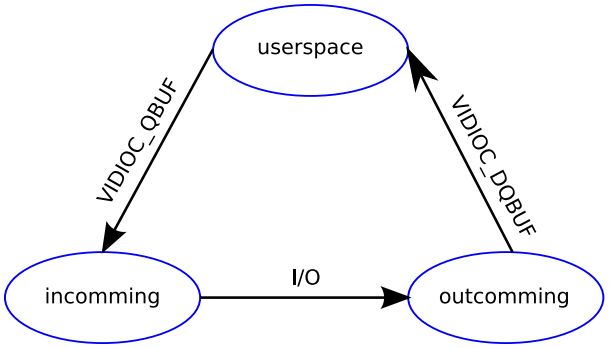
V4L2 programovací rozhraní definuje tři různé způsoby, jak se aplikace dostane k datům zachyceným driverem.

1. Mapování paměti *mmap*. Driver V4L2 zařízení alokuje paměť pro buffer. Aplikace následně požádá o namapování alokovaných bufferů do uživatelského prostoru. Pak k nim jednotlivě přistupuje za podmínky, jestliže byl již naplněn. Nejprve však musí být vyřazen z fronty odpovídajícím *ioctl* voláním.

2. Předáním uživatelského ukazatele *userpointer*. Aplikace si v uživatelském prostoru alokuje paměť pro buffery. Ukazatele na tyto buffery předá driveru zařízení, který si danou paměť zamkne, aby k ní nemohl nikdo přistupovat, a driver je začne plnit daty. Následně je k bufferům přistupováno jako při *mmap*.

3. Metoda *read*. Tímto způsobem lze přečíst pouze určité množství dat specifikované při volání metody *read*. Převážně se pro toto volání alokuje speciální buffer požadované velikosti a naplní se daty, které se zkopírují do uživatelského prostoru.

První a druhý způsob se nazývají streamované a k datům se aplikace dostane na základě stavů bufferů a *ioctl* volání. Typicky bývá v V4L2 driveru fronta bufferů, do které se bufferu dostane pomocí *ioctl* volání *VIDIOC_QBUF*. Když je buffer umístěn může jej driver naplnit daty. Aplikace tedy čeká na naplnění bufferu a jakmile jsou data v bufferu, čekání se zastaví a aplikace může odebrat buffer z fronty *ioctl* voláním *VIDIOC_DQBUF* a data si přečíst. Následně se buffer vrací do fronty opakováním *VIDIOC_QBUF*. Streamovaný způsob lze znázornit obrázkem 4.2.



Obrázek 4.2 Streamovaní ve V4L2

4.4 Videobuf API

Programovací rozhraní *videobuf* je implementováno v jádře Linuxu pro snadnější práci s video daty (video buffery). Typicky se snímky zachytávají do bufferů, které se následně předávají pro další zpracování, kde buffer představuje jeden snímek z obrazového senzoru. Pokud se používá metoda *mmap* nebo *userpoint* musí být při zachytávání videa přítomny minimálně dva buffery. Jeden, který je aktuálně plněn tzv. aktivní (active) a druhý, který je zobrazitelný tzv. zařazený ve frontě (queued), aby bylo snímaní videa plynulé. Většinou

se však používá větší počet bufferů. *Videobuf* modul poskytuje správu nad těmito buffery jako například

- Inicializaci bufferu
- Alokace paměti pro buffer
- Řazení do fronty
- Udržuje informace o aktuální stavu bufferu
- Polling bufferu
- Uvolnění bufferu

Základ tvoří struktura *videobuf_buffer*, která je nadefinovaná v souboru *linux/include/media/videobuf-core.h*.

```
struct videobuf_buffer {
  unsigned int            i; /* číslo bufferu */
  u32                     magic; /* identifikační číslo shodné pro buffery */
  /* info o bufferu */
  unsigned int            width;  /* šířka obrazu */
  unsigned int            height; /* výška obrazu */
  unsigned int            bytesperline; /* počet bytů na řádek */
  unsigned long           size; /* velikost */
  unsigned int            input;
  enum v4l2_field         field; /* typ pole */
  enum videobuf_state     state; /* aktualní stav */
  struct list_head        stream;  /* pro razeni bufferu ve streamu - QBUF/DQBUF list */
  /*  */
  struct list_head        queue; /* řazení do fronty */
  wait_queue_head_t       done; /* čekaní na dokončení */
  unsigned int            field_count;
  struct timeval          ts;
  /* typ paměti MMAP, USERPOINTER */
  enum v4l2_memory        memory;
  /* velikost buffer */
  size_t                  bsize;
  /* buffer offset */
  size_t                  boff;
  /* adresa userspace ukazatele na buffer) */
  unsigned long           baddr;
  /* mapování bufferu */
  struct videobuf_mapping *map;
  /* Privátní data (obrazová data)*/
                intprivsize;
  void                  *priv;
};
```

Videobuf poskytuje řadu metod, které odpovídají voláním V4L2 zařízení. Například pro metodu *mmap* ve V4L2 driveru lze využít *videobuf_mmap_mapper* nebo *ioclt* volání *STREAMON* lze implementovat metodou *vidoebuf_streamon* z *videbuf* atd. Všechny metody, které *videobuf* poskytuje driverům jsou nadefinované v hlavičkovém souboru *linux/include/media/videobuf-core.h*.

Před použitím *videobuf* v V4L2 driveru musí být provedena inicializace vnitřní struktury *videobuf* modulu *videobuf_queue*. Inicializace se provádí funkcí *videobuf_queue_core_init*. Jako hlavní parametre se předává struktura *videobuf_queue_ops*, jež definuje metody pro práci s buffery. Metody této struktury se deklarují až v samotném driveru V4L2 zařízení a předávají se až při inicializaci. *Videobuf* pak tyto metody aplikuje při voláních V4L2 metod. Těmito funkcemi lze docílit řazení, inicializaci a rušení bufferů ve ve V4L2 driveru.

```
struct videobuf_queue_ops {
  /*volána při funkci videobuf_reqbuf*/
  int (*buf_setup)(struct videobuf_queue *q,
       unsigned int *count, unsigned int *size);
  /*volána při funkci videobuf_qbuf*/
  int (*buf_prepare)(struct videobuf_queue *q,
         struct videobuf_buffer *vb,
         enum v4l2_field field);
  /*volána při funkci videobuf_streamon*/
  void (*buf_queue)(struct videobuf_queue *q,
        struct videobuf_buffer *vb);
  /*volána při funkci  videobuf_streamoff*/
  void (*buf_release)(struct videobuf_queue *q,
          struct videobuf_buffer *vb);
};
```

Struktura *videobuf_queue* je navržena také pro streamování. Obsahuje frontu, do které se přidá každý buffer při *ioctl* volání *VIDIOC_QBUF*, a tím si *videobuf* drží informaci o pořadí bufferů ve streamu. Pokud se buffer naplní a následně je zavoláno aplikací *VIDIOC_DQBUF*, buffer se vyřadí z této fronty a stává se volným pro práci v uživatelském prostoru. Když se data zpracují může být buffer opět zařadit do fronty.

Protože *videobuf* poskytuje velké množství takovýchto podpůrných operací velmi často se používá při implementaci V4L2 driverů.

5 DMA přenosy na procesoru i.MX

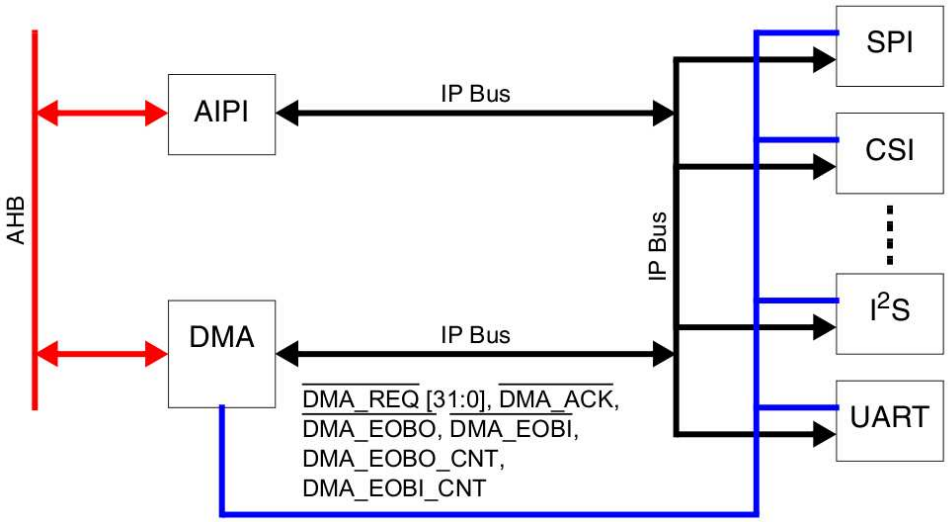
V této kapitole je uveden význam a popis DMA řadiče na procesoru i.MX. Dále je popsán způsob programování v operačním systému Linux na procesoru i.MX.

5.1 Princip a architektura DMA

Pro přenos velkých objemů dat se používá DMA řadič. Jedná se o subsystém procesoru, který je připojen na systémové sběrnici. Jeho hlavní činností je přenést data ze zařízení na systémové sběrnici do operační paměti bez použití procesoru. Tomu také odpovídá anglický název DMA (Direct Memory Access) tedy přímý přístup do paměti.

Právě proto, že se při přenosu dat z nějakého zařízení nepoužívá procesor, zvyšuje rychlost celého systému. Procesor pouze naplánuje přenosy, tedy odkud a kam se bude přenášet, a předá úlohu DMA řadiči. Ten ihned po přenesení všech dat upozorní procesor, že data jsou přenesena a procesor může k přeneseným datům přistoupit.

Architektura DMA na procesoru i.MX je na obrázku 5.1. DMA řadič je připojen k jednotlivým zařízením dalšími doplňujícími signály, které plní funkci žádosti o DMA přenosu. DMA řadič na i.MX podporuje přenos více bytů, proto má každé zařízení vyrovnávací paměť, která se postupně plní, a když je zaplněna požádá DMA řadič, aby provedl přenos.



Obrázek 5.1 Blokové schéma DMA

Další signály slouží jako potvrzení příjmu bytů. Signál DMA_INT, který generuje při kompletním přenosu, vyvolává přerušení, ze kterého lze detekovat konec přenosu.

Na architektuře ARM má DMA řadič 11 kanálů pro jednotlivá zařízení. Každý kanál představuje možnost použití nějakého zařízení. Kanály jsou seřazeny podle priority, kterou se určuje přednostní vyřízení žádosti o přenos (kanál 1 má nejvyšší prioritu a kanál 11 nejnižší) a priority se přidělují podle požadavku aplikace.

Pro každý kanál se je nutné nastavit několik registrů specifikujících parametry přenosu

- cílovou adresu
- zdrojovou adresu
- šířka přenášeného slova
- počet bytů v jednom přenosu
- celkový počet bytů
- řídící registry kanálu

5.2 DMA v operačním systému Linux

Použití DMA přenosů v operačním systém je složitější. Hlavní problémem se stává stránkování paměti[13] operačním systémem. Při virtualizaci paměti je problémem alokovat větší paměť (např. pro obrazová data) než je stránka. Pak může nastat situace, že fyzické adresy na sebe nenavazují, ale jsou různě rozprostřeny v celé paměti a DMA řadič dokáže přenášet pouze do fyzicky kontinuální paměti.

Proto se používá tzv. scatter-gather list (rozděl-spoj seznam). Seznam představuje posloupnost segmentů, které po sjednocení tvoří kontinuální virtuální paměť, avšak fyzické adresy nenavazují. Každý prvek seznamu tak obsahuje údaje o segmentu fyzické paměti (adresu, velikost segmentu), kde se nachaázejí fyzické adresy. Při plnění paměti pomocí scatter-gather listu se postupně prochází tento seznam. Začíná se prvním prvkem od jeho adresy. Jakmile se zaplní celý segment paměti, dáno velikostí, DMA řadič vyvolá přerušení a vynutí si další segment další segment v seznamu scatter-gather. To se opakuje až do konce seznamu.

Převod virtuální paměti na scatter-gather list se provádí pomocí funkce *vmalloc_to_sg*, která je implementována v jaderném API *videobuf*.

```
static struct * vm_to_sg(unsigned char *virt, int nr_pages)
{
  struct scatterlist *sglist;
  struct page *pg;
  int i;
  /*alokování paměti pro sg list*/
  sglist = kcalloc(nr_pages,
      sizeof(struct scatterlist),
      GFP_KERNEL);
  if (NULL == sglist){
    return NULL;
  }
  /*inicializace sg listu*/
  sg_init_table(sglist, nr_pages);
```

[13]Stránkování paměti je technika adresace operační paměti, která umožňuje zobrazit virtuální paměťový prostor (tzv. virtuální paměť) do fyzického adresového prostoru operační paměti. Mapování paměťového prostoru se provádí po stránkách, stránky mají zpravidla stejnou velikost (např. 4kB).

```
  /*vkládání jednotlivých segmentů do sg listu*/
  for (i = 0; i < nr_pages; i++, virt += PAGE_SIZE) {
    //pg=virt_to_page(virt);
    pg = vmalloc_to_page(virt);
    if (pg==NULL){
      goto err;
    }
    BUG_ON(PageHighMem(pg));
    /*vloženi záznamu*/
    sg_set_page(&sglist[i], pg, PAGE_SIZE, 0);
  }
  return sglist;
err:
  kfree(sglist);
  return NULL;
}
```

Podmínkou pro tento převod je použití funkce *vmalloc* pro alokování virtuální paměti. Některé DMA řadiče mají přímo podporu pro používání scatter-gather listu, ale procesor i.MX tuto podporu nemá. Proto je pro i.MX naprogramován modul, který obsahuje jaderné API emulující scatter-gather list na tuto architekturu.

5.3 Programování DMA pro i.MX

DMA jaderný modul také implementuje všechny obslužné funkce související s architekturou procesoru i.MX.

- žádost o kanál
- nastavení kanálu
- povolení dma
- handlery pro konec či chybu přenosu

```
int imx_dma_request_by_prio(imx_dmach_t * pdma_ch,
          const char *name,
          imx_dma_prio prio)
```

Funkce požádá o volný kanál. Alokovaný kanál vrátí v argumentu *pdma_ch*. Argument *name* určuje jméno driveru, který si kanál alokuje, a *prio* je priorita zabíraného kanálu.

```
void imx_dma_free(imx_dmach_t dma_ch)
```

Funkce uvolní DMA kanál.

```
void imx_dma_disable(imx_dmach_t dma_ch)
```

Zákaz DMA přenosů pro kanál daný parametrem *dma_ch* (číslem kanálu) získaným z *request_by_priority*.

```
void imx_dma_enable(imx_dmach_t dma_ch)
```

Zákaz DMA přenosů pro kanál daný parametrem *dma_ch* (číslem kanálu) získaným z *request_by_priority*.

```
int imx_dma_setup_handlers(imx_dmach_t dma_ch,
           void (*irq_handler) (int, void *),
           void (*err_handler) (int, void *, int),
           void *data)
```

Nastaví funkce pro obsluhu přerušení konce nebo chyby přenosu. Do funkcí je možné předat argument data pomocí argumentu *data*.

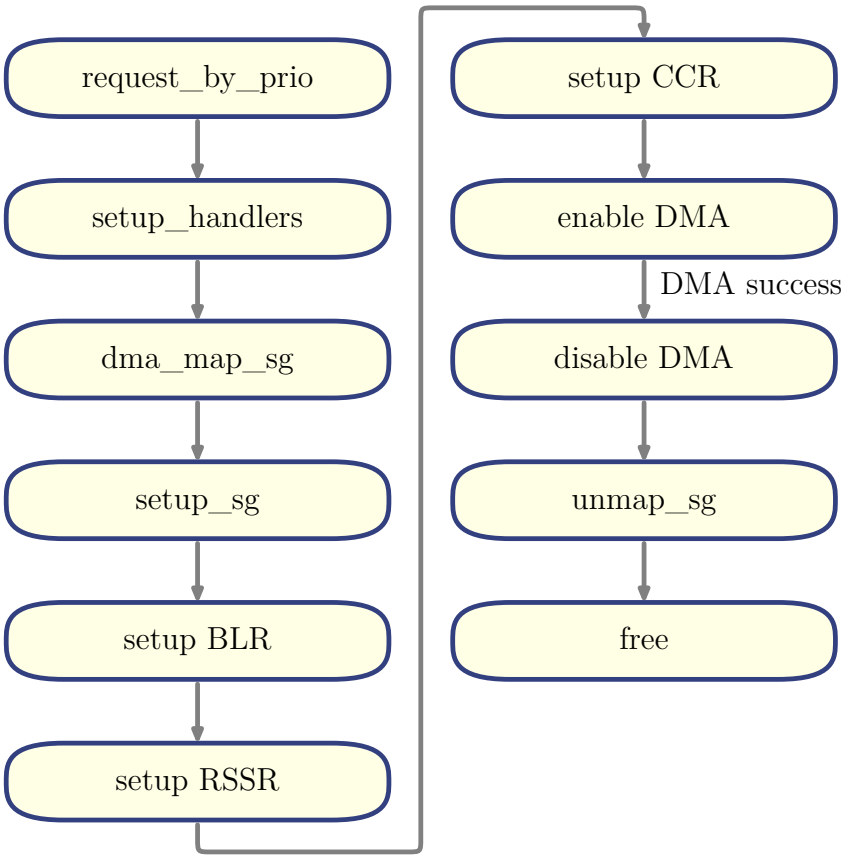
```
int imx_dma_setup_sg(imx_dmach_t dma_ch,
     struct scatterlist *sg, unsigned int sgcount, unsigned int dma_length,
     unsigned int dev_addr, dmamode_t dmamode)
```

Poslední funkcí je nastavení scatter-gather listu **sg, sgcount* na daný kanál *dma_ch*. Důležité je uvést přesnou adresu zařízení *dev_addr*, odkud se budou data při přenosu číst. Pro CSI je to adresa *CSIRX* registru. Argument *dmamode* specifikuje směr přenosu od či ze zařízení.

Protože tyto funkce nezajišťují kompletní nastavení DMA kanálu musí se použít makra nadefinovaná pro architekturu ARM v souboru
linux/include/asm-arm/arch-imx/imx-regs.h.

- CCR (Control register) - nastavení módu přenosu, šířka zdrojové adresy, šířka cílové adresy a povolení žádosti o přenos.
- RSSR (request source select register) - nastavení čísla zařízení, které žádá o DMA přenos
- BLR (burst lenght register) - nastavení počtu bytů pro jeden DMA přenos

Celá příprava pro DMA přenosy se děje podle vývojového diagramu na obrázku 5.2.



Obrázek 5.2 Vývojový diagram pro DMA na i.MX

6 Návrh hardwaru

V této kapitole se čtenář dozví o postupu návrhu plošného spoje, který byl vytvořen v rámci této diplomové práce. Zahrnuje všechny teoretické předpoklady a výpočty navrženého obvodu pro propojení senzoru s deskou PiMX. Na konci kapitoly jsou uvedeny obrázky vyrobeného a osazeného plošného spoje.

6.1 Přizpůsobení napájení

Aby bylo možné připojit vybraný senzor OV7660 k desce PiMX. Bylo nutné vytvořit plošný spoj, který bude osazen senzorem s doplňujícími součástkami a konektorem pro spojení s deskou PiMX.

Na konektor pro připojení kamerky na desce PiMX je vyvedeno napájecí napětí 3 a 5 voltů. Obrazový senzor však pracuje s třemi napájecími napětími dle tabulky 6.1.

Typ	Typické napětí
Napájení jádra	1,8V
Napájení digitální části	2,6V
Napájení analogové části	2,6V

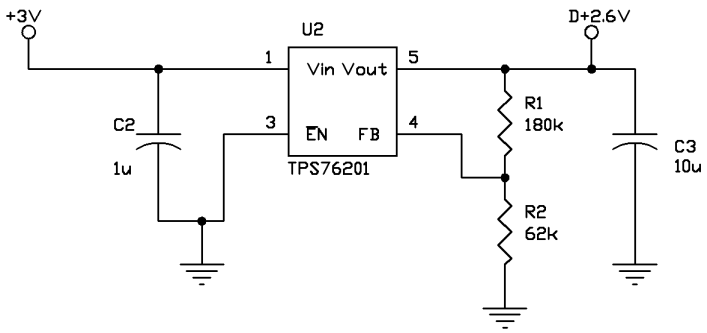
Tabulka 6.1 Napájecí napětí senzoru OV7660

Proto se muselo napájecí napětí přizpůsobit senzoru.

V konečném řešení byla zvolena varianta s lineárním stabilizátorem napětí s malým úbytkem napětí (low drop). Tato varianta byla zvolena, protože lineární stabilizátory mnoho neruší oproti DC-DC konvertoru a protože hlavní nevýhoda lineárních stabilizátorů, tj. velké ztráty, se při velmi malém odběru senzoru neprojeví. Byla tedy vytvořena dvě napájecí napětí pro jádro $1,8$V a pro digitální a analogovou část společné $2,6$V.

Aby se snížil přenos rušivých kmitů z digitálního na analogové napájení byl vytvořen LC filtr, který snižuje tyto kmity a napomáhá k vyšší stabilitě analogového napájení.

Pro stabilizaci napětí byl zvolen obvod TPS76201 od firmy Texas Instruments. Jde o nízko úbytkový lineární stabilizátor určený pro malá výstupní napětí v rozsahu 0,5 až 5 voltů, kde velikost velikost výstupního napětí určuje odporový dělič na výstupu (viz obrázek 6.1) podle vztahu (6.1).



Obrázek 6.1 Zapojení stabilizátoru napětí TPS76201

$$V_{OUT} = V_{REF}(1 + \frac{R_1}{R_2}). \tag{6.1}$$

V tomto vztahu je V_{REF} vnitřní referenční napětí stabilizátoru a má hodnotu $0,6663$V a hodnoty odporů musí být desítky kOhmů.

Po vyjádření odporu R_1 z rovnice (6.1) a zvolení odporu R_2 z manuálu [6] dostanemo pro napájecí napětí jádra $(1,8$V$)$

$$R_1 = R_2 \left(\frac{V_{OUT}}{V_{REF}} - 1 \right) = 68 \cdot 10^3 \left(\frac{1,8}{0,6663} - 1 \right) = 120k\Omega \tag{6.2}$$

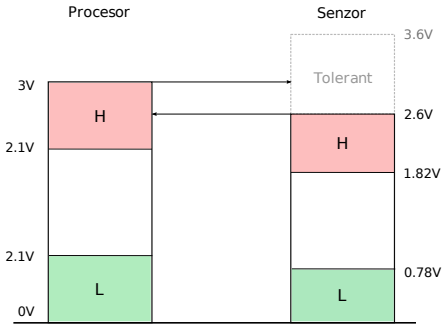
a pro napájecí napětí digitální a analogové části $(2,6$V$)$

$$R_1 = R_2 \left(\frac{V_{OUT}}{V_{REF}} - 1 \right) = 62 \cdot 10^3 \left(\frac{2,6}{0,6663} - 1 \right) = 180k\Omega \tag{6.3}$$

6.2 Kontrola logických úrovní

Důležité bylo také ověření napěťových úrovní jednotlivých signálů jdoucích k resp. od desky PiMX (procesoru ARM). Senzor má všechny vstupy +1V tolerantní, to znamená, že je schopen akceptovat na vstupu napěťovou úroveň o 1V vyšší než je napájení jeho digitální části $(2,6$V$)$. Na druhé straně spojení je digitální část procesoru i.MX napájena třemi volty, a tak dokáže senzor rozpoznat logické úrovně.

Zpětná kompatibilita je rovněž zaručena díky napěťovým úrovním CMOS logiky. Vstupy procesoru jsou schopné přijmout jako logickou úroveň H napětí $0,7U_{CC}$. Pro logickou nulu je kompatibilita zaručena, jelikož na výstupu CMOS senzoru je při log. 0 napětí $0,1U_{CC}$. Kompatibilitě napěťových úrovní odpovídá obrázek 6.2. Jak je vidět z obrázku mohou být signály z resp. k procesoru připojeny přímo bez jakýchkoliv konvertorů úrovní.

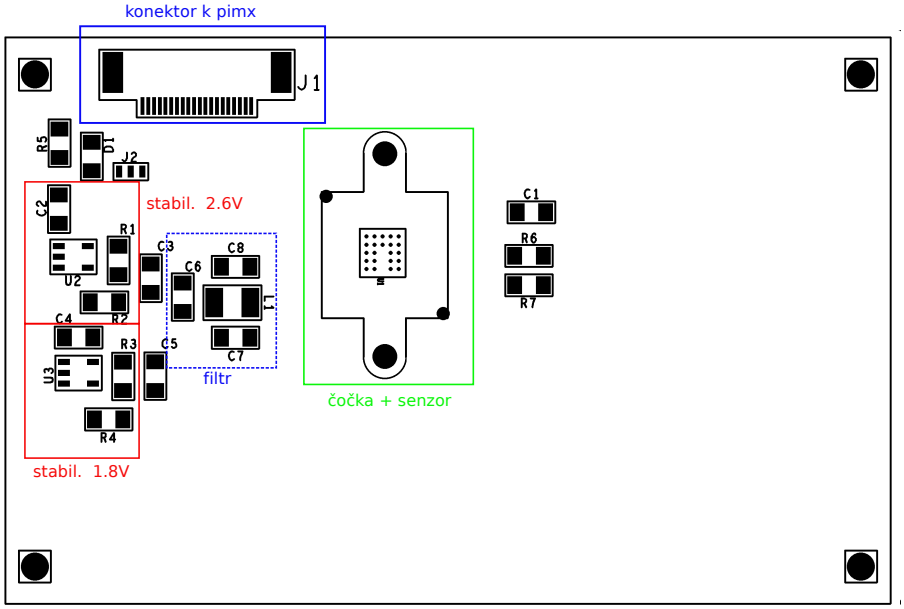


Obrázek 6.2 Logické úrovně procesoru a senzoru

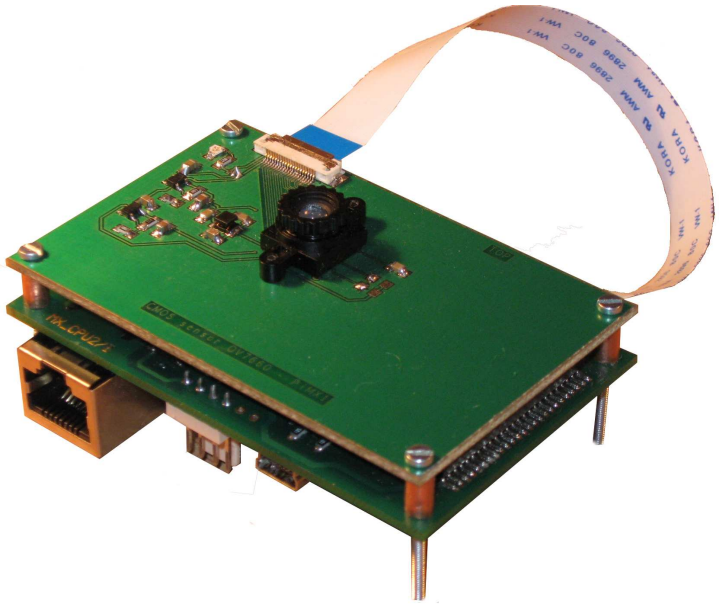
6.3 Konečná realizace desky pro senzor OV7660

Deska byla navržena jako dvouvrstvý plošný spoj stejného rozměru jako deska PiMX. Tudíž musely být zachována stejné pozice montážních otvorů. Navíc jsou na desce připraveny otvory pro optickou čočku, bez níž by nešlo snímat reálný obraz. Kompletní schéma

zapojení včetně hodnot součástek je umístěno v přílohách A a B. Příloha také obsahuje horní, spodní vrstvu, osazovací výkres navrženého spoje a hodnoty součástek. Protože byl celý plošný spoj byl navrhován v softwaru ORCAD je adresářová struktura projektu na přiloženém CD.



Obrázek 6.3 Osazovací popis desky



Obrázek 6.4 Kompletní realizace PiMX s OV7660

7 V4L2 drivery

V této kapitole je uveden popis driverů pro navržený systém. Vysvětluje se zde princip vývoje driverů na desce PiMXpřes NFS a samotná architektura driverů vyvinutých v rámci diplomové práce. Tato kapitola obsahuje hlavní část činnosti na této práci. Popsané drivery jsou obsaženy na přiloženém CD.

7.1 Vývoje driverů pro embedded systémy

Pro snazší vývoj embedded systémů s linuxovým jádrem se používá (Network file system) NFS, kde kořenový souborový systém (root filesystem) je umístěn na vzdáleném počítači (serveru) typicky pracovní stanice vývojáře a klient (embedded systém), pro který se vyvíjí driver či aplikace, přistupuje souborům pomocí sítě ethernet. Tak se může přistupovat na souborový systém z klienta i ze serveru. Tím také odpadá nutnost mít souborový systém na cílové zařízení ve FLASH paměti, což značně ztěžuje programování v důsledku neustálého nahrávání jádra do FLASH paměti.

Deska PiMX obsahuje ve své FLASH paměti bootloader (zavaděč), který dokáže zajistit proces bootování po ethernetové síti. Server tak neobsahuje pouze souborový systém, ale obsahuje také předkompilované jádro, které si procesor při bootování nejprve natáhne a vloží ho do operační paměti. Po uložení do paměti je již možné pracovat operačním systémem.

Jelikož nemá procesor i.MX žádný grafický výstup je systémová konzole přesměrována na sériovou linku. Pokud tedy chceme přistupovat do spuštěného jádra musíme se připojit přes terminál sériové linky k cílovému zařízení.

Další způsob, jak pracovat na vzdáleném procesoru i.MX, je přes TCP spojení (SSH,Telnet), které oproti sériové lince rychlejší, ale je zprovozněno až po kompletním nabootování jádra.

Postup jak správně tento proces provést byl zdokumentován v [7].

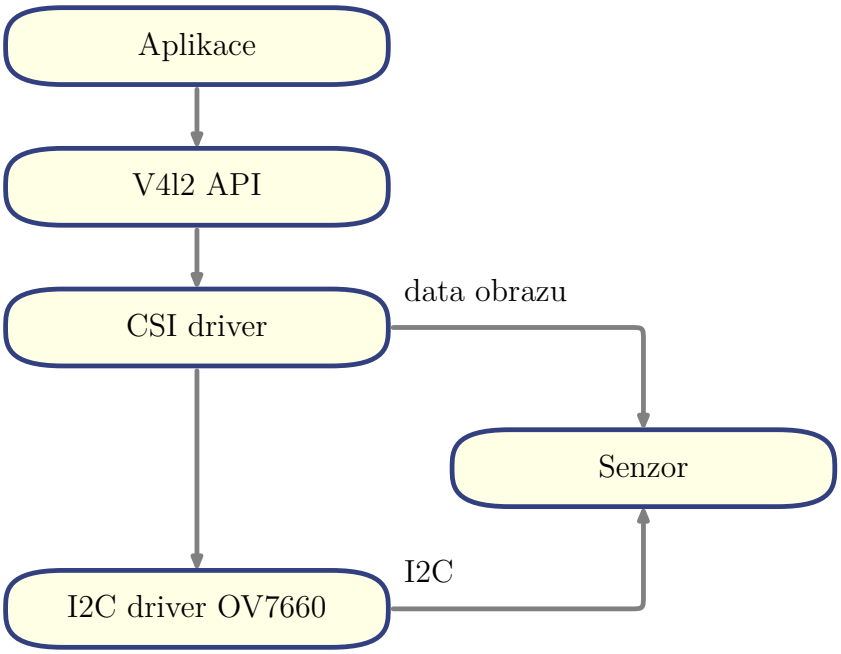
Typicky se překládá aplikaci či driver na serveru a následně se zkopíruje do root filesystemu. Následně se vývojář připojí přes sériovou linku k procesoru a otestuje daná aplikaci či driver.

Vývoj driverů byl prováděn na jádře Linuxu 2.6.24. Aby bylo toto jádro kompatibilní s architekturou ARM a speciálně desku PiMX, je nutné aplikovat na jádro patch, který přizpůsobí jádro pro architekturu i.MX a jsou dostupné z [8]. Před kompilací jádra pro procesor se také musí nastavit konfigurační soubor specifikující jaké části budou obsaženy v jádře. V konfiguračním souboru získaným z [8] není zakomponováno V4L2 API potřebně pro vývoj driveru pro video a podporu I2C sběrnice. Správně nastavený konfigurační soubor s jádrem, na kterém jsou aplikovány všechny patche je na přiloženém CD.

7.2 Architektura driverů

Celý proces ovládání a zachytávání dat je rozděleno do dvou driverů. První driver se stará o komunikaci se senzorem po I2C sběrnici. Druhý driver spravuje zachytávání dat CSI modulu a předávání dat aplikacím. Tyto dva drivery musí být však spojeny, protože některé vlastnosti je nutné předat jak CSI modulu tak i I2C driveru. Dobrý příkladem

je určitě nastavování velikosti obrazu. Požadavek na formát přijímá CSI driver, protože musí alokovat buffery odpovídající velikosti. Aby však dokázal zajistit změnu formátu zavolá funkci I2C driveru, která uskuteční změnu formátu. Pakliže se podaří nastavit formát, umožní se CSI driveru potvrdit změnu formátu aplikaci. Tato struktura lze vyjádřit schématem na obrázku 7.1



Obrázek 7.1 Struktura driverů

Drivery byly kompilovány jako jaderné moduly, které je možné vložit do jádra operačního systému za běhu. To znamená, že není nutné překompilovat celé jádro a znovu ho bootovat.

Takto zvolené uspořádání driverů zavádí problém při vkládání modulů do jádra. Ze zapojení je zřejmé, že senzor nepracuje bez hodinového signálu, a tak bez hodinového signálu nefunguje ani I2C spojení se senzorem. Hodinový signál může však spustit pouze CSI driver zapsáním do registru. Driver OV7660 musí být vložen do jádra dříve než driver obsluhující CSI, protože CSI driver používá funkce exportované driverem OV7660.

V důsledku toho je nejprve vložen OV7660 driver. Jelikož nejsou však spuštěny hodiny nedojde ke komunikaci s senzorem a I2C driver se nezaregistruje do systému. Po vložení CSI driveru se splní závislosti mezi modulu a spustí se hodiny, ale není navázána komunikace.

Tento problém je vyřešen vyexportováním pomocné funkce z OV7660 modulu, která se pokusí navázat komunikaci. Postup je následující:

1. zavedení modulu OV7660 do jádra
   - chyba komunikace (senzor bez hod signálu)
2. zavedení modulu CSI do jádra

- spuštění hodin
- **žádost o nové navázání komunikace**

7.3 CSI driver

CSI modul je připojen na vnitřní sběrnici procesoru, která není kompatibilní PCI sběrnicí. Proto takové případy je v linuxovém jádře nedefinována imaginární pseudo-sběrnice tzv. *platform-bus*, na které jsou umístěna *platform-device* (zařízení) a k nim odpovídající *platform-driver*. Tyto platform struktury jsou vytvořeny tak, aby byly kompatibilní se strukturou ovladačů. Proto je CSI modul naprogramován jako *platform-device*. Pro každé *platform_device* musí být v jádře vyplněna struktura charakterizující vlastní zařízení. Tato struktura se nachází v hlavičkovém souboru *platform-device.h*. Pro CSI zařízení vypadá vyplněná struktura následovně.

```
static struct platform_device imx_csi_device = {
  .  name= "imx-csi", /* jmeno zarizeni */
  .id = 0, /*identifikacni cislo*/
  /* device struktura */
  .dev = {
     .dma_mask = 0xffffffffUL, /*dma masky*/
    .coherent_dma_mask = 0xffffffff,
  }, /*device struktura*/
  .  num_resources= ARRAY_SIZE(imx_csi_resources),
  .  resource= imx_csi_resources,
};
```

Důležitým parametrem *platform_device* je položka *resource*. Ta obsahuje zdroje patřící zařízení. CSI modul disponuje dvěma typy zdrojů

- paměťovým prostorem pro registry a paměti FIFO,
- přerušením.

Oba zdroje se vyplní do pole resourců a předají struktuře *plaform_device*.

```
static struct resource imx_csi_resources[] = {
  [0] = {
    .start = 0x00224000, /* pocatecni adresa pameti CSI */
    .end = 0x002240FF, /* koncová adresa paměti CSI */
    .flags = IORESOURCE_MEM, /* typ resourcu */
  },
  [1] = {
    .  start= (CSI_INT), /* pro CSI je zdroj přerušení pod číslem 6*/
    .  end= (CSI_INT),
    .  flags= IORESOURCE_IRQ, /*typ resourcu*/
  },
};
```

Nadefinování platform zařízení se provádí buď dynamicky nebo se definice uvedou v některých inicializační modulech, aby bylo platform zařízení již v jádře při závádění platform driveru. U CSI modulu je definice uveden v souboru *mach-imx/generic.c*.

CSI driver je tedy naprogramován jako platform driver v souboru *csi-imx.c*. Odděleně byl vytvořen hlavičkový soubor *csi-imx.h* obsahující definice adres registrů CSI modulu

společně s maskami bitů jednotlivých registrů, kterými lze nastavovat funkce CSI modulu podle kapitoly 2.

7.3.1 Platform driver

Platform-driver musí mít nadeklarovány dvě základní funkce. První funkce (*probe*), která se pokusí zavést driver na odpovídající platform zařízení, a druhá funkce (*remove*), jež zařídí odebrání driveru ze systému. Struktura *platform_driver* poskytuje také další funkce, které však nejsou pro CSI implementované.

Než přistoupíme k vysvětlení metod *probe* a *remove*, musí být popsány základní struktura CSI driveru, se kterou se pracuje v metodách *probe* či *remove*.

```
struct csi_imx_dev {
  struct video_device *vd; /* video zařízení */
  struct platform_device *pdev; /* platform zařízení */
  struct mutex lock; /* zámek při více uživatelích */
  int users;  /* počet uživatelů */
    imx_dmach_tdma;  /* DMA kanál */
  int dma_allocated; /* příznak alokovaného DMA kanálu */
  struct csi_imx_dmaqueue cidq; /* fronty pro buffery */
  struct resource *res; /* zdroje platform zařízení */
  int irq; /* číslo přerušení */
  void __iomem *base; /* počáteční adresa CSI */
  struct i2c_client *i2c_chip; /* přístup k I2C driveru senzoru */
```

Struktura obsahuje všechny potřebné ukazatele pro práci s CSI zařízením. *Platform_device* pro obsluhu platform zařízení tedy přístupy do registrů atd. Nesmí zde také chybět struktura *video_device*, jež se používá pro V4L2 driver. Další prvky struktury slouží k plnění bufferů pro data až na poslední, kterým je ukazatel na I2C klienta. Přes tento ukazatel může CSI driver požádat I2C driver (viz kapitola 7.4) o komunikaci se senzorem.

7.3.2 Metody *probe* a *remove*

Metodu *probe* volá systém při registraci driveru funkcí *platform_driver_register* a naopak metoda *remove* při *platform_driver_unregister*. Slouží jako zaváděcí či odebírací funkce driveru.

Probe

V této metodě se inicializují všechny prvky struktury *csi_imx_dev*. Nejprve se z ukazatele na *platform_device*, předávané v argumentu metody, dostaneme na resource CSI zařízení nadefinovaný v *generic.c*.

Pro resource na pamět se provede remapování, aby se driver dostal k I/0 adresám, na kterých se nachází registry driveru. Jestliže se remapování provede základní adresa se uloží do proměnné *base* struktury *csi_imx_dev*. Obdobně se vyčte z *platform-device* resource přerušení, pro který se následně zaregistruje handler (obslužná funkce) pro jeho obsluhu.

Pokud úvodní sekvence proběhne v pořádku, pokouší se driver alokovat výstupní piny procesoru pro užití CSI modulu a také nastaví do počáteční podmínky registry CSI modulu viz kapitola 2.

Registry jsou nastaveny v tabulce 7.1[14]. Úrovně FIFO pamětí jsou zvoleny tak, aby systém zvládal osluhovat DMA přenosy bez ztráty dat. Také dělička hodin jdoucích k senzoru je nastavena s ohledem na tuto podmínku. Ostatní parametry jsou přizpůsobeny zvolenému senzoru OV7660.

Bity registru CSI_CR1	Nastavená hodota (hex)
EN	1
MCLK_DIV	8
RXFF_LEVEL	1
REDGE	1
GCLK_MODE	1
MCLKEN	1
SOF_POL	1

Tabulka 7.1 Nastavení registru *CSI_CR1*

Dále se žádá o DMA kanál (viz kapitola 5), přes který se budou přenášet data v průběhu zachytávání, a pro tento kanál se zaregistrují handlery oznamující konec DMA přenosu a nebo chybu při přenosu.

Dalším krokem je registrace video zařízení metodou *video_device_register*, v níž se spojí CSI driver s V4L2 vrstvou. Hlavní parametrem funkce je struktura *video_device* zmíněná v kapitole 4.1.

V posledním krokem se vytvoří soubor ve adresáři */proc/csi/stat*, ze kterého lze z uživatelského prostoru vyčíst některé aktuální údaje o CSI driveru.

Když se některý z těchto kroků nepovede, celá inicializace je provedena špatně a nedojde ke startu driveru s tím, že se uvolní všechny dosud alokované zdroje a paměťové prostory. Celý proces metody *probe* je na obrázku 7.2.

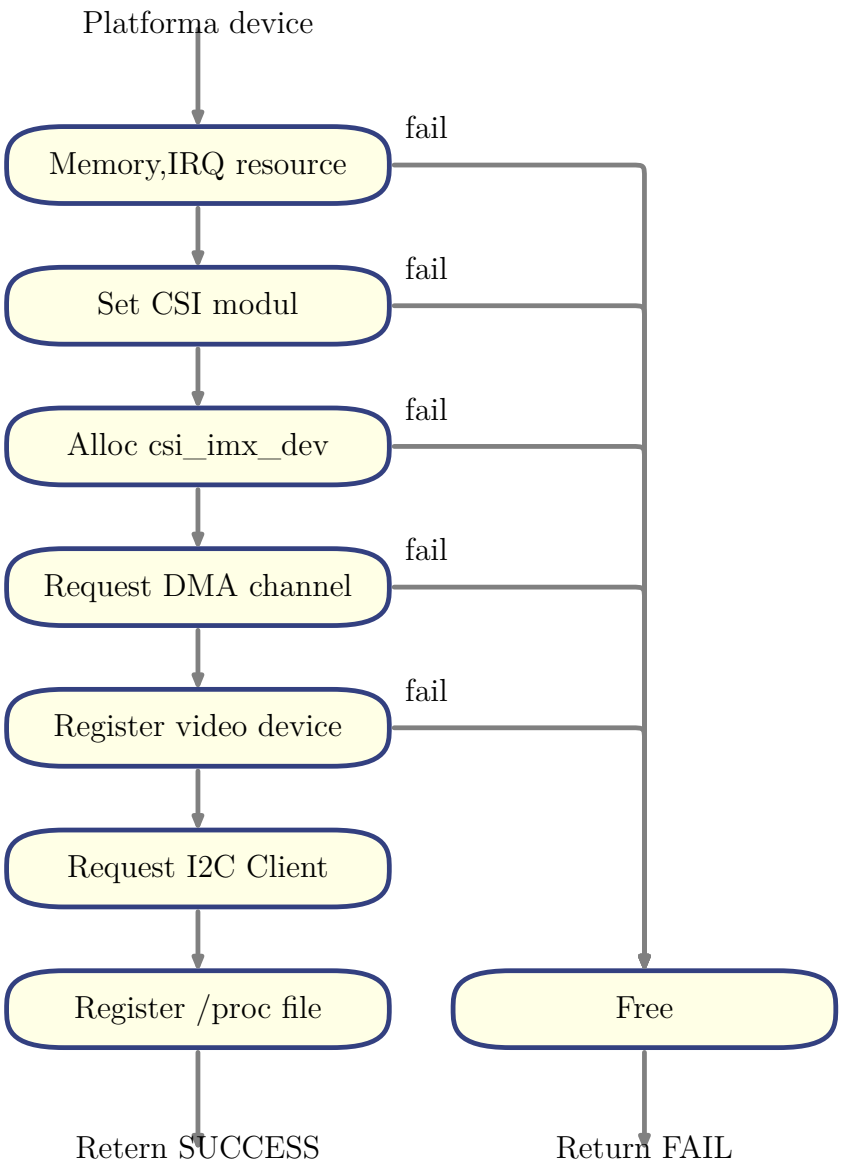
Metoda *remove* naopak uvolní všechny zdroje, které si driver zarezervoval v průběhu metody *probe*. Aplikuje opačný postup oproti *probe*.

7.3.3 Funkce *video_device*

Ihned po úspěšném provedení funkce *video_device_register* lze přistupovat k driveru přes soubor v adresáři */dev/videoX*, kde místo X je minor číslo zařízení. Právě proto, že se z uživatelského přistupuje k zařízení přes soubor, definuje se pro *video_device* struktura *file_operations*, která se předá video zařízení. V těchto metodách pro práci se souborem je naprogramováno chování driveru při volání z aplikace. Vysvětlení jednotlivých metod je v kapitole 4.1.

CSI driver využívá ke své činnosti *videobuf* modul popsaný v kapitole 4.4. Díky němu není nutné implementovat některé souborové funkce týkající se předání zachycených dat do bufferů. *Videobuf* k nim totiž poskytuje alternativy

[14]Bity, které nejsou uvedeny v tabulce jsou vynulovány.



Obrázek 7.2 Metoda *probe* CSI driveru

- *mmap* - funkce *videobuf_mmap_mapper*,
- *poll* - funkce *videobuf_poll_stream* a
- *read* - funkce *videobuf_read_stream*.

Ostatní souborové metody je však nutné implementovat.

Metoda *open* vykonává otevření souboru inicializuje *videobuf* a vytvoří tak řídící strukturu *csi_imx_fh* (filehandler), která se předává do dalších funkcí. A obsahuje informace o stavu otevřeného video zařízení.

```
struct csi_imx_fh {
  struct csi_imx_dev *cidev; /* csi_imx_dev zařízení */
  unsigned int width;   /* šířka obrázku */
  unsigned int height; /* výška obrázku */
  unsigned int fourcc; /* v4l2 id formátu */
  struct videobuf_queue vb_vidq; /* videobuf fronta */
  enum   v4l2_buf_typetype; /* typ bufferů */
};
```

Ioctl volání, které se vztahují k bufferům jsou také naprogramovány z funkcí *videobuf* modulu. Mezi ně patří například *ioctl* volání *VIDIOC_REQ*, *VIDIOC_QBUF*, *VIDIOC_DQBUF* ad.

Na zbylé *ioctl* metody týkající se senzoru (formát, velikost obrazu,...) je použito funkcí I2C driveru senzoru OV7660.

Jak už bylo zmíněno v kapitole 5 DMA přenosy vyžadují znalost scatter-gather listu paměťové oblasti, do níž se data ukládají. Struktura *videobuf* neobsahuje prostor, kde by se scatter-gather list dal uložit. Z tohoto důvodu CSI driver definuje vlastní strukturu pro video buffery.

```
struct csi_imx_buffer {
  struct videobuf_buffer vb; /*původní videobuf*/
  struct * sg_list; /*scatter-gather list*/
  int sg_len; /*délka sg-listu*/
};
```

Při alokaci bufferu se vykoná převod na scatter-gather list uvedený v 5 a uloží se do struktury *csi_imx_buffer*. Každý buffer vlastní svůj scatter-gather list, používaný při DMA přenosech.

7.3.4 Proces zachytávání dat v CSI driveru

Po té co aplikace požádá driver o alokování bufferu, driver si připraví buffery a nastaví jejich stav na *NEEDS_INIT*. Tím se však ještě nealokovalo paměťové místo pro data. To nastane až při požadavku na zařazení do fronty. Driver rezervuje paměť pro obrazová data a zároveň se vytvoří scatter-gather list pomocí funkce *vm_to_sg*, protože paměťový prostor je alokován funkcí *vmalloc*. Buffer změní stav na *PREPARED*. Pak se čeká pouze na *ioctl* příkaz aplikace k začátku spuštění streamu.

Protože bylo složité detekovat jaké buffery jsou momentálně připraveny k plnění a které jsou již naplněny, CSI driver si vytváří dvě fronty inicializované v metodě *probe*.

1. fronta *queued* - buffery zařazeny do fronty - jsou v ní zařazeny buffery všechny buffery, které nejsou naplněny nebo které již byly naplněny ale aplikace je již přečetla. Stav bufferů je nastaven jako *QUEUED*.

2. fronta *done* - buffery již naplněné - jsou v ní buffery, které byly naplněny a čekají na přečtení aplikací. Stav bufferů v této forntě je nastaven na *DONE*.

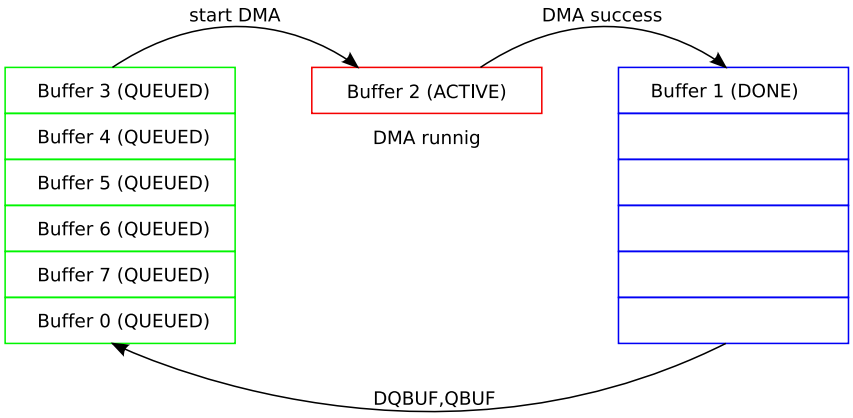
Když se poprvé spustí streamování všechny buffery jsou umístěny do fronty *queued*.

Zachycení snímku

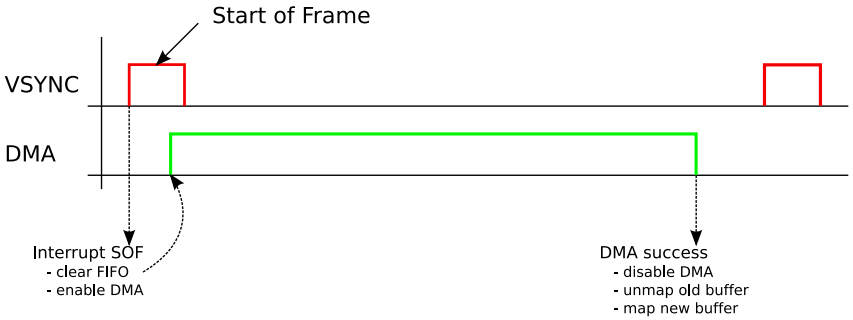
Jakmile se přidá první buffer do fronty *queued* naplánuje se první zachycení snímku. Namapuje se scatter-gather list ze struktury odpovídajícího buffer, nastaví se parametry DMA přenosu a přepíše se stav bufferu na *ACTIVE*. Pak se povolí přerušení od signálu SOF (start of frame). Když nastane toto přerušení povolí se DMA přenosy. Tak je zaručeno, že každý buffer bude začínat shodně se začátkem snímku.

Když se úspěšně dokončí DMA přenos, spustí se handler ukončení DMA. Ten zakáže DMA přenosy a odmapuje scatter-gather list naplněného bufferu, změní se jeho stav na *DONE* a přesune do fronty *done*. V tom samém handleru se ověří, zda-li není prázdná fronta *queued*, a pokud má alespoň jeden prvek vezme se první buffer v této frontě a připraví se na další DMA přenos.

Tento proces se opakuje dokud není zastaven *ioctl* voláním aplikace o zastavení streamování.



Obrázek 7.3 Přesuny bufferů ve frontách



Obrázek 7.4 Časový průběh zachytávání snímku

7.4 OV7660 driver

Druhou částí navržené architektury je I2C driver pro senzor OV7660, který obstarává nastavení senzoru. Driver implementuje jak V4L2 tak I2C jaderné API. V4L2 musí být v driveru obsaženo kvůli formátům obrazu a *ioctl* voláním. Jak je vidět z obrázku 7.1 driver přijímá *ioctl* volání pro změnu formátu a dalších přes CSI driver. Následně rozhodne, o které *ioctl* volání se jedná a komunikací přes I2C provede změnu. Protože senzor obsahuje velké

množství registrů byl odděleně vytvořen hlavičkový soubor *ov7660.h*, v němž jsou zahrnuty všechny registry i s jejich nastavovacími hodnotami definovanými jako makra.

Základ driver OV7660 tvoří I2C driver, jehož struktura je nadefinovaná následovně.

```
static struct i2c_driver ov7660_driver = {
  .driver = {
    .name = "ov7660", /* jméno */
  },
  .id     = I2C_DRIVERID_OV7670,  /* identifikační číslo */
  .class     = I2C_CLASS_CAM_DIGITAL, /* třída driveru */
  .attach_adapter = ov7660_attach, /* připojení driveru */
  .  detach_client= ov7660_detach, /* odebrání driveru */
  .  command= ov7660_command, /* příkazy driveru */
};
```

7.4.1 Funkce OV7660 driveru

Než bude možné vysvětlit jednotlivé funkce driveru, je nutné ujasnit způsob komunikaci po I2C sběrnici z operačního systému Linux. Komunikaci zajišťuje vrstva *i2c*, která je přítomna v jádře.

Pro komunikaci na sběrnici stačí použít funkce nadefinovanou v souboru */linux/include/linux/i2c.h*. Pro čtení se funkce jmenuje

```
int i2c_master_send(struct i2c_client *client,const char *buf ,int count);
```

a pro zápis

```
int i2c_master_recv(struct i2c_client *client, char *buf ,int count);
```

Parametr *client* je struktura, která je driveru předána při jeho registraci, pointer *buf* ukazuje na odesílaný respektive přijímaný buffer a v argumentu *count* se předává počet znaků.

Protože však vrstva *i2c* používá při čtení repeated start (viz kapitola 3.2), deklaruje si OV7660 driver vlastní funkce pro zápis či čtení.

Funkce pro čtení

```
static int ov7660_read(struct i2c_client *c, unsigned char reg,
    unsigned char *value)
{
  int ret;
  unsigned char tmp;
  ret = i2c_master_send(c,&reg,1); /* odeslání adresy */
  if (ret != 1){
    dprintk(2,"Error send");
  }
  ret = i2c_master_recv(c,&tmp,1); /* přečtení hodnoty */
  if (ret != 1){
    dprintk(2,"Error receive register");
  }
  if (ret >= 0)
    *value = (unsigned char) tmp;
  return ret;
}
```

V této funkci se obchází *repeated start* podmínka, kterou senzor OV7660 neumí akceptovat. Nejprve se odešle adresa registru, který má být změněn, funkcí pro zápis a následně se funkcí pro čtení vyčte hodnotudaného registru.

Funkce pro zápis

```
static int ov7660_write(struct i2c_client *c, unsigned char reg,
    unsigned char value)
{
  int ret;
  unsigned char tmp[2];
  tmp[0]=reg;
  tmp[1]=value;
  /* odeslání dat */
  ret = i2c_master_send(c,tmp,2);
  if (ret >= 0){
    dprintk(2,"Error send");
  }
  return ret;
}
```

U funkce pro zápis nenastává problém s repeated start, proto lze pomocí funkce *i2c_master_send* odeslat jak číslo registru, tak i jeho hodnotu.

Obě funkce čtení i zápis jsou blokující. To znamená, že nelze provádět jinou operaci, jestliže nebyla dokončena tato funkce.

7.4.1.1 Funkce *ov7660_attach*

Funkce *attach_addapter* se volá při zavádění driveru do systému. Jejím úkolem je detekovat senzor OV7660 na sběrnici. K tomu slouží adresa zařízení, která je určena výrobcem. Pro senzor OV7660je adresa 42(hex).

Každý senzor obsahuje několik registrů, které jednoznačně určují I2C zařízení. Pro senzor OV7660 jsou to registry zmíněné v kapitole 3 registry *PID*, *VER*, *MIDH*, *MIDL*. Driver se pokusí z této adresy přečíst údaje identifikující OV7660 senzor a pokud souhlasí s nadefinovanými údaji je zajištěno, že se jedná daný senzor OV7660 a driver může být přidán do systému.

Celá metoda *attach* probíhá dle vývojového diagramu na obrázku 7.5.

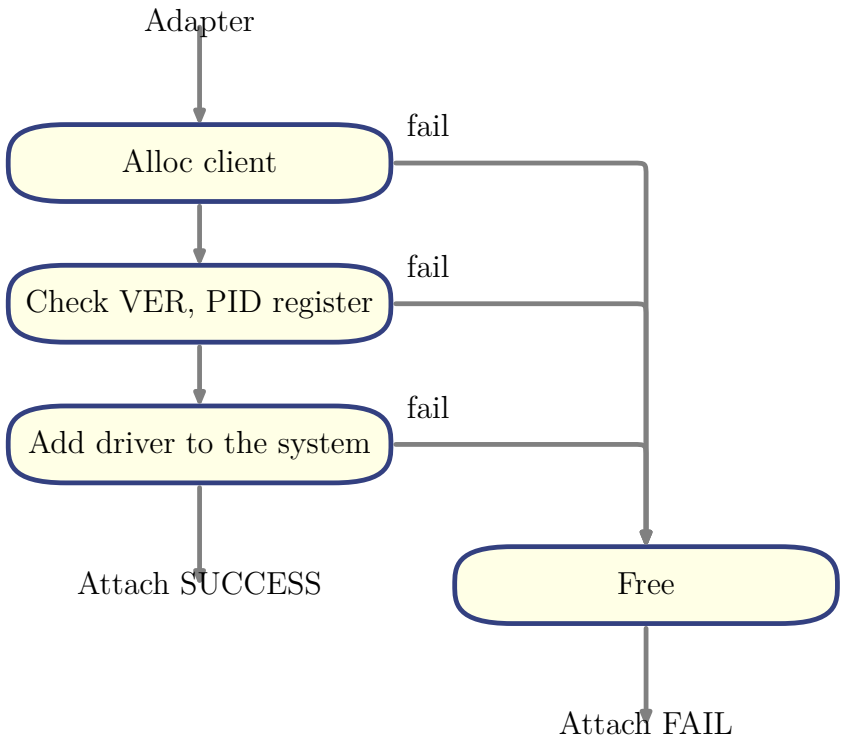
7.4.1.2 Funkce *ov7660_detach*

Metoda odebere driver ze systému a uvolní paměť po alokovaných strukturách driveru.

7.4.1.3 Funkce *ov7660_command*

Tato metoda je volána, je-li požadována komunikace se senzorem týkající se V4L2 *ioctl* funkcí. Ve funkci je implementován příkaz *switch*, který roztřiďuje jednotlivá V4L2 volání a podle daného identifikátoru určuje jaká funkce OV7660 driveru se bude provádět.

Driver podporuje pouze *ioctl* volání související s vlastnostmi senzoru, které lze nastavit



Obrázek 7.5 Vývojový diagram metody *ov7660_attach*

- *VIDIOC_ENUM_FMT*
- *VIDIOC_TRY_FMT*
- *VIDIOC_S_FMT*
- *VIDIOC_QUERYCTRL*
- *VIDIOC_S_CTRL*
- *VIDIOC_G_CTRL*
- *VIDIOC_S_PARM*
- *VIDIOC_G_PARM*

Ioctl volání jsou popsané v kapitole 4.1.

7.4.2 Principy nastavování registrů senzoru

Každá změna v nastavení senzoru vyžaduje zásah do nějakého z registrů. Aby bylo zřejmé jaký registr nastavit na správnou hodnotu je definována struktura

```
struct regval_list {
  unsigned char reg_num; /*číslo registru*/
  unsigned char value; /*hodnota registru*/
};
```

Pro většinu případů se musí změnit více jak jeden registr, proto se deklarují pole těchto struktur.

7.4.2.1 Změna formátu dat obrazu

Formáty výstupních dat jsou definovány jako pole

```
/* pole pro formát YUV422 */
static struct regval_list ov7660_fmt_yuv422[];
/* pole pro formát RGB565 */
static struct regval_list ov7660_fmt_rgb565[];
/* pole pro formát RGB555 */
static struct regval_list ov7660_fmt_rgb555[];
/* pole pro formát Bayer formát */
static struct regval_list ov7660_fmt_raw[];
```

kde každé pole obsahuje svou vlastní sadu registrů s jejich nastavovacími hodnotami.

Změna formátu je pak jednoduchá. Ověří se, zda daný formát podporuje senzor a odešlou se registry i s hodnotami po I2C sběrnici.

7.4.2.2 Změna rozlišení obrazu

Odlišná je situace pro změnu rozlišení snímaného obrazu (viz kapitola 3.4). Pro změnu rozlišení používá senzor pouze pět registrů

- *COM7* - nastavení rozlišení
- *VSTART* - první sloupec, který bude vzorkován do obrazu
- *VSTOP* - poslední sloupec, který bude vzorkován
- *HSTART* - první řádek, který bude zahrnut do obrazu
- *HSTOP* - poslední řádek, který bude zahrnut do obrazu,

a tak každé rozlišení deklaruje strukturu

```
static struct ov7660_win_size {
  int width;  /* šířka obrazu */
  int height;  /* výška obrazu */
  unsigned char com7_bit;  /* nastavovací bit v registru COM7 */
  int hstart;  /* start horizontálních hodnot */
  int hstop;  /* stop horizontálních hodnot */
  int vstart;  /* start vertikálních hodnot */
  int vstop;  /* stop vertikálních hodnot */
  struct regval_list *regs; /* pomocné nastavovací registry */
}
```

Pro každé rozlišení, který je podporován senzorem OV7660, je nadeklarována tato struktura s odpovídajícími hodnotami registrů.

O změnu rozlišení se stará driver při stejném *ioctl* volání jako změna formátu. Ve funkci se zjistí daný typ rozlišení a pokud se nalezne odpovídající formát v driveru, je nastaven do senzoru odesláním registrů s hodnotami po sběrnici I2C.

7.4.2.3 Exportované funkce driver OV7660

Jak bylo navrženo v již v architektuře, driver OV7660 poskytuje funkce pro nadřazený CSI driver. Aby by se dalo využít funkce I2C driveru, musely být z něho vyexportovány funkce, které má CSI driver zahrnuty ve svých definicích z *ov7660.h*.

Driver OV7660 se dostává ke svým funkcím přes strukturu *i2c_client*, která se inicializuje v metodě *attach*. Aby mohl nadřazený driver CSI volat funkce tohoto I2C driveru, měl by znát právě tuto strukturu. Pro tento účel byla vytvořena funkce, která zaručí předání struktury *i2c_client*. Funkce se nazývá *ov7660_get_i2c_client*. Pokud se provede úspěšně metoda *attach* je *i2c_client* uložen do globální proměnné a při volání této funkce se pouze předá ukazatel na globální proměnnou. Pokud se však stane, že klient ještě nebyl inicializován požádá funkce o nové provedení metody *attach*, a tím i nový pokus o alokování klienta. Pokud je toto úspěšné předá se ukazatel na nově vytvořeného klienta. Tato složitá inicializace musela být vytvořena z důvodu problému při nastavení hardwaru CSI modulu (viz kapitola 7.2)

```
struct i2c_client * ov7660_get_i2c_client(void)
{
  struct i2c_adapter *adp;
  adp=i2c_get_adapter(0);
  if(!global_client){
    printk(KERN_INFO"I2C client driver OV7660");
    if(ov7660_driver.attach_adapter(adp)){
      dprintk(1,"error attach driver");
      return NULL;
    }
  }
  return global_client;
}
```

Jestliže již CSI driver vlastní ukazatel na strukturu *i2c_client* může použít další exportované funkce driveru OV7660, která zaručí zavolání funkce *command* starající se o *ioctl* volání.

```
int ov7660_v4l2_ioctl(struct i2c_client *client, unsigned int cmd, void *arg)
{
  return ov7660_command(client,cmd,arg);
}
```

Ta pouze provede komunikaci po I2C sběrnice a pokud se povede nastavit odpovídající registry vrátí návratovou hodnotu nadřazenému CSI driveru.

8 Aplikace pro zpracování videa

Tato kapitola uvádí příklad aplikace napsané pro zachytávání dat z navrženého systému. Popisuje základní principy inicializace zařízení z uživatelského prostoru, vlastní zachytávání obrazu a ukončovací sekvenci. Jako příklad byl vybrán zachytávání obrazu do souboru. Výsledné snímky obrazu jsou zobrazeny v závěru kapitoly.

8.1 Aplikace pro zachycení obrazu

Po úspěšném zavedení driveru CSI do systému se v adresáři */dev* objeví soubor představující dané video zařízení (např. */dev/video1*). Pokud chce aplikace používat toto zařízení musí k němu přistupovat přes souborové operace. Aplikace používá V4L2 API tak, že zahrnuje definice z hlavičkového souboru *linux/videodev2.h*. Přes tento soubor smí aplikace volat jednotlivé identifikátory *ioctl* volání.

Aplikace se dá rozdělit do několika základních částí prováděné postupně za sebou.

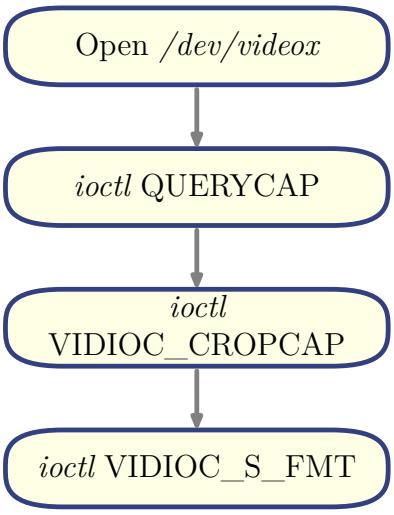
1. otevření zařízení
2. inicializace zařízení
3. nastavení čtecí metody
4. spuštění streamu
5. zpracování dat
6. zastavení streamu
7. zavření zařízení

Aplikace je naprogramována obecně pro všechny tři druhy čtení dat ze zařízení (*mmap, userpointer, read*), ale protože CSI driver má implementovanou metodu *mmap* bude vysvětlena pouze tato metoda.

8.1.1 Otevření a inicializace

Pro všechny metody přístupu k datům však platí inicializační princip pro V4L2. Tento princip spočívá v základní inicializaci V4L2 zařízení, která je na obrázku 8.1

Před začátkem práce s video zařízením se nejprve otevře operací souborovou funkcí *open*. Jestliže zařízení funguje vrátí funkce filedescriptor, který si aplikace uchová pro další funkce. K zjištění o jaké zařízení se jedná se používá *ioctl* volání *VIDIO_QUERYCAP*. Když aplikace porovná vrácenou hodnotu z tohoto volání a ujistí se, že dané zařízení umožňuje zaznamenávat obraz, pokusí se udělat základní nastavení formátu *VIDIO_TRY_FMT*. Jestliže se povedou i tyto *ioctl* funkce je aplikace schopna přejít k nastavení čtení dat.



Obrázek 8.1 Vývojový diagram inicializace video zařízení

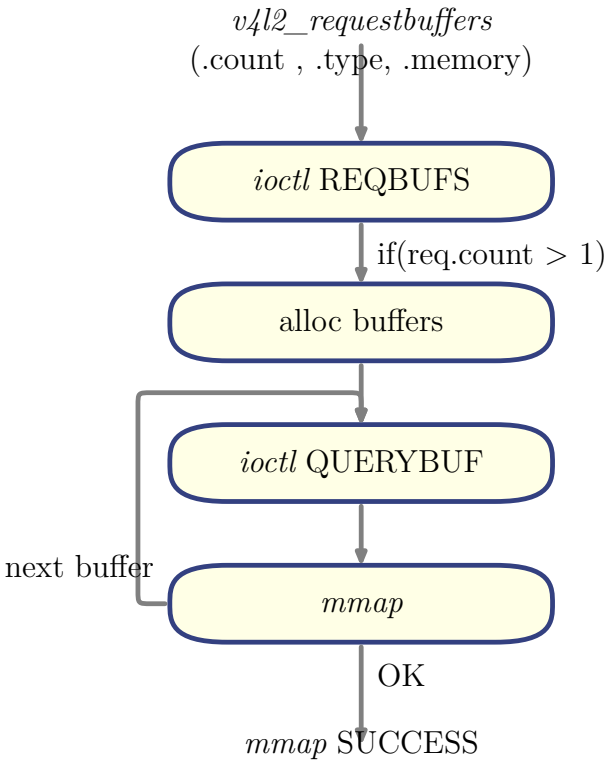
8.1.2 Nastavení čtení pro *mmap*

Pro tento druh čtení se připraví buffery *ioctl* voláním. Na začátek se vyplní struktura *v4l2_requestbuffers*, která předává požadavky na buffery driveru, v položkách *count*(10) pro požadovaný počet bufferů, *.type* na *V4L2_BUF_TYPE_VIDEO_CAPTURE* pro typ bufferů a *memory* na *V4L2_MEMORY_MMAP* pro typ paměti.

S touto strukturou v argumentu se zavolá *ioctl VIODIO_REQBUF*. Protože driver může počet driveru změnit v důsledku nedostatku paměti, musí být zkontrolováno, jestli jsou připraveny minimálně dva buffery. Tento údaj je obsažen v struktuře *v4l2_requestbuffers* po provedení *ioctl*. Pokud tedy jsou alespoň dva buffery připraveny přikročí se k alokování struktury *buffer* reprezentující protějšek k bufferů alokovaným v jádře. Jejich počet je shodný s počtem bufferů v jádře. Následně se v cyklu pro všechny provádí *ioctl* volání *VIDIO_QUERYBUF* a namapování bufferů z uživatelského prostoru na buffery z jádra pomocí funkce *mmap*. Celá příprava čtení je znázorněna na obrázku 8.2.

```
for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
  struct v4l2_buffer buf;
  CLEAR (buf);
  /* určení typu bufferu */
  buf.type        = V4L2_BUF_TYPE_VIDEO_CAPTURE;
  buf.memory      = V4L2_MEMORY_MMAP;
  buf.index       = n_buffers;
  /*Dotaz na stav bufferu v jádře*/
  if (-1 == xioctl (fd, VIDIOC_QUERYBUF, &buf)){
    errno_exit ("VIDIOC_QUERYBUF"); printf("error Vidioc_query");
  }
  /* nastaveni velikosti buffer */
  buffers[n_buffers].length = buf.length;
```

```
  /* namapování bufferu */
  buffers[n_buffers].start = mmap (NULL /* start anywhere */,
      buf.length, PROT_READ | PROT_WRITE /* required */,
      MAP_SHARED /* recommended */,
      fd, buf.m.offset);
  /* test úspěšnosti provedení funkce */
  if (MAP_FAILED == buffers[n_buffers].start){
    errno_exit ("mmap"); printf("Chyba mmap ");
  }
}
```



Obrázek 8.2 Vývojový diagram mapování bufferů

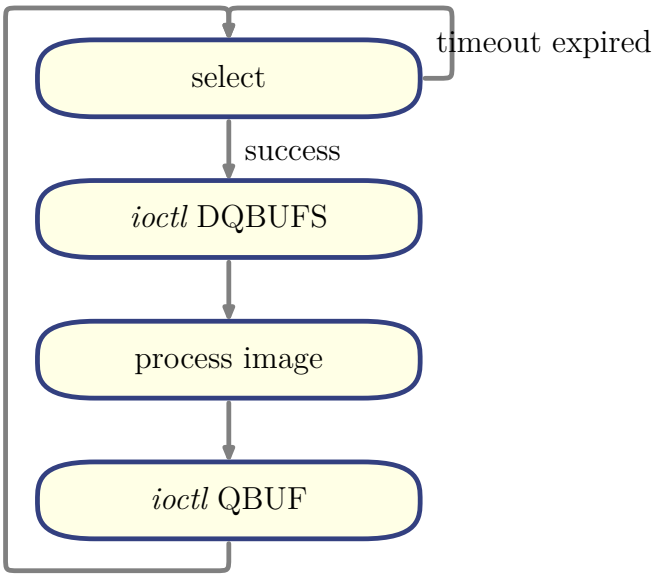
8.1.3 Zachytávání dat

Před vyčítáním obrazových dat z driveru se ještě musí provést dvě *ioctl* volání, bez kterých by zachytávání obrazu nefungovalo. Jako první je nutné přidat všechny buffery do fronty pomocí volání *VIDIOC_QBUF*. To se opět provádí ve smyčce pro všechny buffery. Pak už stačí zavolat poslední *ioctl* funkci *VIDIOC_STREAMON*, která spustí zachytávání. Ihned po úspěšném provedení posledního volání je schopna aplikace přijímat data.

V hlavní smyčce na aplikace se provádí vyčítání snímků z driveru. Driver implementuje *polling* na buffer, proto lze použít metodu *select*, která čeká na dokončení operace (naplnění bufferu). Do metody *select* se přidává časový interval, do kterého musí být dokončena. Tím je zaručeno, že čtení nebude blokující.

Jakmile driver naplní nějaký buffer metoda *select* vrátí kladnou hodnotu a aplikace se dozví, že si může přečíst naplněný buffer.

Čtení bufferu se děje podle scénáře vysvětleného v kapitole 4.3.2 a obrázku 8.3.



Obrázek 8.3 Vývojový diagram čtení bufferů

8.1.4 Zpracování obrazu

V části *procces data* smí aplikace provádět jakékoliv operace nad obrazovými daty. Pro mobilní roboty se používá například hledání hran a další. Ukázková aplikace naprogramovaná pro testování celého zaznamenává obraz a ukládá jej do souboru.

Výsledky zachytávání obrazu jsou zobrazeny v kapitole 8.2.

8.1.5 Ukončení aplikace

Před ukončením celé aplikace by měly být správně provedeny tyto kroky

1. *ioctl* volání *VIDIOC_STREAMOFF*
2. odmapování bufferů funkcí *unmap*
3. uvolnění bufferů v aplikaci
4. zavření filedescriptoru zařízení

až po té je možné aplikaci regulérně ukončit.

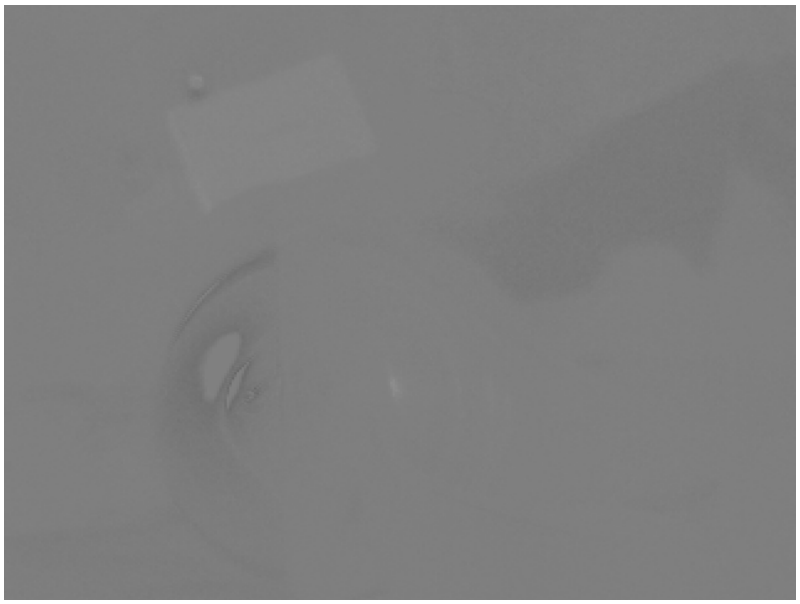
8.2 Výsledky zaznamenávání obrazu

Pro otestování celého systému byl zvolen formát YUV422 a rozlišení 640x480 pixelů. Nevýhodou tohoto formáty je, že se musí přepočítávat na RGB.

Nasnímaný snímek je na obrázku 8.7.

Obrázek 8.4 Jasová složka obrazu Y

Obrázek 8.5 Složka obrazu U



Obrázek 8.6 Složka obrazu V

Obrázek 8.7 Výsledný snímaný obraz v RGB

9 Závěr

Při testování zachytávání dat nastávaly problémy s tím, že se při velkých snímkovacích frekvencích přícházela data ze senzoru příliš rychle a DMA řadič nestíhal vyprazdňovat FIFO paměť, a tak docházelo k přetečení a ztratě těchto dat. To však mělo za následek znehodnocení celého snímku.

Proto se musela být zvolena snímkovací frekvence obrazu tak, aby operační systém dokázal zvládat přenosy obrazových dat. V konečném řešení se podařilo zachytávat obrázky s frekvencí 1 snímek za sekundu pro maximální rozlišení 640x480. I přesto se občas vyskytne nějaký špatný snímek.

Bylo by zajímavé vyvinutý embedded systém aplikovat do nějaké aplikace, která by opravdu řídila reálného mobilního robota. Takový robot by mohl například z obrazu vypočítávat směr pohybu nebo zjišťovat jiná obrazová data pro svou činnost.

V průběhu vývoje této práce se vyskytl v novém jádře operačního systému Linux (2.6.30) driver, který se stará o zachytávání videa na procesoru i.MX (*mx1-camera.c*), který odpovídá CSI driveru vyvíjenému v rámci této diplomové práce.

Pokud bychom porovnávali navržený driver s driverem v novém jádře, zjistili bychom, že v novém jádře se vyskytuje pozměněné V4L2 rozhraní. Tedy i driver je odlišný. Navíc v novém jádře je podpora rozhraní, které je náhradou za *videobuf* a poskytuje scatter-gather list. Autor driveru také používá pro přerušení od SOF rychlé přerušení, což umožňujší rychlejší reakci.

Co se týče zachytávání dat je podstata stejná jako ve vyvíjeném driveru. Vytváří se opět fronty bufferů, které se plní a předávají postupně aplikaci. Rozdíl je však v předávání dat. Driver v novém driveru používá *userpointer* oproti *mmap*.

Odlišností v *mx1_camera.c* je použití rozhraní *soc_camera*, které usnadňuje práci s programováním V4L2 driverů a také zahrnuje přístup na sběrnici I2C pro komunikaci s obrazovým senzorem.

Pro další vývoj by bylo zajímavé použít nový driver v jádře 2.6.30 na navržený hardware se senzorem OV7660. To by ale znamenalo naportovat nové jádro na desku PiMX.

10 Reference

[1] Pikron s.r.o. *Deska PiMX*[online]. Poslední revize (11.12.2004) [cit.20.4.2009]. http://www.pikron.com/

[2] J. Corbet, G. Kroah-Hartman and A. Rubini. *Linux device drivers, 3rd edition*. 2005.

[3] Freescale. *i.MX Reference manuál*[online]. Poslední revize (8.3.2007) [cit.20.4.2009]. http://www.freescale.com

[4] OmniVision Technology. *Datasheet OV7660*[online]. Poslední revize (24.1.2004) [cit.20.4.2009]. http://www.ovt.com

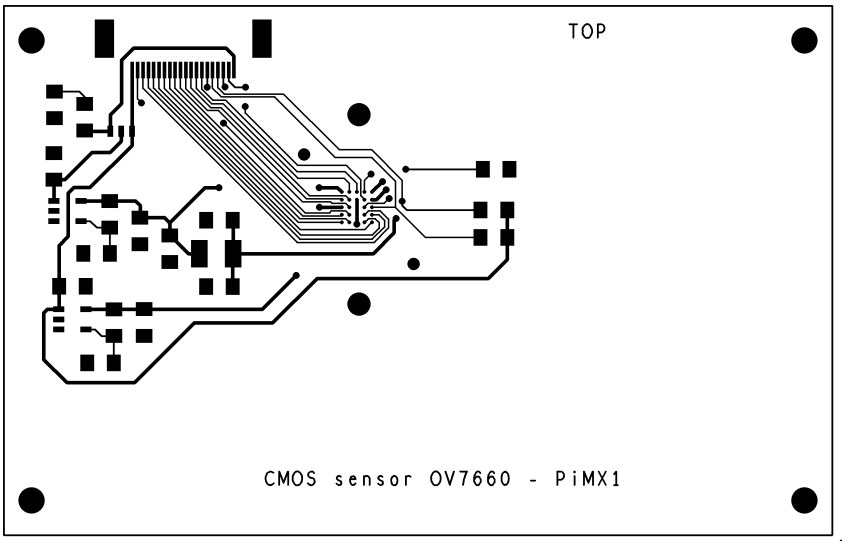
[5] M. Shimek, B. Dirks, H. Verkuil and M. Rubli. *Video for linux Two Specifications*[online]. Poslední revize (2008) [cit.20.4.2009]. http://v4l2spec.bytesex.org/spec/

[6] Texas Instruments. *Datasheet TPS76201*[online]. Poslední revize (1.2.2001) [cit.20.4.2009]. http://www.ti.com

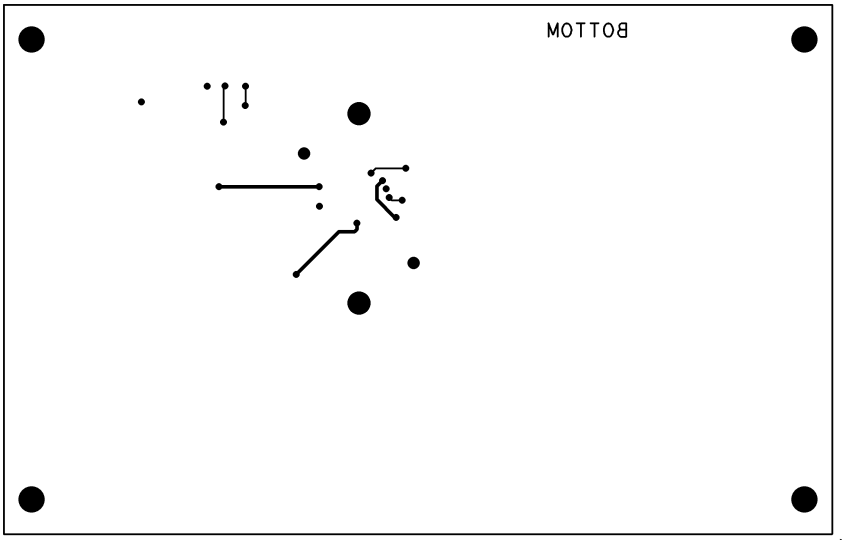
[7] A. Zikmund. *PiMX*[online]. Poslední revize (9.11.2008) [cit.20.4.2009]. http://rtime.felk.cvut.cz/hw/index.php/PiMX1

[8] P. Píša. *Patches*[online]. Poslední revize (3.8.2006) [cit.20.4.2009]. http://rtime.felk.cvut.cz/repos/ppisa-linux-devel

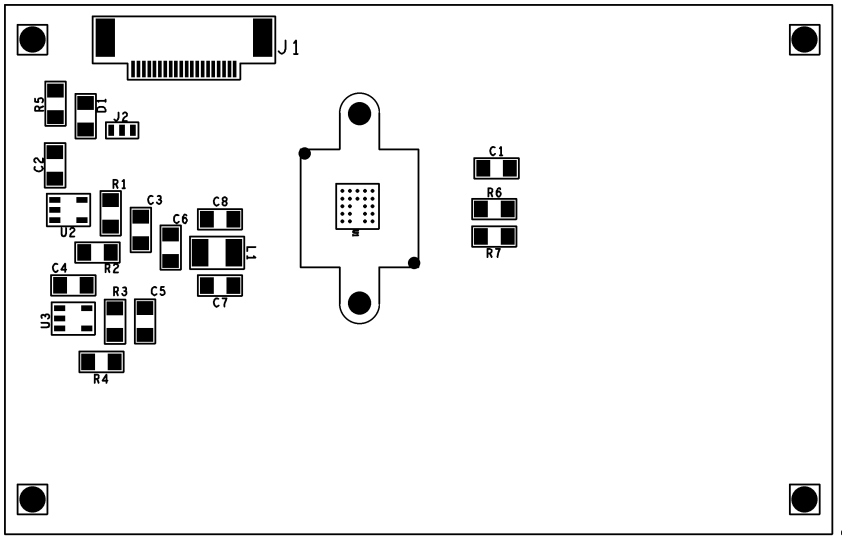
Příloha A Plošný spoj pro obrazový senzor



Horní vrstva



Spodní vrstva

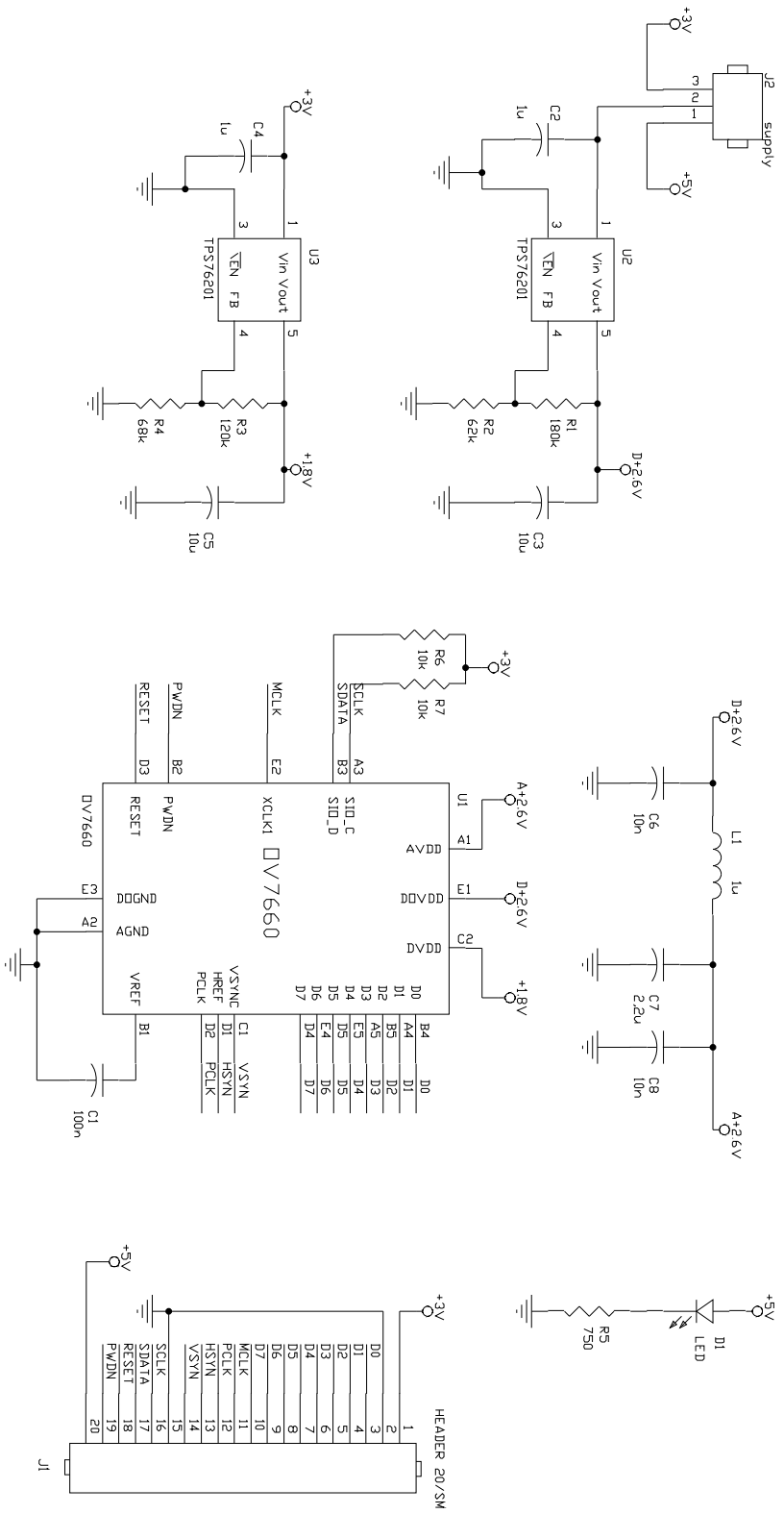


Osazovací výkres

C1	ker.	SMD0805	100nF
C2	ker.	SMD0805	1uF
C3	ker.	SMD0805	10uF
C4	ker.	SMD0805	1uF
C5	ker.	SMD0805	10uF
C6	ker.	SMD0805	10n
C7	ker.	SMD0805	2,2uF
C8	ker.	SMD0805	10nF
D1		SMD0805	LED dioda RED
R1		SMD0805	180kOhm
R2		SMD0805	62kOhm
R3		SMD0805	120kOhm
R4		SMD0805	68kOhm
R5		SMD0805	750Ohm
R6		SMD0805	10kOhm
R7		SMD0805	10kOhm
L1		SMD1210	1,5uH
U1			OV7660
U2,U3			TPS76201
J1 konektor MOLEX			0528922096

Hodnoty součástek

Příloha B Schéma zapojení hardwaru senzoru



Příloha C Obsah CD

- *linux* - linuxové jádro 2.6.24, patche pro PiMX, root filesystem
- *hardware* - návrh plošného spoje - adresáře ORCAD
- *drivers* - obsahuje naprogramované drivery pro CSI a OV7660 + makefile
- *appl* - aplikace pro zachytávání dat + makefile
- *toolchain* - vývojový řetězec pro i.MX
- *picture* - obrázky
- *datasheets* - manuály k elektronikcým součástkám
- *diplomovaprace.pdf* - vlastní diplomová práce