Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství

Bakalářská práce

Softwarové řešení pro quadrokoptéru s řídící jednotkou Raspberry Pi

Jiří Kukačka

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

15. května 2014

Poděkování

Chtěl bych poděkovat Ing. Pavlu Kubalíkovi za zapůjčení hardware potřebného pro vývoj aplikace, rodině a přátelům za podporu v průběhu studia a svým vedoucím v práci za vstřícnost ve studijních záležitostech.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2014 .....

Odkaz na tuto práci

Kukačka, Jiří. *Softwarové řešení pro quadrokoptéru s řídící jednotkou Raspberry Pi*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

Abstrakt

Cílem této práce je zvolit vhodný operační systém a připravit základní knihovnu funkcí potřebných pro let kvadrokoptéry a analyzovat možnosti rozšíření tohoto modelu o letové řízení pomocí AI a dalších různých pluginů.

Klíčová slova kvadrokoptéra, RTOS, AI, letové řízení

Abstract

The aim of this thesis is to choose suitable operating system and prepare basic library required for quadcopter flight and to analyse possibilities of extending this model by flight control AI and various other plugins.

Keywords quadcopter, RTOS, AI, flight control

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Kvadrokoptéra [2], jinak též multirotorová helikoptéra, kvadrotorová helikoptéra nebo jednoduše kvadrotor je varianta letadla, která v nedávné době začala velice získávat na popularitě a velkém rozvoji. Jedná se o jistou obdobu vrtulníku ve tvaru písmene X, na jehož vrcholech jsou umístěny rotory a v jehož středu je řídící jednotka.

Při porovnání kvadrokoptéry proti klasickému vrtulníku je kvadrokoptéra stabilnější (Nejen z pohledu vlastních otřesů modelu, ale i z pohledu schopnosti udržovat statické místo ve vzduchu. Díky této vlastnosti je kvadrokoptéra velice často využívána jako nosič záznamových zařízení i pro profesionální kamerové záběry a většina dnes dostupných modelů kvadrokoptér disponuje alespoň základním záznamovým zařízením.), dokáže rychleji a pružněji reagovat na pokyny změny směru (Vrtulník se musí otočit za pomoci ocasního rotoru v ose Z, kvadroptéra jen mění úhel natočení v osách X a Y zrychlením či zpomalením jednoho nebo dvou rotorů.) a oproti vrtulníku dokáže kvadrokoptéra provádět manévry jako jsou loopingy nebo backflipy. Další velmi ceněnou vlastností kvadrokoptéry je výrazně lepší crash-recovery, tedy oprava modelu po pádu (Dle zkušeností modelářů trvá vrtulník po havárii opravit týden, kvadrokoptéry den až dva – zpravidla výměnou ramene za nové, tedy výměny kousku kovu a jednoho rotoru bez větší nutnosti opravovat elektroniku.).

Kvadrokoptéry v porovnání s helikoptérami mají však i negativa. Největším je výrazně vyšší spotřeba, protože kvadrokoptéru udržují ve vzduchu čtyři rychle rotující malé vrtule, helikoptéru pouze jedna, která má v porovnání s kvadrokoptérou nižší otáčky. Helikoptéry mají také vyšší maximální rychlost a lepší manévrovatelnost v přímém letu.

Přestože mezi modeláři kvadrokoptéry teprve získávají přívržence, jejich samotný koncept byl vynalezen již před více než sto lety. První model kvadrokoptéry totiž vznikl již v roce 1907 a jejím autorem je Louis Breguet. Tento model byl ovšem schopný vznášet se jen několik stop nad zemí.

Roku 1920 kvadrokoptéry začaly slavit první úspěchy, když francouzský

inženýr Etienne Oehmichen sestrojil model, který byl schopný cestovat již z počátku desítky, později stovky metrů. Tyto modely byly ovšem prakticky spíše helikoptérou vybavenou více rotory, na otáčení soudobých kvadrokoptér se využívalo ocasu, stejně jako je tomu u helikoptér.

V roce 1956 vznikl první model kvadrokoptéry bez řídících ocasních rotorů, ovládaný pouze čtyřmi primárními rotory, což je model využívaný do současnosti. Tento koncept využívá stejného směru otáčení rotorů umístěných na stejné ose a současně protisměru vůči ose druhé.

Od té doby kvadrokoptéry neslavily žádné větší úspěchy ani nebyly nijak majoritně využívány. Až v posledním desetiletí opět narůstá jejich význam, a to převážně jako bezpilotních letounů, schopných buď samostatné navigace, nebo asistovaného řízení.

V dnešní době jsou kvadrokoptéry díky svým malým rozměrům převážně řazeny do kategorie UAV [3], ačkoli existuje mnoho pokusů o přepravu posádky.

Cílem této bakalářské práce je vybrat vhodný operační systém a následně implementovat knihovny, které budou umožňovat podporu asistovaného řízení a navigace za pomoci umělé inteligence. Tato práce by měla být základem pro další výzkum a vývoj létacích zařízení, ať již kvadrokoptér nebo vrtulníků, s asistencí AI. Jádro aplikace musí být přizpůsobené požadavkům operačního systému reálného času, aby bylo možné dynamicky reagovat na vzniklé situace, jako je například let proti překážce.

Aby bylo možné realizovat cíle stanovené v odstavci výše, je nutné zvolit vhodný operační systém, nad kterým bude spuštěna aplikace umělé inteligence. Operační systém je nutno zhodnotit z hlediska správy procesů, paměti, ale i přenositelnosti. Primární platforma, pro kterou bude systém vyvíjen, je Raspberry Pi, ale je možné, že tato platforma pro některá možná rozšíření nebude dostačovat, a je tedy potřeba umožnit snadné přenesení aplikace na platformu jinou.

Vzhledem k možným přenosům aplikace mezi hardware je vhodné navrhnout i vhodnou knihovnu pro správu komunikace s periferiemi. Výhodou Raspberry Pi je možnost kombinace I2C sběrnice i sériové linky, základní moduly budou tedy cílené na tyto komunikační kanály, primárně na sériovou linku.

Kvůli faktu, že neexistuje žádný konkrétní model, pro který by se knihovna připravovala, je součástí práce i návrh komunikačního protokolu pro sériovou linku, který by měl případný model dodržovat. Protokol bude poskytovat všechny potřebné udaje pro bezpečný let a případnou automatickou navigaci a současně musí poskytovat dostatečné zabezpečení, aby bylo možné snadno detekovat chyby v reálném čase.

Kapitola 1

Specifikace cíle

Cíle této práce jsou:

1. Vybrat vhodný operační systém.
2. Navrhnout schéma propojení Raspberry Pi s periferiemi.
3. Navrhnout protokol komunikace mezi Raspberry Pi a periferiemi.
4. Navrhnout strukturu jádra programu.
5. Připravit jádro programu.
6. Připravit ukázkovou knihovnu pro komunikaci mezi Raspberry Pi a periferiemi.
7. Analyzovat možné využití algoritmů pro navigaci v prostoru.
8. Vytvořit testy pro realizované algoritmy.

Seznam potřebných periferií nezbytně nutných k realizaci:

1. Gyroskop
2. Řídící deska pro rotory
3. Barometr
4. Čidlo vzdálenosti umístěné na spodek kvadrokoptéry
5. Akcelerometr

Kapitola 2

Rešerše

V současné době existuje velké množství projektů, které se snaží o ovládání či letové řízení vrtulových letounů, ať už se jedná o kvadrokoptéry, jiné multirotory nebo o klasické helikoptéry. Bohužel většina těchto projektů je vystavěna nad jednodeskovým počítačem Arduino nebo nad vlastní deskou osazenou jiným ARM procesorem. Řešení, které by pracovalo pouze s jednodeskovým počítačem Raspberry Pi prakticky neexistuje z důvodu potřeby zpětnovazebního řízení rotorů (nutnost neustálé korekce rotorů, která ač rychlá, zabírala by příliš výpočetního času), a variant, které jsou podobné námi požadovanému modelu, je jen málo. Proto do tohoto krátkého porovnání zahrneme i ostatní kvadrokoptéry s programovatelnou řídící jednotkou.

2.1 APM copter neboli Arducopter

Projekt Arducopter [4] začal v roce 2010 a jeho primární určení je pro DYI (Do It Yourself) drony od společnosti 3D Robotics. Jak již název napovídá, tyto multirotory byly určeny k sestavení v domácím prostředí, spíše než k dodávání hotových celků (ačkoli tato možnost je k dispozici od distributorů také). Cílem tohoto projektu byla v počátcích asistence letu v podobě drobných vylepšení (automatické udržování výšky atd). V pozdějších fázích se ale již jedná o plně automatizovanou letovou kontrolu s pokročilým plánováním letu z pozemního stanoviště, a to na základě interaktivní mapy, GPS souřadnic a v případě asistovaného řízení i na základě kamery. V aktuální verzi má tento projekt již označení UAV (Unmanned Aerial Vehicle neboli Bezpilotní letoun), umožňuje plánování úloh, pasivní sledování letu i aktivní řízení. Základem řídící jednotky je jednodeskový počítač Arduino, jako ovládací jednotka se používá dálkové ovládání v kombinaci s počítačem (existují programy pro plánování a letovou kontrolu na Windows, Linux i Mac) a projekt jako takový je opensource pod licencí GNU GPL. Výhodou tohoto řešení je již vytvořené GUI, komplexní funkce pro navigaci, multiplatformní klientský software a funkcionality otestované mnoha uživateli po celém světě.

Nevýhodou je absence větší škály čidel, protože DYI drony jako takové spoléhají pouze na GPS s kompasem a gyroskop.

2.2 Owenquad

Podobně jako připravovaný projekt, i Owenquad [5] využívá Raspberrry Pi jako jednotku letového řízení a druhou řídící desku pro ovládání motorů. Touto deskou je jako v ostatních projektech jednodeskový počítač Arduino, který slouží k rychlému zpracování zpětné vazby a vyvažování kvadrokoptéry. Raspberry Pi zde slouží pouze pro zprostředkování uživatelského rozhraní za pomoci FOSS technologií distribuované po síti WiFi, určené pro ovládání z mobilních telefonů, prioritně s operačním systémem Android. Toto uživatelské rozhraní prakticky jen nahrazuje klasické ovládání pro modely, nepřináší nic z námi požadovaných vlastností. Projekt by byl dobrým základem, jeho autor bohužel však zdrojové kódy přemístil do soukromého repozitáře a nedaří se jej kontaktovat.

2.3 Seriál „Stavíme kvadrokoptéru s Raspberry Pi a Arduino Nano“

Tento čtyřdílný seriál, vydávaný na serveru root.cz [6], popisuje stavbu kvadrokoptéry od základu. Připravuje tak půdu pro česky mluvícího kutila, který by si chtěl kvadrokoptéru vyrobit doma. Pro pochopení návodu je potřeba základní znalost práce s jednodeskovým počítačem Arduino, základy programování v jazyce C a Python. Samotný seriál je psaný spíše formou postupného směrování k cíli, než jako přímý návod, je tedy vhodným zdrojem pro někoho, kdo potřebuje pouze nasměrovat a veškerou práci si je ochotný udělat sám. Co se týká využití Raspberry Pi, je v tomto seriálu zmíněno pouze okrajově, a to jako server pro rozhraní vytvořené v Pythonu, kterým by se měla kvadrokoptéra ovládat. Bohužel v seriálu je pouze zmíněno, jak prostředí pro Python připravit, chybí zde ale jakákoli zmínka o propojení mezi Raspberry Pi a Arduinem nebo prakticky cokoli o řízení či letové asistenci. V úvodu seriálu je také zmíněno, že Raspberry Pi je možné využít jako zdroj videa pro možnou kontrolu letu, bohužel tomuto tématu se dále již autor nevěnuje vůbec. Při kontaktu autora jsem zjistil, že prakticky ani kvadrokoptéru nedostavěl a již se nadále stavbě nevěnuje a v nejbližší době ani věnovat nebude.

2.4 Starlino články

Autoři článků Starlino [7] se věnují tvorbě různých robotických zařízení ovládaných jednočipovými mikropočítači PIC. K nalezení zde jsou řešení pro kvadrokoptéru, a dokonce i pokusný projekt kombinující kvadrokoptéru

s helikoptérou za účelem získání stabilizovanějšího letu. Autor uvádí nejen pracovní postupy, ale i zdrojové kódy, a to pod licencí MIT, takže je možné dostupné kódy dále dle potřeby upravovat. Pro reprodukování zde uvedených projektů je potřeba disponovat programátorem PIC mikročipů. Při tvorbě kvadrokoptéry nejsou použité žádné pokročilé plánovací techniky letu ani žádné navigační funkce, celý projekt je z tohoto pohledu jen drobnou asistencí řízení v podobě zpětnovazebního vyvažování a vyrovnávání s velkým důrazem na stabilizaci celé kvadrokoptéry. A samozřejmě je mikroprocesor využíván ke zpracování a distribuci řídících signálů. Velkou předností této práce jsou široké matematické podklady a rozbory kvadrokoptéry a je tedy vhodná jako zdroj informací pro možná vylepšení. Nevýhodou je program samotný, který je vytvořený pro procesory PIC a bylo by nutné vytvořit jeho port pro Raspberry Pi.

2.5 Vjaunet

Vjaunet quadrotor [8] je zajímavý projekt, blížící se specifikací požadavkům této bakalářské práce, založený částečně na projektu Owenquad. Využívá Raspberry Pi jako řídící jednotku ovládanou pomocí WiFi přes telefon s operačním systémem Android. Cílem projektu je připravit prostředí vytvořené v Pythonu na webové platformě a pomocí tohoto prostředí ovládat let kvadrokoptéry. Do budoucna si tento projekt ukládá za cíle rozšířit ovládání o živý přenos videa za pomoci Raspberry Pi. V současné chvíli jsou data vyčítána z senzorové desky MPU6050, která obsahuje gyroskop a akcelerometr, což jsou údaje dostatečné pro asistovaný let (vyrovnávání kvadrokoptéry), nikoli však pro autonomní letové řízení. O stabilizaci se stará program v C++ vytvořený pro blíže neurčený operační systém, k dispozici pro veřejnost pod licencí MIT.

2.6 Picopter

Pod tímto názvem se neskrývá jeden, ale hned několik různých projektů, které se snaží vytvořit kvadrokoptéru řízenou jednodeskovým počítačem Raspberry Pi. Název je tvořen spojením slov Raspberry Pi a Quadcopter, a proto je využíván více autory pro jejich projekt. Tyto různé projekty většinou nemívají vlastní prezentaci, pouze zmínku na webových fórech zabývajících se tematikou kvadrokoptér nebo Raspberry Pi o tom, že některý z uživatel pracuje na projektu Picopter. V některých případech se ovšem dají najít i zdrojové kódy, bohužel i zde jsou pouze funkce letové asistence a ne autonomního řízení. Zajímavostí je projekt [9], který staví základní projekt v C++ a navazuje na něj simulací určenou pro MATLAB, pomocí kterého řeší zpětnovazební vyvažování matematickými výpočty spíše než hrubou silou, jak to dělají ostatní projekty. Z pohledu samotné stabilizace kvadrokoptéry

je tedy toto řešení nejlepší, vyžaduje však výpočetní výkon zaměřený na matematickou stabilizaci modelu a odečítání hodnot z gyroskopu a accelerometru, které zvyšuje zátěž na vstupně/výstupních portech Raspberry Pi. Pokud by tímto způsobem bylo využíváno Raspberry Pi pouze k asistenci letového řízení, bylo by to ideální, ovšem využívat ho současně ještě jako řídící jednotku autonomního letu a jako zdroj videosignálu pro uživatelskou kontrolu letu by bylo příliš náročné na jeho 700MHz procesor.

Kapitola 3

Analýza

3.1 Výběr vhodného operačního systému

Tato sekce se věnuje porovnání operačních systémů typu RTOS a GPOS, a zvážení jejich možného využití pro řízení letové asistence kvadrokoptéry.

3.1.1 Operační systémy typu GPOS

Do kategorie GPOS se řadí běžné operační systémy, využívané v osobních počítačích. Nejčastějšími jsou Microsoft Windows, Mac OS a Linux. Linux do této kategorie patří nejméně, protože jeho jádro obsahuje i podporu RTOS funkcí. GPOS jsou typické svým přizpůsobením uživateli – například zobrazování GUI vyžaduje část výkonu počítače, a ačkoli nemá vliv na výsledky aplikací (nebereme-li v potaz potřebu vstupů uživatele, ale jen samotný běh algoritmů), v dnešní době je nutné, aby GUI reagovalo na požadavky uživatele (při delším běhu výpočtu uživatel běžně přepíná do jiné aplikace, aby se zabavil). Například bude-li uživatel v programu renderovat video, můžeme očekávat, že renderování bude mít vysokou prioritu. Protože renderování trvá dlouho, uživatel může chtít v jiné aplikaci zobrazit internetovou stránku. Tato aplikace bude mít malou prioritu, protože není z pohledu systému důležitá. Aby počítač dokázal efektivně zpracovat obě aplikace, musí jim určit přibližnou dobu běhu na procesoru – důležité delší, méně důležité kratší. Protože však v jeden moment přijde požadavek na načtení stránky, tato méně důležitější aplikace dostane větší množství dat ke zpracování a její běh na procesoru bude delší, než je plánováno. Toto chování systému nevadí, renderování bude sice trvat trochu déle, ale uživatel si mezi tím může prohlížet webové stránky, a je tedy spokojený.

3.1.1.1 Microsoft Windows

Protože Microsoft Windows [10] nejsou primárně určené pro procesory typu ARM (s výjimkou Windows 8, který má ale větší nároky na paměť, než je

Raspberry Pi schopno nabídnout), neexistuje žádná oficiální verze, která by byla pro Raspberry Pi určena. Existuje ovšem neoficiální projekt, který se pro něj snaží připravit Windows CE 7, což je ale operační systém typu RTOS, viz bod 1.2.1.

3.1.1.2 Mac OS

Mac OS [11] (a jeho další vývoje v podobě OS X) nejsou určené pro procesory ARM, a protože jádro tohoto operačního systému nemá veřejné zdrojové kódy, nejsou možné ani žádné neoficiální porty pro Raspberry Pi.

3.1.1.3 Linux

Existuje mnoho verzí operačního systému Linux, pro Raspberry Pi se nejčastěji vyskytují tyto distribuce: Debian [12], Fedora, Arch Linux. Žádná z těchto distribucí není ovšem čistě GPOS, ale obsahuje i soft RT podporu, budou tedy probrány podrobněji až později. Z pohledu využití OS v režimu bez RT vlastností jsem otestoval distribuci Debian, protože tato distribuce je dále využívána jako soft RT, a po rozšíření jádrem Xenomai i jako hard RT.

Distribuci Debian můžeme najít již předpřipravenou [13] k nasazení na Raspberry Pi, což začátečníkům s Linuxem a osobám nezkušeným s kompilováním jádra pro různé architektury procesorů usnadní mnoho práce.

3.1.2 Operační systémy typu Soft RTOS

3.1.2.1 Microsoft Windows CE

Operační systém Windows CE [14] obsahuje v jádře realtimovou podporu, ale protože Windows jsou určené primárně jako operační systém s grafickým rozhraním, režijní nároky pro běh operačního systému jsou příliš veliké, než aby se daly Windows používat jako operační systém reálného času. Jedinou výjimkou jsou Windows CE, což je operační systém určený pro embeedované aplikace a mobilní telefony. Tento operační systém je přímo určený k použití jako operační systém reálného času, a na rozdíl od ostatních verzí Windows tato verze má kernel licencovaný jako shared source, a je tedy za splnění určitých podmínek k dispozici vývojářům, kteří mohou Windows CE připravit pro cílové platformy. Díky tomu může existovat projekt, který se snaží připravit tento OS pro Raspberry Pi, ale protože tento projekt nese zatím pilotní alpha verzi, není vhodné ho zahrnovat do testů.

3.1.2.2 Linux-based OS – Debian, Fedora, Arch Linux

Operační systém Linux je v porovnání s MS Windows mnohem méně náročný, dokáže totiž běžet v CLI módu již s 64MB paměti a 1GB diskového prostoru a 1GHz rychlostí procesoru (minimální požadavky Debianu a Arch Linuxu,

Fedora má nároky trochu vyšší). Operační systémy této kategorie mají všechny v jádře integrovanou podporu soft RT. Podporují více úrovní priority, implementují základní POSIX API. Bohužel tyto operační systémy mají velkou velikost stránkování a nemohou se tedy přibližovat atomicitě přepínání mezi vlákny/procesy. Také nedisponují zaručeným plánovačem, aby bylo možno klást požadavky na systém běžné pro hard RT.

I přes tyto drobné nevýhody je z testů patrno, že dnešní operační systémy mají základy nutné pro běh aplikací v režimu reálného času.

3.1.3 Operační systémy typu Hard RTOS

3.1.3.1 ChibiOS

Operační systém ChibiOS [15] je jedním ze dvou kandidátů vhodných pro tak náročnou úlohu, jako je řízení letu. Jeho výhodou je rychlost, nízké nároky a portabilita. Jedná se o samostatný a úplný operační systém, podporující všechny požadované rozšiřující porty počítače. ChibiOS je založen na statickém mikrokernelu, který po svém načtení spustí uživatelský program. Z tohoto chování je již na první pohled vidět, že se jedná o operační systém určený k běhu v embeedovaných zařízeních. Navíc ChibiOS disponuje pouze základním shellem, který není ve výchozím stavu spuštěný (a je potřeba ho spouštět až z uživatelského programu), a který umožňuje připojení k interaktivní konzoli pomocí sériového portu. Samotné jádro operačního systému včetně podpůrných souborů potřebných k běhu se vejde do 4MB diskového prostoru, jedná se tedy o opravdu miniaturní OS. Pro použití v Raspberry Pi se ChibiOS překládá běžným C/C++ kompilátorem, ale doporučuje se využít tzv. cross compiler, který umožňuje kompilovat program určený pro jednu architekturu na architektuře druhé (tedy program pro ARM kompilovat na i86 procesorech). Pro podporu realtime funkcí má ChibiOS vlastní API, bohužel nekompatibilní s rozhraním POSIX normy. Přenositelnost aplikací na tuto architekturu vyžaduje úpravy ve zdrojovém kódu ve větším rozsahu, než jen v podobě typovacích maker. Navíc zde ještě vzniká problém při nasazení, protože všechny návody a testy dostupné na internetu jsou připravované pro starší verzi Raspberry Pi, nová verze vyžaduje jiné podpůrné soubory nekompatibilní s původními.

Bohužel neexistují žádná řešení připravená k nasazení na Raspberry Pi, k dispozici je pouze několik rad [16]. Tyto rady jsou bohužel dnes již zastaralé, vzhledem k nedávnému update firmware na Raspberry Pi, který vyžaduje jiný bootloader.

3.1.3.2 Xenomai framework

Xenomai [17] vlastně není operační systém jako takový, ale bezešvá implementace hard real-time podpory do jádra operačního systému Linux. Toto rozšíření vyžaduje vlastní bootovací prostředí, protože rozšiřuje nejen

plánovač, ale mění i velikost stránkování, přidává mnoho úrovní priorit a rozšíření o hodiny reálného času. Protože se jedná o nadstavbu operačního systému Linux, může mít tato varianta RTOS potíže se správou existujících ovladačů, pokud nejsou připravené pro reálný čas. Framework Xenomai umožňuje využívat rozšíření operačního systému o rozhraní POSIX normy a navíc obsahuje Nucleus API, takže je možné na operační systém s nadstavbou Xenomai přenášet aplikace vytvořené pro RTOS Nucleus. Díky přímému rozšíření operačního systému Linux je možno kompilovat aplikace určené ke spuštění na dané konfiguraci přímo, bez nutnosti využívat ke kompilaci jiný systém. Vývoj a testování aplikace tedy probíhá na jednom stroji, což omezuje šanci vzniku chyb při přenosu programu.

3.1.4 Testování operačních systémů vhodných pro aplikaci

Testováním a porovnáváním RTOS a GPOS se zabývá úzce specializovaná komunita, například společnost ElectronicsHub [18] nebo Open Source Automation Development Lab [19], protože pro běžného uživatele je RTOS zbytečný, a mnohdy i nevyužitelný.

Testy jsou inspirované diplomovou prací Petra Šumbery [20], jenž je shrnuta ve článku na stránce odbornecasopisy.cz [21]. Zaměřené jsou na rychlost přepínání mezi vlákny a práci s mutexy, protože tyto funkce budou prioritně potřeba v systému pro řízení letu kvadrokoptéry. Všechny testy jsem prováděl na Raspberry Pi, protože v emulovaném systému (například ve virtuálních strojích) by tyto testy nemusely poskytovat skutečné hodnoty. Další z testů je zápis a čtení dat na disk, aby bylo možné pracovat s velkým datovým objemem (například pro tvorbu virtuální navigační mapy pro algoritmy vyhledávající optimální cestu), generování velkého množství dat (tento test by měl ukázat schopnost vytvářet velké množství objektů v paměti a ideálně otestovat schopnosti stránkování paměti a přepínání stránek), a pro rozšíření systému o aplikace reálného času také test spolehlivosti přepínání mezi vlákny (cílem je vytížit procesor drobnými úlohami, které by mohly dostat přednost před dlouhodobou úlohou s vyšší prioritou, viz chování typické pro soft RTOS).

3.1.4.1 Testování na rychlost přepínání mezi vlákny

Při práci s vlákny v reálném čase na jednojádrovém procesoru je stěžejní doba přepnutí kontextu, která je v běžné aplikaci zanedbávána. Přepnutí kontextu by mělo být za ideální situace atomické, protože je však potřeba udržovat paměť a jádro procesoru nezávislé na aktuálně běžícím vlákně (v tom smyslu, že aktuálně běžící vlákno nemůže být ovlivněno vláknem běžícím před ním a nemůže přistupovat k proměnným jiného vlákna), je spojeno s režijními náklady v podobě času přepínání kontextu. Tento čas můžeme změřit, vytvoříme-li dvě vlákna, jejichž jediným smyslem existence bude

Tabulka 3.1: Měření rychlosti zamykání a odemykání mutexů.

OS	Minimum	Průměr	Maximum
Xenomai	0,23	0,24	0,31
Debian	0,01	0,02	0,02

uvolnění procesoru pro opačné vlákno. V případě výhradního běhu aplikace můžeme získat poměrně přesná data o době trvání přepínání, v režimu soft RT nebo úplně bez podpor funkcí reálného času budou tato data nepřesná, protože se do nich bude promítat ještě běh ostatních aplikací/vláken. Toto zkreslení nám ale nevadí, protože ve skutečnosti lépe reflektuje stav a chování aplikace za reálných podmínek.

Měření rychlosti přepínání vláken v obou systémech dopadlo obdobně, doba přepínání kontextu je přibližně 300 µs.

3.1.4.2 Práce s mutexy

Protože uzamčení mutexu není z pohledu user-space aplikace jednoduchá záležitost, může nás také zajímat, jak dlouhou dobu bude kernel trávit odemykáním, zamykáním a testováním mutexů. Pro tento test stačí aplikace, která bude opakovaně zkoušet odemknutí a zamknutí několika mutexů. Test na zjišťování hodnoty mutexů je nutno postavit nezávisle, protože doba mezi čekáním na mutex a pouhým zamykáním a odemykáním se může lišit. Jelikož připravovaná aplikace pro řízení kvadrokoptéry bude využívat v mnoha případech právě mutexy, je potřeba zjistit, jak se s nimi systém bude chovat a kolik času na nich bude trávit (Vzhledem k potřebě okamžitých reakcí je potřeba minimalizovat čas strávený na systémových záležitostech a maximalizovat čas efektivní pro výpočet.).

Měření práce s mutexy probíhalo na nezatíženém počítači, a výsledky3.1 vycházejí výrazně jiné proti očekávání.

3.1.4.3 Zápis a čtení dat na disku

Testování zápisu a čtení dat na disku je na první pohled jednoduchá věc, je však potřeba si uvědomit, že čas, který řadič pevného disku tráví vyhledáváním, je možno využít k výpočtům mimo čekající proces. Při testech využíváme Raspberry Pi v základní konfiguraci, která jako záznamové médium využívá paměťové karty, což výrazně urychluje přístup k datům. Je tedy potřeba porovnávat skutečný čas využitý testovací aplikací, nikoli čas od spuštění aplikace po její ukončení.

Měření opět probíhalo v nezatíženém systému, a výsledky3.2 jsou si již podobnější, ale pořád neodpovídají předpokladům.

Tabulka 3.2: Měření rychlosti zápisu do souboru.

OS	Minimum	Průměr	Maximum
Xenomai	3,31	3,51	3,89
Debian	2,86	2,87	2,87

Tabulka 3.3: Měření rychlosti vytváření a uvolňování velkých bloků paměti.

OS	Minimum	Průměr	Maximum
Xenomai	0,55	0,64	0,68
Debian	0,1	0,1	0,13

3.1.4.4 Testy práce s pamětí

V této sérii testů jde o srovnání operačních systémů z hlediska práce s pamětí – jak rychle umožní vytvářet objekty (jestli se v rychlostech nějak operační systémy liší), jak pracují s uvolňováním objektů a jak se systém chová při práci s pamětí při více vláknech. Vzhledem k navržené struktuře řídícího programu je potřeba porovnat, jak se v modelových situacích bude operační systém chovat. Navržené testy reflektují možné skutečné využití práce s pamětí v programu řízení letu.

I výsledky3.3 měření práce s pamětí překvapily výrazně jiným výsledkem, než jaký je prezentován ve všech citovaných srovnáních.

3.2 Návrh schématu propojení periferií k řídící jednotce, protokolu komunikace s periferiemi a struktury řídicího programu

Protože cílem této bakalářské práce není vytvořit kompletní systém včetně řízení a regulování motorů, je její realizace o mnoho snazší, než by tomu bylo v opačném případě. Pokud by bylo potřeba vytvořit řízení motorů, bylo by nutné oddělit komunikační kanály pro regulaci a pro zjišťování ostatních letových údajů, jako je například GPS. V takovém případě by se totiž jednalo o zpětnovazební řízení, kdy by oddělená úloha regulovala chod motorů na základě požadavků z řídícího vlákna a také na základě vlivů okolí, jako jsou například vzdušné proudy nebo samotný pohyb kvadrokoptéry. Pro takovéto řízení by bylo potřeba neustále vyčítat hodnoty z gyroskopu, akcelerometru a výškoměru a na jejich základě neustále upravovat drobné odchylky, které v průběhu letu budou vznikat od požadovaných hodnot. Jelikož jednodeskový počítač Raspberry Pi, pro který je tento řídící systém určený, disponuje širokou škálou komunikačních portů, bylo by možné celé řízení realizovat jen v rámci tohoto počítače, vhodnější však bude využít teoretický model kvadrokoptéry, který má integrované zpětnovazební řízení a od navigační jednotky přebírá

pouze požadavky k nastavení náklonu a rychlosti. Tím bude možné věnovat volnou část výpočetního výkonu navigační jednotky na ostatní úlohy, jako je výpočet optimální cesty, spolupráce s mapovými podklady, navigace v prostoru a další možné úlohy. Dále tedy budeme předpokládat model kvadrokoptéry jako celek, který je schopný vracet hodnoty jednotlivých čidel, nastavení rychlosti rotorů a kapacity baterie. Tento model také podporuje zápis, ale pouze některých hodnot, které dávají smysl (Například není možné, aby se zapisovala hodnota GPS, když GPS je dána čtením z čidla.). Dále tento teoretický model musí ještě podporovat možnost získání informací o systému jako takovém, například detekci stavu systému (inicializace, připraveno k letu, porucha). Tento teoretický model budeme dále v textu označovat zkratkou QC, navigační jednotku budeme dále označovat zkratkou RPi.

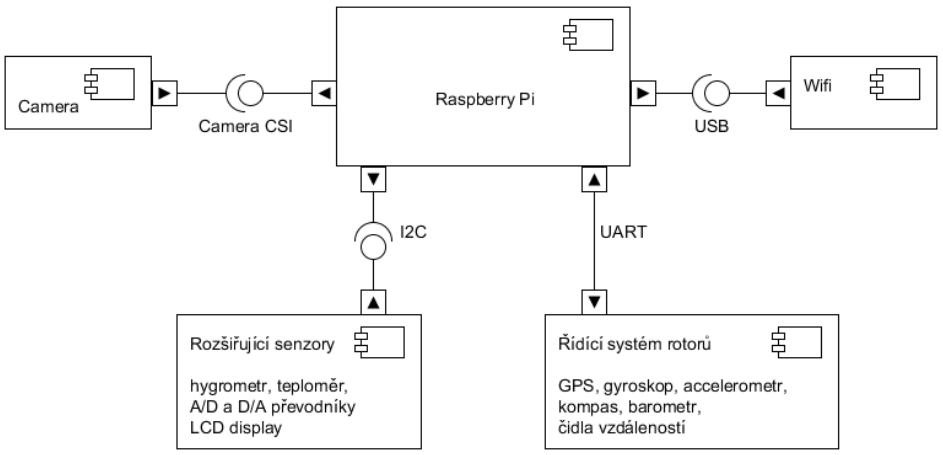
3.2.1 Návrh schématu propojení RPi s QC

Možností, jak propojit QC s RPi je velké množství, protože Raspberry Pi disponuje nejen standardními periferiemi jako jsou USB, ethernetový adaptér a sériová linka, ale oproti běžnému počítači disponuje navíc sběrnicí SPI, I2C a dalšími 8 vstupně-výstupními piny. Teoreticky lze všech těchto periferií využít ke komunikaci s rozšiřující deskou, prakticky je pro toto využití nevhodný ethernetový adaptér, protože by bylo nutné vytvořit vlastní protokol. Protokoly rodiny IP totiž nejsou vhodné k přenosu krátkých dat mezi dvěma zařízeními, mají na to moc velké režijní náklady. Dále můžeme pro účely běžných přenosů vyloučit vstupně-výstupní piny, protože by se přenos musel řídit navíc ještě odděleným hodinovým signálem. Rozumně využitelné možnosti jsou tedy USB, sériová linka, SPI a I2C. Všechny tyto periferie patří do kategorie sériového přenosu dat a práce s nimi bude v určitém smyslu obdobná. V této práci se dále soustředíme na sériovou linku a připravíme vzorový komunikační modul rozšiřitelný na kterékoli rozhraní 3.1.

Volbou spojení v podobě sériového rozhraní budeme muset předem určit parametry sériové linky, jako je velikost dat v jednom bloku, zakázání či povolení a případně typ parity, rychlost vysílání a další potřebná nastavení. Samozřejmě by bylo možné zahrnout do programu autodetekci těchto hodnot, ale vzhledem k velkému množství různých kombinací by autodetekce zbytečně prodlužovala inicializaci programu. Po stanovení těchto hodnot je ještě nutno určit poslední parametr, a to frekvenci vysílání (Pozor, neplést frekvenci s rychlostí vysílání – rychlost vysílání určuje počet pulzů za sekundu, kdežto frekvence určuje, kolikrát za sekundu dochází k přenosu zpráv mezi QC a RPi.).

3.2.2 Navrhněte protokol pro komunikaci

Pro komunikaci mezi QC a RPi potřebujeme jednoduchý protokol, který nebude zatěžovat QC přidanými výpočty navíc, ideálně takový, který bude



Obrázek 3.1: Schéma navrhovaného propojení RPi s QC

poskytovat data v surové podobě, určená dále ke zpracování v RPi. Jediné, co potřebujeme zachovat, je druh přenášených dat, aby všechna čidla nezávisle na typu přenášela data se stejnou jednotkou a nedocházelo k chybám vlivem použité soustavy. Protokol musí podporovat obousměrný přenos hodnot a také dotazování těchto hodnot. Protože cílem je nevytěžovat QC, je nutné také dbát na vhodnou velikost packetů přenášených mezi QC a RPi a obsah packetů. Pokud by protokol byl příliš složitý, řídící deska QC by trávila příliš mnoho času parsováním dat a nezbýval by výpočetní výkon pro vyčítání hodnot z senzorů i následné úpravy zpětné vazby pro rotory. Dále je potřeba protokol připravit pro možná rozšíření o další čidla a senzory (Což by se stávat nemělo, protože taková komunikace by zbytečně vytěžovala linku pro řízení, ale pokud by byla QC připojena například přes sériový port, ostatní senzory by byly připojené pomocí I2C a nově připojovaná periferie by neumožňovala komunikaci přes I2C nebo USB, mohlo by dávat větší smysl připojit ji do QC a zařadit toto nové zařízení do komunikace mezi QC a RPi.) a případně i o odebírání senzorů (Například pro aplikace závislé na přesných zeměpisných souřadnicích může být vhodnější propojit GPS přes nezávislý port a umožnit tak častější komunikaci mezi GPS modulem a RPi.), umožnit zpracovávat data o proměnné délce a také přenášet různé datové typy. Pokud budeme souhlasit s přenášením dat o proměnné délce, což se v mnoha případech může zdát výhodné, máme-li většinu zpráv krátkých, například jedno nebo dvoubitových, musíme též do úvahy zahrnout, zda potřebujeme indikovat konec zprávy. Pokud bude použita sekvence bitů jako konec zprávy, je nutno zvážit, zda tato sekvence nemůže nastat uvnitř dat a tedy vytvořit falešný konec zprávy. Protože se bude jednat o prostá data, mohlo by dávat smysl zahrnout do protokolu kontrolní paritní bit, v extrémnějším případě zvolit

nějakou samoopravovací variantu kódu, pokud by docházelo k silnějšímu rušení, ale pokud chceme omezit co nejvíce přenášená data, prvotní návrh se kontrolou zabývat nebude. Dále by bylo dobré mít v protokolu zahrnuté potvrzování příkazů ze strany QC, aby bylo zaručeno, že vyslané příkazy nebyly ztraceny (například z důvodu přeplnění či vyprázdnění bufferu na straně QC), protože zašleme-li příkaz k zápisu do QC, nebyla by definována odpověď, a tedy by se vysílací okénko plně nevyužilo. Dalším cílem je tedy také plné využití komunikačního okénka, protože v případě nízkofrekvenční komunikace je okénko drahé.

Z hlediska zakomponování protokolu do programu je ještě nutno zvážit stránku realtime zpracování, a to hned z několika pohledů. Zaprvé je nutno se zamyslet nad faktem, že špatně navržený protokol by mohl vyžadovat delší výpočetní dobu, a tedy by program mohl zůstat příliš dlouhou dobu v kritické sekci a bránit důležitějším částem navigačního programu ve vykonávání. Dále je nutno vzít v úvahu extrémní případ, kdy by bylo při každém cyklu okénka potřeba větší množství paměti pro výpočty a tuto paměť by bylo nutno inicializovat, což by mohlo ovlivnit rychlost přepínání kontextu mezi vlákny. Z těchto požadavků je zřejmé, že využití komplexnějšího protokolu pro přenos dat mezi QC a RPi nepřipadá v úvahu a protokol by měl být vymyšlen pro maximální jednoduchost a transparentnost.

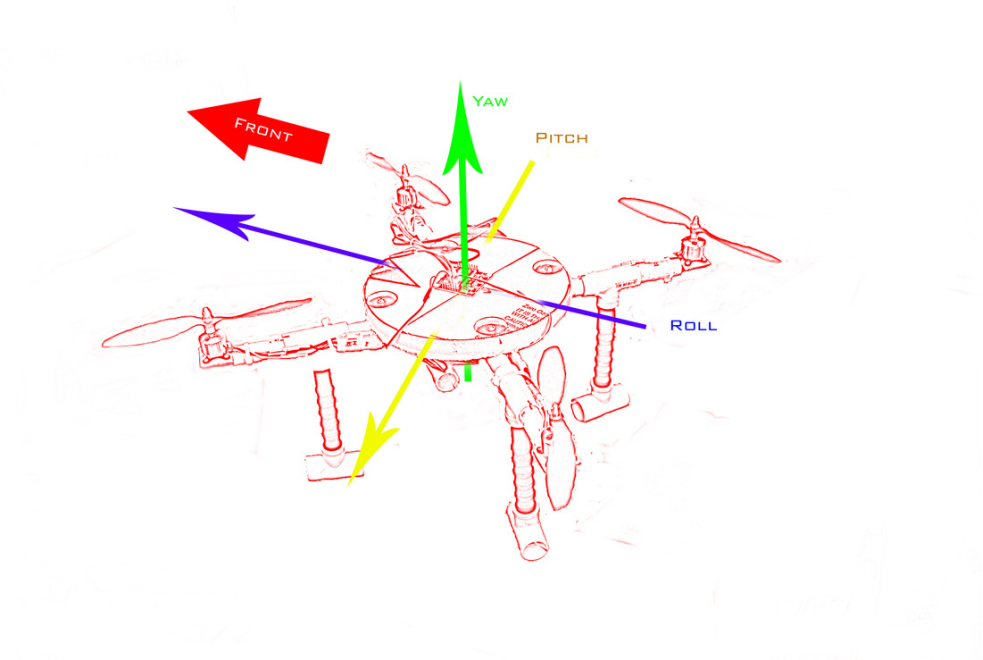
Posledním hlediskem, jak můžeme nahlížet na volbu protokolu, je z pohledu QC samotné. Důvod k využití Raspberry Pi jako řídícího počítače je ten, že QC nedisponuje příliš výkonným procesorem, který nutně nemusí zvládat pokročilejší matematické funkce (a nebo je nemusí zvládat v optimálním čase), jakými by mohlo být například využívání hashů. Tento procesor také nemusí disponovat příliš velkou pamětí programu a dat, a protože kromě komunikace s RPi musí zastávat hlavně funkci zpětnovazebního řízení, je lépe nechat větší prostor pro tuto jeho hlavní funkci. Navíc pro rozpoznávání příchozích packetů je lepší rozhodování na základě předem známých hodnot, než dlouhý dynamický výpočet a přiřazování adres jednotlivých periferních zařízení (čidla, senzory).

3.2.3 Struktura jádra řídícího programu

Nejprve budeme potřebovat pár základních údajů o kvadrokoptéře. Řízení probíhá za pomoci nastavování několika údajů - rychlosti rotorů, a náklonu kvadrokoptéry. Pro rychlost můžeme použít RPM, ale efektivněji můžeme posílat jen procento nastavení rychlosti, protože řídící deska rotorů ve skutečnosti stejně pracuje jen s touto hodnotou, a nikoli s RPM. Nastavení náklonu budeme řešit dle standardních leteckých os - yaw, pitch, roll 3.2

Jádro řídícího programu musí obsahovat údaje o těchto základních položkách:

1. aktuálně nastavená rychlost rotorů



Obrázek 3.2: Letecké osy promítnuté do modelu kvadrokoptéry [1]

2. aktuální výška
3. požadovaná výška
4. aktuální rychlost letu kvadrokoptéry
5. požadovaná rychlost letu kvadrokoptéry
6. aktuální náklon kvadrokoptéry v ose X a Y
7. požadovaný náklon kvadrokoptéry v ose X a Y
8. aktuální směr letu (v podobě odchylky od severu)
9. požadovaný směr letu (v podobě odchylky od severu)
10. aktuální natočení kvadrokoptéry v ose Z
11. aktuální hodnota nabití baterie

Toto jsou základní položky, bez kterých by se řízení prakticky neobešlo, ovšem seznam těchto položek můžeme dále rozšiřovat, obzvláště pokud naším cílem je řízení s asistencí letu, tedy prevence nabourání do překážky a GPS souřadnice pro určování cílů. V případě řízení za pomoci bezdrátového přenosu mezi RPi a pozemní řídící jednotkou (například při řízení pomocí WiFi) by

bylo dobré udržovat si přehled o stavu signálu, aby kvadrokoptéra nelétala mimo oblast signálu. Dále ještě může být vhodné mít k dispozici výchozí souřadnice, protože v případě nízkého stavu baterie může kvadrokoptéra doletět na místo odletu, aby bylo snazší vybitou kvadrokoptéru získat zpět.

Program také musí v základu obsahovat možnost provádění komunikace mezi RPi a QC, a to nezávisle na frekvenci komunikace. Tato část může být nejvíce problematická, protože zatímco různé řídící algoritmy mohou požadovat velké množství informací, pokud možno okamžitě, frekvence komunikace mezi RPi a QC může být velikou překážkou, obzvláště pokud komunikační protokol nemá možnost získání všech informací najednou. Je tedy potřeba zvolit vhodný způsob dotazování požadovaných hodnot, který bude udržovat data aktuální. Jako spravedlivý by se mohl zdát stavový automat, který bude každé informaci přidělovat počet a pořadí dotazů na cyklus přes všechny veličiny. Alternativním přístupem by mohlo být využití fronty, ať už obyčejné nebo například prioritní. Tento přístup by se mohl jevit jako dobrý nápad, ovšem ve skutečnosti bude vznikat několik problémů – pokud by byla využita obyčejná fronta, požadavky na méně důležité věci by mohly dostávat prioritu před důležitějšími požadavky, například kdyby kvadrokoptéře hrozila srážka a fronta by byla zaplněna dotazy na rychlost, náklon, nabití baterie a podobně. Prioritní fronta může tyto situace předvídat a velmi efektivně je řešit, obzvláště s využitím dynamicky přiřazované priority dle situace. Problém s prioritní frontou bude ovšem nastávat v případě, kdy by se nízkoprioritní dotazy prakticky nedostávaly na řadu, ačkoli by jejich údaje byly v určitých chvílích potřebné (Například manévrování mezi překážkami v obtížném terénu má jistě přednost před vyčítáním hodnoty nabití baterie, ale pokud by toto trvalo příliš dlouho, mohlo by dojít k situaci, že již nebude zbývat energie na návrat, ačkoli je návrat požadovaný.). Tento nedostatek lze sice dále řešit pomocí zvyšování priority dotazů, které se pro nízkou prioritu nedostanou na řadu, největší problém fronty ovšem ještě nebyl zmíněn, a tím je fronta samotná. Při častém dotazování a nízkofrekvenční komunikaci mezi RPi a QC bude totiž docházet k postupnému plnění fronty a postupnému zhoršování aktuálnosti dotazů – bude-li fronta plná již před příletem k překážce, a překážka bude náhodou detekována, ve frontě může být ještě spousta nevyřízených dotazů, které budou zabírat paměť, zvyšovat velikost fronty a mohlo by docházet k hromadění dotazů dle jejich typu.

Praktické řešení by mohlo být spojit předchozí dva přístupy, a vytvořit omezenou frontu, dlouhou stejně jako údaje, které se snažíme získat. Pro každý údaj by existovala priorita, která by se časem zvyšovala, a na základě toho by se odesílal vždy aktuální dotaz, aniž by docházelo k diskriminaci dotazů s nižší prioritou nebo plnění fronty jedním typem dotazů. Každý prvek by měl svou počáteční prioritu, prioritu odvozenou dle situace (což by prakticky byla počáteční priorita upravená o hodnotu dle situace), a nakonec prioritu odesílání, která by se resetovala při každém odeslání dotazu. Tento způsob provedení komunikace by se mohl zdát spravedlivý a snadno

aplikovatelný pro vícevláknovou aplikaci, ale i zde vzniká problém, a to nastavování hodnot. Zatímco pro vyčítání hodnot je tento přístup optimální, je ještě nutno zauvažovat nad možnostmi, kdy by priorita vyčítání mohla ohrozit provedení nastavení. Tato situace by mohla nastat v případě, že by došlo k nutnosti náhlé změny letu, například pro vyhnutí se překážce, a dotazování by mělo aktuálně vysokou prioritu. Pro tyto případy je vždy možné využít „nouzové“ priority, vyhrazené a za normální situace nezískatelné.

Vzhledem k aktuálně navrženému komunikačnímu protokolu, který má pouze 8 základních dotazů, 25 možných dotazů celkem a z toho 5 položek s možným zápisem, tato kombinace se jeví jako dobré řešení.

Poslední položkou, která nás může zajímat při tvorbě základního programu, je rozšiřitelnost o další funkce. Rozšiřování přímé, tedy úpravou zdrojového kódu jádra o přidané funkce by mohlo snadno dojít k situaci, kdy se bude vykonávat spousta kódu rozšíření, ale výkonný kód jádra bude čekat na vyřízení všech rozšíření. Již dle první analýzy je patrné, že sekvenční návrh programu je v tomto případě špatná volba, a je nutno program rozdělit do nezávislých vláken. Kdyby k tomuto rozdělení nedošlo, celý koncept operačního systému reálného času by poté absolutně nedával smysl.

Přijmeme-li tedy předpoklad, že vhodným modelem pro rozšiřování programu jsou moduly se samostatným vláknem, můžeme připravit prvotní model těchto modulů. Protože předem nemůžeme určit, co by který modul mohl vyžadovat, můžeme snadno požadavky přenechat jednotlivým modulům a jejich přípravu zajistit inicializační funkcí. Po přípravě můžeme modul spustit funkcí pro běh, která si sama zajistí vytvoření vlákna. Tímto dosáhneme jednotného rozhraní bez nutnosti vytvářet abstraktní modely. Prioritu tohoto vlákna může modul mít předdefinovanou, ale jádro by mělo mít možnost tuto prioritu měnit nejen při spouštění, ale i za běhu, nastane-li situace, která by si takovou změnu vyžadovala.

Pokud by modulů bylo větší množství, za zvážení by mohl stát model, který by spouštěl vlákna s přiřazenou prioritou, a tato vlákna by sekvenčně zpracovávala moduly se stejnou prioritou. Toto řešení by sice mělo nevýhodu, že dlouhotrvající operace jednoho modulu by zpomalovaly běh ostatních modulů se stejnou prioritou, ale zajistilo by souběh mnoha modulů bez nutnosti přepínání kontextu velkého množství vláken. Jako další možné vylepšení této myšlenky by mohlo být spuštění vlastního vlákna pro moduly s dlouhými výpočty, o kterých můžeme předpokládat, že jich nebude příliš velké množství, a můžeme tak zaručit souběh dlouhotrvajících modulů s krátkými, aniž bychom krátké moduly omezovali na výpočetním čase. Tento model nebude implementován jako součást této bakalářské práce, protože systém vytvořený k této bakalářské práci nebude obsahovat natolik velké množství modulů, a tedy nebude potřeba řešit jejich režijní náklady.

Kapitola 4

Návrh řešení

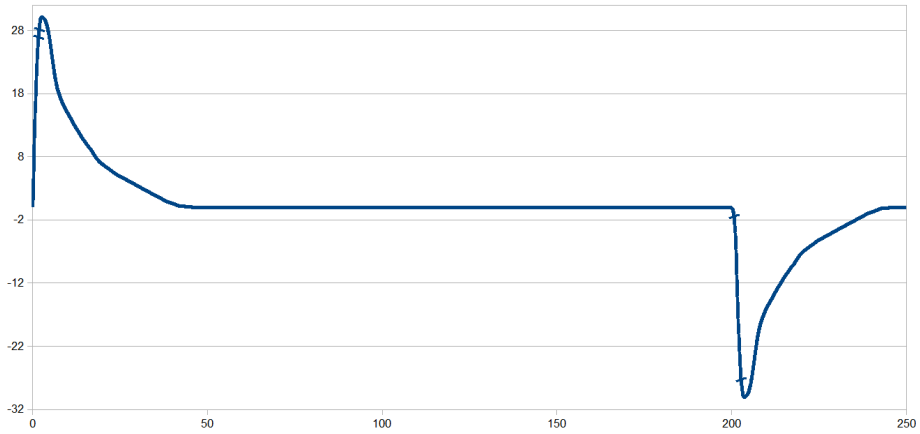
4.1 Řídící struktura programu

Na základě podrobné analýzy v bodě 3 můžeme snadno odhadnout rozsah potřebného řešení. V první řadě je nutno připravit řídící strukturu a podpůrné funkce pro ni. Realizace tohoto cíle je přímočará a bude částečně reflektovat položky v protokolu komunikace (hlavně z pohledu datových typů). Jako vhodné řešení bude postačovat datový typ struct s výčtem položek a odvozených nejčastěji používaných hodnot (Například úhlový směr letu vůči ose Z lze určit na základě natočení kvadrokoptéry v osách X a Y, ale je vhodné si tento úhel udržovat mimo, protože se na jeho základě určují potřebná čidla pro detekci překážek v dopředném směru letu.). Podpůrnými funkcemi můžeme v tomto bodě rozumět funkce zajišťující aktuálnost odvozených dat při změně dat základních.

4.2 Jádro programu

Jádrem programu můžeme rozumět sadu funkcí pro základní navigaci kvadrokoptéry:

1. změna výšky
2. přepočet požadovaného směru letu na natočení kvadrokoptéry
3. nouzové přistání
4. inicializace
5. příprava a spuštění modulů
6. příprava a spuštění komunikace mezi RPi a QC
7. výpočet aktuální rychlosti letu



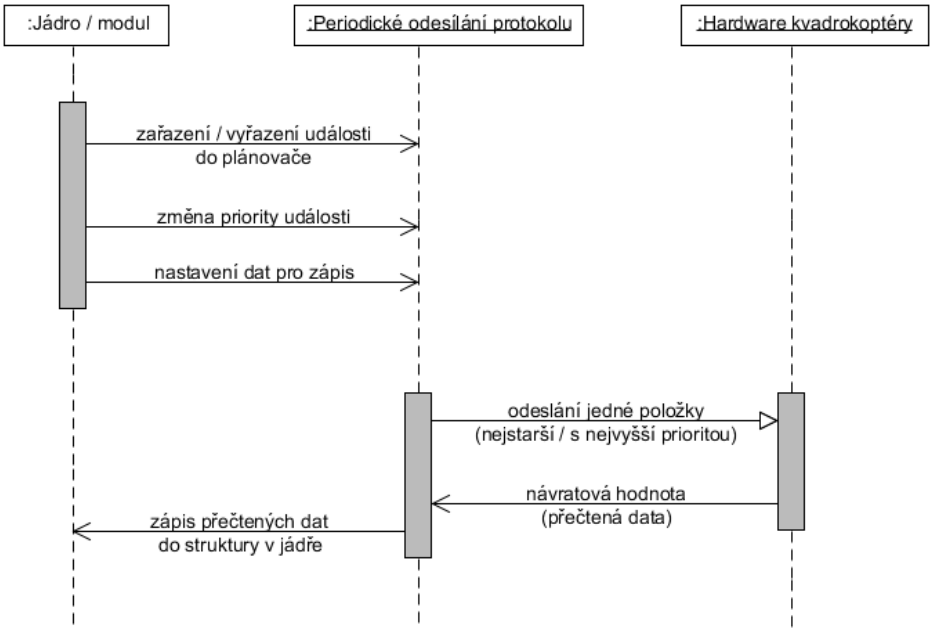
Obrázek 4.1: Teoretický graf hodnot accelerometru při zrychleném a následně zpomaleném pohybu

Pro výpočet aktuální rychlosti letu můžeme použít buď údaje z GPS, které ale mohou být zkreslené, a na krátké lety nebude GPS inicializována, a nebo můžeme využít accelerometr. Teoretické údaje z accelerometru pro pohyb zrychlený a následně zpomalený můžeme vidět na grafu 4.1. Okamžitou rychlost lze na grafu spočítat jako hodnotu určitého integrálu od 0 po bod, ve kterém chceme rychlost zjistit. Protože však tento graf (a žádný graf v praxi) nemá žádnou snadno odhadnutelnou funkci, je vhodné použít metodu Riemannova integrálu. Tato metoda má navíc velikou výhodu, že při posunu výpočtu v ose X do bodu x+a nám stačí znát hodnotu integrálu v bodě x, a dopočítat jen rozdíl mezi body x a x+a. Jediná nevýhoda této metody je při pohybu konstantní rychlostí, protože accelerometr v praxi bude vykazovat drobné oscilace. Praktickým měřením lze určit mezní hodnotu, které této oscilace dosahují, a cokoli menšího zanedbat, a tím výrazně omezit nepřesnost.

4.3 Komunikační protokol

Dalším důležitým bodem je vytvoření funkcí implementujících komunikační protokol a zaručujících aktuálnost dat v řídící struktuře programu. Řešení se bude sestávat z vhodné implementace prioritní fronty, periodického zpracování požadavků v této frontě, podpůrných funkcí pro práci s frontou (umístění požadavku do fronty, správa priorit) a podpůrných struktur pro převod a zasílání dat (Námi zvolená sériová linka zasílá zprávy jako byte-stream, ale data v řídící struktuře jsou ukládána v různých datových typech.).
Pro protokol jsou navržena následující pravidla:

1. Pokud se jedná o dotaz, zasílá se vždy jen číslo dotazu



Obrázek 4.2: Schéma navrhovaného propojení RPi s QC

2. Pokud se jedná o zápis, zasílá se číslo dotazu s přičtenou bitovou maskou 80h + data
3. Odpověď na dotaz je vždy číslo dotazu + data odpovědi
4. Odpověď na zápis je buď jen číslo dotazu, nebo číslo dotazu + aktuální data
5. Protool je jednostranný, RPi zasílá požadavky, QC vyřizuje odpovědi
6. Jedinou výjimkou, kdy QC neodpoví na dotaz je zaslání stavu 127 s nastaveným error bitem namísto odpovědi

Pro lepší porozumění fungování protokolu je v tabulce 4.2. Zpracování zpráv probíhá na základě sekvenčního modelu priorit 4.2.

Tento protokol je navržen ve spolupráci s Jakubem Halákem, který aktuálně vyrábí hardware kvadrokoptéry jako svou diplomovou práci, a má tedy praktické zkušenosti s periferiemi. Mou prací je vytvoření protokolu, Jakubovým příspěvkem je jeho korekce z pohledu datových typů.

Tabulka 4.1: Protokol komunikace verze 1.1

Číslo (max 128)	Význam	Datový typ	Priorita	Poznámka
1	gyro	3x int16	20	(x, y, z)
2	gyro_x	int16	19	(roll)
3	gyro_y	int16	19	(pitch)
4	gyro_z	int16	19	(yaw)
11	speed	int8	20	
čísla vyšší než 50 jsou pro položky pouze pro čtení				
51	gps	2x double, float	10	(lat, lon, elev)
52	gps_lat	double	9	
53	gps_lon	double	9	
54	gps_elevation	float	8	
61	battery_max	int8	0	(napětí)
62	battery_current	int8	5	(procento nabití)
63	altitude	float	15	(na základě barometru)
64	compas	int16	10	(odchylka natočení „předku“ vůči
71	accelerometer	3x float	18	(x, y, z)
72	accelerometer_x	float	17	
73	accelerometer_y	float	17	
74	accelerometer_z	float	17	
čísla vyšší než 100 jsou rezervovaná pro vzdálenostní senzory				
101	sensor_front	int16	10	(cm od překážky)
102	sensor_right	int16	10	(cm od překážky)
103	sensor_back	int16	10	(cm od překážky)
104	sensor_left	int16	10	(cm od překážky)
105	sensor_around	4x int16	13	(front, right, back, left)
106	sensor_top	int16	11	(cm od překážky)
107	sensor_bottom	int16	12	(cm od překážky)
nejvyšší možné číslo je rezervováno pro přenos stavů				
127	state	int8	2	(ready, error, 6x reserved)

Tabulka 4.2: Ukázka komunikace navrženým protokolem

hodinový signál	RPi	QC	Vysvětlení
1	01h	01 0000 0000 0000h	RPi zjišťuje stav natočení kvadrokoptéry, inicializ
2	0Bh	0B 00h	RPi zjišťuje aktuální rychlost otáčení rotorů, QC
3	3Eh	3E 64h	RPi zjišťuje aktuální nabití baterie, QC vrací 100
4	7Fh	7F 80h	RPi zjišťuje, zda je kvadrokoptéra připravena k le
5	8B 10h	0B	RPi dává pokyn ke startu rotorů, QC potvrzuje
6	6Ah	6A 0016h	RPi dotazuje výšku, QC vrací aktuální výšku
7	6Ah	6A 0034h	RPi dotazuje výšku, QC vrací aktuální výšku
8	6Ah	6A 003Eh	RPi dotazuje výšku, QC vrací aktuální výšku
9	...	...	

Kapitola 5

Řešení požadovaného řídícího software

Následující oddíl popisuje jednotlivé bloky jádra včetně schémat komunikačních diagramů mezi těmito bloky.

5.1 Stavový registr

Stavový registr je tvořen strukturou 5.1, která obsahuje záznamy pro jednotlivé položky dotazů, výsledky pomocných výpočtů, potřebné globální zámky a seznam použitých modulů.

Rychlost kvadrokoptéry – rychlost je vyčítána pomocí tříosého gyroskopu, který udává momentální rychlost v jednotkách gravitace, přičemž 1 g = 9.80665 m/s^2. K získání rychlosti kvadrokoptéry můžeme využít integrálu funkce zrychlení. Pro přibližný výpočet můžeme použít aproximaci na základě známé doby mezi jednotlivými měřeními a zrychlení při dvou měřeních.

Natočení kvadrokoptéry – protože od řídící desky QC požadujeme stabilitu na základě zpětnovazebního řízení, můžeme pro zjednodušení počítat se situací, že pokud zašleme požadavek k nastavení, který bude úspěšně vykonán, a kvadrokoptéra bude jednou natočena na požadovanou hodnotu, tato hodnota zůstane konstantní do dalšího požadavku na změnu. Hlídat natočení kvadrokoptéry tedy dává smysl pouze po požadavku na změnu, a to jen na dobu nezbytně nutnou (tedy dokud se kvadrokoptéra neotočí do tolerance požadovaného úhlu). Přenosovou jednotku zde počítáme jako úhel ve stupních. Kvůli proudění vzduchu, toleranci přesnosti měření a otřesům nemá smysl se zabývat jemnějším dělením. Pokud by se ovšem v praxi zjistilo, že by jemnější dělení dávalo smysl, jsou hodnoty přenášeny jako šestnáctibitová čísla, což zaručuje dostatečný prostor pro zpřesnění (Teoreticky až na dvě desetinná místa, ovšem předpoklad je, že by se využívalo maximálně jednoho desetinného

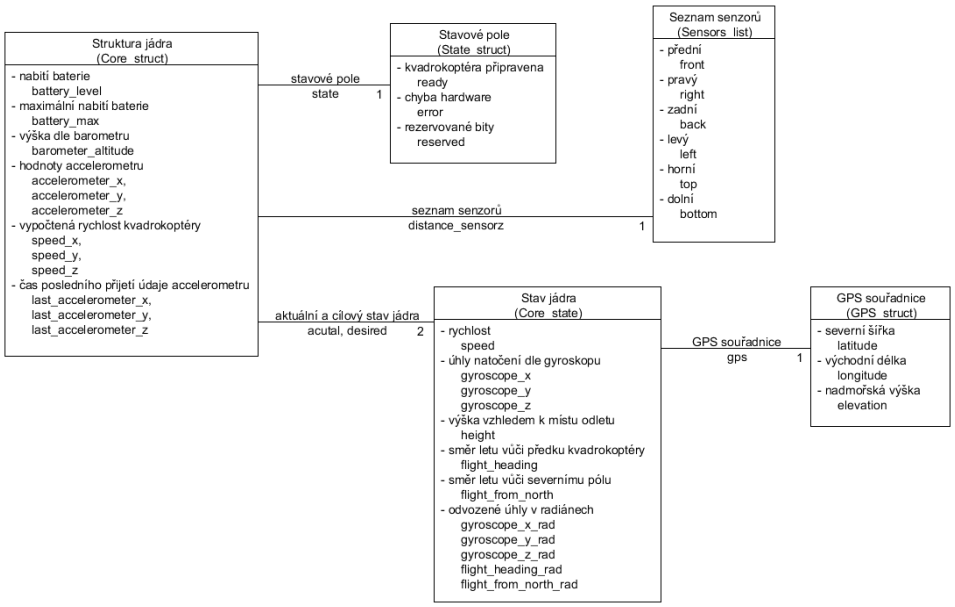
místa, a nebo modulární aritmetiky, například modulo 2 pro dělení po polovině stupňů.).

Výška kvadrokoptéry – pro stanovení výšky kvadrokoptéry máme několik možností – dokud je kvadrokoptéra v dosahu spodního čidla na zem, máme výšku udánu poměrně přesně. Výšku můžeme stanovovat také na základě barometrického měření, které je ještě možné zpřesnit předchozí kalibrací. Výšku nám také udává GPS, ale její přesnost není příliš spolehlivá. Základní jednotkou výšky jsou cm nad zemí, ukládané jako int16_t, předpokládá se, že kvadrokoptéra nebude létat ve větších výškách než 650m (I kdyby kvadrokoptéra měla přelétávat běžné budovy ve městě, tak nejvyšší z budov v Praze má cca 220 metrů a rezerva je tedy dostatečná). Největším problémem může být start kvadrokoptéry z vysokého místa a následně let pod úrovní startu. Pro řešení tohoto problému je zaváděna prostá kontrola výšky na základě priority výškového vzdálenostního čidla před barometrickým. Pokud kvadrokoptéra doletí na úroveň nižší než start, posune tuto úroveň na základě prosté podmínky – jakmile se kvadrokoptéra ocitne na úrovni blízké nulové výšce a čidlo vzdálenosti od země hlásí, že země je vzdálenější, než má být, počáteční výška se posune o rozdíl počátku a nižšího bodu, pokud čidlo není schopno detekovat zem vůbec, rozdíl bude stanoven na základě předem známého dosahu čidla a počáteční polohy.

Směr letu – je důležité, aby kvadrokoptéra měla jasně určený směr letu. Důvodem je určení předního čidla vzdáleností, které by mělo detekovat možnou kolizi, protože postupné dotazování všech čidel je zbytečná zátěž na komunikaci mezi RPi a QC. Přední čidlo můžeme určit na základě nastavení natočení kvadrokoptéry, protože to přímo určuje směr letu. Jediný problém může vznikat při brzdění natočením do protisměru, ale tuto výjimku můžeme snadno odchytit, a při brždění zasílat dotazy na všechna čidla.

Baterie – monitorování úbytků na baterii je důležité zejména pro plánování další cesty kvadrokoptéry, a také pro nouzové přistávací manévry. Více se tomuto tématu budeme věnovat v kapitole 6.1, kde se řeší zpětná navigace kvadrokoptéry na výchozí pozici. Jádro kvadrokoptéry bude v tomto smyslu implementovat pouze nouzové přistávání, a to za situace, kdy úroveň baterie poklesne pod bezpečnou úroveň pro další let. Za nouzové přistání může být považováno zastavení dalšího letu a kolmé přistání na zem. V kapitole 6.2 je nastíněný postup, jak by mohlo být nouzové přistání vylepšeno, aby se zamezilo pokusům přistát na nemožném místě (v koruně stromu, na lampě, na vrcholku bleskosvodů, atd).

Rychlost rotorů – rychlost rotorů je udávaná jako int8_t číslo, které je offsetem rychlosti skutečné. Offset je využit kvůli faktu, že nezanedbatelná část rychlosti rotorů není pro kvadrokoptéru efektivní, protože není schopná překonat gravitační sílu a tedy neumožňuje let. Rozsah rychlosti rotorů se může lišit dle použitého typu, stejně jako konstanta offsetu (offset řeší QC), takže v případě změny typu rotorů by mohlo být nutné rozšířit tento registr spolu s úpravou komunikačního protokolu.



Obrázek 5.1: Doménový model řídící struktury s využitím asociačních tříd

5.2 Základní funkce pro podporu letu

Změna výšky – prvotní funkce, bez které se kvadrokoptéra neobejde, je změna výšky. Kvadrokoptéra v prvotním modelu implementuje naivní řešení, které se pokusí přiblížit požadované výšce a postupně bude brzdit rotory, než stanoví jejich rychlost potřebnou pro udržení této výšky. Jako lepší řešení by mohlo být využití tabulkového modelu, kde každé výšce bude přiřazena rychlost, jakou rotory potřebují pro vyrovnání gravitační síly. Nejlepším možným řešením by byl matematický výpočet uvažující skládání vektorů sil působících na kvadrokoptéru se započtením potřebné změny výšky.

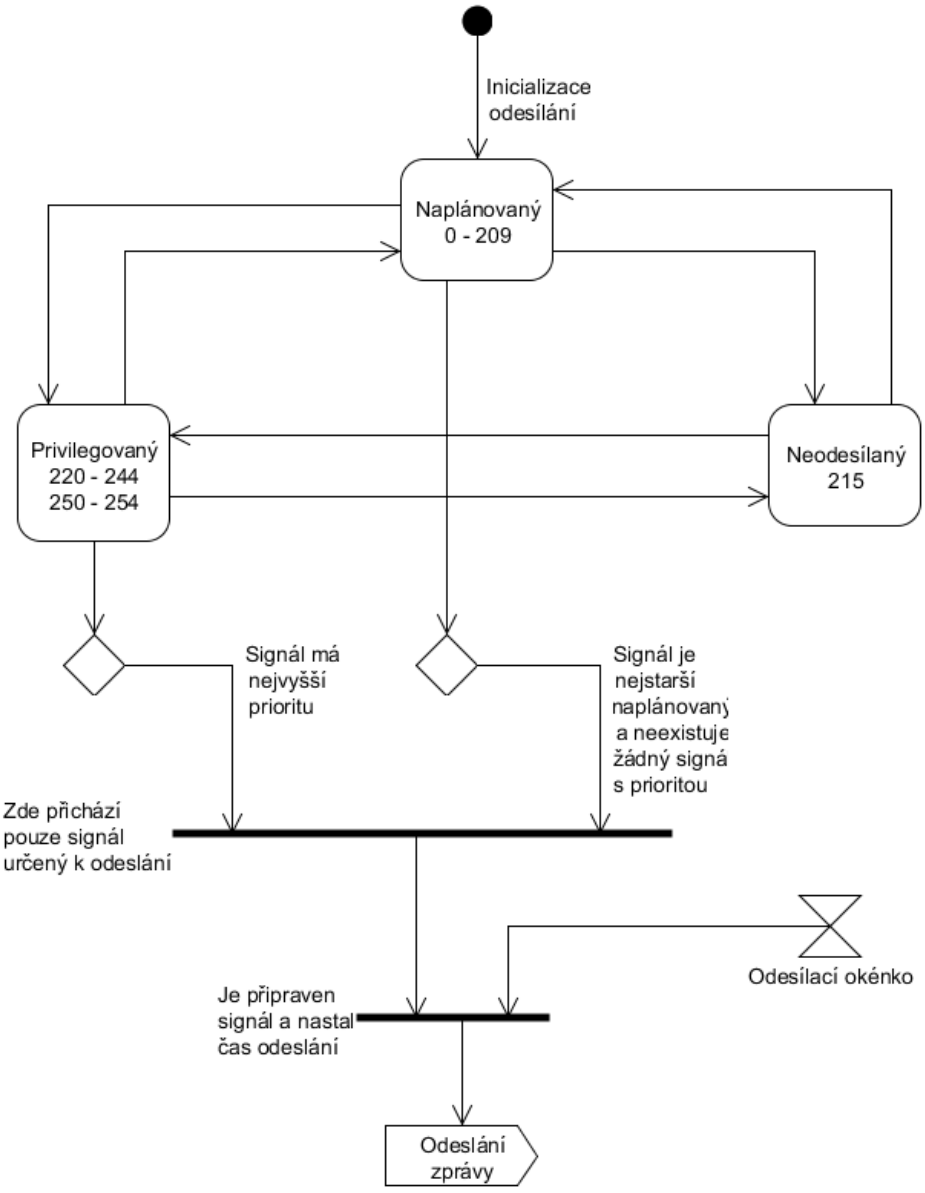
Let v udaném směru – druhá nejzákladnější funkce pro pohyb kvadrokoptéry, implementovaná jako naivní řešení přepočítávající udaný úhlový směr na natočení kvadrokoptéry v osách X a Y. Protože mohou existovat různé požadavky na pohyb kvadrokoptéry v udaném směru, existuje přepínač definující, zda se funkce má snažit o zachování výšky, nebo zda tato snaha není požadována (a v takovém případě je doporučené hlídat výšku mimo na této funkci). Stejně jako v případě změny výšky, i zde by lepší řešení představoval model založený na skládání vektorů, protože by snadno mohl reflektovat změny potřebné k dosažení cílového směru s úvahou iniciálního směru. Další možnou optimalizací je započítání možné změny v ose Z a dále také optimalizace z pohledu energetické náročnosti pohybu v požadovaném směru. Na základě analýzy je možné určit, jaké natočení kvadrokoptéry vybíjí baterii nejméně (s přihlédnutím k požadované rychlosti letu, vektoru letu atd).

Akustické signály – velmi užitečnou funkcí je též akustická signalizace, zejména pro vzlétání a přistávání, a nebo signalizaci překážky. Ačkoli tato funkce nemá žádný vliv na samotný chod programu, je nezbytná pro uživatelskou bezpečnost, protože rotory kvadrokoptéry mohou snadno někoho zranit, a to i když jsou opatřeny ochrannou zástěnou. Signalizační funkce může být ovšem využita i pro rozšiřující moduly, je tedy lepší tuto funkci připravit, než ji pak integrovat do výše zmíněných funkcí nezávisle.

Detekce překážek v okolí – jednoduchá funkce, která má za cíl varování osob v okolí v případě, že se přibližují ke kvadrokoptéře. Funkce rozlišuje 3 zóny, a to bezpečnou, nebezpečnou a kritickou. Bezpečná zóna je do vzdálenosti 75 cm a kvadrokoptéra o přítomnosti objektu v této vzdálenosti informuje krátkým pípnutím. Nebezpečná zóna je v rozsahu 75-30 cm a je indikována dvěma dlouhými tóny. Kritická zóna je pro objekty bližší jak 30 cm indikována sérií kratších tónů a kvadrokoptéra dostává příkaz k odletu v protisměru objektu. V případě, že je kvadrokoptéra z obou stran uzavřena do nebezpečné zóny (například nacházela-li by se mezi člověkem a zdí), indikace se mění na jeden nepřetržitý tón, a kvadrokoptéra dostává pokyn k nouzovému vertikálnímu přistání.

5.3 Aktualizace stavového registru

Pro zajištění aktuálnosti informací je využito algoritmu navrženého v bodě 2.5, konkrétně metody využívající omezené prioritní fronty. Z důvodů zrychlení programu je však priorita řešena stárnutím bodu na periodické časové ose v modulární aritmetice nad datovým typem int8_t. Stáří záznamu je tedy určeno výpočtem (aktuální_čas – čas_záznamu) modulo 210, a hodinový signál (aktuální čas) je inkrementován s každým vysíláním s výpočtem modulo 210 po inkrementu. Není tedy potřeba inkrementovat priority jednotlivých signálů zvlášť. Zápis nového požadavku se používá v podobě (aktuální_čas – priorita) modulo 210 a požadavky se zapisují vždy, když dojde k odeslání hodnoty na základě jejího stáří. Existuje navíc ještě 25 priorit pro případ nouzových požadavků (hodnoty 220-244), které mohou být nastaveny pro přednostní vyřizování požadavků od modulů. Dále existují ještě rezervované priority o hodnotách 250-255, které nesmí využívat žádný modul, a jsou určeny ke zpracování požadavků jádra (například nouzové přistání nebo ověření stavu baterie při delším pozastavení hodinového signálu). Při využití prioritních požadavků je inkrementace hodinového signálu pozastavena a požadavky jsou vyřízeny na základě priority. Hlavním důvodem využití priorit je řešení krizových situací, jako například hrozící kolize. Kromě priorit ještě existuje rezervovaná hodnota 215, která je určena pro informaci o odmítnutí dalšího plánování. Nachází-li se položka ve stavu 215, není určena k periodickému dotazování (Do těchto stavů přechází například dotazy na jednotlivá čidla, která nejsou ve směru letu.). Pro zařazení nového dotazu do fronty existuje



Obrázek 5.2: Stavový diagram položek prioritní fronty

funkce plan_now, která zařadí požadovaný dotaz do fronty, a jako argument je možno přidat momentální prioritu (použití funkce plan_now(7, 10) tedy přidá 7. položku v tabulce dotazů do časové osy s offsetem stáří -10). Přechody mezi jednotlivými stavy položky v prioritní frontě jsou naznačeny v obrázku 5.2

Tento návrh aktualizace stavového registru je nezávislý na použité

přenosové technologii a spravedlivě řeší distribuci požadavků k přenosu dotazů/nastavení.

Kapitola 6

Rozbor možností navigace v prostoru

Existuje nepřeberné množství algoritmů pro navigaci v prostoru, většinou jsou ovšem navržené pro navigaci ve dvou rozměrech a může být tedy nutná jejich úprava. Situaci navíc komplikuje

6.1 Zpětná navigace

Pod pojmem zpětné navigace kvadrokoptéry na výchozí souřadnice můžeme rozumět dvě možnosti – buď pokus o přímý let k cíli, nebo backtracing. Obojí může mít své opodstatnění a různé výhody. Pokud bychom použili přímý let k cíli (a samozřejmě i v kombinaci s nějakou z chytřejších navigačních variant, aby bylo možné se vyhýbat překážkám, atd.), můžeme narazit na závažný problém v podobě překážky, které se nebude možno snadno vyhnout. V případě letu v otevřeném prostoru se dá předpokládat, že každou překážku bude možno obletět nebo přeletět, v případě letu v prostoru uzavřeném by ovšem tento problém mohl snadno vzniknout. Pro představu můžeme mít jednoduché bludiště tvaru U, kde kvadrokoptéra se nachází uvnitř U a cíl na spodní vnější straně. Navigace přímého letu by říkala, že kvadrokoptéra má letět nejkratší cestou, jenže ta je uzavřena. Takovéto bludiště snáze řeší metoda backtracingu, kdy si kvadrokoptéra uchovává informace o instrukcích dosavadního letu a v případě požadavku na vrácení na výchozí pozici jen tyto instrukce začne zpracovávat v opačném pořadí. Každý z těchto algoritmů může být v určité situaci výhodnější. V případě backtracingu přesně víme (za předpokladu, že se nemění okolí), jakým postupem se vrátit na výchozí pozici a můžeme předpokládat, že na návrat spotřebujeme přibližně stejně energie, jako jsme spotřebovali na postup do požadavku na návrat. Prakticky tato volba návratu dává možnost využít maximálně 50% baterie pro pohyb vpřed, a rezervovat 50% pro návrat (v praxi bude potřeba rezerva ještě o trochu větší,

protože spotřeba nemusí být lineárně úměrná po celou dobu letu a baterie se bude vybíjet rychleji, čím více bude prázdná). Metoda navigace přímého letu bude energeticky úspornější, protože můžeme-li odhadnout vzdálenost mezi aktuálním bodem a počátkem a známe-li spotřebu energie na základě vzdálenosti, můžeme rezervu dynamicky nastavit dle těchto požadavků na minimum. Z rozboru je patrno, že na základě různých situací může být vhodnější jedna nebo druhá metoda, a pokud neznáme okolí, není možné jednoznačně určit, která z metod bude výhodnější.

6.2 Nouzové přistání

K nouzovému přistání může docházet z mnoha různých důvodů, efekt je ale vždy stejný – kvadrokoptéra dostane požadavek na kolmé přistání, který realizuje na základě zjištěné výšky, a přistání vyhladí za pomoci spodního vzdálenostního čidla. Jak již bylo naznačeno v předešlé kapitole, takové přistání není zcela jednoznačné, může se totiž stát, že se kvadrokoptéra pokusí přistát v místě, které k přistání není vhodné, a protože vzdálenostní čidlo hlásí jen nejbližší překážku, není možné snadno rozpoznat tvar přistávací plochy. Na základě znalosti vysílacího rozptylu čidla můžeme ale provést analýzu plochy pod kvadrokoptérou, a to tím přesnější, čím blíže přistávací ploše kvadrokoptéra bude. Pokud se kvadrokoptéra bude postupně natáčet v osách X a Y, bude možné analyzovat terén pod kvadrokoptérou, viz nákres. Tímto postupem je možné poměrně nenáročně a rychle získat základní představu o přistávací ploše, takže by se tento přístup mohl vyplatit i v případě nouzového přistání. Dále je ještě možno detekovat, zda se kvadrokoptéra nepokouší přistát na nepřístupném místě, jako například v koruně stromů. Postup pro detekci by mohl být například takový, že kvadrokoptéra oblétne vybrané místo přistání v několika kruzích, a na základě tvaru místa můžeme předpokládat, že jestliže toto místo má kopulovitý tvar, mohlo by se jednat o strom, a mohlo by tedy být rozumné se pokusit přistát v nějakém vhodně zvoleném místě poblíž.

6.3 Vyhýbání se překážkám

Algoritmů k vyhýbání se překážkám je opět velké množství a jsou podrobně popsány na internetu, zmíníme tedy jen základní myšlenky algoritmů. Mezi ně určitě bude patřit wall-following, algoritmus určený k následování stěny, surrounding-conscious algoritmy, které se nejprve snaží dostupnými senzory zmapovat okolí a na jeho základě vybrat nejvhodnější cestu nebo map-oriented algoritmy, které na základě známé mapy můžou určit nejvhodnější cestu mezi překážkami. Základním problémem většiny algoritmů je ovšem fakt, že typicky řeší problémy v rovině, nikoli ve třídimenzionálním prostoru. Na první pohled by se mohlo zdát, že typicky dobré řešení bude překážku přeletět, než řešit

její oblétávání. Nejčastější překážkou v otevřené přírodě budou ovšem stromy, u nichž se přelétávání zpravidla nebude vyplácet a navíc k nalezení překážky dojde typicky v případě, kdy kvadrokoptéra už nemůže vzlétnout kvůli koruně stromu.

Kapitola 7

Testování

7.1 Testování programu

Vývoj celé aplikace probíhal na základě předpokládaných hodnot. Bohužel pro praktické testy a potřebné reálné odladění řídících funkcí chybí existující model kvadrokoptéry, na kterém by bylo možno testy realizovat. Veškeré funkce jsou z tohoto důvodu vybaveny podrobným logováním, aby při nasazení na skutečný model bylo možné snadno získávat data potřebná pro další optimalizaci a reflektování reálného letu do programu.

7.2 Návrh testování v praxi

Jakmile bude existovat reálný model kvadrokoptéry, bude možné vyzkoušet v praxi všechny vytvořené funkce. Test by měl probíhat metodicky, začít od nejjednodušších funkcí, a pokračovat ke složitějším a jejich kombinacím. Logické pořadí testů by mělo vypadat takto:

1. Otestovat roztočení rotorů na rychlost 1, a následně je vypnout. Ověřit logování hodnot stavu jádra.

2. Provést vzlétnutí bez kontroly výšky, pouze změnou rychlosti rotorů, a následně opět přistát stejným postupem. Ověřit hodnoty natočení kvadrokoptéry kvůli oscilacím / nepřesnostem / vibracím.

3. Vzlétnout do výšky 20 cm a opět přistát. Ověřit přesnost výšky na základě spodního čidla vzdálenosti.

4. Vzlétnout do výšky 2 m a opět přistát. Ověřit přesnost výšky na základě barometru.

5. Vzlétnout do výšky 0.5 m, ověřit funkci nouzového přistání.

6. Vzlétnout do výšky 0.5 m, nastavit směr na let dopředu, přistát. Ověřit správnost nastavení osy.
7. Vzlétnout do výšky 0.5 m, nastavit směr na let vpravo, přistát. Ověřit správnost nastavení osy.
8. Stanovit vzdálenost 20m vpřed, proletět letem vpřed ve výšce 0.5 m, změřit čas letu, ověřit přesnost výpočtu rychlosti.
9. Aktivovat kvadrokoptéru, vypnout po 10 minutách, ověřit správnost údajů ve struktuře stavu jádra (načtení GPS, měření baterie, atd).

Závěr

Protože během tvorby této práce neexistoval skutečný model, na kterém by bylo možno testovat realizaci, zaměřil ji spíše na přípravu teoretických podkladů, rozbor algoritmů a řídících struktur a studium alternativních možností spíše než na praktickou realizaci.

Operační systém jsem zvolil na základě výsledků dostupných v citovaných odkazech, nikoli z testů mnou provedených, a podle praktických zkušeností s nasazením a využitím operačního systému na Raspberry Pi. Jako nejlepší možnost se jeví operační systém Debian s jádrem opatřeným patchem Xenomai framework. Nepřesnost mnou provedených testů může být způsobena špatně nastaveným operačním systémem a nebo rozdíly mezi jednotlivými programy pro testování.

Navržené schéma propojení RPi s QC je dostatečně obecné, aby bylo možné jej nasadit na jakýkoli hardware. Schéma preferuje využití sériového portu pro komunikaci mezi RPi a QC, a připojení ostatních kritických periferií (např. WiFi) jiným způsobem.

Vytvořený protokol jsem v průběhu jeho návrhu konzultoval s Jakubem Halákem, který aktuálně vyrábí hardware kvadrokoptéry jako svou diplomovou práci, a který mi pomohl s ustanovením datových typů pro jednotlivé dotazy na základě jeho zkušeností s periferiemi.

Návrh a realizace jádra programu byla nejtěžší částí této práce. Jádro je nyní realizováno nejjednodušší cestou, aby se omezil potřebný výpočetní výkon k jeho udržování, a uvolnily se prostředky k implementaci rozšiřujících algoritmů. V návrhové části jsem se zabýval popisem alternativních řešení pro případ, že by se realizované části ukázaly jako nevhodné řešení.

Ukázková knihovna pro komunikaci mezi RPi a QC implementuje práci se sériovým portem, je založena na odděleném vlákně, a realizuje metody inicializace, spuštění, a zastavení (a současně zrušení). Úprava pro jiné komunikační médium spočívá pouze v úpravě těchto tří částí, a knihovna je tedy vhodným základem pro realizaci.

Druhou složitou částí v této bakalářské práci je analýza algoritmů

pro navigaci v prostoru. Běžně dostupné algoritmy se zabývají navigací v dvourozměrném prostoru, ale kvadrokoptéra se pohybuje v prostoru třírozměrném. Součástí analýzy dostupných řešení je tedy i upozornění na možné problémy, které mohou při implementaci jednotlivých algoritmů s ohledem na třídimenzionální prostor nastat.
Možná pokračování práce:

1. praktické testy a možná úprava řídících funkcí
2. realizace algoritmů pro navigaci v prostoru
3. rozšíření algoritmů pro navigaci v prostoru
4. tvorba řídícího GUI a návrh vhodného přenosového média
5. rozšíření periferií kvadrokoptéry o audio a videozáznam

Literatura

[1] Miki: Yaw, pitch, roll. 2013, file: ./images/axes.jpg. Dostupné z: http://mechatronicmiki.weebly.com/uploads/2/1/6/1/21613894/3161634_orig.jpg

[2] Wikipedia: Quadcopter -- Wikipedia, The Free Encyclopedia. 2014. Dostupné z: http://en.wikipedia.org/w/index.php?title=Quadcopter&oldid=606259311

[3] Wikipedia: Unmanned aerial vehicle -- Wikipedia, The Free Encyclopedia. 2014. Dostupné z: http://en.wikipedia.org/w/index.php?title=Unmanned_aerial_vehicle&oldid=608153796

[4] Copter, A.: ArduCopter | Multirotor UAV. 2014. Dostupné z: http://copter.ardupilot.com/

[5] Dr.G.O...@gmail.com: owenquad - Raspberry Pi based quadcopter with full walkthrough coding tutorial - Google Project Hosting. 2013. Dostupné z: https://code.google.com/p/owenquad/

[6] Štrauch, A.: Stavíme kvadrokoptéru s Raspberry Pi a Arduino Nano - Root.cz. 2013. Dostupné z: http://www.root.cz/clanky/stavime-kvadrokopteru-s-raspberry-pi-a-arduino-nano/

[7] Electronics, S.: Starlino Electronics. 2014. Dostupné z: http://www.starlino.com/

[8] vjaunet: vjaunet/QUADCOPTER. 2014. Dostupné z: https://github.com/vjaunet/QUADCOPTER

[9] big5824: big5824/Picopter. 2013. Dostupné z: https://github.com/big5824/Picopter

[10] Microsoft: Microsoft Windows - Microsoft Windows. 2014. Dostupné z: http://windows.microsoft.com/cs-CZ/windows/home

[11] Inc., A.: Apple - OS X Mavericks - Do even more with new apps and features. 2014. Dostupné z: https://www.apple.com/osx/

[12] Inc., A.: Debian - Univerzální operační systém. 2014. Dostupné z: http://www.debian.org/index.cs.html

[13] komunita, R. P.: Raspberry Pi. 2013. Dostupné z: http://www.raspberrypi.org/documentation/installation/installing-images/README.md

[14] dboling: Windows Embedded Compact BSP for Raspberry Pi - Home. 2013. Dostupné z: https://ceonpi.codeplex.com/

[15] chibios: ChibiOS/RT Homepage [ChibiOS/RT free embedded RTOS]. 2014. Dostupné z: http://www.chibios.org/dokuwiki/doku.php

[16] stevebate: Getting Started With ChibiOS/RT on the Raspberry Pi. 2013. Dostupné z: http://www.stevebate.net/chibios-rpi/GettingStarted.html

[17] Xenomai: Xenomai. 2013. Dostupné z: http://www.xenomai.org/

[18] elktros: Real Time Operating System - Hard RTOS and Soft RTOS. 2013. Dostupné z: http://www.electronicshub.org/real-time-operating-system-rtos/

[19] Dr. Jeremy H. Brown, B. M.: How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. 2012. Dostupné z: https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf

[20] Šumbera, I. P.: *Porovnání RT vlastností operačních systémů.* Dipomová práce, Ústav informatiky a výpočetní techniky FEL VUT, Brno, 2000.

[21] Šumbera, I. P.: Odborné časopisy - Porovnávací test operačních systémů reálného času. 2010. Dostupné z: http://www.odbornecasopisy.cz/index.php?id_document=27836

Příloha **A**

Seznam použitých zkratek

QC Kvadrokoptéra - bez počítače letové asistence

RPi Raspberry Pi - využito jako počítač letové asistence

GPOS General Purpose Operating System - Operační systém všeobecného využití

RTOS Real Time Operating System - Operační systém reálného času

RPM Revolutions per minute - Otáčky za minutu

OS Operating System - Operační systém

Příloha **B**

Obsah přiloženého CD

- `bakalarska_prace` . Adresář se zdrojovou formou práce ve formátu LaTeX
  - `chapters` ..... Adresář obsahující jednotlivé kapitoly práce
  - `images` ..... Adresář obsahující obrázky použité v práci
- `desky` ..... Adresář se zdrojovou formou desek ve formátu LaTeX
- `zdrojove_kody` ..... Zdrojové kódy implementace
- `BP_Kukacka_Jiri_2014.pdf` ..... Text práce ve formátu PDF
- `Makefile` ..... Makefile sloužící k automatickému kompilování zdrojových kódu a generování dokumentace, pro seznam podporovaných příkazů slouží make help
- `Doxyfile` ... Konfigurační soubor systému Doxygen sloužící ke generování dokumentace do adresáře dokumentace
- `ostatni` ..... Adresář obsahující blíže neurčená užitečná data, jako např. dokumentace OS Xenomai