České Vysoké Učení Technické v Praze

fakulta elektrotechnická, Katedra řídící techniky

Zpracování a vizualizace letových dat pro model vrtulníku

DIPLOMOVÁ PRÁCE

2009

Vypracoval: Bc. Petr Svoboda

Vedoucí práce: Ing. Ondřej Špinka

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Petr Svoboda**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Kybernetika a měření, blok KM1 - Řídicí technika

Název tématu: **Zpracování a vizualizace letových dat pro model vrtulníku**

Pokyny pro vypracování:

1. Prostudujte komunikační protokol telemetrie modelu vrtulníku a seznamte se se stávajícím programem pro zpracování letových dat.
2. Navrhněte a realizujte komplexní systém pro zpracování a vizualizaci letových dat z modelu vrtulníku s grafickým rozhraním.
3. Systém bude schopen pracovat on-line (vizualizace všech žádaných parametrů v reálném čase, zakreslování pozice do mapy) a bude schopen ukládat data v požadovaném formátu pro pozdější zpracování.
4. Realizovaný systém důkladně otestujte v praxi.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Ondřej Špinka

Platnost zadání: do konce zimního semestru 2009/10

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 27. 2. 2009

Abstrakt

Cílem této práce je návrh, implementace a otestování vizualizačního softwaru pro model vrtulníku. Data z vrtulníku přicházejí TCP spojením a software je zpracovává a ukládá do souboru. Letová data jsou následně zobrazována na časově závislých grafech a informačních panelech, virtuálních leteckých přístrojích a jako cesta na mapě.

Software je napsán v jazyce `C++` a grafické části využívají knihovnu Qt. Platformou softwaru je operační systém Linux. Celý systém byl zveřejněn pod licencí GNU GPL v3.

Abstract

Goals of this work are to design, implement and test visualization software for helicopter model. The data are received trought TCP connection and software process them and stores into a log file. The flight data are shown on time-flow charts and status bars, board instruments like gyro-horizont and as a path in map.

The software is written in `C++` language and graphic parts use Qt library. The main operation platform is Linux. Whole system was released under the GNU General Public License version 3.

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 **22.5.2009**

. . . . . .

(podpis autora)

Poděkování

Chtěl bych poděkovat všem lidem, bez nichž by má diplomová práce nemohla vzniknout. Děkuji především vedoucímu diplomové práce Ing. Ondřeji Špinkovi za jeho vedení i samotný podnět k této práci. Významný dík patří také Josefu Kufnerovi za odborné konzultace. V neposlední řadě děkuji rodině i přátelům za morální podporu.

1 Úvod

Autorovou osobní motivací pro vytvoření této práce je nadšení pro projekt RAMA - Otevřený řídicí systém pro bezpilotní létající prostředky (UAV[1]) [4].

Cílem práce je vytvořit „divácky atraktivní“ vizualizační software použitelný pro projekt RAMA. Zároveň však zachovat co největší univerzálnost systému, aby byl použitelný i pro jiné aplikace.

Požadavky na vizualizační software jsou

- Schopnost příjmu dat z telemetrie modelu vrtulníku.
- Vizualizace letových dat v reálném čase.
- Ukládání letových dat do souboru.
- Vysoká spolehlivost softwaru.

V rámci této práce byl software navržen, implementován i otestován. Platformou pro software je operační systém Linux. Grafická část je postavena na knihovně Qt, což umožňuje eventuální portovatelnost softwaru i na pro operační systém Windows.

V následující kapitole jsou popsány hlavní koncepční myšlenky. Dále pak jsou uvedeny všechny použité principy a implementační podrobnosti.

V příloze A je návod, jak systém z přiloženého CD uvést do provozu.

Obrázek 1.1: Obrázek bezpilotního vrtulníku v projektu RAMA.

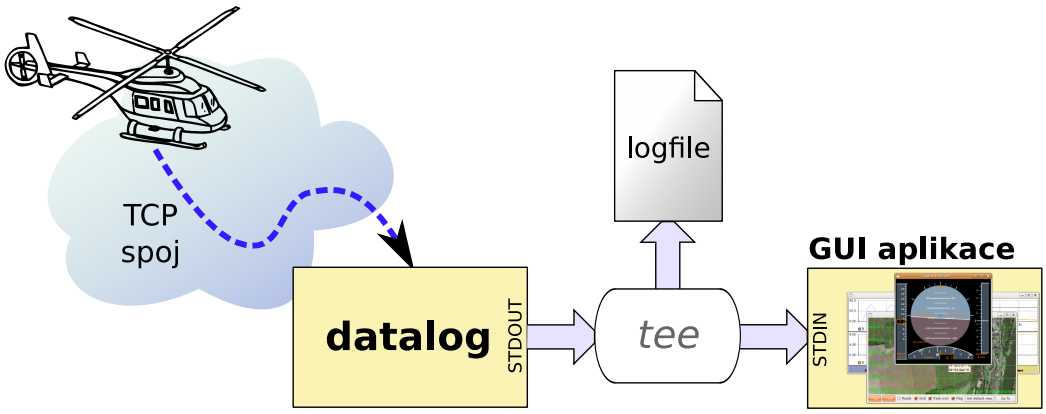
[1] Unmanned Aerial Vehicle

2 Koncepce

Základní koncepční myšlenkou je dekompozice celého systému na malé, samostatně pracující aplikace. Namísto jedné komplexní aplikace tak vznikne soubor utilit. Každá utilita pak řeší pouze izolovaný podproblém, zato však velice kvalitně. Modularita takového systému umožňuje snadnou rozšiřovatelnost, zaručuje flexibilitu použití[2] a v neposlední řadě i usnadňuje hledání a odstraňování chyb.

Samostatnou problematiku tvoří příjem dat prostřednictvím síťového TCP spojení a jejich interpretace dle popsaného protokolu. Grafické aplikace se tak mohou zabývat pouze vizualizací dat a interakcí s uživatelem.

Koncepce je schématicky znázorněna na obrázku 2.1. Program `datalog` přijímá data po síťovém socketu a vypisuje je na svůj standardní výstup. Ten je pak možné pomocí roury přesměrovat na vstup grafické aplikace, která se postará o zobrazení dat. Na obrázku je znázorněno použití standardní linuxové utility *tee*, čímž lze bezpracně získat funkcionalitu ukládání letových dat do souboru.



Obrázek 2.1: Koncepční schéma softwaru coby souboru utilit

Příklad další výhody modulárního systému je na obrázku 2.2. Zde je vidět opět bezpracný způsob off-line přehrávání letových dat ze souboru[3].

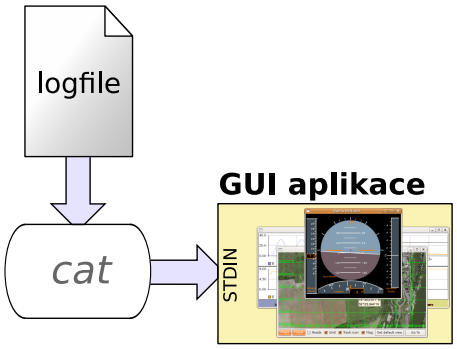
Pro vizualizaci letových dat byly zvoleny tři formy.

- Aplikací `graph` jako časově závislé grafy.
- Aplikací `horiznot` vyobrazení na virtuálních palubních přístrojích.
- Aplikací `map` jako poloha/cesta na mapě.

Ukázky grafických aplikací jsou na obrázku 2.3.

Aplikace při svém spuštění načtou veškeré nastavení z konfiguračního souboru. Pokud není řečeno jinak, pokusí se najít `*.conf` soubor stejného jména, jako je název aplikace.

[2] Celý systém je lépe konfigurovatelný, popřípadě jednotlivé utility naleznou uplatnění i samostatně.
[3] *cat* je standardní linuxová utilita, která čte soubor a jeho obsah posílá na svůj standardní výstup.

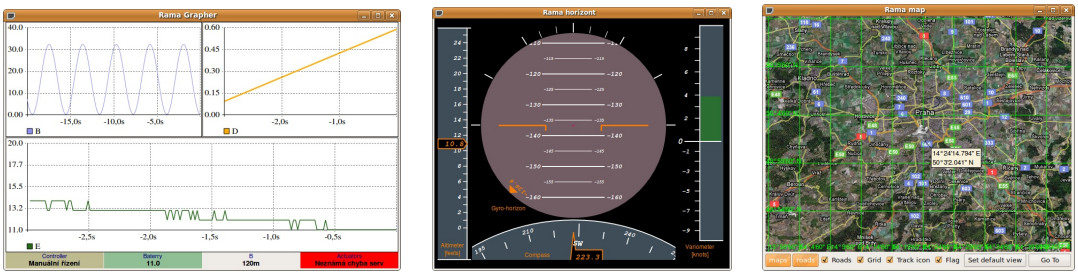


Obrázek 2.2: Demonstrace možností modulárního systému: přehrávání letových dat.

Jako jedna z voleb je tzv. *transparentní* mód - vše, co je přijato na standardním vstupu, je okamžitě vypsáno i na standardní výstup. To umožňuje použití všech utilit zřetězeně elegantním voláním:

```
./datalog | ./graph | ./horizont | ./map > letova_data.log
```

Výpis 2.1: Příklad spuštění systému utilit



a) b) c)

Obrázek 2.3: Ukázka grafických aplikací a) `graph`, b) `horiznot`, c) `map`

2.1 Popis síťového komunikačního protokolu telemetrie modelu vrtulníku

Síťový komunikační protokol byl pevně stanovený zadáním.

Jedná se o jednosměrný textový komunikační protokol napsaný v aplikační vrstvě. Komunikace probíhá skrze TCP/IP spojení. Tento protokol tedy řeší pouze problematiku formátu dat. Potvrzování zpráv či validita dat je obstarávána nižšími vrstvami.

Data se přenáší po zprávách vždy zahájených znakem `'@'` a zakončených znakem[4] `'\0'`.

Uvnitř zprávy jsou pouze textové znaky vyjadřující hexadecimální čísla. Tedy `0` až `9` a `a` až `f`, popřípadě `A` až `F`. První dvě hexadecimální číslice mají speciální význam.

[4]Znakem s ASCII hodnotou 0. Obdobně, jako textové řetězce v jazyce `C`.

Označují tzv. *ID*, unikátní identifikátor typu zprávy. Zbytek zprávy tvoří tělo, v kterém jsou zakódována data. Délka celé zprávy a obsah těla jsou předem stanoveny a závisí na ID. Příklad zprávy s ID 42 je na obrázku 2.4.

Obrázek 2.4: Příklad zprávy s ID 42.

Zprávy můžeme dělit na dvě skupiny:

- *Synchronní* zprávy, které přicházejí pravidelně s určitou časovou periodou. Tyto zprávy nesou vlastní telemetrická data.
- *Asynchronní* zprávy, které přicházejí nepravidelně. Tyto zprávy zpravidla nesou informace o chybách či jiných asynchronních událostech[5]. Tělo asynchronní zprávy může být i prázdné nebo obsahuje pouze jedinou hodnotu.

2.1.1 Kódování dat v těle zprávy

Server při sestavování zprávy nejprve poskládá všechny přenášené proměnné do jednoho pole bajtů. Poté toto pole převede do hexadecimální podoby a tím získá tělo zprávy.

Když zpětně dekódujeme tělo zprávy, musíme znát jaké proměnné obsahuje, jaká je jejich velikost v B i jaké je jejich zarovnání[6]

Server i klient proto musí být předem domluveni na ID zpráv a počtu, typu a zarovnání proměnných, které budou zprávami přenášeny. Tyto informace se komunikačním protokolem nepřenášejí[7].

Příklad zakódování zprávy[8] popsané ve výpisu 2.2 je na obrázku 2.5.

```
ID = 28 (dec.)
- speed, type: short, alignment: little endian
- direction, type: short, alignment: big endian
- altitude, type: float, alignment: little endian
```

Výpis 2.2: Příklad struktury zprávy

2.2 Popis meziprocesní textové komunikace

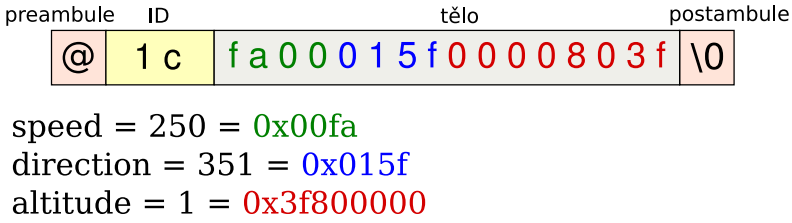
Meziprocesní textovou komunikací je myšleno to, v jakém formátu budou jednotlivé aplikace vypisovat data na svém standardním výstupu. Zároveň se jedná o popis formátu, v jakém budou aplikace schopné akceptovat data na svém standardním vstupu.

[5] Například přepnutí řízení vrtulníku na automatiku.

[6] Tím je myšleno, jestli jsou data v proměnných uložena ve formátu little nebo big endian.

[7] Na straně serveru jsou tyto informace napevno ve zdrojových kódech. Na straně klienta, tedy aplikace `datalog`, jsou tyto informace v konfiguračním souboru.

[8] Jedná se pouze o příklad zprávy. Telemetrie vrtulníku žádnou zprávu s ID 28 neposílá.



Obrázek 2.5: Příklad zakódování zprávy.

Protože se data do záložního souboru ukládají přímo ve formátu, v jakém je aplikace `datalog` vypisuje, vznikají tím požadavky na formátování dat. Výsledný soubor s daty musí být[9] jednoduše zpracovatelný v Matlabu.

Výše zmíněný požadavek vedl k stanovení následujícího formátu dat:

- Jednotlivé veličiny jsou zapsány do sloupců oddělených tabulátory.
- Řádky souboru reprezentují jednotlivé časové okamžiky. V posledním řádku jsou vždy nejaktuálnější data.
- První řádek souboru tvoří *hlavička*. Ta začíná znaky `%%` a dále obsahuje číslo a jméno každého sloupce. Čísla jsou vždy uvedena v kulatých závorkách před názvem a jednotlivé sloupce jsou od sebe odděleny tabulátorem.
- Jakákoli další informace, která nepatří do dat, je vždy na samostatném řádku a uvedena znakem `%`. Takto jsou od datového toku zařazovány asynchronní zprávy. Každá asynchronní zpráva začíná znaky `%ASYNC(`, poté následuje ID zprávy, poté zpětné lomítko `/`, dále hodnota z těla zprávy, znaky `):` a nakonec textová reprezentace asynchronní zprávy.

Datový soubor v takovémto formátu se dá načíst do Matlabu jako dvojrozměrné pole hodnot. Všechny řádky začínající znakem `%` bere Matlab jako komentáře, a proto jsou ignorovány. Ukázka meziprocesní textové komunikace je na výpisu 2.3.

```
%%(1)sample_number  (2)battery_voltage  (3)latitude (4)longitude
896 12.73910    0.8734893945014739  0.2513599464293202
897 12.37410    0.8734893913353606  0.2513607992865035
898 12.01330    0.8734893886221152  0.2513615934561571
899 11.65692    0.8734893859095488  0.2513624607550316
900 11.30520    0.8734893836149947  0.2513632672588618
901 10.95835    0.8734893813560431  0.251364146831632
%ASYNC(17/1):Servo unit error
902 10.61659    0.8734893793329942  0.2513650319197129
903 10.28013    0.8734893776749574  0.2513658533019912
```

Výpis 2.3: Ukázka textové meziprocesní komunikace

[9] Dle požadavků zadavatele.

2.3 Výběr programovacího jazyka a grafické knihovny

Výběr grafické knihovny, potažmo programovacího jazyka, v kterém jsou jednotlivé aplikace naprogramovány, se řídil následujícími požadavky (uvedeny dle důležitosti):

1. Operační systém Linux jako cílová platforma.
2. Perspektivita grafické knihovny do budoucna.
3. Výkonnost a HW nenáročnost.

První podmínka vyřazuje ze hry .NET framework a jazyk `C#`, nebo třeba `Object Pascal` a vývojové prostředí Delphi. Poslední podmínka vylučuje interpretované jazyky jako `Java`, `(Visual) Basic`, nebo `Python`.

Z portfolia běžných programovacích jazyků zbývá tedy `C/C++`. S přihlédnutím na druhou podmínku vycházejí dvě možnosti:

- C, GTK+
- C++, Qt

Používání GTK+ má v poslední době[10] lehce sestupnou tendenci. Naopak knihovna Qt je na vzestupu a má velice kvalitní dokumentaci, viz [1]. Proto volba padla na toolkit Qt a programovací jazyk `C++`.

V následujícím textu budou používány pojmy související s programovacím jazykem `C++` a knihovnou Qt, jako například *třída*, *template*, *widget* či *signál-slotový* mechanismus. Potřebná terminologie je uvedena v [1] a [2].

[10] Polovina roku 2008 až začátek roku 2009. Jedná se spíše o osobní dojem autora, než podložený průzkum.

3 Programové moduly

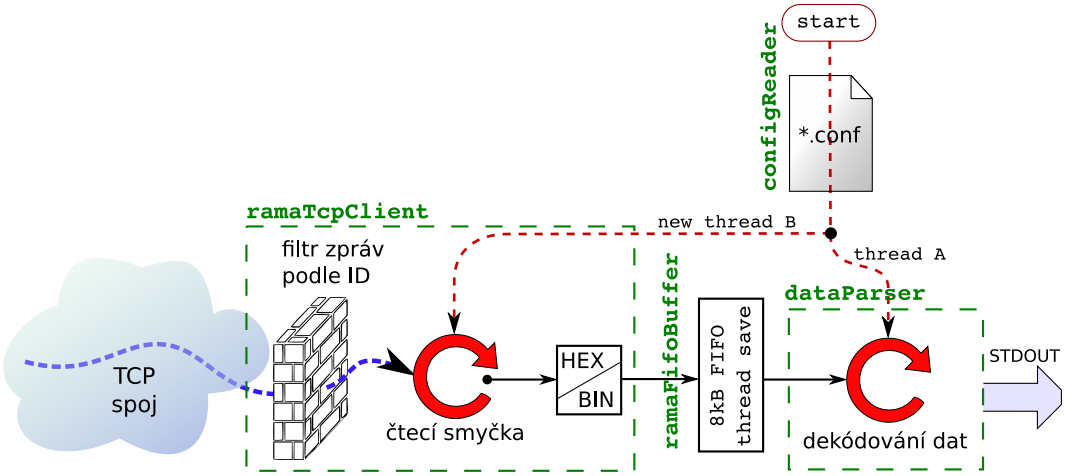
V této kapitole budou do detailu popsány jednotlivé apliace. Jmenovitě `datalog`, `graph`, `horizont`, `map` a podpůrné utility `concater`, `atee` a `gct`.

3.1 Aplikace `datalog`

Účelem aplikace `datalog` je příjem dat prostřednictvím síťového TCP spojení (viz kapitola 2.1) a jejich výpis na standardní výstup ve formátu popsaném v kapitole 2.2. Protože `datalog` nemá grafický interface, je napsán v čistém `C++`, bez použití grafické knihovny Qt.

Vnitřní struktura programu `datalog` je na obrázku 3.1. Zeleně jsou označeny názvy tříd, červeně cesta vykonávání programu. Nejprve program načte nastavení z konfiguračního souboru. Pokud vše proběhne v pořádku, vytvoří se druhé vlákno, které se připojí na server a skončí ve čtecí smyčce třídy `ramaTcpClient`. Původní vlákno skončí ve smyčce dekódování dat třídy `dataParser`.

Přijdou-li data ze serveru, jsou vyhodnocena ve čtecí smyčce třídy `ramaTcpClient`. Je-li přijata celá zpráva a má správné ID i délku, je v kontextu vlákna B zavolána callback funkce nesoucí data nové zprávy. Jako tato callback funkce je zaregistrována metoda `write` třídy `ramaFifoBuffer`, která tvoří komunikační kanál mezi oběma vlákny. „Z druhé strany“ čeká vlákno A na blokujícím metodě `waitRead` třídy `ramaFifoBuffer`. Zápisem do FIFO bufferu se dekódovací smyčka vlákna A uvolní, příchozí data dekóduje a vypíše je na standardní výstup.



Obrázek 3.1: Vnitřní funkce programu `datalog`

Program `datalog` je koncepčně navržen tak, že je schopen přijímat právě jednu synchronní zprávu[11]. Předpokládejme, že by tomu tak nebylo. Řekněme, že `datalog` by přijímal dvě synchronní zprávy. Kupříkladu zprávu α s periodou $0.1\,s$ a zprávu β s periodou $5\,s$. Vyvstalo by tak mnoho otázek.

[11] Asynchronních zpráv může přijímat libovolný počet.

- Kolik řádků má mít výstup? Tolik, kolik přišlo zpráv α, nebo tolik, kolik přišlo zpráv β?
- Jak má v takovém případě vypadat hlavička výstupu (první řádek)? Má obsahovat pouze sloupce odpovídající datům ve zprávě α, nebo β, či spojení obou?
- Pokud jsou na jednom řádku data z obou zpráv, co se má vypisovat, přišla-li zpráva α, ale zpráva β ještě nedorazila? Má se s výpisem celého řádku čekat, než dorazí obě zprávy? Nebo vypsat řádek okamžitě a místo neznámých dat vypsat `NaN`? Nebo 0? Nebo poslední známou hodnotu?

Tato problematika je obecně obtížně řešitelná a hlavně, aplikaci `datalog` ani nenáleží ji řešit.

Pokud je třeba přijímat více synchronních zpráv, musí se pro každou zprávu vytvořit instance programu `datalog`. Vznikne tak několik oddělených datových proudů. Je pak na uživateli, jakým způsobem a zda-li vůbec data sloučí.

V souvislosti s přijímanými zprávami byly zavedeny tři čítače: *accepted*, *ignored* a *trash*. Každý z těchto čítačů si pamatuje číslo odpovídající počtu přijatých bytů. Pokud vlákno B v pořádku přijme zprávu a rozpozná její ID[12], navýší čítač *accepted* o velikost zprávy. Zprávy, které nejsou přijaty v pořádku (například mají nesprávnou délku), nebo mají neznámé ID, jsou označeny jako *trash*. Konečně zprávy, které jsou přijaty v pořádku, ale jejich ID se má ignorovat[13], jsou připočteny do čítače *ignored*. Na konci programu jsou obsahy jednotlivých čítačů vypisovány.

Seznam všech tříd aplikace `datalog` a jejich stručný popis je v tabulce 3.1.

Název třídy	**Stručný popis**
`configReader`	Načítá konfiguraci z `.conf` souboru.
`ramaTcpClient`	Realizuje příjem dat skrze síťové spojení.
`ramaFifoBuffer`	Zajišťuje mezivláknovou komunikaci a bufferuje data.
`dataParser`	Dekóduje data z těla zpráv a vypisuje je na standardní výstup.

Tabulka 3.1: Seznam všech tříd programu `datalog` a jejich stručný popis.

[12] ID bylo deklarováno v konfiguračním souboru.
[13] Seznam ignorovaných ID se nastavuje v konfiguračním souboru.

3.1.1 Třída configReader

Třída configReader obstarává načtení informací z konfiguračního souboru. Jedná se především o popis zpráv, které bude program datalog přijímat. Dále pak i informace typu: IP adresa serveru, port, velikost přijímacího bufferu ...

Příklad konfiguračního souboru je na výpisu 3.1. Jen podotkněme, že v následujícím textu bude pod pojmem „data packet“ myšlena synchronně přicházející zpráva.

```
IP_ADDRESS=192.168.100.1
PORT=9735
# declaration of synchronous message, also called "data packet"
DATA_PACKET=0x35
 uint32 L sample_num
 uint8 L servo_mask
 float L roll
 float B pitch
DATA_PACKET_END
ASYNC_PACKET=0x10 uint8 L Controller
 0 Manual
 255 Automatic
ASYNC_PACKET_END
ASYNC_PACKET=0x04 uint8 L GPS
 -1 GPS error!
ASYNC_PACKET_END
IGNORE 0x64 0x40
```

Výpis 3.1: Příklad konfiguračního souboru aplikace datalog

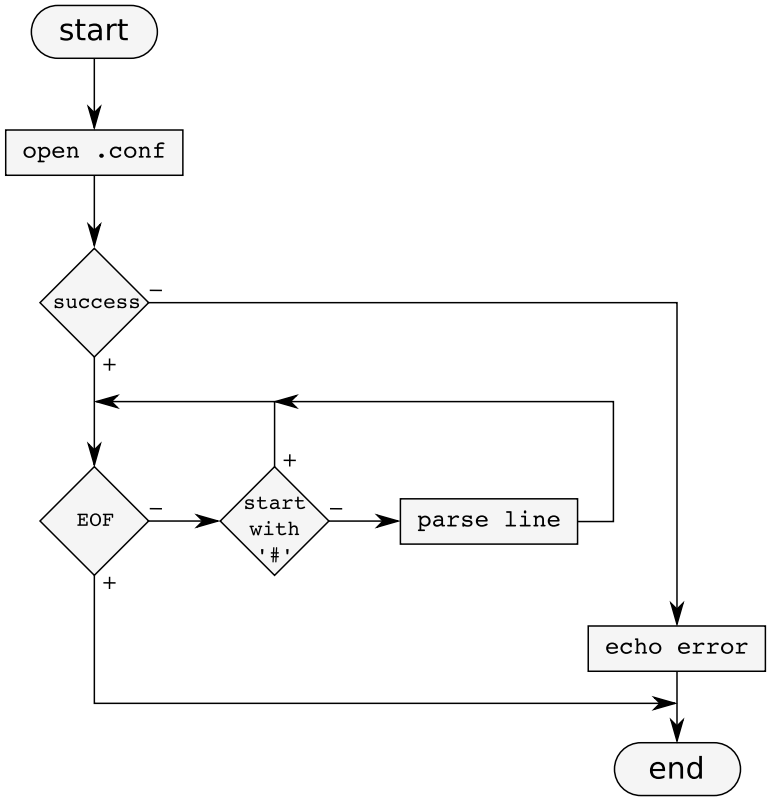
Zjednodušená deklarace[14] třídy configReader je na výpisu 3.2.

```
class configReader
{
    public:
        configReader(const char *filename, int verbose = 1);
        int getDataPacketNumber();
        vector<DataPacketRecord> getDataPacketRecods();
        vector<AsyncPacket> getAsyncPackets();
        vector<int> getIgnoreList();
    private:
        void parseLine();
};
```

Výpis 3.2: Zjednodušená deklarace třídy configReader

[14]Deklarace obsahuje pouze nejpodstatnější položky a proto se liší od deklarace ve zdrojových kódech.

V konstruktoru třídy `configReader` se provede otevření konfiguračního souboru a jeho zpracování po řádcích metodou `parseLine`. Ještě předtím se ale zkontroluje, jestli se nejedná o řádek s komentářem. Celá situace je vidět na obrázku 3.2.



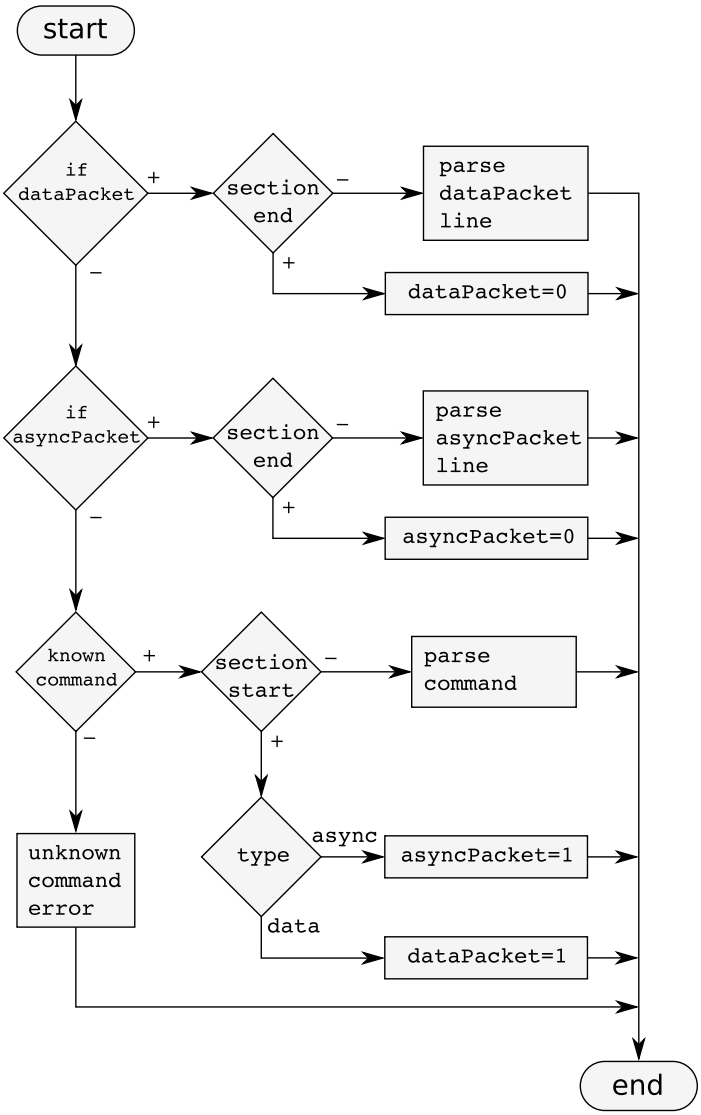
Obrázek 3.2: Zpracování konfiguračního souboru v konstruktoru třídy `configReader`.

Metoda `parseLine` vyhodnocuje obsah aktuálně zpracovávaného řádku. Algoritmus je znázorněn na obrázku 3.3. Jedná se prakticky o stavový automat, který má tři stavy:

- Zpracování obecného konfiguračního řádku. Tím jsou myšlena klíčová slova jako `IP_ADDRESS=`, `PORT=`, `DATA_PACKET=` nebo třeba `IGNORE`.
- Zpracování řádku uvnitř deklarace data packetu (synchronní zprávy).
- Zpracování řádku uvnitř deklarace asynchronní zprávy.

Metoda `parseLine` není příliš zajímavá. Jedná se spíše o poměrně dlouhou pasáž podmínek, testování textových řetězců na obsah klíčových slov, načítání čísel z textových řetězců ...

Pro uložení informací o přijímaných zprávách byly vytvořeny speciální struktury, které jsou vidět na výpisech 3.3, 3.4 a 3.5.



Obrázek 3.3: Zpracování jednoho řádku konfiguračního souboru metodou `parseLine`.

```
struct DataPacketRecord
{
    string DataName;
    int DataTypeCode;
    int DataTypeLength;
    int LittleEndian;
    int Ignore;
};
```

Výpis 3.3: Struktura pro uložení informací o položce v těle datového packetu.

Struktura `DataPacketRecord` odpovídá jedné datové položce v těle synchronní zprávy. Nalezneme v ní počet bytů použitých pro uložení této položky, datový typ položky a zarovnání. Krom toho také název položky, který se v budoucnu použije při výpisu hlavičky, a příznak `Ignore`. Pokud je tento příznak nastaven, bude při zpracování těla zprávy přeskočen příslušný počet bytů. Tato volba nalezne uplatnění, pokud tělo zprávy obsahuje mezery - byty jež nenesou žádnou informaci, pouze tvoří rezervu pro budoucí použití.

```
struct AsyncPacket
{
    int identificator;
    int DataTypeCode;
    int DataTypeLength;
    int LittleEndian;
    string Message;
    vector<AsyncSubPacket> aSubP;
};
```

Výpis 3.4: Struktura pro uložení informací o příchozí asynchronní zprávě

```
struct AsyncSubPacket
{
    int64_t subIdentificator;
    string Message;
};
```

Výpis 3.5: Struktura popisující možný obsah asynchronní zprávy

Každé asynchronní zprávě odpovídá jedna instance struktury `AsyncPacket`. Tělo asynchronní zprávy smí obsahovat právě jednu položku. Proto jsou informace o datovém typu a zarovnání dat v těle zprávy obsaženy přímo v této struktuře.

Při příjmu asynchronní zprávy chceme, aby se na výstup vypsalo textové hlášení v závislosti na hodnotě datové položky v těle zprávy. To je patrné z výpisu konfiguračního souboru 3.1. Například, pokud přijde asynchronní packet s ID 16 (0x10) a v jeho těle je hodnota 0, na výstup je třeba vypsat následující asynchronní hlášení:

```
%ASYNC(16/0):Manual
```

Výpis 3.6: Příklad výpisu asynchronní zprávy na výstup

Proto struktura `AsyncPacket` dále obsahuje seznam struktur `AsyncSubPacket`.

Po úspěšném načtení konfiguračního souboru můžeme získat seznam datových položek v těle synchronní zprávy metodou `getDataPacketRecods`. Obdobně pak můžeme získat seznam asynchronních zpráv metodou `getAsyncPackets` a seznam ID zpráv určených k ignorování metodou `getIgnoreList`.

3.1.2 Třída ramaTcpClient

Třída ramaTcpClient má za úkol navázat síťové spojení se serverem a pokud uspěje, vytvořit vlákno vyhrazené k příjmu dat skrze síťové spojení.

Zjednodušená deklarace třídy ramaTcpClient je na výpisu 3.7.

```
class ramaTcpClient
{
    public:
        ramaTcpClient(configReader *cr);
        int connectToServer();
        int registerCallBackFunction(void(*func)(int, unsigned char*, unsigned int));
    private:
        vector<msgIdAndLens> msgs;
        pthread_t listener_thread;
};
```

Výpis 3.7: Zjednodušená deklarace třídy ramaTcpClient

Konstruktoru třídy ramaTcpClient je jako parametr předávána konfigurace - objekt configReader. z toho objektu si ramaTcpClient načte adresu a port serveru a dále seznam přijímaných zpráv. Protože třída ramaTcpClient neinterpretuje těla zpráv, nepotřebuje znát jeji strukturu. Není ani třeba odlišovat, jedná-li se o synchronní či asynchronní zprávu. Je pouze potřeba rozhodnout, jestli zprávu akceptovat a poslat dál, nebo ignorovat, či označit za *trash*. Pro tento účel si ramaTcpClient vytvoří seznam struktur msgIdAndLens. Struktura msgIdAndLens je na výpisu 3.8.

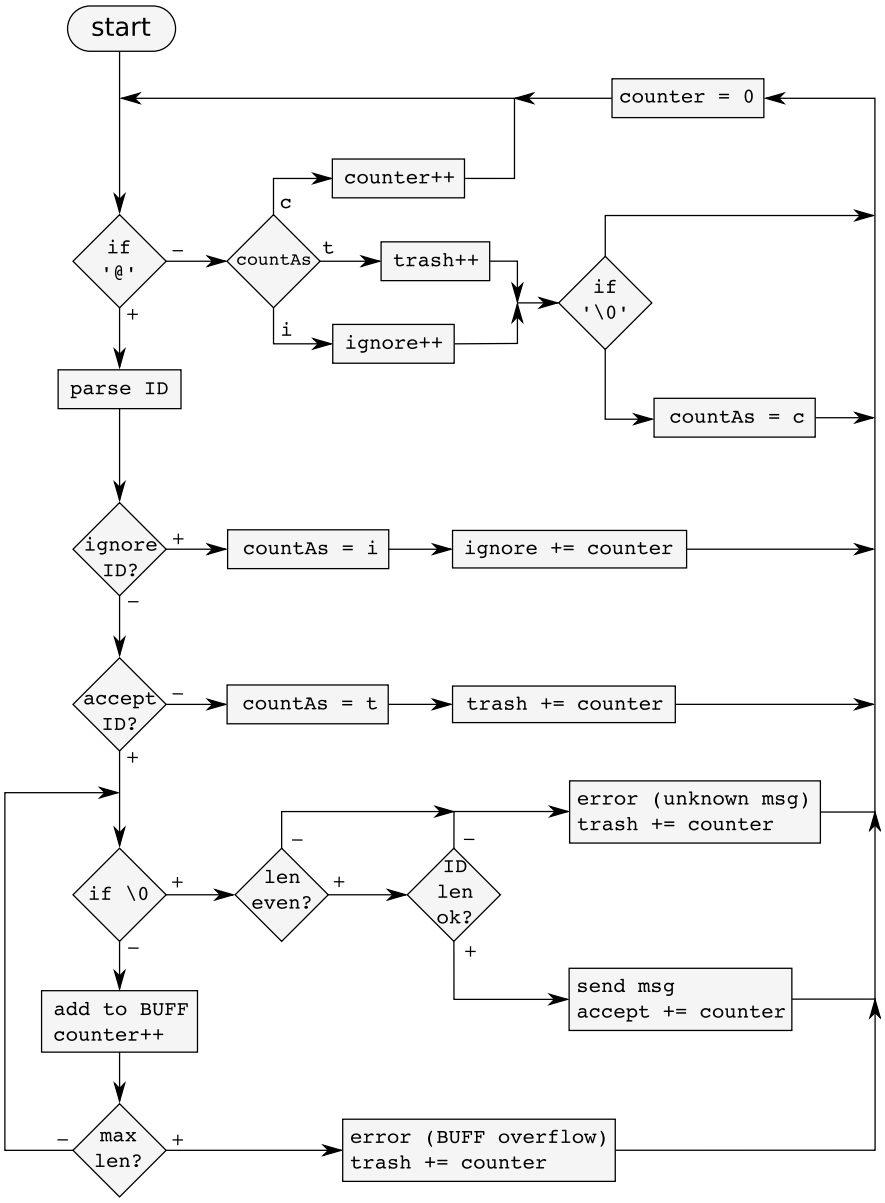
```
struct msgIdAndLens
{
    int msgID;
    bool ignore;
    unsigned int minLen, maxLen;
};
```

Výpis 3.8: Zjednodušená deklarace struktury msgIdAndLens

Pro připojení k serveru slouží metoda connectToServer. Tato metoda po úspěšném navázání spojení vytvoří přijímací vlákno, které vykonává čtecí smyčku na obrázku 3.4.

Ve čtecí smyčce se očekává přijetí znaku '@'. Dokud se tak nestane, je pouze zvyšována hodnota v čítači counter, případně ignore či trash. Po přijetí znaku '@', jsou zpracovány první dva byty, které obsahují ID zprávy. Pokud je zpráva určena ke zpracování, je ukládána do kruhového bufferu, dokud není obdržen znak \0, který označuje konec zprávy. Nakonec je provedeno ověření, jestli má tělo zprávy správnou délku[15]

[15] Délka těla například nesmí být liché číslo. Protože je tělo v hexadecimálním textovém kódování, každému bytu odpovídají právě dva hexadecimální znaky.

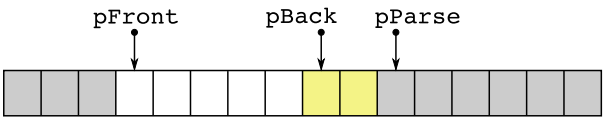


Obrázek 3.4: Algoritmus čtecí smyčky třídy `ramaTcpClient`

(v závislosti na ID). Pokud ano, je zpráva odeslána k dalšímu zpracování. Pokud ne, je označena za *trash*.

Uživatel si může definovat v konfiguračním souboru čtecí *timeout*. Pokud do stanoveného času není přijata synchronní zpráva, třída `ramaTcpClient` uměle vygeneruje tzv. *prázdnou zprávu* a odešle ji k dalšímu zpracování. *Prázdná zpráva* má ID 0 a tělo nulové délky. Na výstup bude místo každé datové položky vypsán `NaN` (znak pro *Not a Number*).

Kruhový buffer, který používá třída `ramaTcpClient` k dočasnému ukládání příchozích dat, je na obrázku 3.5. Buffer je použit z důvodu možné fragmentace příchozích zpráv. Hlavní myšlenkou je akumulovat příchozí data do doby, než bude celá zpráva k dispozici a teprve poté pokračovat v jejím zpracování.



Obrázek 3.5: Kruhový buffer používaný třídou `ramaTcpClient`

Při práci s bufferem na obrázku 3.5 jsou použity tři ukazatele: `pBack`, `pParse` a `pFront`. Ukazatele `pBack` a `pFront` vymezují v bufferu místo použité k uložení právě příchozí zprávy. Ukazatele `pParse` a `pFront` vymezují tělo zprávy.

Po přijetí celé zprávy a zdárném ověření její délky, je tělo zprávy převedeno z hexadecimálního textového kódování do přímé binární formy. Pokud je metodou `registerCallBackFunction` třídy `ramaTcpClient` zaregistrována callback funkce, provede se její zavolání. Jako parametry funkce je předloženo ID zprávy, tělo zprávy a délka těla zprávy. Operace převedení těla zprávy do přímé binární formy a zavolání callback funkce jsou ve vývojovém diagramu na obrázku 3.4 souhrnně označeny heslem `send msg`.

3.1.3 Třída `ramaFifoBuffer`

Úkolem třídy `ramaFifoBuffer` je obstarávat komunikaci mezi oběma vlákny aplikace `datalog`. Zjednodušená deklarace třídy je na výpisu 3.9.

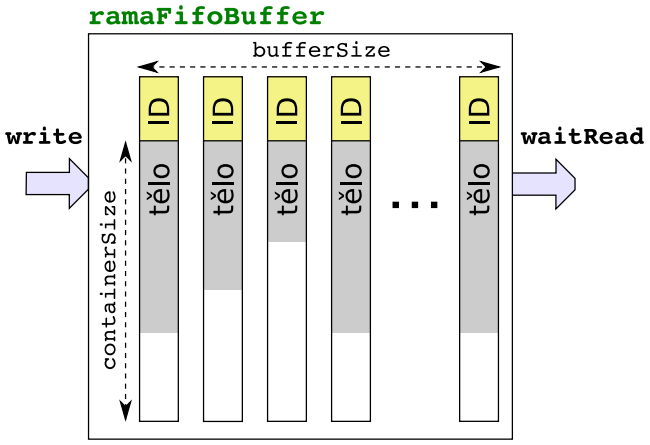
```
class ramaFifoBuffer
{
    public:
        ramaFifoBuffer(unsigned int containerSize, unsigned int bufferSize);
        unsigned int write(int mesID, unsigned char *data, unsigned int dataSize);
        unsigned int waitRead(int *mesID, unsigned char *data, unsigned int dataSize);
    private:
        msgContainer** buf;
        volatile unsigned int ifront, iback;
        sem_t readyToRead;
};
```

Výpis 3.9: Zjednodušená deklarace třídy `ramaFifoBuffer`

Třída `ramaFifoBuffer` nerealizuje pouhý FIFO buffer. Byla napsána tak, aby měla následující vlastnosti:

- Metody pro zápis a čtení se mohou volat z kontextu dvou různých vláken. Metody však nejsou reentrantní. Při vícenásobném volání metody pro zápis nebo čtení dojde k serializaci.
- Metoda `write` je neblokující. Při neúspěchu pouze vrátí nulu. Oproti tomu metoda `waitRead` je blokující v tom případě, že buffer neobsahuje žádná data.

- Zápis i čtení probíhá „po zprávách“. Přečíst či zapsat nelze libovolný objem dat, ale právě jednu zprávu - tedy ID a tělo. Tato vlastnost je dobře patrná z obrázku 3.6.



Obrázek 3.6: Použití třídy `ramaFifoBuffer`

Jako parametr konstruktoru třídy `ramaFifoBuffer` zadáváme maximální velikost těla zprávy a počet zpráv, které se dají do bufferu uložit. Toto není omezující, protože v okamžiku volání konstruktoru `ramaFifoBuffer` máme k dispozici informace z konfiguračního souboru. Je tedy možné přesně určit, jak dlouhé bude tělo nejdelší zprávy.

Vnitřně třída `ramaFifoBuffer` uchovává informace v kontejnerech - strukturách `msgContainer`, viz výpis 3.10.

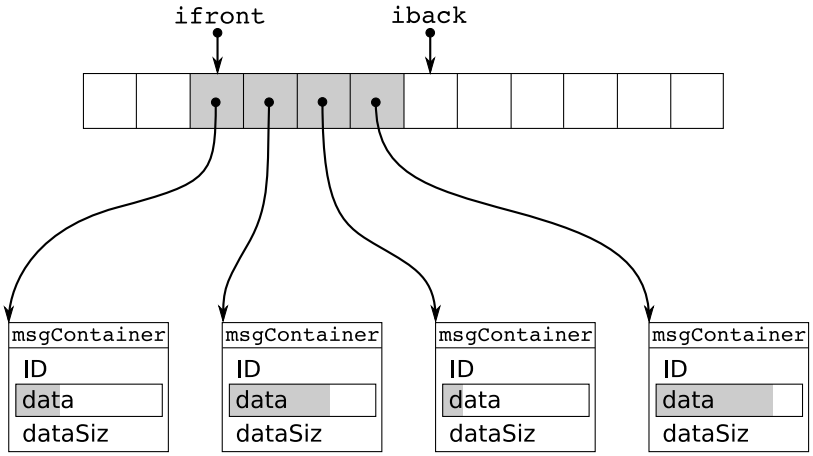
```
struct msgContainer
{
    int msgId;
    unsigned int dataSiz;
    unsigned char *dat;
};
```

Výpis 3.10: Zjednodušená deklarace kontejneru `msgContainer`

V konstruktoru je vytvořeno pole ukazatelů na tyto struktury stejně tak, jako je inicializován příslušný počet struktur. Třída `ramaFifoBuffer` obsahuje dva ukazatele[16] `ifront` a `iback`, které používá k indexování v poli ukazatelů na kontejnery. Situace je znázorněna na obrázku 3.7.

Funkcionalitu blokujícího módu metody `waitRead` zajišťuje Posixový semafor.

[16] Ve skutečnosti se jedná pouze o indexery pro pole ukazatelů na kontejnery, nikoliv o ukazatele ve smyslu jazyka `C/C++`.



Obrázek 3.7: Vnitřní struktura třídy `ramaFifoBuffer`

3.1.4 Třída `dataParser`

Třída `dataParser` přijímá zprávy od objektu `ramaTcpClient`, dekóduje jejich těla a vypisuje dekódovaná data na standardní výstup. Zjednodušená deklarace třídy `dataParser` je na výpisu 3.11.

```
class dataParser
{
    public:
        dataParser(configReader *configFile, char valuesSeparator);
        void parseAndEcho(ostream &o);
        void echoHeader(ostream &o);
    private:
        vector<DataPacketRecord> dr;
        vector<AsyncPacket> ap;
};
```

Výpis 3.11: Zjednodušená deklarace třídy `dataParser`

Jako první parametr konstruktoru třídy `dataParser` je předáván objekt s konfigurací, z kterého je načtena struktura příchozích zpráv. Druhý parametr konstruktoru je znak, jímž budou datové hodnoty při výpisu na výstup odděleny. Pokud není řečeno jinak, je použit tabulátor jako oddělovač.

Použití třídy `dataParser` spočívá v zavolání metody `echoHeader` a následně v cyklickém voláním metody `parseAndEcho`.

Metoda `echoHeader` vypíše na výstup jeden řádek začínající znaky `%%` a obsahující jména všech datových položek v těle synchronní zprávy tak, jak jsou uvedeny v konfiguračním souboru.

Algoritmus uvnitř metody `parseAndEcho` je na obrázku 3.8. Jako první je volána metoda blokujícího čtení `waitRead` z mezivláknového FIFO bufferu. Dále se rozhodne,

jedná-li se o *prázdnou zprávu*, synchronní zprávu, nebo asynchronní zprávu[17].

V případě prázdné zprávy se vypíše tolik hodnot `NaN`, kolik datových položek[18] obsahuje tělo zprávy.

V případě příjmu zbylých dvou typů zprávy je zpracováno jejich tělo. K tomu je použita struktura z výpisu 3.12.

```
struct dataValue
{
    int DataTypeCode, DataTypeLen, LittleEndian;
    union
    {
        uint8_t t0;
        uint16_t t1;
        uint32_t t2;
        uint64_t t3;
        int8_t t4;
        int16_t t5;
        int32_t t6;
        int64_t t7;
        float t8;
        double t9;
    };
    dataValue (const AsyncPacket & ap);
    dataValue (const DataPacketRecord & dp);
    void loadFromBuf(const unsigned char *buf, int siz);
    friend ostream & operator << ( ostream & out, const dataValue &a );
};
```

Výpis 3.12: Zjednodušená deklarace struktury `dataValue`

Struktura `dataValue` umí načíst informace z `AsyncPacket` nebo `DataPacketRecord`. Na základě načtených informací je možno zpracovat část těla zprávy. K tomu slouží metoda `loadFromBuf`. Ta, jednoduše řečeno, „ukousne“ z těla zprávy příslušný počet bytů a ty přetypuje (s ohledem na zarovnání) na odpovídající datový typ (t0 $\sim$ t9). Strukturu `dataValue` je potom možné vypsat pomocí přetypovaného operátoru `<<`. Při výpisu se zohledňuje aktuální datový typ[19].

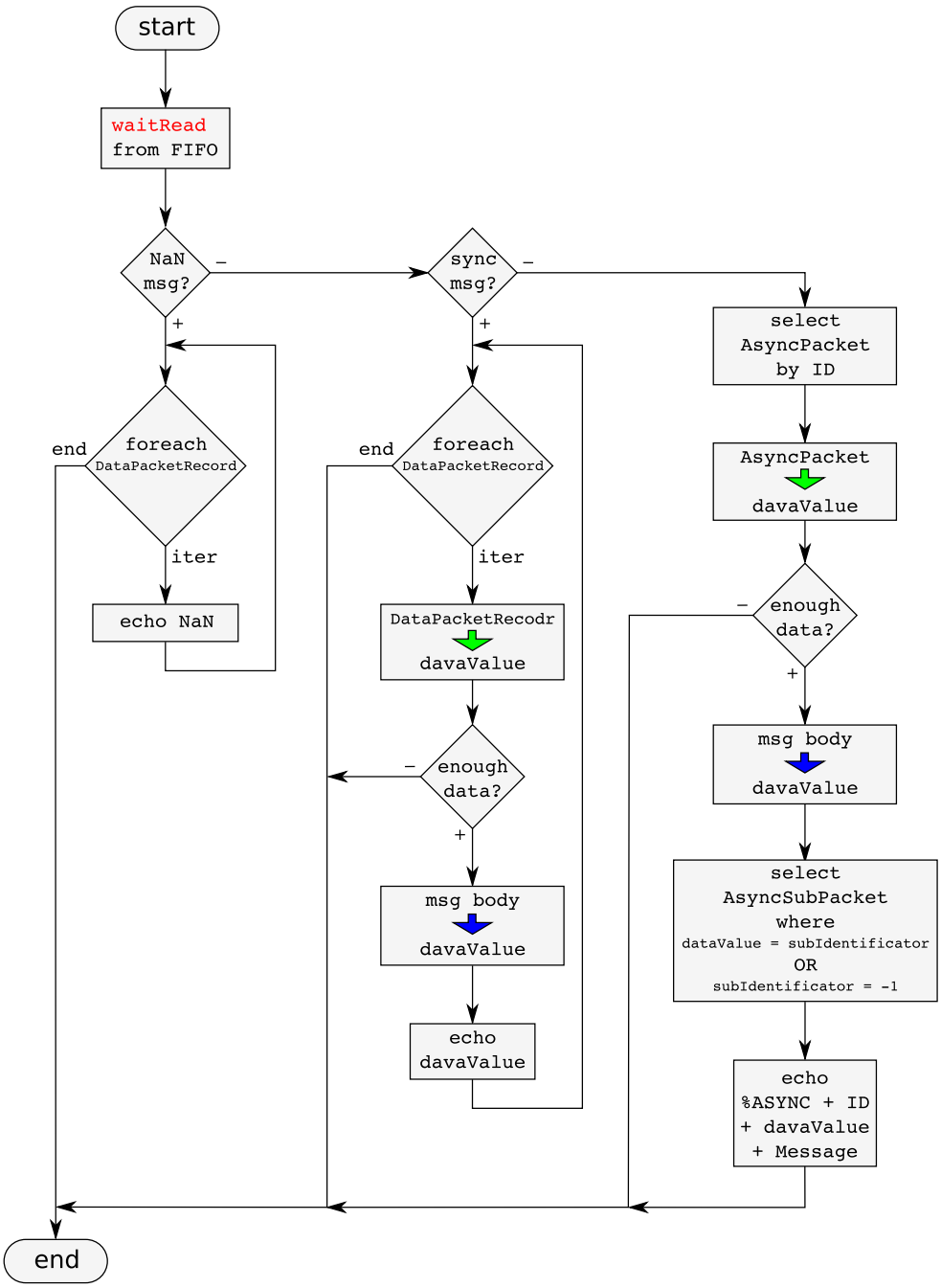
Při zpracování synchronní zprávy je iterováno přes celý seznam datových položek těla zprávy a s využitím struktury `dataValue` jsou data z těla zprávy načtena a následně vypsána.

[17] Pokud zpráva není prázdná, nebo synchronní, musí se jednat o asynchronní. Objekt `ramaTcpClient` zasílá puze tyto tři typy zpráv.

[18] Datových položek bez příznaku ignore.

[19] Například `t9` sevypisuje na větší počet desetinných míst tak, aby byla zachována přesnost.

Při zpracování asynchronní zprávy se načte pouze jedna položka[20] z těla zprávy. Poté se porovná hodnota této položky se seznamem struktur `AsyncSubPacket`. Pokud je nalezena shoda, tak se při výpisu asynchronní zprávy připojí textový řetězec z těla struktury `AsyncSubPacket`.



Obrázek 3.8: Smyčka zpracování zpráv - metoda `parseAndEcho` třídy `ramaFifoBuffer`

[20] Asynchronní zpráva nemůže v těle obsahovat více, jak jednu datovou položku.

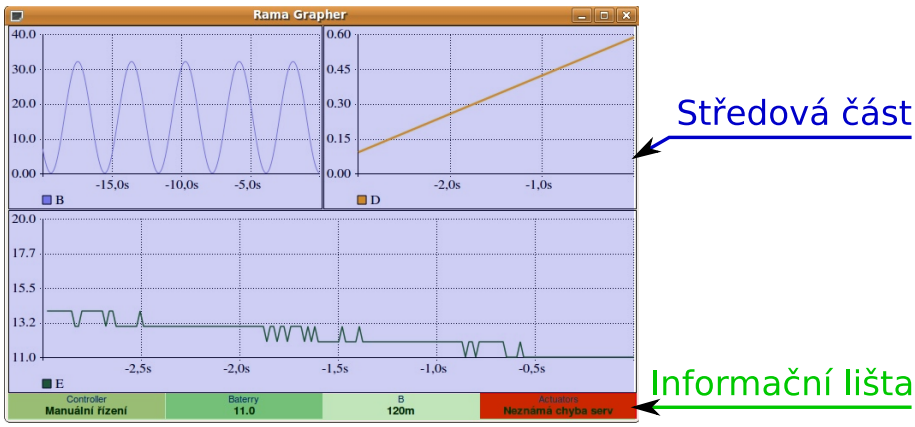
3.2 Aplikace graph

graph je grafická aplikace nad knihovnou Qt, která zobrazuje uživateli vybrané informace v podobě časově závislých grafů a informačních panelů[21]

Vyvstává otázka „Proč takováto aplikace vznikla, když už velice podobné věci existují?“. Ano, velice podobné aplikace existují. Nejblíže z nich je pravděpodobně *gnuplot* [3]. Problém ale je, že *gnuplot* není určen na on-line vykreslování grafů, kdy je požadováno až 64 překreslení za sekundu[22]. Dále pak není možné pomocí aplikace *gnuplot* zobrazit informační panel s varovným hlášením.

Aplikace graph, včetně všech její komponent, byla naprogramována „od nuly“. Opět zde vyvstala otázka, zda-li nepoužít například knihovnu *qwt* [5]. *qwt* je sada widgetů pro knihovnu Qt, která již vykreslování grafů umožňuje. Bohužel, výkon knihovny *qwt* se projevil jako nedostačující[23].

Aplikace graph se skládá z hlavního středového panelu, na kterém jsou vykreslovány grafy, a z informační lišty vespodu okna, na které jsou informační panely. Informační panely slouží k rychlému zobrazení chyb a upozornění. Typickým příkladem je upozornění na podpětí baterie. Aplikace graph se zvýrazněním jednotlivých částí je na obrázku 3.9.



Obrázek 3.9: Aplikace graph s popisem jednotlivých částí.

Pro aplikaci graph byly vytvořeny dva widgety, GraphObject a AStatus, které obstarávají vykreslování grafu a informačního panelu. Ve středové části aplikace graph je tak možné umístit libovolný počet grafů - instancí třídy GraphObject. Obdobně je možné vytvořit libovolný počet panelů na informační liště.

Panely na informační liště jsou umístěny zleva do prava v pořadí, v jakém jsou definovány v konfiguračním souboru.

Umístění grafů ve středové části okna se definuje v konfiguračním souboru obdobně, jako se indexují prvky matice. Interně je pro správu polohy a velikosti grafů použit objekt QGridLayout.

Seznam všech tříd aplikace graph a jejich stručný popis je v tabulce 3.2.

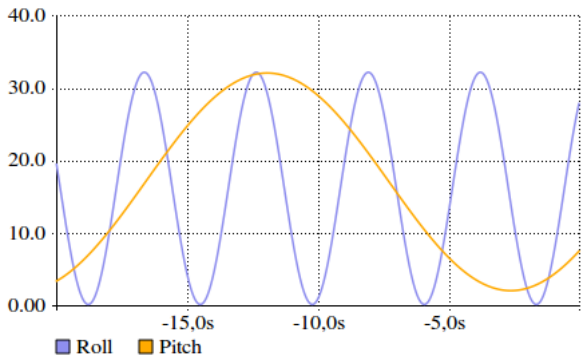
[21] Anglicky mnohem výstižněji *status-bars*.
[22] 64Hz je frekvence s kterou jsou odesílána data z modelu vrtulníku
[23] Vykreslování grafu s 200 tisíci vzorky při obnovovací frekvenci 64Hz.

Název třídy	Stručný popis
`GraphObject`	Zobrazuje jeden graf včetně legendy, mřížky a popisků.
`GScaler`	Stará se o měřítka os grafů.
`AStatus`	Zobrazuje jeden panel na informační liště.
`CIOReader`	Čte data ze standardního vstupu.
`MainWidget`	Tvoří podkladovou plochu a načítá `.conf` soubor.

Tabulka 3.2: Seznam všech tříd programu `graph` a jejich stručný popis.

3.2.1 Třída `GraphObject`

Obstarává vykreslování jednoho grafu. Graf může vykreslovat zároveň více veličin, které se liší v barvě čáry. Na pravé straně grafu jsou vždy nejaktuálnější hodnoty a na levé straně historicky nejstarší data. Kladný vývoj časové osy je tak zleva doprava. Při vykreslování křivka „pluje“ směrem zprava doleva. Popisky na časové ose vyjadřují „stáří“ dat. Legenda je umístěna pod grafem a zarovnána k levému okraji, viz obrázek 3.10.



Obrázek 3.10: Detail grafu vykreslovaného třídou `GraphObject`.

Pro celý graf je možné nastavit sílu čar, rozměr časové osy, či jestli se při vykreslování čar má použít antialiasing.

Zjednodušená deklarace třídy `GraphObject` je na výpisu 3.13.

Pro třídu `GraphObject` je klíčová struktura `curve`, jejíž zjednodušená deklarace je na výpisu 3.14. Struktura `curve` reprezentuje jednu křivku v grafu a obsahuje v sobě n posledních hodnot určité veličiny. Dále uchovává informaci o názvu veličiny, šířce, délce a barvě čáry.

Každá struktura `curve` v sobě obsahuje pointer na následující strukturu. To umožňuje vytvořit ze struktur `curve` jednosměrný spojový seznam. Pointer na první položku je obsažen v objektu `GraphObject`.

```
class GraphObject : public QWidget
{
public:
    void activateCol(unsigned int col, QColor color);
    void setXTime(float timeInterval);
public slots:
    void addDataSlot(RamaColDataPacket data);
    void addColSlot(unsigned int col, QString name);
protected:
    curve *curves;
    float globalMax, globalMin;
    QPixmap *background;
    QPointF *points;
    void paintCurve(curve *c, QPainter * painter);
    void drawRaster();
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
};
```

Výpis 3.13: Zjednodušená deklarace třídy `GraphObject`

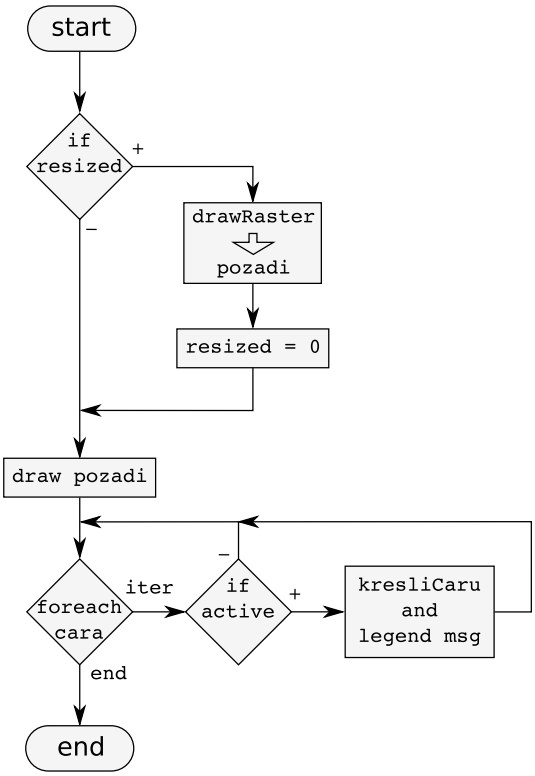
```
struct curve
{
    float *data;
    unsigned int dataSize;
    float curveWidth;
    bool active;
    QColor curveColour;
    QString name;
    curve *next;
};
```

Výpis 3.14: Zjednodušená deklarace struktury `curve`

V objektu `GraphObject` jsou překryty funkce `paintEvent` a `resizeEvent` z předka `QWidget`. Tím objekt přebírá kontrolu nad vlastním vykreslováním. Algoritmus uvnitř metody `paintEvent` je na obrázku 3.11.

Metoda `paintEvent` prakticky jen projde celý seznam struktury `curve` a ty, které mají nastavený příznak `active`, vykreslí pomocí metody `paintCurve`. Pro každou vykreslenou křivku se ještě vypíše její jméno do legendy.

Metoda `paintEvent` drobně optimalizuje při vykreslování mřížky. Pokud nedošlo ke změně měřítka osy Y, nebo ke změně rozměrů widgetu, vypadá mřížka včetně popisků stejně. Proto se její vykreslení provede pouze jednou, a to do proměnné `background`. Při každém dalším překreslení objektu `GraphObject` je mřížka i s popisky pouze „ozrca-



Obrázek 3.11: Algoritmus metody `paintEvent` třídy `GraphObject`.

dlena“ z proměnné `background`. Příznak `resized` indikuje, je-li třeba mřížku překreslit nebo jestli je možné použít již vykreslenou z proměnné `background`. Tento příznak je nastavován uvnitř metody `resizeEvent`, a také, pokud dojde ke změně měřítka osy Y.

Již zmíněná metoda `paintCurve` vnitřně používá pole bodů `points`. Jedná se o souřadnice v pixelech uvnitř okna. X souřadnice pixelů čáry jsou časově neměnné[24] a shodné pro všechny čáry grafu. Proto jsou dopředu předpočítány ve funkci `resizeEvent`. Metoda `paintCurve` pouze nastaví Y souřadnice pro pole `points` tak, aby odpovídalo vykreslované čáře. Při určování Y souřadnice se provádí lineární přepočet, který bere v úvahu velikost widgetu a aktuální měřítko osy Y.

Již bylo řečeno, jak třída `GraphObject` vykresluje do grafu křivky odpovídající strukturám `curve`. Nyní zbývá objasnit, jak struktury `curve` vznikají, a jak se plní daty.

Struktury `curve` vznikají uvnitř metody (slotu) `addColSlot`. Jako parametry této metody se předává sloupec, ve kterém je proměnná, a název proměnné. Viz protokol meziprocesní textové komunikace 2.2.

Tyto informace jsou k dispozici v prvním řádku dat příchozích skrze standardní vstup. Třída `CIOReader`, která se stará o čtení vstupu, rozesílá pří startu aplikace signály právě s těmito informacemi.

Aby se struktura `curve` mohla vykreslovat, je třeba jí ještě aktivovat. To se provede voláním metody `activateCol`, kde specifikujeme sloupec dat, který chceme vypisovat.

[24] Pokud nedojde k události `resizeEvent`.

Metodou je možné ještě redefinovat barvu čáry. Pokud tak neučiníme, čáře zůstane automaticky přidělená barva[25].

Počet zobrazovaných vzorků, a tím pádem i velikost pole `GraphObject::points` či `curve::data`, je nastavován uvnitř metody `setXTime`. Jako parametr metody se předává požadovaný rozměr časové osy v sekundách.

Metoda `setXTime` volí počet vykreslovaných vzorků s ohledem na počet pixelů widgetu. Je tak možné, že metoda `setXTime` byla volána s parametrem 30, ale rozměr osy času bude ve výsledku třeba 32,6 sekund, protože při tomto rozměru vychází právě 2 pixely na jeden vzorek.

Jako výchozí rozměr časové osy je takový, aby vycházel právě jeden pixel na vzorek.

Data jsou do struktur `curve` ukládána v metodě (slotu) `addDataSlot`. Jako parametr se předává struktura `RamaColDataPacket`, která obsahuje všechna data z právě příchozího synchronního packetu.

```
struct RamaColDataPacket
{
    float data[RAMA_MAX_COLS];
    unsigned int cols;
};
```

Výpis 3.15: Zjednodušená deklarace struktury `RamaColDataPacket`

Metoda `addDataSlot` projde seznam struktur `curve` a do všech aktivních přidá jeden vzorek.

Poslední funkcionalita třídy `GraphObject` je volba měřítka osy Y. Jsou přitom řešeny dvě otázky:

- Kdy se má změnit měřítko osy Y?
- Jaké hodnoty zvolit jako krajní?
  Pokud se například zobrazují data v rozmezí $-9.85 \cdot 10^{-3}$ až $8.34 \cdot 10^{-3}$, bylo by vhodné volit krajní hodnoty osy Y $\pm 10 \cdot 10^{-3}$.

Třída `GraphObject` obsahuje dvě proměnné (`globalMax` a `globalMin`), ve kterých si udržuje největší a nejmenší hodnotu z dat aktivních struktur `curve`. Pokud příchozí data přesáhnou maximální hodnotu, nebo poklesnou pod minimální hodnotu, je vyvolán přepočet mezí osy Y. Aby se osa Y i „smršťovala“, přepočet je také vyvolán po určitém počtu přijatých vzorků.

Pro určení mezí osy Y je použita třída `GScaler`.

[25] Automatické přidělování barev probíhá při vytváření struktury `curve` a jsou cyklicky používány barvy z palety, jež je napevno stanovená ve zdrojových kódech.

3.2.2 Třída GScaler

Hlavním úkolem třídy GScaler je nalezení vhodných mezí pro Y osu grafu. Zjednodušená deklarace třídy je na výpisu 3.16.

```
class GScaler
{
public:
    GScaler (float gMax, float gMin);
    void makeNiceLimits(unsigned int scalePolitics);
    float getGlobalMin();
    float getGlobalMax();

    void static separateMantisaExp(float num, float *mantisa, int *exp);
  float static getUpperNice(float cislo);
private
    float globalMin, globalMax;
};
```

Výpis 3.16: Zjednodušená deklarace třídy GScaler

Jádrem třídy je metoda separateMantisaExp, která rozdělí reálné číslo typu float na mantisu a exponent[26]. Mantisa je navrácena jako reálné číslo v intervalu $\pm\langle 1; 0.1)$ a exponent jako celé číslo.

Metoda getUpperNice zaokrouhluje reálné číslo na vyšší „kulatou“ hodnotu. Vnitřně metoda getUpperNice využívá metodu separateMantisaExp, pomocí které nejprve separuje mantisu a exponent. Mantisa je zaokrouhlena (směrem nahoru) do řady

$$\pm(0.1,\ 0.2,\ 0.3,\ 0.4,\ 0.5,\ 0.6,\ 0.7,\ 0.8,\ 0.9,\ 1)$$

a následně vynásobena číslem 10^p, kde p je exponent.

Metody separateMantisaExp a getUpperNice jsou užitečné i samostatně. Proto jsou definované jako *statické* a mohou tak být použity nezávisle na instanci třídy GScaler.

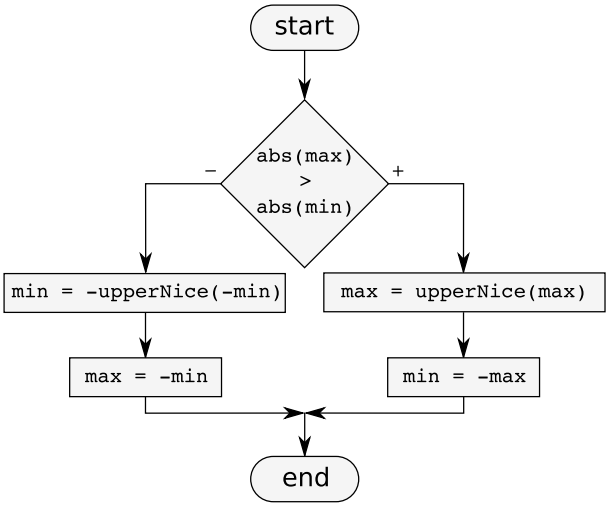
Metoda getUpperNice *nemá* svého protějška getLowerNice, byť to k tomu logicky vybízí. Metoda getLowerNice by se uplatnila při zaokrouhlování dolní meze. Implementovat getLowerNice není třeba, protože platí getLowerNice(x) = -getUpperNice(-x).

Pro volbu mezí osy grafu se vytvoří instance třídy GScaler, kde se konstruktoru předá největší a nejmenší hodnota zobrazovaných dat. Po volání metody makeNiceLimits je možné získat nové meze metodami getGlobalMin a getGlobalMax.

Meze je třeba volit tak, aby horní i dolní mez byla „pěkná“ čísla a zároveň i jejich rozdíl byl „pěkné“ číslo. Uvažme příklad, kdy maximální hodnota v zobrazovaných datech je 4.8 a minimální 0.32. Samostatně zaokrouhlené meze by vycházely 5 a 0.3. To by například při pěti dílcích mřížky vedlo na hodnotu 0.94 na jeden dílek.

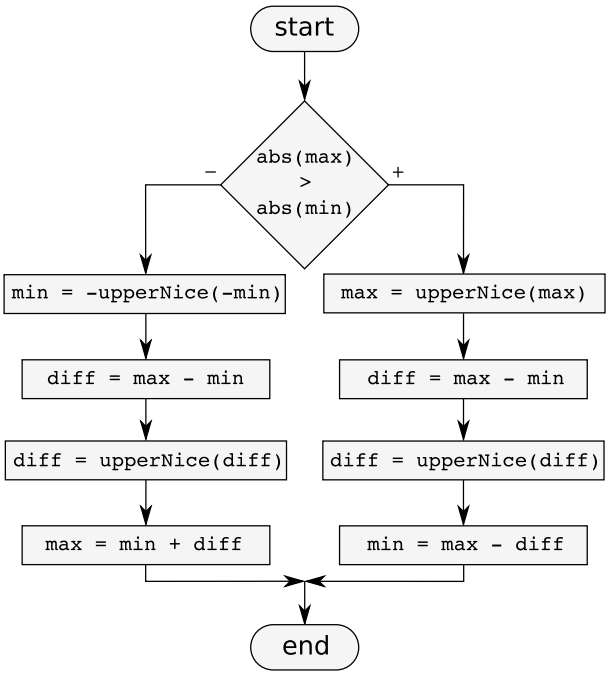
[26] O desítkovém základu.

Metodě `makeNiceLimits` je možné „napovědět“, jaký charakter mají zobrazovaná data. Pokud například víme, že zobrazovaná data jsou téměř symetrická vůči nule, můžeme metodě `makeNiceLimits` doporučit symetrickou politiku volby mezí. Obrázek 3.12 znázorňuje algoritmus metody `makeNiceLimits` pro takový případ.



Obrázek 3.12: Algoritmus metody `makeNiceLimits` třídy `GScaler` pro symetrickou osu.

Politiku volby mezí osy Y můžeme specifikovat v konfiguračním souboru. Pokud tak neučiníme, třída `GScaler` volí obecný algoritmus, který je znázorněn na obrázku 3.13.



Obrázek 3.13: Algoritmus metody `makeNiceLimits` třídy `GScaler` pro automatickou volbu mezí.

3.2.3 Třída AStatus

Widget AStatus zobrazuje obdélník s malým popiskem[27] v horní části a hlavní informací ve středové části. Zobrazovanou informací může být krátký text, nebo jedno číslo. Největší informaci nese barva pozadí, která se mění v závislosti na zobrazeném čísle/textu. Ukázka dvou widgetů AStatus je na obrázku 3.14.

Obrázek 3.14: Příklad vzhledu textového a číselného widgetu AStatus.

Zjednodušená deklarace třídy AStatus je na výpistu 3.17.

```
class AStatus : public QWidget
{
public:
    AStatus(QWidget *parent, int async_data);
    void setLegend(const QString &legend);
    void setBackgroundEvent(bEvent event);
    void valueFrom(unsigned int col);
    void asyncNum(unsigned int ID);
public slots:
    void addDataSlot(RamaColDataPacket data);
    void addMsgSlot(unsigned int ID, const QString &msg);
protected:
    QVector<bEvent> bevents;
    void paintEvent(QPaintEvent *event);
};
```

Výpis 3.17: Zjednodušená deklarace třídy AStatus

Widget AStatus může pracovat ve dvou módech

- *Textový mód* kdy zobrazuje text z asynchronních zpráv.
- *Numerický mód* kdy zobrazuje číslo odpovídající jedné proměnné z těla synchronní zprávy.

V jakém módu bude widget pracovat, se nastavuje druhým parametrem v konstruktoru. 0 pro numerický mód, nenulové číslo pro textový mód.

Pro numerický mód se zobrazovaná proměnná specifikuje metodou valueFrom. Vlastní aktualizace dat má na starost slot addDataSlot.

Pro textový mód se ID asynchronní zprávy, jejíž text widget zobrazuje, nastavuje metodou asyncNum. Aktualizaci textu má na starost slot addMsgSlot. Na tento slot je napojen odpovídající signál třídy CIOReader.

[27] Text tohoto popisku je definovatelný v konfiguračním souboru.

Metodou `setLegend` se nastavuje text popisku v horní části widgetu.

Pro definici barvy pozadí widgetu `AStatus` slouží struktura `bEvent`, jejíž deklarace je na výpisu 3.18.

```
enum backAction { numGreater, numLower, numBetween, numNotBetween,
                  strContains, strNotContains, strIs, strIsNot};
struct bEvent
{
    float dato[2];
    QString msg;
    backAction action;
    QColor barva;
};
```

Výpis 3.18: Deklarace struktury `bEvent`

Struktura `bEvent` obsahuje barvu pozadí a podmínku `action`, při jejímž splnění se příslušná barva pozadí použije.

- Pro widget v *numerickém módu* jsou důležité položky výčtového typu `backAction`, které začínají písmeny `num`. Při vyhodnocování, zda-li se má příslušná barva pozadí použít, se porovnává hodnota aktuálně zobrazovaného čísla s hodnotou/hodnotami v poli `dato`. Zkoumá se přitom, jestli je číslo menší než `dato[0]`, větší než `dato[0]`, mezi `dato[0]` a `dato[1]`, nebo mimo `dato[0]` a `dato[1]`.
- Pro widget v *textový mód* jsou důležité položky výčtového typu `backAction`, které začínají písmeny `str`. Při vyhodnocování, zda-li se má příslušná barva pozadí použít, se porovnává hodnota aktuálně zobrazovaného textu s řetězcem `msg`. Podle hodnoty `action` se zkoumá, jestli text obsahuje řetězec `msg`, neobsahuje řetězec `msg`, je identický s řetězcem `msg`, nebo není identický s řetězcem `msg`.

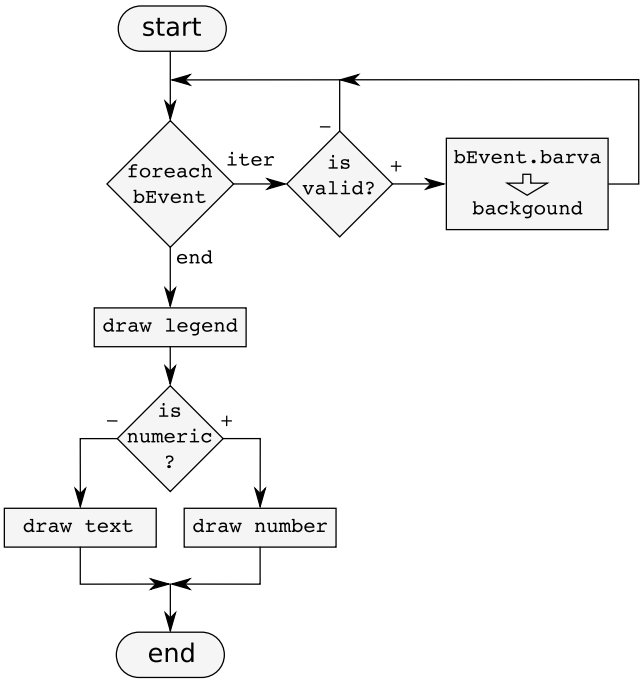
„Události pozadí“ (struktury `bEvent`) se přidávají metodou `setBackgroundEvent`.

Při vykreslování widgetu metodou `paintEvent` se postupně projde seznam struktur `bEvent` v pořadí, jakém byly přidány metodou `setBackgroundEvent`. Pokud je `bEvent` shledán vyhovující vůči aktuální zobrazované hodnotě, je pozadí widgetu přebarveno na barvu obsaženou ve struktuře `bEvent`. Metoda `paintEvent` nakonec vypíše legendu a text/číslo. Algoritmus vykreslování widgetu je na obrázku 3.15.

3.2.4 Třída `CIOReader`

Třída `CIOReader` má za úkol přijímat data přicházející skrze standardní vstup, zpracovávat[28] je a rozesílat, v podobě signálů, ostatním widgetům. Třída `CIOReader` si pro svou práci vytváří nové vlákno, aby příjem a zpracování dat „nezdržoval“ hlavní procesní vlákno od vykreslování widgetů.

[28] Převádět textové řetězce na čísla.



Obrázek 3.15: Algoritmus vykreslování widgetu `AStatus`.

Zjednodušená deklarace třídy `CIOReader` je na výpisu 3.19.

```
class CIOReader : public QThread
{
public:
    CIOReader(QObject *parent, int delay, bool transparent);
public signals:
    void dataReceived(RamaColDataPacket data);
    void headerReceived(unsigned int col, QString name);
    void msgReceived(unsigned int ID, QString msg);
    void endReceived();
public slots:
  void writeStdout(const QString &s);
private slots:
  void readStdin( int );
private:
  QSocketNotifier* notifier;
  QTextStream* cin;
  QTextStream* cout;
  void parseData(const QString &s);
};
```

Výpis 3.19: Zjednodušená deklarace třídy `CIOReader`

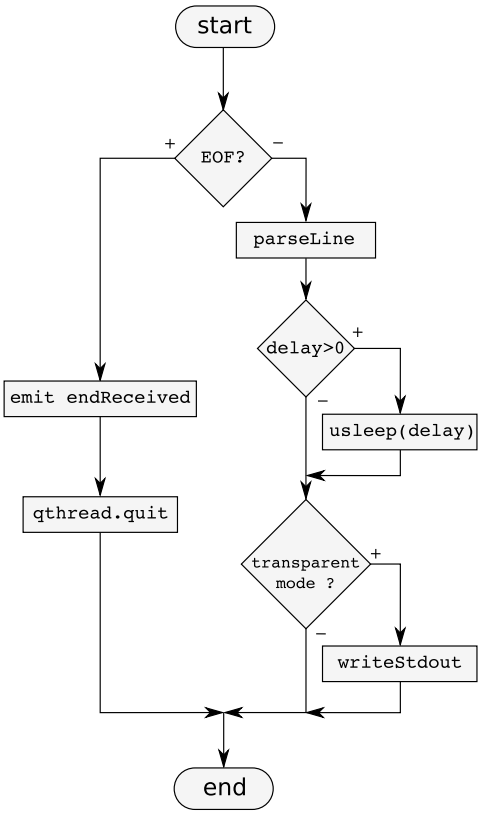
V konstruktoru třídy se sděluje pauza a transparentnost. Transparentní mód znamená, že všechna data příchozí skrze standardní vstup jsou přeposlána na standardní výstup. Pokud je nastavena nezáporná pauza, po každém přijatém řádku je provedeno čekání. Délka čekání v mikrosekundách je rovna hodnotě v proměnné `delay`. Čekání po každém přijatém řádku má smysl, pokud se přehrávají data ze souboru. Bez prokládání řádků čekáním by rychlost přehrávání neodpovídala rychlosti záznamu[29].

Každé nově vzniklé vlákno `QThread` po vytvoření začne vykonávat metodu `run`. Protože objekt `CIOReader` nemá tuto metodu implementovanou, vykoná se metoda `run` jeho předka `QThread`, která pouze zavolá metodu `exec`.

Metoda `exec` je smyčka zpráv. Díky běžící smyčce zpráv je i nově vzniklé vlákno schopno používat signál-slotový mechanismus. Čtení standardního vstupu je provedeno právě pomocí signálů a slotů.

Objekt `CIOReader` se ve svém konstruktoru připojí na standardní vstup a výstup pomocí privátních proměnných `cin` a `cout`. V metodě `writeStdout` je použita proměnná `cout` k výpisu textu na standardní výstup.

Dále je použita třída `QSocketNotifier` v kombinaci s `cin`, která emituje signál vždy, jsou-li data k dispozici na standardním vstupu. Na tento signál je zachycen slot `readStdin`. Algoritmus uvnitř metody `readStdin` je na obrázku 3.16.



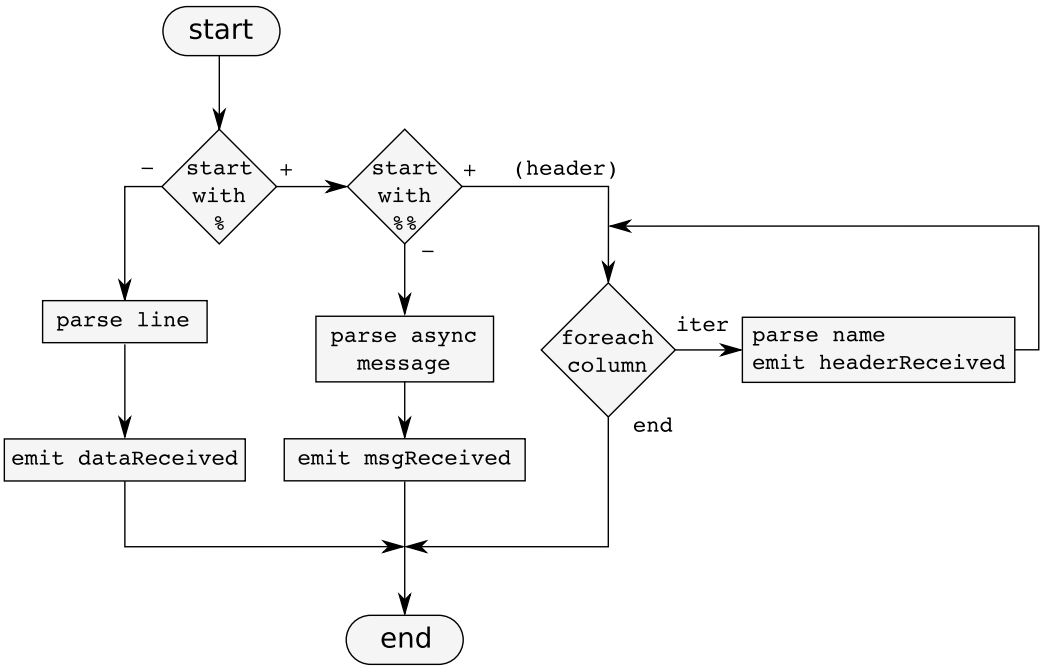
Obrázek 3.16: Čtení standardního vstupu metodou `readStdin` ve třídě `CIOReader`.

[29] Data ze souboru by „protekla“ aplikací maximální možnou rychlostí.

Metoda `readStdin` vždy zkontroluje, nedošlo-li k uzavření standardního vstupu. Pokud ano, emituje signál `endReceived` a ukončí vykonávání čtecího threadu. Pokud je vstup v pořádku, přečte z něho jeden řádek a ten zpracuje metodou `parseData`. Pokud je nastavená pauza, provede se čekání a pokud je požadován transparentní mód, přepošle přečtený řádek také na standardní výstup.

Algoritmus zpracování každého řádku vstupu metodou `parseData` je na obrázku 3.17. Nejprve je rozhodnuto, jestli se jedná o řádek s daty, asynchronní zprávu nebo hlavičku.

- Při příjmu dat jsou textové řetězce oddělené tabulátorem převedeny na čísla, kterými je naplněna struktura `RamaColDataPacke` (3.15). Tato struktura je následně odeslána signálem `dataReceived`.
- Při příjmu asynchronní zprávy je zjištěno její ID a textová zpráva. Obě tyto informace jsou následně odeslány signálem `msgReceived`.
- Při příjmu hlavičky jsou postupně čteny názvy sloupců (proměnných) oddělené tabulátorem. Pořadí a název každého sloupce je odesílán signálem `headerReceived`.



Obrázek 3.17: Zpracování jednoho řádku standardního vstupu metodou `parseData` ve třídě `CIOReader` aplikace `graph`.

3.2.5 Třída `MainWidget`

Třída `MainWidget` v konstruktoru načte nastavení z konfiguračního souboru a vytvoří požadovaný počet instancí `GraphObject`, `AStatus` a jednu instanci `CIOReader`. Nakonec propojí signály třídy `CIOReader` s příslušnými sloty tříd `GraphObject` a `AStatus`.

Zjednodušená deklarace třídy `MainWidget` je na výpisu 3.20.

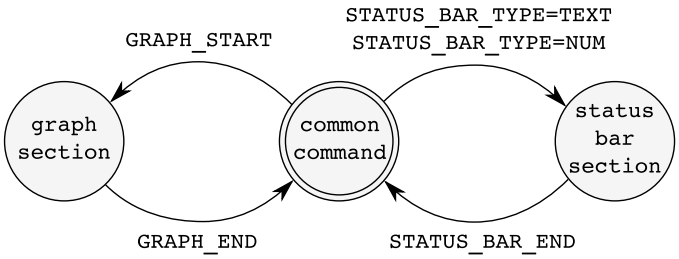
```
class MainWidget : public QWidget
{
public:
    MainWidget(const QString &confFile, QWidget *parent);
private:
    QVector<GraphObject*> gs1;
    QVector<AStatus*> as;
    CIOReader *cior;
    QGridLayout *glayout;
    QVBoxLayout *vlayout;
    QHBoxLayout *hlayout;

    bool parseLine(const QString &s);
};
```

Výpis 3.20: Zjednodušená deklarace třídy `MainWidget`

Zpracování konfiguračního souboru probíhá obdobně, jako ve třídě `configReader` aplikace `datalog` (kapitola 3.1.1).

V konstruktoru je otevřen konfigurační soubor a po řádcích zpracováván metodou `parseLine`, která opět tvoří třístavový automat. Viz obrázek 3.18. Konfigurační soubor obsahuje obecná nastavení (velikost a pozice okna, transparentní mód . . .), definici grafů a definici informačních panelů.



Obrázek 3.18: Stavový automat pro zpracování jednoho řádku konfiguračního souboru metodou `parseLine` ve třídě `MainWidget`.

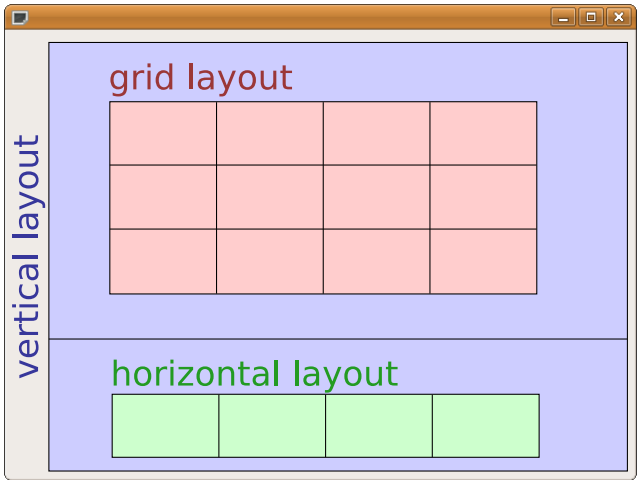
Metody `parseLine` v aplikacích `graph` a `datalog` se liší pouze ve dvou zásadních věcech:

1. Očekávají jiná klíčová slova.

2. `parseLine` v aplikaci `graph` používá přirozených prostředků knihovny Qt. Pro řetězce třídu `QString`, pro soubory třídu `QFile`, pro převod textových řetězců na čísla třídu `QLocale` . . .

Po zdárném načtení konfiguračního souboru třída `MainWidget` vytvoří instance tříd

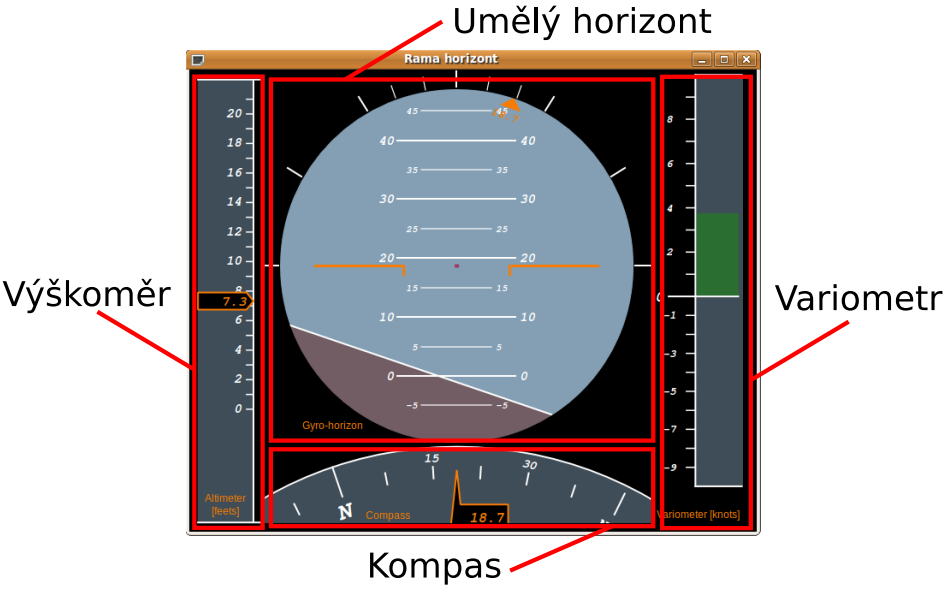
`GraphObject` a `AStatus` a umístí je na odpovídající pozice pomocí *layoutů*. Použité *layouty* a jejich rozložení je na obrázku 3.19.



Obrázek 3.19: Rozložení *layoutů* na widgetu `MainWidget`.

3.3 Aplikace `horizont`

`horizont` je grafická aplikace nad knihovnou Qt, která vykresluje virtuální letecké palubní přístroje. Jmenovitě *umělý horizont*, *kompas*, *výškoměr* a *variometr*, viz obrázek 3.20.



Obrázek 3.20: Aplikace `horizont` s popisem jednotlivých částí.

`horizont` je dedikovaná pro použití v kombinaci s modelem vrtulníku[30] a proto se jedná o nejméně obecnou utilitu ze všech programových modulů.

Vykreslování všech leteckých přístrojů má na starost widget `HorizontWidget`. O čtení standardního vstupu se stará třída `CIOReader`. Třída `MainWidget`, stejně jako v aplikaci `graph`, načítá konfigurační soubor a tvoří podkladovou plochu.

Název třídy	Stručný popis
`HorizontWidget`	Zobrazuje virtuální letecké přístroje.
`CIOReader`	Čte data ze standardního vstupu.
`MainWidget`	Tvoří podkladovou plochu a načítá `.conf` soubor.

Tabulka 3.3: Seznam všech tříd programu `horizont` a jejich stručný popis.

3.3.1 Třída `HorizontWidget`

Vykreslování jednotlivých přístrojů se provádí softwarově pomocí základních primitiv typu *úsečka*, *kruh*, *obdélník* ... Dále jsou použity transformace typu *posun* a *rotace*.

[30] Případně s jiným leteckým modelem.

Všechny tyto funkce poskytuje třída `QPainter`, která tvoří základ knihovny Qt pro vykreslování 2D grafiky.

Zjednodušená deklarace třídy `HorizontWidget` je na výpisu 3.21.

```
class HorizontWidget : public QWidget
{
public:
    HorizontWidget(QWidget *parent = NULL);
    void setPitchTransformation(float k, float q);
    void setRollTransformation(float k, float q);
    void setVarioTransformation(float k, float q);
    void setAltTransformation(float k, float q);
    void setComTransformation(float k, float q);
public slots:
    void setPitch(float pitch);
    void setRoll(float roll);
    void setAltitude(float alt);
    void setCourse(float cou);
    void setAltSpeed(float alts);
protected:
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
private:
    void paintHorizont(QPainter &painter);
    void paintAltimeter(QPainter &painter);
    void paintCompass(QPainter &painter);
    void paintVario(QPainter &painter);
};
```

Výpis 3.21: Zjednodušená deklarace třídy `HorizontWidget`

Třída `HorizontWidget` disponuje sloty pro nastavení veličin, které „hýbou“ jednotlivými přístroji. Každá veličina je uvnitř slotu transformována lineární transformací

$$\hat{x} = kx + q,$$

kde x je původní hodnota veličiny, $\hat{x}$ je nová hodnota a k, q jsou transformační konstanty. To umožňuje přepočet jednotek[31].

Při změně rozměrů widgetu je volána metoda `resizeEvent`, ve které se vypočte poloha a rozměry každého přístroje[32].

[31] Například úhly `Pitch` a `Roll`, udávající polohu umělého horizontu, telemetrie vrtulníku vysílá v radiánech a pro zobrazení jsou použity stupně.

[32] Poloha a rozměr každého přístroje jsou definovány souřadnicí levého horního a pravého dolního rohu v pixelech.

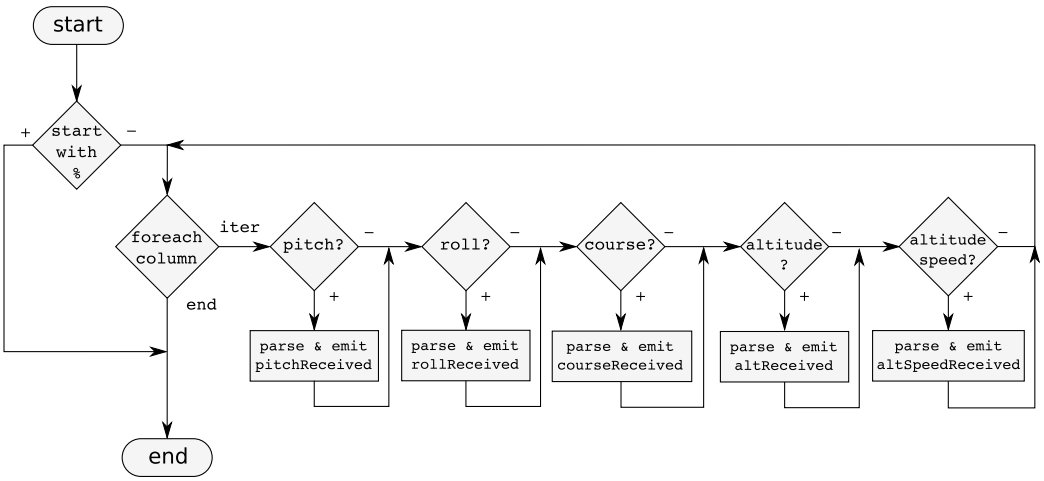
Vykreslení widgetu metodou `paintEvent` spočívá v zavolání metod `paintHorizont`, `paintAltimeter`, `paintCompass` a `paintVario`. Těla těchto metod nejsou příliš zajímavá - pouze obsahují mnoho volání různých metod třídy `QPainter` tak, aby došlo k vykreslení příslušného přístroje.

3.3.2 Třída `CIOReader`

Až na drobné úpravy se jedná o tutéž třídu, jako používá aplikace `graph` (kapitola 3.2.4). Hlavní rozdíl je v tom, že tato třída ignoruje asynchronní zprávy a neodesílá informace obsažené v hlavičce. Dále místo signálu `dataReceived`, který obsahuje všechna data synchronního packetu, jsou vysílány pouze signály obsahující veličiny pro přístroje. Rozdíly jsou vidět na výpisu 3.22 a vývojovém diagramu na obrázku 3.21.

```
class CIOReader : public QThread
{
public:
    CIOReader(QObject *parent, int delay, bool transparent);
public signals:
    void pitchReceived(float val);
    void rollReceived(float val);
    void altReceived(float val);
    void courseReceived(float val);
    void altSpeedReceived(float val);
private:
  void parseData(const QString &s);
};
```

Výpis 3.22: Zjednodušená deklarace třídy `CIOReader`



Obrázek 3.21: Zpracování jednoho řádku standardního vstupu metodou `parseData` ve třídě `CIOReader` aplikace `horizont`.

3.3.3 Třída `MainWidget`

Třída `MainWidget` tvoří podkladovou a načítá nastavení z konfiguračního souboru.

Obdobně, jako třída `MainWidget` aplikace `graph`, v konstruktoru je načten konfigurační soubor.

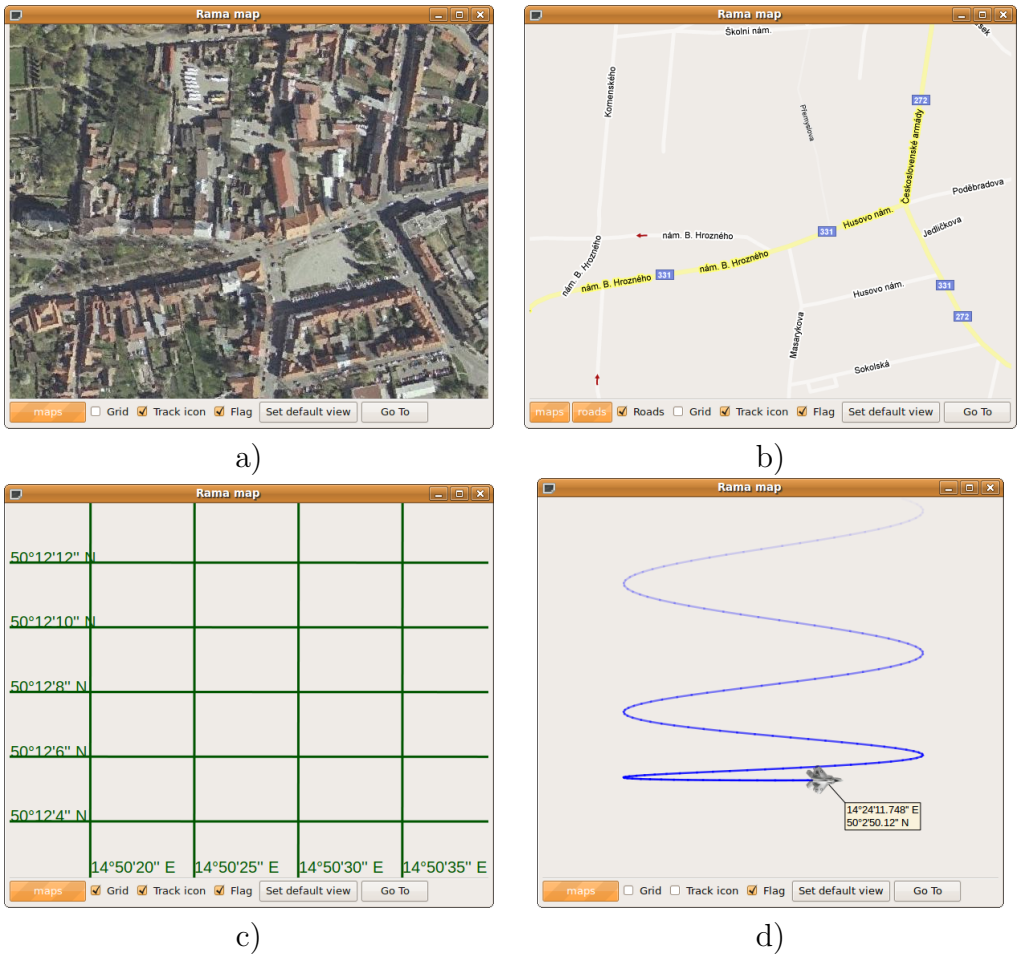
Dále pak vytvoří jednu instanci objektu `CIOReader` a `HorizontWidget` a propojí jejich signály a sloty.

3.4 Aplikace `map`

`map` je grafická aplikace nad knihovnou Qt, která zobrazuje trajektorii pohybu vrtulníku do mapy. Podkladové obrázky s mapou mohou být automaticky načítány z internetu nebo se použijí lokální soubory pro off-line provoz aplikace.

Byť je aplikace `map` primárně uzpůsobena pro zobrazení dráhy na mapě, jedná se o velice obecný nástroj umožňující vykreslování dvourozměrného grafu, kde jako pozadí může být použita libovolná rastrová mapa (jpeg, png ...). Aplikaci `map` by bylo možné použít například pro on-line vizualizaci polohy mobilního robota na hrací ploše, nebo třeba zobrazení ekvipotenciální teplotní hladiny kolem půdorysu budovy.

Aplikace `map` má čtyři vrstvy, do kterých provádí vykreslování. Nejspodnější vrstva je určená pro obrázky orto-foto mapy, poté je vykreslována vrstva s popisky (silnice, názvy ulic ...), následně mřížka a jako poslední je vykreslována vrstva s trajektorií pohybu. Vrchní tři vrstvy jsou průhledné, aby nedošlo k zastínění nejspodnější vrstvy. Ukázky jednotlivých vrstev jsou na obrázku 3.22.



Obrázek 3.22: a) Nejspodnější vrstva s orto-foto mapou, b) vrstva s popisky c) vrstva mřížky d) vrstva s trajektorií pohybu.

Pozadí, které tvoří spodní dvě vrstvy, nazvěme *pracovní plochu*. Pracovní plochu lze

klávesnicí i myší přibližovat, oddalovat a posouvat. *Pracovní plochou* je vždy myšleno celé pozadí, ač v důsledku přiblížení/posunutí je viditelná pouze jeho část. Pokud aplikace `map` zobrazuje dráhu na mapě, tak *pracovní plochou* je povrch celé zeměkoule[33].

Zobrazení vrstvy s popisky a mřížky je volitelné. Výchozí viditelnost/neviditelnost těchto vrstev je nastavitelná v konfiguračním souboru. Vrstvy mohou být zobrazeny nebo skryty i „za běhu“ aplikace pomocí ovládacích prvků ve spodní liště.

Trajektorie má volitelnou délku, barvu, šířku a způsob vykreslování - pomocí úseček nebo pomocí Bézierových křivek. Konec trajektorie (aktuální pozice) může být označen „praporkem“ se souřadnicemi polohy. Při aktivovaném módu *sledování* je navíc konec trajektorie udržován ve středu okna[34]. Trajektorie může postupně „mizet“, aby se odlišila historická data od aktuálních.

Pozadí je automaticky stahováno z internetu pomocí WMS[35]. Každý stažený obrázek je zároveň uložen do souboru, což umožňuje automatické vytvoření repositáře pro off-line provoz.

Souborový repositář může být vytvořen i ručně nahráním souborů s obrázky pozadí do příslušného adresáře.

Popis jednotlivých tříd je v tabulce 3.4. Vzájemné vztahy tříd jsou na obrázku 3.23.

Název třídy	**Stručný popis**
`affineConverter`	Převod ze souřadnicového systém WGS84 (GPS souřadnice).
`CIOReader`	Čte data ze standardního vstupu.
`coordConverter`	Abstraktní třída: Interface pro uživatelský souřadný systém.
`dataCoordSystem`	Abstraktní třída: Interface zobrazení jednotek.
`geographyCoords`	Formátuje a načítá GPS souřadnice.
`goToDialog`	Dialogové okno umoňující skoky v mapě.
`lineSmoother`	Vyhlazuje křivky mřížky.
`mapWidget`	Hlavní widget s mapou.
`mapManager`	Stahuje mapy z internetu a spravuje souborový repositář map.
`mapObject`	Jeden fragment mapy - čtvercový obrázek se souřadnicemi.
`mapObjectTree`	Stromová struktura pro uložení objektů `mapObject` v paměti.
`Widget`	Podkladová plocha + spodní ovládací panel.

Tabulka 3.4: Seznam všech tříd programu `map` a jejich stručný popis.

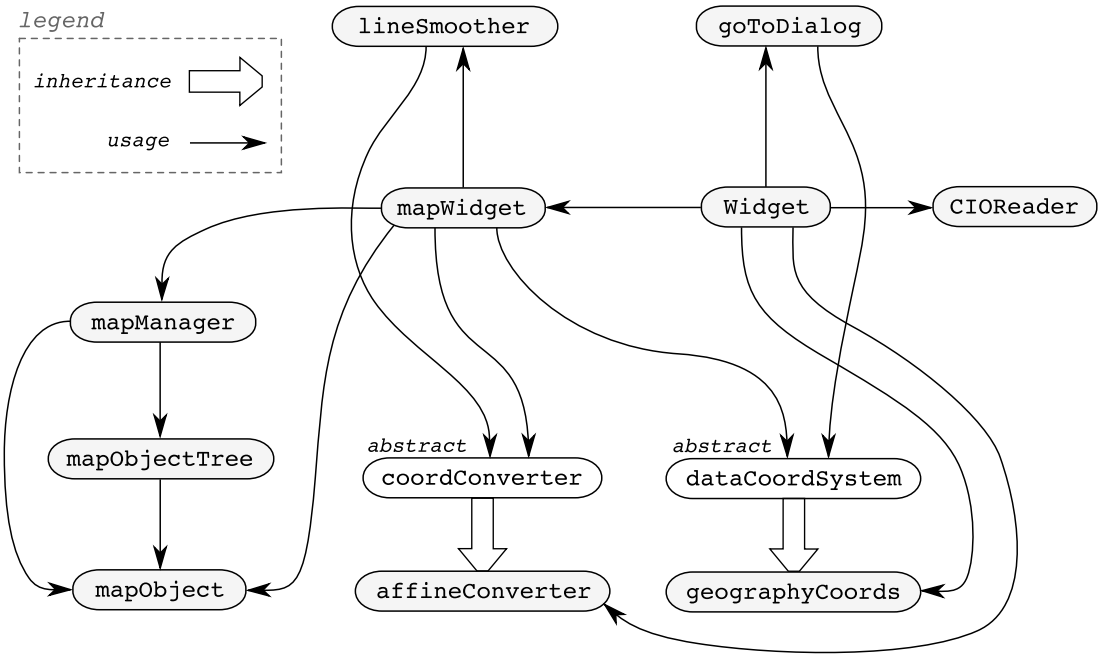
3.4.1 Souřadné systémy používané aplikací `map`

Aplikace `map` používá tři souřadné systémy. Nyní bude objasněn účel a princip jednotlivých souřadných systému.

[33]Pod pojmem „povrch celé zeměkoule“ je myšleno cca 95% povrchu. Díky zvolenému způsobu zobrazení (Mercatorovo) není možné postihnout malé oblasti kolem pólů.

[34]To při pohybu vrtulníku vede k posouvání pozadí.

[35]Web Map Service - otevřený standard pro sdílení geografických informací ve formě rastrových map [8]. Vyvinut a dále podporován organizací Open Geospatial Consortium (OGC) [9].



Obrázek 3.23: Class diagram aplikace `map`.

1. Pro jednoznačnou reprezentaci bodu na *pracovní ploše* jsou použita dvě reálná čísla[36] a, b udávajíc souřadnice v ose X a Y.

$$\begin{bmatrix} a \\ b \end{bmatrix}$$

   Levý horní roh má souřadnici $[0, 0]^T$ a pravý dolní roh má souřadnici $[1, 1]^T$. Vodorovnou osu označme X a její kladný vývoj je zleva doprava. Svislou osu označme Y a její kladný vývoj je shora dolů. Nazvěme tuto soustavu **globálním souřadnicovým systémem**.

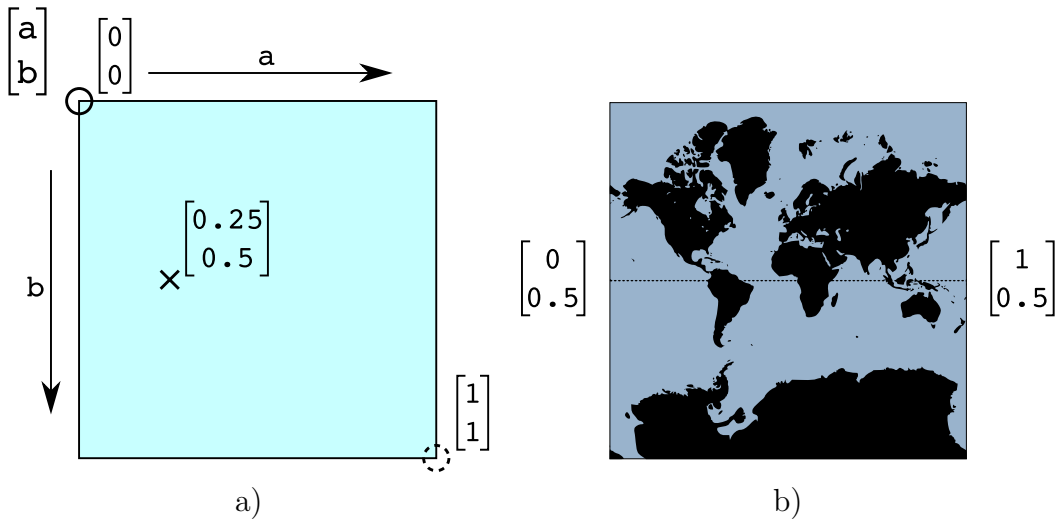
   Pro mapu světa bylo zvoleno *Mercatorovo zobrazení*, umístěné na *pracovní ploše* aplikace `map` tak, že rozvinutý rovník má souřadnici $[0, 0.5]^T$ až $[1, 0.5]^T$.

   Globální souřadnicový systém a způsob projekce povrchu zeměkoule je znázorněn na obrázku 3.24.

2. Pro práci s obrázky pozadí a jejich snadné ukládání do souborů byl vytvořen **souborový souřadnicový systém**. Je tvořen trojicí celých nezáporných čísel

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

[36] Reálná čísla s dvojitou přesností, tedy datové typy `double`.



Obrázek 3.24: a) Popis *pracovní plochy* globálním souřadnicovým systémem. b) Ilustrace projekce povrchu zeměkoule na *pracovní plochu*.

Souřadnice z určuje přiblížení. Celá *pracovní plocha* bez přiblížení má souřadnici $z = 0$. Dvojnásobně přiblížené *pracovní ploše* odpovídá souřadnice $z = 1$. Čtyřnásobně přiblížené odpovídá $z = 2$. Osminásobně $z = 3$ atd ...

Souřadnice x, y odpovídají poloze obrázku na *pracovní ploše*. Nejedná se o souřadnici jednoho bodu, ale celého čtverce - určité výseče *pracovní plochy*. Pro přiblížení $z = 0$ odpovídají souřadnice $[x, y]^T = [0, 0]^T$ celé *pracovní ploše*. Pro přiblížení $z = 1$ odpovídají souřadnice $[x, y]^T = [0, 0]^T$ levé horní čtvrtině *pracovní plochy*. Pravý horní čtverec má souřadnice $[x, y]^T = [1, 0]^T$, levý dolní čtverec $[x, y]^T = [0, 1]^T$ a pravý dolní čtverec $[x, y]^T = [1, 1]^T$. Při zvětšení přiblížení na $z = 2$ se každý čtverec rozpadne na další 4 a způsob adresování jednotlivých podčtverců se rekurzivně opakuje. Situace je znázorněna na obrázku 3.25.

Souborový souřadnicový systém tak vytváří stromovou strukturu souřadnic.

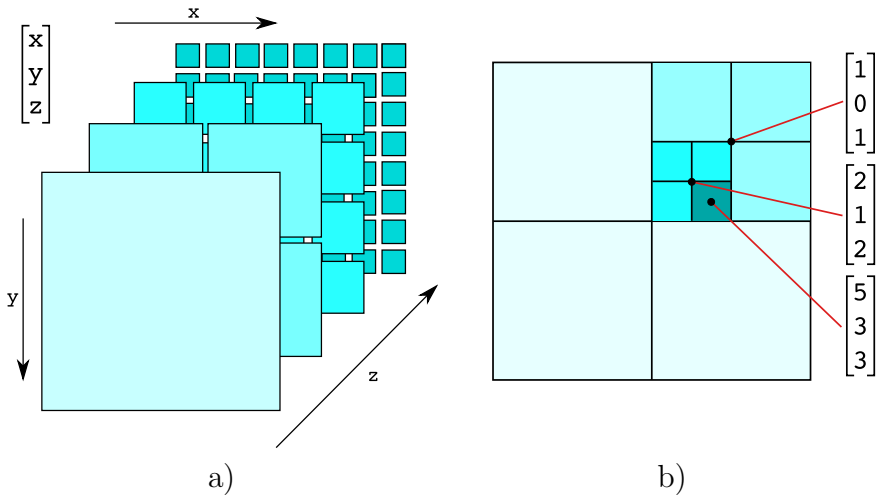
Další možný náhled na jednu souřadnici je, že celá *pracovní plocha* je rozdělena na 2^{2z} čtverců uspořádaných do mřížky. x určuje pořadí obrázku ve směru zleva doprava a y pořadí obrázku ve směru shora dolů.

Ukázka adresace jednotlivých částí pozadí pomocí souborového souřadnicového systému je na obrázku 3.26.

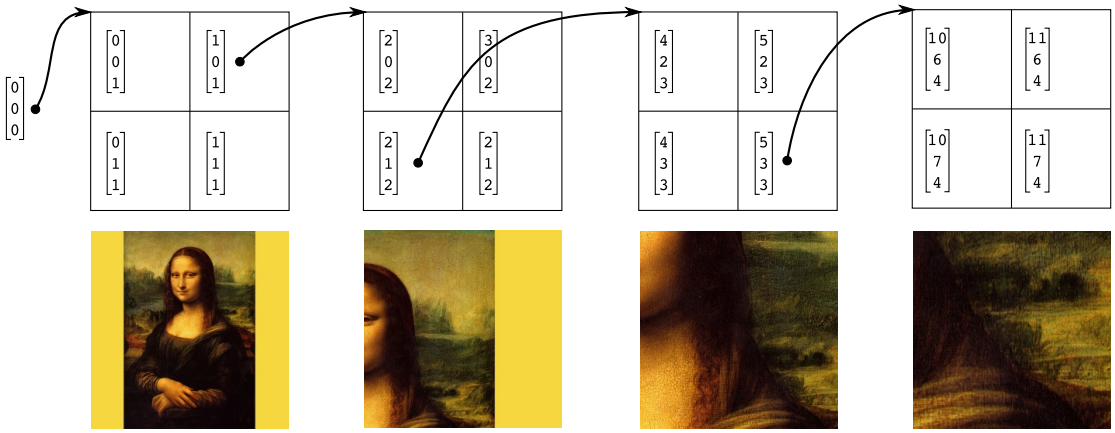
Poloha levého horního rohu obrázku o souřadnicích $[x, y, z]^T$ je v globálních souřadnicích

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x/2^z \\ y/2^z \end{bmatrix}. \tag{1}$$

Pozice ostatních rohů se dají jednoduše odvodit. Například pravý horní roh obrázku je vlastně levým horním rohem pravého souseda o souřadnicích $[x + 1, y, z]^T$. Pro ten platí opět vztah 1.



Obrázek 3.25: a) Schématické znázornění souborového souřadnicového systému. b) Příklad adresování jednotlivých čtverců pozadí.



Obrázek 3.26: Ukázka adresace *pracovní plochy* souborovým souřadnicovým systémem.

3. **Uživatelský souřadný systém** může být zcela libovolný. Je možné jej definovat pomocí tříd `coordConverter` a `dataCoordSystem`. Ty poskytují interface, kde je pouze třeba vyplnit převodní vztah mezi novým souřadným systémem a globálním souřadným systémem.

   Někdy je uživatelský souřadný systém nazýván též **datovým souřadným systémem**, protože v tomto formátu jsou data načítána do aplikace `map`.

   Aplikace `map` má implementovaný souřadný systém WGS84, tedy GPS souřadnice - zeměpisná šířka a délka v radiánech. Příslušné převodní operace provádí třídy `affineConverter` a `geographyCoords`. Nejedná se o plnohodnotnou transformaci, která by platila na povrchu celé zeměkoule. Jedná se o lokální lineární aproximaci, která platí pouze v okolí aproximačního bodu[37]. Vzhledem k velikosti zeměkoule je lokální aproximace funkční v rádiu desítek kilometrů s chybou kolem jednoho

[37] V konfiguračním souboru na přiloženém CD je aproximace provedená ve městě Lysá nad Labem.

metru. Nástroje pro vytvoření lokální aproximace v libovolném bodě na zeměkouli jsou přiloženy na CD.

3.4.2 Třída CIOReader

Jedná se o třídu, jež byla již dvakrát zmiňována v kapitole 3.2.4 a 3.3.2. Účelem třídy je přijímat data přicházející skrze standardní vstup, zpracovávat je a rozesílat pomocí signálů.

Zjednodušená deklarace třídy je na výpisu 3.23.

```
class CIOReader : public QThread
{
public:
  void setActiveCols(int x, int y) {xpos = x; ypos = y;}
signals:
  void dataReceived(double x, double y);
  void endReceived();
};
```

Výpis 3.23: Zjednodušená deklarace třídy `CIOReader` aplikace `map`

Třída neposílá signálem `dataReceived` všechna přijatá data, ale pouze dvě čísla. Jaká data budou posílána, je možné specifikovat metodou `setActiveCols`.

3.4.3 Třída coordConverter

Abstraktní třída `coordConverter` poskytuje interface pro definování uživatelského souřadnicového systému, viz výpis 3.24. Definice souřadnicového systému spočívá pouze v implementaci dvou metod `convert` a `invConvert`. Metoda `convert` převádí bod z uživatelské souřadné soustavy do globální soustavy a metoda `invConvert` realizuje převod opačným směrem. Musí tedy platit `x == invConvert(convert(x))`. Oba převody musí existovat[38] a musí být jednoznačné. Zobrazení z uživatelských souřadnic do globálních souřadnic tedy musí být *bijekcí*, avšak nemusí být lineární.

```
class coordConverter
{
public:
  virtual ~coordConverter() {}
  virtual QPointF convert(QPointF from) = 0;
  virtual QPointF invConvert(QPointF from) = 0;
};
```

Výpis 3.24: Zjednodušená deklarace třídy `coordConverter`

[38] Alespoň pro používanou množinu dat.

3.4.4 Třída affineConverter

Třída affineConverter realizuje *afinní zobrazení*, tedy *posun*, *rotaci*, *zkosení* a *změnu měřítka*, pomocí násobení souřadnice[39] maticí $3x3$.

Jednotlivé prvky matice jsou definovatelné v konfiguračním souboru.

Třída affineConverter je použita k převodu ze souřadnicového systému WGS84 do globálních souřadnic. Jedná se vlastně o nalezení tečné roviny na povrch země. Tato lineární aproximace převodního vztahu je pochopitelně platná pouze lokálně kolem aproximačního bodu.

3.4.5 Třída dataCoordSystem

Uživatelský souřadný systém definovaný pomocí coordConverter může mít i své specifické jednotky. Abstraktní třída dataCoordSystem, viz výpis 3.25, vytváří interface pro definování zacházení s těmito jednotkami.

```
class dataCoordSystem
{
public:
  virtual ~dataCoordSystem() {}
  virtual double getUpperStepX(double siz) = 0;
  virtual double getUpperStepY(double siz) = 0;
  virtual QString labelX(double num) = 0;
  virtual QString labelY(double num) = 0;
  virtual double parseX(const QString &s, bool *convOk = 0) = 0;
  virtual double parseY(const QString &s, bool *convOk = 0) = 0;
  virtual const QString getXAxisName() = 0;
  virtual const QString getYAxisName() = 0;
  virtual const QString getXHint() = 0;
  virtual const QString getYHint() = 0;
};
```

Výpis 3.25: Deklarace třídy dataCoordSystem

Třída dataCoordSystem definuje pět funkcionalit:

- Zaokrouhlování čísel v uživatelském souřadném systému na „pěkné“ hodnoty. Například číslo 0.1572 se dá zaokrouhlit na „pěknou“ hodnotu $0.1\overline{6}$, pokud víme, že zaokrouhlovaná veličina je čas v hodinách. Potom $0.1\overline{6}$ odpovídá času 0:10. Zaokrouhlování na „pěkná“ čísla provádí metody getUpperStepX a getUpperStepY a jsou použity při volbě rozteče mřížky.

- Formátování souřadnic do člověku dobře čitelné podoby (případně i zobrazení jednotek). Pokud by formátovaná veličina byla například velikost souboru, potom

[39] Souřadnice je nejprve rozšířena na *homogenní*.

číslo 78643200 zapsané v člověku čitelné podobě je 75MB. Tuto funkci vykonávají metody `labelX` a `labelY`.

- Převod textových řetězců zpět na čísla. Tuto funkci poskytují metody `parseX` a `parseY`. Jedná se vlastně o inverzní operaci k metodám `label`.
- Pojmenování veličiny - navrácení textového řetězce s názvem veličiny. Tuto funkci poskytují metody `getXAxisName` a `getYAxisName`.
- Poskytnutí nápovědy pro uživatelský vstup. Když uživatel zadává textový řetězec, jímž chce vyjádřit hodnotu určité veličiny, tak metody `getXHint` a `getYHint` by mu v tom měly pomoci. Například u zadávání data by nápověda mohla vypadat: `'YYYY-MM-DD'`.

3.4.6 Třída `geographyCoords`

Třída `geographyCoords` implementuje jednotky soustavy WGS84, tedy stupně, minuty a vteřiny.

Nejvíce ilustrativní jsou příklady volání metod třídy `geographyCoords`, které jsou v tabulce 3.5.

Volání metody	**Výsledek volání**
`getUpperStepX(7.1564)`	`10 (10°0'0'')`
`getUpperStepY(7.1564e-3)`	`8.3333e-3 (0°0'30'')`
`labelX(5.8333e-2)`	`"0°3'30''E"`
`labelY(-5.8333e-2)`	`"0°3'30''S"`
`parseX("0°3'30''W")`	`-5.8333e-2`
`parseY("0°3'30''N")`	`5.8333e-2`
`getXAxisName()`	`"Longitude"`
`getYAxisName()`	`"Latitude"`
`getXHint()`	`"14°25'1.095''E"`
`getYHint()`	`"50°4'35.557''N"`

Tabulka 3.5: Příklady volání metod třídy `geographyCoords` a jejich výsledky.

3.4.7 Třída `mapObject`

Třída `mapObject` v sobě obsahuje jeden čtvercový obrázek a jeho souřadnice[40], což tvoří jeden fragment pozadí.

Jak je vidět z výpisu 3.26, třída `mapObject` se umí načíst ze souboru pomocí konstruktoru a uložit zpět do souboru pomocí metody `saveToFile`. Dále pak je možné třídu naplnit z pole bytů metodou `loadFromBA`, čehož je využito při načítání obrázků z internetu.

[40] V souborovém souřadném systému i v globálním souřadném systému.

Proměnné `coordX`, `coordY` určují polohu středu obrázku v globálních souřadnicích. Proměnná `coord2pixel` definuje poměr jedné jednotky v globálních souřadnicích ku počtu pixelů obrázku. Díky `coord2pixel` je možné stanovit velikost obrázku při vykreslování na okno. `mapObject` dále obsahuje proměnné `fscX`, `fscY` a `fscZ`, které tvoří jednu souřadnici v souborovém souřadném systému. Proměnné [`fscX`, `fscY`,`fscZ`] a [`coordX`, `coordY`, `coord2pixel`] obsahují prakticky stejnou informaci (polohu a rozměr) a jsou tedy mezi sebou navzájem převoditelné.

Tato ambivalence v souřadnicích je pohodlná, protože pro každý druh práce se hodí jiný souřadný systém. Například pro vykreslování je vhodnější globální souřadnicový systém. Při načítání z internetu, ukládání do souboru, nebo zjišťování, je-li obrázek načten, je naopak vhodný souborový souřadný systém. Jelikož při každém překreslení okna jsou potřeba souřadnice v obou formátech, jedná se zároveň o výkonovou optimalizaci.

Na druhou stranu dualita v datech skrývá nebezpečí. K souřadnicím je však přistupováno pouze metodami objektu, které vždy zaručí shodnost dat.

```
class mapObject
{
public:
    mapObject(const QString & fileName);
    void loadFromBA(const QByteArray & ba);
    bool saveToFile(const QString & fileName);
    void paintSelf(QPainter *painter, double left, double top,
                   double right, double bot, double c2p) const;
private:
    QImage img;
    double coordX, coordY, coord2pixel;
    int fscX, fscY, fscZ;
};
```

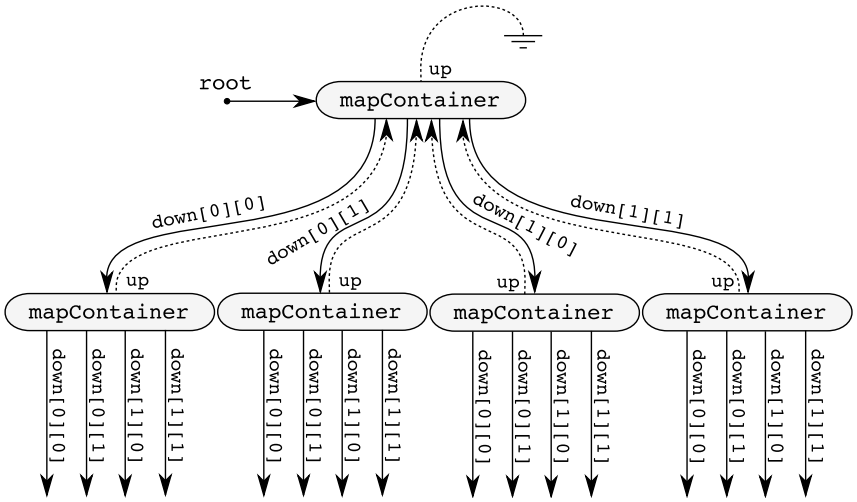
Výpis 3.26: Zjednodušená deklarace třídy `mapObject`

Třída `mapObject` se umí sama vykreslit pomocí metody `paintSelf`, které se jako parametry předají souřadnice levého horního a pravého dolního rohu vykreslované plochy. S těmito souřadnicemi je sdělena i proměnná `c2p` určující poměr velikosti vykreslované plochy v globálním souřadnicovém systému vůči počtu pixelů plochy. Na základě těchto informací `mapObject` určí, kam na plochu se má vykreslit a případně provede zvětšení/zmenšení, aby `c2p == coord2pixel`.

3.4.8 Třída `mapObjectTree`

Třída `mapObjectTree` má na starosti správu tříd `mapObject` v paměti. Vnitřně je implementována pomocí stromového spojového seznamu, který tak opisuje souborový souřadný systém. Situace je znázorněna na obrázku 3.27.

Zjednodušená deklarace třídy je na výpisu 3.27. Pomocí metody `add` je možné objekt uložit do stromu. Pomocí metody `contains` je možné dotázat se na přítomnost objektu



Obrázek 3.27: Vnitřní struktura třídy `mapObjectTree`.

ve stromu a pomocí metody `get` je možné objekt ze stromu vyzvednout[41]. Metody `add`, `contains` a `get` mají logaritmickou časovou složitost.

```
class mapObjectTree
{
public:
    bool add(mapObject* m);
    bool contains(int x, int y, int z);
    mapObject* get(int x, int y, int z);
private:
    mapContainer* root;
    int siz;
};
```

Výpis 3.27: Zjednodušená deklarace třídy `mapObjectTree`

Protože samotná třída `mapObject` nedokáže vytvořit stromovou strukturu, je nejprve `mapObject` uložen do schránky `mapContainer`. Jak je vidět na výpisu 3.28 a obrázku 3.27, struktury `mapContainer` je již možné řadit do stromu.

```
struct mapContainer
{
  mapObject* mapo;
  mapContainer *up, *down[2][2];
};
```

Výpis 3.28: Zjednodušená deklarace struktury `mapContainer`

[41] Pod pojmem vyzvednutí je myšleno pouze navrácení ukazatele. `mapObject` není metodou `get` ze stromu vyjmut.

Do třídy `mapObjectTree` je možné přidat listový objekt (například $[1, 0, 2]^T$), aniž byl předtím přidán jeho rodič ($[0, 0, 1]^T$). Tato situace je řešena tak, že se vytvoří prázdné schránky[42] `mapContainer`, aby existovala cesta od kořene k přidávanému listu.

3.4.9 Třída `mapManager`

Úkolem třídy `mapManager` je „obstarávat“ mapy - příslušné instance třídy `mapObject`. Třída `mapManager` disponuje třemi „zdroji“ požadovaných map:

- Paměť. Pokud již byla jednou instance třídy `mapObject` získána (ze souboru / z internetu), je uložena v paměti, uvnitř třídy `mapObjectTree`.
- Souborový repositář - adresář s obrázky map, který je uvedený v konfiguračním souboru. Každý repositář musí obsahovat *deskriptivní soubor* - textový soubor, kde každý řádek obsahuje jméno jednoho obrázku a následně jeho souřadnice v souborovém souřadném systému.
- Internet. Stahování požadovaných obrázků s mapou z WMS serveru, který je definován v konfiguračním souboru.

Protože získávání obrázků map může být časově náročné, pracuje třída `mapManager` v samostatném vláknu. Zjednodušená deklarace třídy je na výpisu 3.29.

```
class mapManager : public QThread
{
public:
    mapManager(QString repositary, QString fileFormat, QString repositaryFile,
               QString server, int maxZoom, QObject *parent);
    void coverArea(double centerX, double centerY, double radius, double c2p_ratio);
public slots:
    void webReplyFinished(QNetworkReply *rep);
public signals:
    void mapUpdated(QList<mapObject*> newList);
private:
    QList<mapObject*> tempMapOList;
    QList<QPair<QString,imgCoords> > reposFile;
    QNetworkAccessManager *manager;
    mapObjectTree mot;
    bool loadReposDesc();
    bool saveMapToFile(mapObject *map);
    bool getMapFromWeb(int x, int y, int z);
    bool getMapFromFile(int x, int y, int z);
};
```

Výpis 3.29: Zjednodušená deklarace třídy `mapManager`

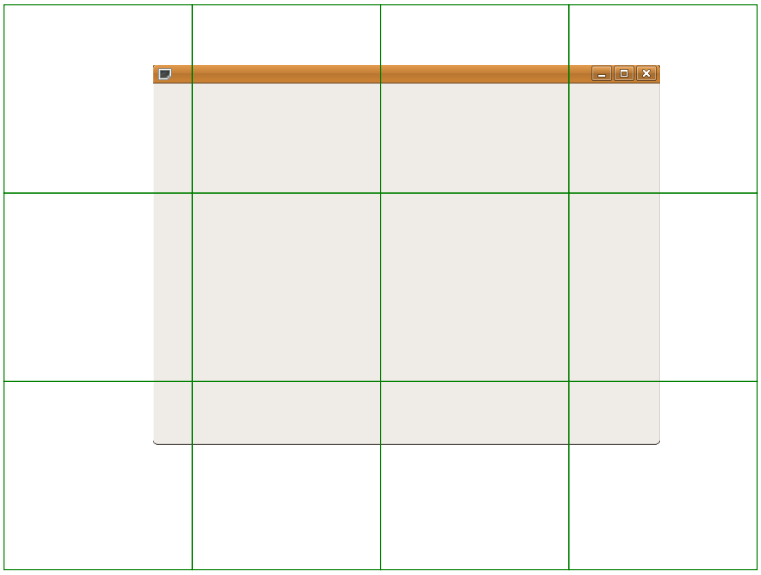
[42] Schránky, které neobsahují `mapObject`. Jejich proměnná `mapo == NULL`.

Třída `mapManager` má vlastní smyčku zpráv, což umožňuje používání signál-slotového mechanismu. Ten je použit ke komunikaci s ostatními vlákny, čímž odpadají problémy s mezivláknovou komunikací. Signál-slotový mechanismus je již *thread safe*.

V konstruktoru třídy `mapManager` se specifikují zdroje obrázků. `repositary` určuje adresář s obrázky - souborový repositář. Pokud adresář neexistuje, je automaticky vytvořen. `fileFormat` sděluje, jaký formát obrázků je v repositáři používán (`jpeg`, `png`, `bmp` ...). To určuje, v jakém formátu se budou obrázky do repositáře ukládat a v jakém formátu budou obrázky získávány z internetu. `repositaryFile` ukazuje na deskriptivní soubor repositáře. Pokud neexistuje, je automaticky vytvořen a repositář je považován za prázdný. Čtvrtý parametr `server` je adresa WMS serveru, ze kterého budou obrázky stahovány. Pokud adresa není uvedena, stahování map z internetu nebude použito. Poslední parametr `maxZoom` určuje, do jakého maximálního přiblížení jsou obrázky na WMS serveru k dispozici. Číslo má význam třetí souřadnice souborového souřadnicového systému.

Vlastní požadavek na „získání“ map se provádí metodou `coverArea`. Jako parametr se specifikuje souřadnice středu plochy, kterou chceme pokrýt obrázky. Dále pak je uveden rozměr plochy a poměr jednotky globálního souřadnicového systému na počet pixelů, z čehož se vyvodí, v jakém přiblížení je třeba obrázky dodat.

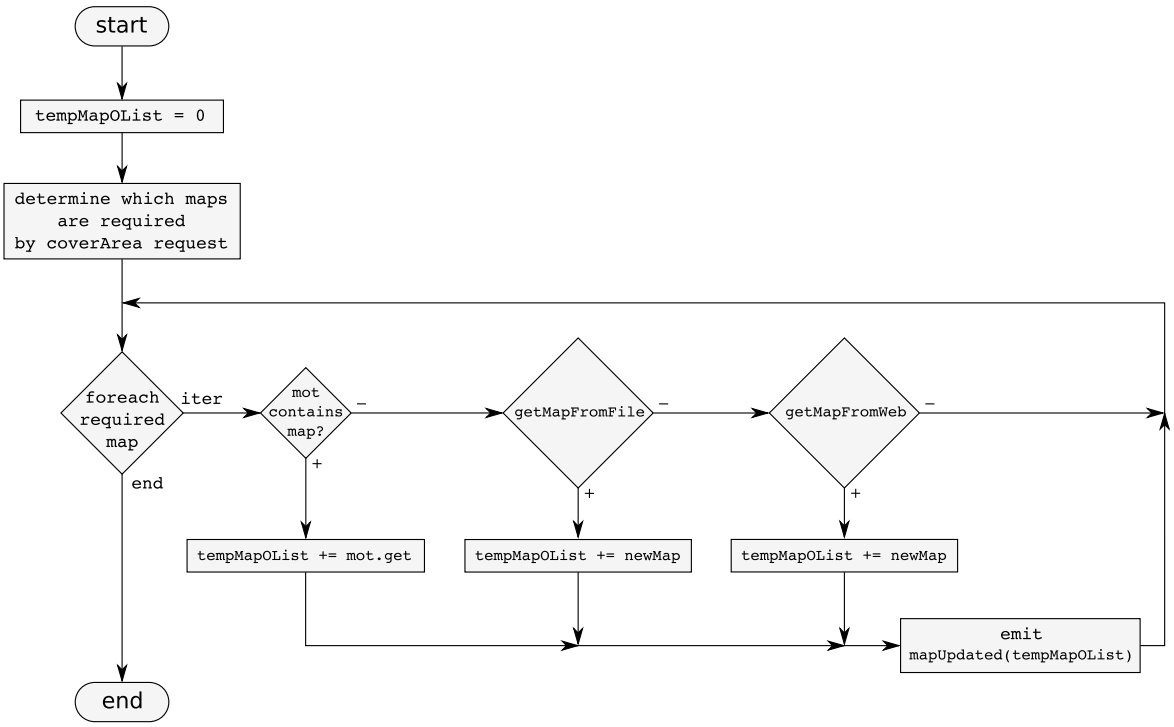
Odpověď na volání metody `coverArea` je emitování signálu `mapUpdated`, který obsahuje nové mapy. Ukázka pokrytí plochy mapami obsaženými v signálu `mapUpdated` je na obrázku 3.28.



Obrázek 3.28: Ukázka pokrytí okna obrázky obsaženými v signálu `mapUpdated`.

Způsob, jakým metoda `coverArea` obstará jednotlivé mapy, je na obrázku 3.29. Metoda nejprve určí, které mapy budou pro pokrytí požadované oblasti potřeba. Poté začne jednu mapu po druhé obstarávat. Nejprve se podívá do paměti, proměnné `mot`, není-li

již mapa načtena. Pokud ano, přiřadí ji do seznamu a odešle signál informující o aktualizaci map. Pokud mapa není k dispozici v paměti, použije se metoda `getMapFromFile`, která se pokusí mapu načíst ze souborového repositáře. Pokud ani tentokrát není mapa získána, dojde k pokusu o její stažení z WMS serveru pomocí metody `getMapFromWeb`.



Obrázek 3.29: Algoritmus získání map metodou `coverArea`.

Metoda `getMapFromFile` využívá proměnné `reposFile`, která obsahuje deskriptivní soubor z repositáře. Ozrcadlení obsahu deskriptivního souboru do proměnné `reposFile` je provedeno metodou `loadReposDesc` v konstruktoru. Proměnná `reposFile` je seznam dvojic {jméno souboru, souřadnice v souborovém souřadném systému}. Souřadnici reprezentuje struktura `imgCoords`, viz výpis 3.30.

```
struct imgCoords
{
 int x,y,z;
};
```

Výpis 3.30: Zjednodušená deklarace struktury `imgCoords`

Metoda `getMapFromWeb` získává obrázky z WMS serveru pomocí třídy `QNetworkAccessManager` (součást knihovny Qt od verze 4.4.). WMS server navrátí obrázek mapy jako odpověď na klasický HTTP požadavek. Úkolem metody `getMapFromWeb` tedy je sestavit příslušný HTTP dotaz. Pro komunikaci se serverem byla zvolena souřad-

nicová soustava `EPSG:102067`[43].

Odpověď WMS serveru je obsloužena metodou `webReplyFinished`, kerá vytvoří instanci třídy `mapObject` a načte do ní příslušný obrázek. Nově vzniklá instance `mapObject` je uložena do paměti pomocí proměnné `mot` a zároveň i do souborového repositáře pomocí metody `saveMapToFile`.

Metoda `saveMapToFile` se krom uložení do souboru postará o aktualizaci proměnné `reposFile` i samotného deskriptivního souboru v repositáři.

3.4.10 Třída `mapWidget`

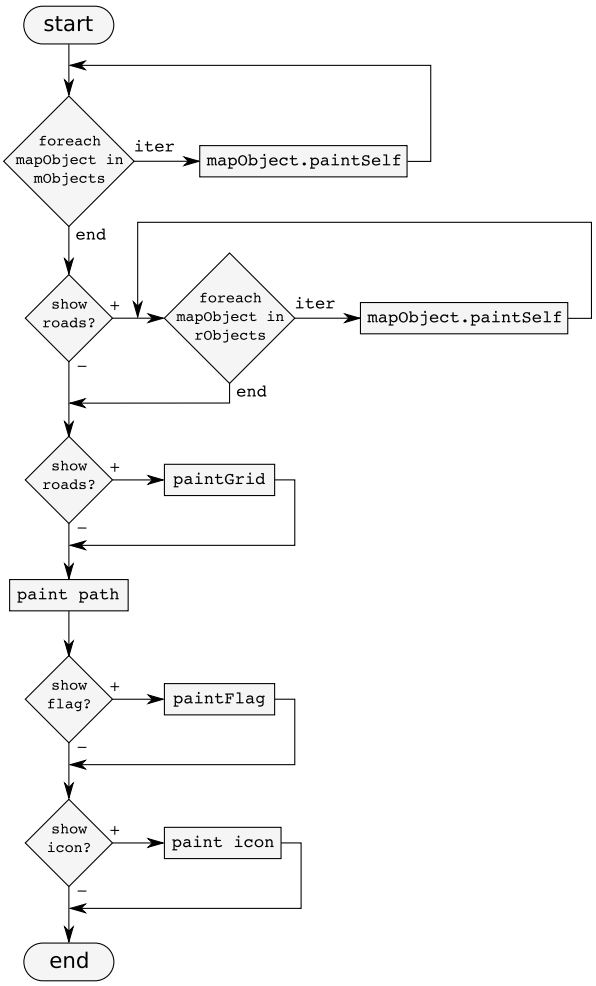
Úkolem třída `mapWidget` je vykreslovat dvě vrstvy mapy, mřížku a trajektorii pohybu. Zjednodušená deklarace tříty `mapWidget` je na výpisu 3.31.

```
class mapWidget : public QWidget
{
public:
    void setIcon(const char *iconFile);
public slots:
    void addDataSlot(double a, double b);
    void backgroundUpdate(QList<mapObject*> newList);
    void roadsUpdate(QList<mapObject*> newList);
private:
    QList<mapObject*> mObjects, rObjects;
    QList<gridLine> grid;
    QPointF *pathPoints;
    QPointF *pathData;
    unsigned char *pathAlpha;
    coordConverter *coordConv;
    mapManager *mManag, *rManag;
    dataCoordSystem *gcoords;
    mapPathStyle mps;
    QPixmap *icon;
    double scaleAbsolute, centerX, centerY;
    void transformPath();
    void createNewGrid();
    void paintGrid(QPainter &painter);
    void paintFlag(QPainter &painter);
    void resizeBy(double resizeFactor);
    void shiftBy(QPoint offset);
    void countNewCoords();
};
```

Výpis 3.31: Zjednodušená deklarace třídy `mapWidget`

[43] Viz [10].

Překreslení widgetu probíhá podle algoritmu z obrázku 3.30. Nejprve jsou vykresleny všechny instance třídy `mapObject` obsažené v seznamu `mObjects`. Volitelně pak může být vykreslen seznam `rObjects`. Dále pak může být vykreslena mřížka metodou `paintGrid` a následně je vykreslena cesta - trajektorie pohybu. Nakonec je vykreslen praporek metodou `paintFlag` a ikona.



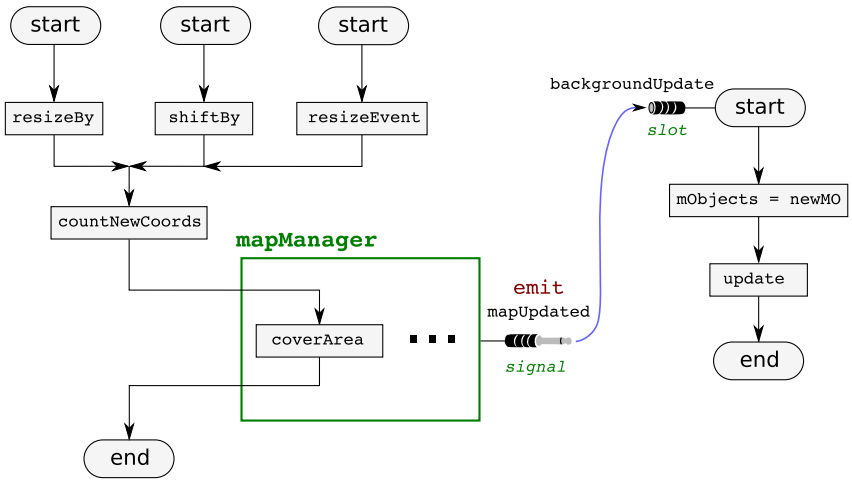
Obrázek 3.30: Algoritmus překreslení widgetu `mapWidget`.

Souřadnice středu plochy udržují proměnné `centerX` a `centerY`. Přiblížení určuje proměnná `scaleAbsolute`, která vyjadřuje poměr velikosti plochy v globálních souřadnicích ku velikosti plochy v pixelech.

Třída `mapWidget` odchytává události stisknutí klávesy a kliknutí myší, na základě kterých posouvá a přibližuje/oddaluje náhled na *pracovní plochu*. To je prováděno pomocí metod `resizeBy` a `shiftBy`. Tyto metody upravují proměnné `centerX`, `centerY` a `scaleAbsolute` a následně volají metodu `countNewCoords`. Metoda `countNewCoords` i při události `resizeEvent`.

Metoda `countNewCoords` přepočítá globální souřadnice levého horního rohu widgetu a pravého dolního rohu widgetu. Následně provede volání metody `coverArea` proměn-

ných mManag a rManag. Tímto způsobem jsou získávány mapové podklady. Celá situace je znázorněna na obrázku 3.31.



Obrázek 3.31: Algoritmus získání nových mapových podkladů třídy mapWidget.

Volání metody coverArea způsobí, že proměnné mManag a rManag emitují signál s novými mapami. Na tyto signály jsou zachyceny sloty backgroundUpdate a roadsUpdate, které uloží nové seznamy map a vynutí další vykreslení voláním update.

Vykreslování *cesty* - trajektorie pohybu

Cesta je vytvářena z dat načtených pomocí slotu (metody) addDataSlot. Tento slot je napojen na odpovídající signál třídy CIOReader. Jednotlivé body cesty jsou uchovávány v polích pathData a pathPoints. Pole pathData udržuje data v původním formátu, tedy v uživatelské souřadné soustavě. Pole pathPoints obsahuje data v pixelových souřadnicích vůči levému hornímu rohu widgetu. Přepočet z uživatelské souřadnicové soustavy do globální souřadnicové soustavy i do pixelových současnic[44] je proveden v těle metody countNewCoords.

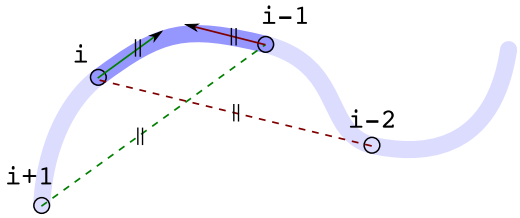
Cesta se může vykreslovat dvěma způsoby - pomocí úseček nebo pomocí Bézierových křivek. To definuje proměnná mps, která je výčtového typu mapPathStyle, viz výpis 3.32.

```
enum mapPathStyle {simpleLines, simpleCurves};
```

Výpis 3.32: Deklarace výčtového typu mapPathStyle

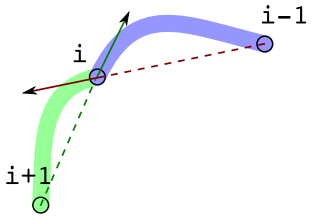
Při vykreslování cesty pomocí Bézierových křivek je třeba znát v každém bodě křivky její směrnici. Ta se stanoví jako rozdíl souřadnic okolních bodů, viz obrázek 3.32. Jedná se o určení oboustranné diference pro každý bod. To má za následek, že koncové body cesty nemohou být vykresleny. Při on-line vykreslování to má za následek, že vykreslování cesty je o jeden vzorek zpožděno za aktuálním stavem.

[44] Na základě znalosti polohy okna: [centerX,centerY])



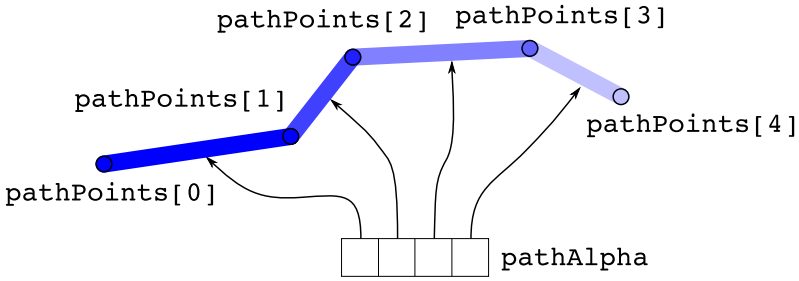
Obrázek 3.32: Vykreslování cesty pomocí Bézierových křivek ve třídě `mapWidget`.

Pokud by se použila pouze jednostranná diference k určení směrnice křivky, nedošlo by k hladkému napojení jednotlivých úseků. Diference v jednom bodě křivky bude různá zleva i zprava. Situaci demonstruje obrázek 3.33.



Obrázek 3.33: Ukázka nevhodnosti použití jednostranné diference při vykreslování cesty ve třídě `mapWidget`.

Vykreslovaná cesta může postupně slábnout, aby se odlišila historická data od aktuálních. Toho je dosaženo pomocí barevného alfa kanálu, který určuje průhlednost. Každému úseku cesty odpovídá jedna hodnota z pole `pathAlpha`, které obsahuje informaci o alfa kanálu. Hodnoty v poli `pathAlpha` představují klesající posloupnost čísel. Situaci znázorňuje obrázek 3.34.



Obrázek 3.34: Efekt „ztrácející se“ cesty pomocí barevného alfa kanálu, třída `mapWidget`.

Vykreslení ikony

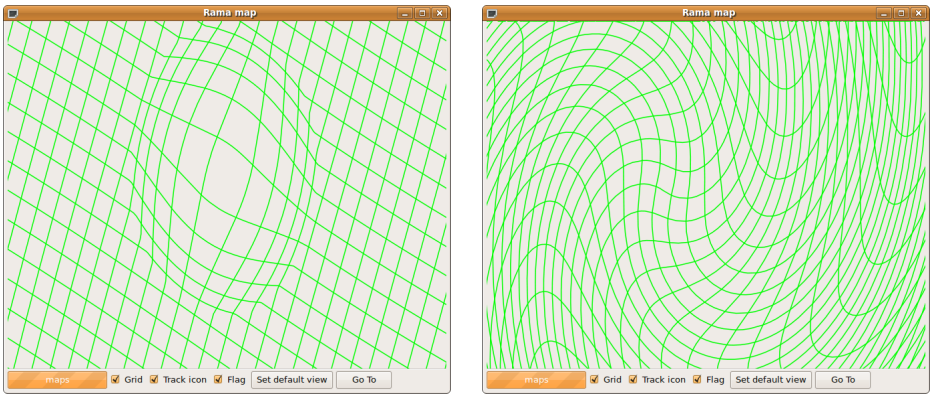
Pomocí metody `setIcon` je možné načíst malý obrázek, který bude vykreslován jako dekorace začátku cesty - aktuální pozice. Jako parametr metody se předává cesta k souboru s ikonou. Může se jednat o libovolný formát, doporučovaný však je `png` z důvodů průhlednosti pozadí obrázku.

Pokud je požadováno, ikona je před vykreslením orotována podle směrnice v prvním bodě cesty. Tato směrnice je počítána pomocí jednostranné diference.

Voliteně je u ikony vykreslován i praporek s číselnou informací o poloze. O formátování souřadnic polohy se stará proměnná `gcoords`, instance třídy `dataCoordSystem`. Vykreslení praporku má na starost metoda `paintFlag`.

Vykreslení mřížky

Transformace z uživatelských souřadnic do globálních nemusí být lineární a mřížka musí svým zakřivením tuto nelinearitu respektovat. Na obrázku 3.35 jsou ilustrativní ukázky chování mřížky pro nelineární transformace.



Obrázek 3.35: Ukázky zakřivení mřížky při nelineární transformaci z uživatelské souřadnicové soustavy.

Jedna křivka mřížky je reprezentována strukturou `gridLine`, jejíž deklarace je na výpisu 3.33.

```
struct gridLine
{
    QPointF* points;
    int pointsCount;
    QString label;
    QPointF textPos;
};
```

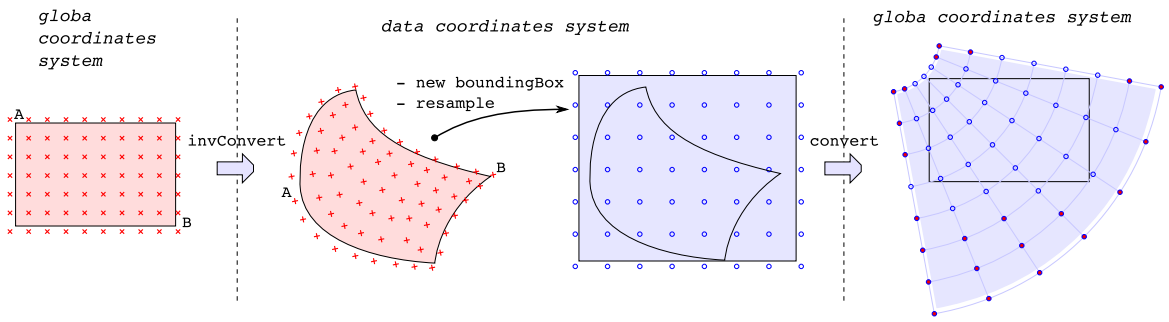
Výpis 3.33: Zjednodušená deklarace struktury `gridLine`

Struktura `gridLine` obsahuje body křivky, popisek a polohu popisku. Celá mřížka je uložena v proměnné `grid` - spojovém seznamu struktur `gridLine`. Vykreslení mřížky je prováděno metodou `paintGrid`.

Protože výpočet mřížky může být relativně náročný, je mřížka napočítána předem metodou `createNewGrid`.

Algoritmus metody `createNewGrid` je znázorněn na obrázku 3.36. Viditelná oblast je pravidelně ovzorkována a vzniklá množina bodů je převedena zpět do uživatelského

(datového) souřadnicového systému pomocí metody `invConvert` proměnné `coordConv`. Dále je určen obdélník, který uzavírá všechny transformované body. Obdélníková oblast v uživatelském souřadnicovém systému je opět pravidelně navzorkována a tyto body jsou převedeny zpět do globálního souřadnicového systému. Spojením vzniklých bodů získáme mřížku, která respektuje nelinearitu uživatelské transformace.



Obrázek 3.36: Algoritmus metody `createNewGrid` pro výpočet nové mřížky.

Jak je vidět na obrázku 3.36, některé body mřížky, vypočítané pomocí metody `createNewGrid`, mohou ležet mimo viditelnou oblast. Aby se při vykreslování mřížky nevykreslovaly neviditelné úseky, jsou z mřížky vyjmuty všechny body, které neleží ve viditelné oblasti ani nemají přímého souseda, který leží ve viditelné oblasti. Tyto body jsou na obrázku označeny červenými kroužky.

3.4.11 Třída `lineSmoother`

Třída `lineSmoother`, viz výpis 3.34, má za úkol vyhladit křivky mřížky přidáváním dalších bodů nebo naopak zjednodušovat křivky odebíráním bodů.

```
class lineSmoother
{
public:
    lineSmoother(coordConverter *coordConv);
    void setSmoothLimits(double min, double max);
    void load(QPointF *buf, unsigned siz);
    void copyTo(QPointF *buf, unsigned siz);
    int makeNice();
private:
    coordConverter *cc;
    container *first;
    double aproxLen;
    double deviation(container *c);
};
```

Výpis 3.34: Zjednodušená deklarace třídy `lineSmoother`

Jako parametr konstruktoru se předává instance třídy `coordConverter` (přesněji řečeno jeho potomka), která sděluje transformační vztah mezi uživatelským a globálním souřadnicovým systémem.

Vnitřně třída `lineSmoother` uchovává jednotlivé body uložené v oboucestném spojovém seznamu. To umožňuje rychlé přidávání a ubírání bodů z libovolného místa křivky. Spojový seznam je tvořen pomocí struktur `container`, viz výpis 3.35. Každá instance struktury `container` udržuje právě jeden bod křivky v proměnné `point` a ukazatele na předchozího a následujícího souseda ve spojovém seznamu.

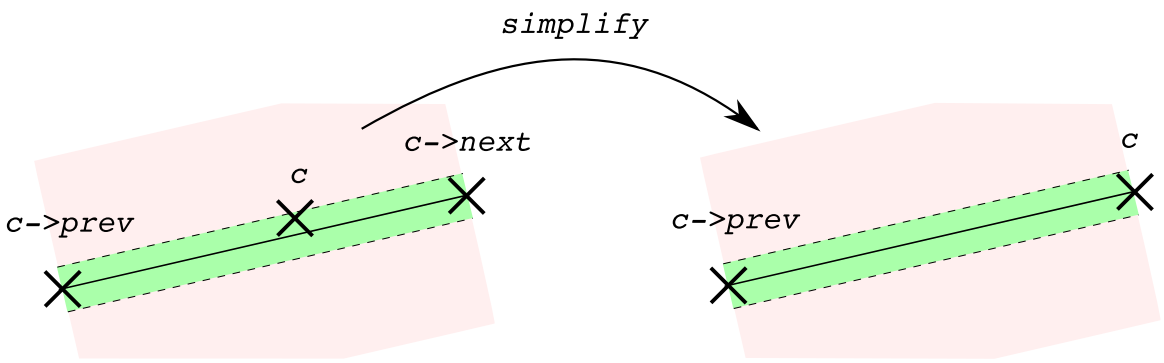
```
struct container
{
    QPointF point;
  container *prev, *next;
};
```

Výpis 3.35: Zjednodušená deklarace struktury `container`

Pomocí metody `load` třída `lineSmoother` načte body křivky z pole bodů `buf` a pomocí metody `copyTo` je nakopíruje zpět do pole bodů `buf`. Metody `load` a `copyTo` realizují převod mezi klasickým polem bodů typu `QPointF*` a spojovým seznamem. Metoda `load` navíc spočítá přibližnou délku načítané křivky a uloží ji do privátní proměnné `aproxLen`.

Vlastní vyhlazení/zjednodušení provádí metoda `makeNice`. Ta postupně projde celou křivku a pro každý bod[45] podle potřeby provede *zjednodušení* nebo *expanzi*.

Operace *zjednodušení* je znázorněna na obrázku 3.37. Pro bod `c` jsou nalezeni jeho nejbližší dva sousedé, označme je `c->prev` a `c->next`. Body `c->prev` a `c->next` je proložena přímka a následně je určena vzdálenost bodu `c` od této přímky. Pokud vzdálenost nepřekročí určitou toleranci, považuje se, že body `c->prev`, `c` a `c->next` leží na jedné přímce. V takovém případě je možné bod `c` odebrat, aniž by to ovlivnilo tvar křivky.



Obrázek 3.37: Operace zjednodušení křivky prováděná metodou `makeNice`.

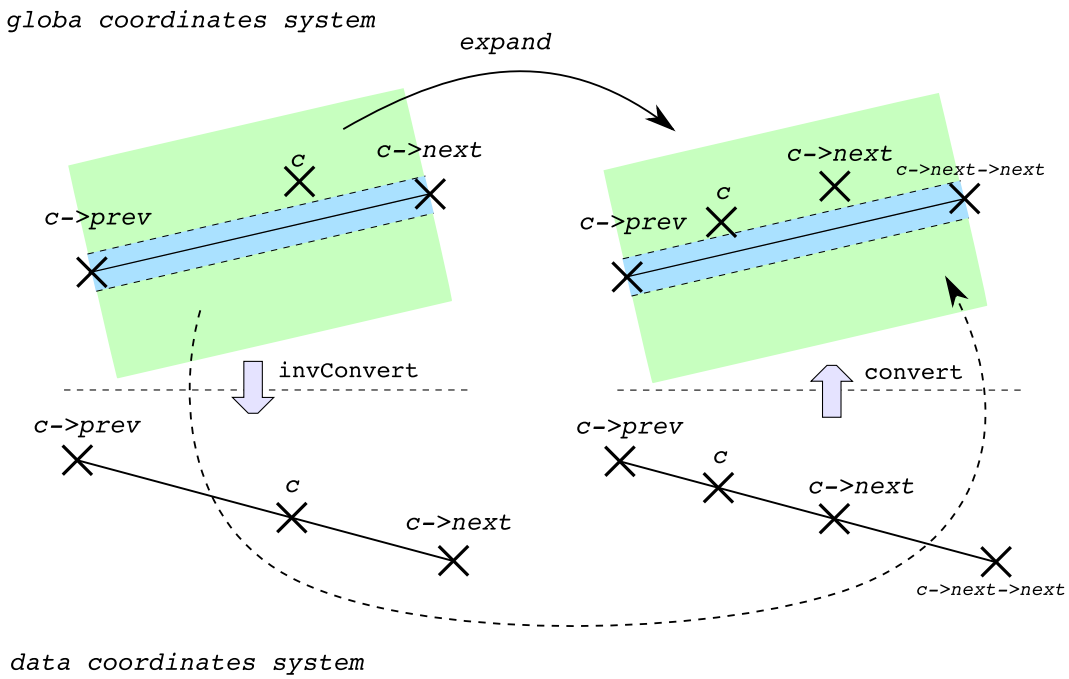
Operace *expanze* je znázorněna na obrázku 3.38. Pro bod `c` jsou nalezeni jeho nejbližší dva sousedé, označme je `c->prev` a `c->next`. Body `c->prev` a `c->next` je proložena

[45] Pro každý bod vyjma koncových dvou.

přímka a následně je určena vzdálenost bodu `c` od této přímky. Pokud vzdálenost překročí určitou mez, je přidán další bod křivky. Přidání nového bodu probíhá v uživatelské souřadnicové soustavě, kde leží všechny body na jedné přímce. Souřadnice nově vznikajícího bodu `newC` lze jednoduše určit jako

$$newC = \frac{c \rightarrow prev + c}{2}. \tag{2}$$

Nový bod `newC` je přidán do spojového seznamu a převeden zpět do globální souřadnicové soustavy.



Obrázek 3.38: Operace expanze křivky prováděná metodou `makeNice`.

Při operaci *zjednodušení* a *expanze* je třeba určovat vzdálenost bodu od přímky. Tuto operaci provádí privátní metoda `deviation`, které se jako parametr předá bod křivky a ona vrátí jeho vzdálenost od přímky procházející sousedními body.

Při rozhodování, provede-li se *zjednodušení* či *expanze*, se používají meze, které se dají nastavit pomocí metody `setSmoothLimits`. Pokud je vzdálenost bodu od křivky menší než `min`, je bod odebrán operací *zjednodušení*. Pokud je vzdálenost větší než `max`, je bod přidán operací *expanze*. Tyto vzdálenosti jsou přitom brány relativně, vůči délce křivky uložené v proměnné `aproxLen`.

Metoda `makeNice` vrací jako návratovou hodnotu počet bodů vyhlazené křivky. Tato informace udává, jak velké pole bodů je třeba připravit, aby následné volání `copyTo` proběhlo úspěšně.

3.4.12 Třída `Widget`

Třída `Widget` tvoří podkladovou plochu pro `mapWidget` a načítá nastavení z konfiguračního souboru. Dále se stará o vytvoření instancí všech potřebných tříd a propojení jejich signálů a slotů. Plní tedy stejnou funkci, jako třídy `MainWidget` v aplikacích `graph` a `horizont`, viz kapitola 3.2.5.

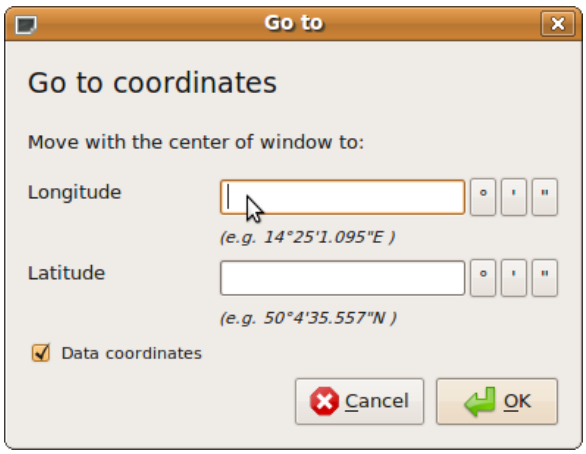
Krom výše zmíněného navíc třída `Widget` zobrazuje stavovou lištu s ovládacími a informativními prvky. Viz obrázek 3.39.

Obrázek 3.39: Stavová lišta aplikace `map`.

Na stavové liště nalezneme prvky pro ovládání vykreslování druhé vrstvy s popisky, mřížky a praporku. Dále můžeme zapnout/vypnout sledovací mód[46], vyvolat zobrazení `goToDialogu` a uložit aktuální pozici náhledu na mapu. Uložení náhledu provede zápis souřadnic středu okna do konfiguračního souboru. Při příštím spuštění aplikace se střed okna automaticky posune na tyto souřadnice.

3.4.13 Třída `goToDialog`

Třída `goToDialog` má za úkol zobrazovat dialogové okno, viz obrázek 3.40, které nabízí posunutí středu náhledu na libovolnou pozici. Souřadnice mohou být zadávány v globálním souřadnicovém systému nebo v uživatelském souřadnicovém systému. Při použití uživatelského souřadnicového systému jsou využity metody třídy `geographyCoords`, které udávají názvy souřadnic, převádějí textové řetězce na čísla a poskytují uživatelskou nápovědu[47].



Obrázek 3.40: Dialogové okno „posunu na lokaci“ aplikace `map`.

[46] Začátek vykreslované cesty je vždy ve středu okna.
[47] Text v závorce pod každým `textBoxem`.

3.5 Pomocné utility `concater`, `atee` a `gct`

Kromě hlavních aplikací `gatalog`, `graph`, `horizont` a `map` je systém utilit doplněn o několik malých pomocných nástrojů, které zjednodušují práci nebo přizpůsobují výstup, aby přesně odpovídal zadání.

3.5.1 Utilita `concater`

Jak bylo zmíněno v popisu aplikace `datalog` v kapitole 3.1, jednou instancí programu `datalog` je možné přijímat právě jeden synchronní packet.

Řídicí jednotka vrtulníku přitom posílá dva synchronní packety:

- Packet s číslem `0x35` o frekvenci 64Hz, který obsahuje telemetrické informace.
- Packet s číslem `0x40` o frekvenci 1Hz, který obsahuje informace z GPS jednotky.

Pro příjem obou synchronních packetů jsou vytvořeny dvě instance programu `datalog` a vznikají tak dva datové proudy. V zadání je požadováno spojit tyto dva proudy dat a uložit je do jediného souboru.

Utilita `concater` akceptuje dva vstupní soubory a sloučeně je vypisuje na standardní výstup.

Sloučený soubor obsahuje na jednom řádku data z obou synchronních packetů. V okamžiku, kdy nejsou k dispozici aktuální GPS data, je na výstup vypsána poslední známá hodnota. To má za následek, že ve sloupcích odpovídajících GPS packetu je vždy 64x zopakována tatáž hodnota.

3.5.2 Utilita `atee`

Utilita `atee` funguje jako standardní Linuxová utilita `tee`. Všechna data ze standardního vstupu opakuje na standardní výstup a zároveň je zapisuje do souboru.

Kromě toho má utilita `atee` jednu speciální vlastnost. Pokud se ukončí libovolná instance `atee`, dojde k ukončení i všech ostatních instancí `atee`.

Utilita `atee` najde uplatnění, pokud je paralelně spuštěno několik instancí `datalog`, na které jsou zachyceny grafické aplikace. Pokud dojde k zavření jedné grafické aplikace, automaticky se zavřou i ostatní. Použití demonstruje výpis 3.36.

```
./datalog telemetrie.conf | ./atee tdata | ./graph &
./datalog gps.conf | ./atee gdata | ./map &
```

Výpis 3.36: Příklad spuštění svou instancí aplikace `datalog`

Utilita `atee` je napsána v čistém jazyce `C`. Ke komunikaci mezi jednotlivými instancemi je použit systémový semafor.

Při spuštění utilita `atee` zažádá o systémový semafor. Pokud neexistuje, vytvoří jej. Následně nastaví jeho hodnotu na 0. Pokud dojde k ukončování aplikace `atee`, navýší se hodnota v semaforu. Pokud jiná instance `atee` za běhu zjistí, že semafor není na nule, ukončí se také.

3.5.3 Utilita `gct`

Utilita `gct` úzce souvisí s lokální aproximací souřadného systému WGS84 pro aplikaci `map`.

Tato aproximace se provádí tak, že se na mapě v místě, kde chceme lokální aproximaci provést, najde co možná nejvíce opticky výrazných bodů u nichž je známá GPS souřadnice. Nabízí se objekty jako komíny továren, křižovatky, rohy velkých budov atd.

Tyto objekty se naleznou i v aplikaci `map` a pomocí pravého tlačítka myši se určí jejich poloha v globálním souřadném systému.

Následně pak pomocí scriptu[48] `rmtf.m` pro Matlab se nalezne matice afinní transformace tak, aby byla minimální suma čtverců chyb. Tuto matici pak zbývá pouze přepsat do konfiguračního souboru.

Aby se co nejvíce zjednodušil a zrychlil proces lokální aproximací souřadného systému WGS84, byla vytvořena jednoduchá coman-line aplikace `gct`. Ta převádí GPS souřadnice zapsané ve stupních, minutách a vteřinách do reálných čísel, která se dají načíst do Matlabu.

Utilita `gct` využívá již vytvořené třídy `geographyCoords` z aplikace `map`, viz kapitola 3.4.6. Utilita `gct` akceptuje dva vstupní parametry - čísla sloupců, kde se nacházejí data zeměpisné šířky a délky. Data k přeformátování jsou přijímána na standardním vstupu a vypisována na standardní výstup.

Ukázka použití utility `gct` je na obrázku 3.41.

```
$ cat data.txt
14°50'29.479"E  50°12'7.203"N
14°50'19.497"E  50°12'5.409"N
14°50'36.652"E  50°12'4.024"N
14°50'32.571"E  50°12'2.529"N
14°50'34.917"E  50°12'10.092"N
14°50'24.516"E  50°12'12.846"N
14°50'19.917"E  50°12'7.87"N
$ cat data.txt | ./gct 0 1
14.84152194     50.20200083
14.83874917     50.2015025
14.84351444     50.20111778
14.84238083     50.2007025
14.8430325      50.20280333
14.84014333     50.20356833
14.83886583     50.20218611
```

Obrázek 3.41: Příklad použití utility `gct` na převod souřadnic.

[48] Script je na CD v příloze

4 Použití a testování celého systému

Dosud byly samostatně popisovány jednotlivé aplikace. Nyní bude uvedeno, jak všechny utility používat dohromady.

V zadání je také výslovně řečeno, že se má celý systém důkladně otestovat. Bude tedy uvedeno, jak probíhalo testování.

4.1 Použití

Pro snadné používání všech utilit byly vytvořeny scripty, které usnadňují obsluhu celého systému. Při letových testech vrtulníku stačí spustit jediný script pro vizualizaci i záznam dat.

Všechny aplikace jsou v adresáři `tools`. Spouštěcí script je na výpisu 4.1.

```
#!/bin/bash
TOOLSDIR='tools'
DATALOG='./datalog'
GPSDATALOG='./datalog datalog_gps.conf'
GRAPH='./graph'
HORIZONT='./horizont'
MAP='./map'
CONCATER='./concater'
MY_TEE='./atee'
TMP1='data.tmp'
TMP2='gps.tmp'
if [ -n $1 ]; then
    cd $TOOLSDIR
    $DATALOG | $MY_TEE $TMP1 | $GRAPH | $HORIZONT &
    $GPSDATALOG | $MY_TEE $TMP2 | $MAP &
    PID1=$!
    wait $PID1 $!
    $CONCATER $TMP1 $TMP2 > ../$1
    rm $TMP1 $TMP2
fi
```

Výpis 4.1: Script pro vizualizaci a záznam letových dat.

Spouštěcí script akceptuje jako parametr název souboru, do kterého budou uložena letová data.

Script spustí dvě instance programu `datalog`. První instance přijímá telemetrická data a posílá je do přes utilitu `atee` do aplikací `graph` a `horizont`. Druhá instance přijímá GPS data a přes druhou instanci utility `atee` je posílá k zobrazení do aplikace `map`.

Poté se čeká na ukončení obou instancí `datalog`, aby se mohl zavolat `concater`, který sloučí výsledná data.

Pro přehrání dat souží script na výpisu 4.2.

```
#!/bin/bash
TOOLSDIR='tools'
GRAPH='./graph graph_replay.conf'
HORIZONT='./horizont horizont_replay.conf'
MAP='./map map_replay.conf'
if [ -n $1 ]; then
    cd $TOOLSDIR
    cat ../$1 | $GRAPH | $HORIZONT | $MAP &
    wait $!
fi
```

Výpis 4.2: Script pro přehrání letových dat.

Jako parametr scriptu se udává název souboru s daty. Tento soubor je pomocí standardní linuxové utility `cat` přepsán na vstup grafických aplikací.

Aplikace se pro přehrávání spouští s jinými konfiguračními soubory než při záznamu. Důvodem je požadavek lehce odlišného chování[49].

4.2 Testování

Testování softwaru byl relativně problém, protože během jeho vývoje byl model vrtulníku nefunkční. Proto byl vytvořen „simulátor“ modelu vrtulníku. Jedná se o jednoduchý program, který simuluje server z modelu vrtulníku a vysílá náhodná data. Při tvorbě tohoto simulátoru se vycházelo ze zdrojových kódů skutečného serveru z modelu vrtulníku.

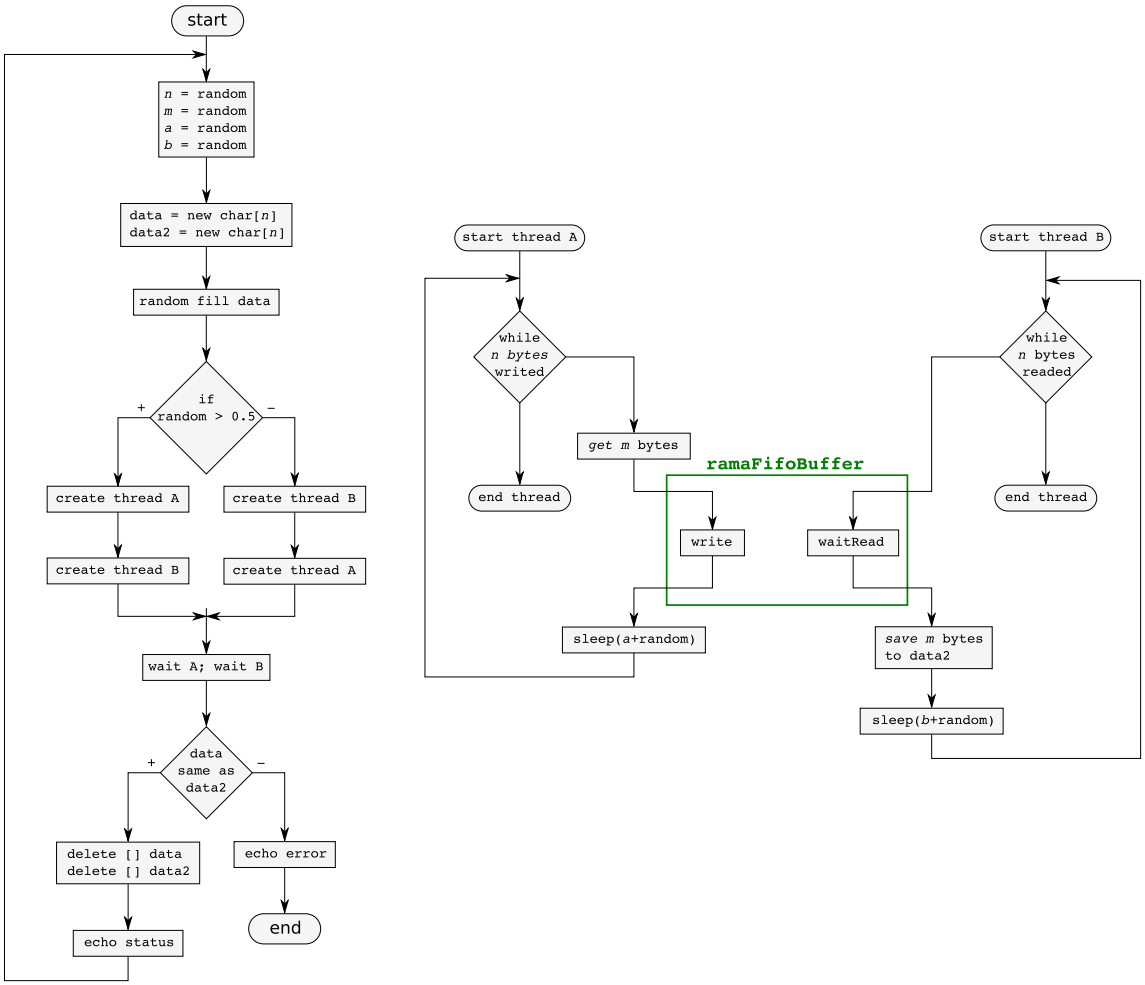
Simulátor vrtulníku je na přiloženém CD a s jeho pomocí je možné si celý systém vyzkoušet.

4.2.1 Testování jednotlivých komponent

Vzhledem k vysokým nárokům na spolehlivost softwaru byly samostatně testovány i jednotlivé komponenty. Příkladem je třída `ramaFifoBuffer` aplikace `datalog`. Tato třída zajišťuje mezivláknovou komunikaci a to je vždy nebezpečná a obtížně testovatelná programová pasáž. Proto byl vytvořen program `conditionRaceTest`, jehož algoritmus je na obrázku 4.1.

Program `conditionRaceTest` vytvoří pole `data` náhodné velikosti, naplní jej náhodnými daty a po náhodně velkých blocích jej zkopíruje do pole `data2`. Ke kopírování je použita testovaná třída `ramaFifoBuffer`. Zápis a čtení probíhá ve dvou různých vláknech. Rychlost zápisu a čtení je taktéž náhodná. Doba čekání zapisovacího vlákna je zvětšeno o konstantu `a` a doba čekání čtecího vlákna je zvětšena o konstantu `b`. Pokud je `a >> b`, `ramaFifoBuffer` je neustále prázdný, protože čtecí vlákno je „agresivnější“

[49] Například je požadován `self-timing`, který zpomaluje tok dat tak, aby rychlost odpovídala záznamu.



Obrázek 4.1: Algoritmus testování třídy `ramaFifoBuffer`.

než zapisovací. Pokud je `a << b`, `ramaFifoBuffer` je neustále plný, protože zapisovací vlákno je „agresivnější“ než čtecí.

Testovací program běžel bez jediné chyby dva dny a třídou `ramaFifoBuffer` bylo překopírováno přes 60GB dat.

5 Závěr

Celý systém byl otestován i proti modelu vrtulníku a fungoval nad očekávání dobře. Díky vysoké univerzálnosti celého systému vznikl projekt PILOT [12], který distribuuje všechny uvedené utility pod GLP licencí [11].

Systém tak nejen splňuje zadání diplomové práce, ale potenciálně může najít i širší uplatnění. Samostatná aplikace `map` může zobrazovat například cestu automobilu, cyklisty, chodce ...

Mapové podklady pro aplikaci `map` byly získány z WMS serveru Českého úřadu zeměměřického a katastrálního.

`http://geoportal.cuzk.cz/wmsdata13/RASORTOFOTO/wms.asp?`

Jedná se o placenou službu a na mapové podklady se vztahují obchodní podmínky užití digitálních a tištěných produktů Zeměměřického úřadu. Pro neziskové účely diplomových prací jsou poskytovány mapové podklady v omezené míře zdarma. Toho bylo využito při testování softwaru v této práci.

Seznam obrázků

1.1 Obrázek bezpilotního vrtulníku v projektu RAMA. 6
2.1 Koncepční schéma softwaru coby souboru utilit 7
2.2 Demonstrace možností modulárního systému: přehrávání letových dat. . . 8
2.3 Ukázka grafických aplikací a) `graph`, b) `horiznot`, c) `map` 8
2.4 Příklad zprávy s ID 42. 9
2.5 Příklad zakódování zprávy. 10
3.1 Vnitřní funkce programu `datalog` 12
3.2 Zpracování konfiguračního souboru v konstruktoru třídy `configReader`. . 15
3.3 Zpracování jednoho řádku konfiguračního souboru metodou `parseLine`. . 16
3.4 Algoritmus čtecí smyčky třídy `ramaTcpClient` 19
3.5 Kruhový buffer používaný třídou `ramaTcpClient` 20
3.6 Použití třídy `ramaFifoBuffer` 21
3.7 Vnitřní struktura třídy `ramaFifoBuffer` 22
3.8 Smyčka zpracování zpráv - metoda `parseAndEcho` třídy `ramaFifoBuffer` 24
3.9 Aplikace `graph` s popisem jednotlivých částí. 25
3.10 Detail grafu vykreslovaného třídou `GraphObject`. 26
3.11 Algoritmus metody `paintEvent` třídy `GraphObject`. 28
3.12 Algoritmus metody `makeNiceLimits` třídy `GScaler` pro symetrickou osu. 31
3.13 Algoritmus metody `makeNiceLimits` třídy `GScaler` pro automatickou volbu mezí. 31
3.14 Příklad vzhledu textového a číselného widgetu `AStatus`. 32
3.15 Algoritmus vykreslování widgetu `AStatus`. 34
3.16 Čtení standardního vstupu metodou `readStdin` ve třídě `CIOReader`. . . . 35
3.17 Zpracování jednoho řádku standardního vstupu metodou `parseData` ve třídě `CIOReader` aplikace `graph`. 36
3.18 Stavový automat pro zpracování jednoho řádku konfiguračního souboru metodou `parseLine` ve třídě `MainWidget`. 37
3.19 Rozložení *layoutů* na widgetu `MainWidget`. 38
3.20 Aplikace `horizont` s popisem jednotlivých částí. 39
3.21 Zpracování jednoho řádku standardního vstupu metodou `parseData` ve třídě `CIOReader` aplikace `horizont`. 41
3.22 a) Nejspodnější vrstva s orto-foto mapou, b) vrstva s popisky c) vrstva mřížky d) vrstva s trajektorií pohybu. 43
3.23 Class diagram aplikace `map`. 45
3.24 a) Popis *pracovní plochy* globálním souřadnicovým systémem. b) Ilustrace projekce povrchu zeměkoule na *pracovní plochu*. 46
3.25 a) Schématické znázornění souborového souřadnicového systému. b) Příklad adresování jednotlivých čtverců pozadí. 47
3.26 Ukázka adresace *pracovní plochy* souborovým souřadnicovým systémem. 47
3.27 Vnitřní struktura třídy `mapObjectTree`. 52
3.28 Ukázka pokrytí okna obrázky obsaženými v signálu `mapUpdated`. 54
3.29 Algoritmus získání map metodou `coverArea`. 55

3.30 Algoritmus překreslení widgetu `mapWidget`. 57
3.31 Algoritmus získání nových mapových podkladů třídy `mapWidget`. 58
3.32 Vykreslování cesty pomocí Bézierových křivek ve třídě `mapWidget`. 59
3.33 Ukázka nevhodnosti použití jednostranné diference při vykreslování cesty ve třídě `mapWidget`. 59
3.34 Efekt „ztrácející se“cesty pomocí barevného alfa kanálu, třída `mapWidget`. 59
3.35 Ukázky zakřivení mřížky při nelineární transformaci z uživatelské souřadnicové soustavy. 60
3.36 Algoritmus metody `createNewGrid` pro výpočet nové mřížky. 61
3.37 Operace zjednodušení křivky prováděná metodou `makeNice`. 62
3.38 Operace expanze křivky prováděná metodou `makeNice`. 63
3.39 Stavová lišta aplikace `map`. 64
3.40 Dialogové okno „posunu na lokaci“aplikace `map`. 64
3.41 Příklad použití utility `gct` na převod souřadnic. 66
4.1 Algoritmus testování třídy `ramaFifoBuffer`. 69

Seznam tabulek

3.1 Seznam všech tříd programu `datalog` a jejich stručný popis. 13
3.2 Seznam všech tříd programu `graph` a jejich stručný popis. 26
3.3 Seznam všech tříd programu `horizont` a jejich stručný popis. 39
3.4 Seznam všech tříd programu `map` a jejich stručný popis. 44
3.5 Příklady volání metod třídy `geographyCoords` a jejich výsledky. 50

Seznam výpisů

2.1 Příklad spuštění systému utilit 8
2.2 Příklad struktury zprávy 9
2.3 Ukázka textové meziprocesní komunikace 10
3.1 Příklad konfiguračního souboru aplikace `datalog` 14
3.2 Zjednodušená deklarace třídy `configReader` 14
3.3 Struktura pro uložení informací o položce v těle datového packetu. 16
3.4 Struktura pro uložení informací o příchozí asynchronní zprávě 17
3.5 Struktura popisující možný obsah asynchronní zprávy 17
3.6 Příklad výpisu asynchronní zprávy na výstup 17
3.7 Zjednodušená deklarace třídy `ramaTcpClient` 18
3.8 Zjednodušená deklarace struktury `msgIdAndLens` 18
3.9 Zjednodušená deklarace třídy `ramaFifoBuffer` 20
3.10 Zjednodušená deklarace kontejneru `msgContainer` 21
3.11 Zjednodušená deklarace třídy `dataParser` 22
3.12 Zjednodušená deklarace struktury `dataValue` 23
3.13 Zjednodušená deklarace třídy `GraphObject` 27
3.14 Zjednodušená deklarace struktury `curve` 27
3.15 Zjednodušená deklarace struktury `RamaColDataPacket` 29
3.16 Zjednodušená deklarace třídy `GScaler` 30
3.17 Zjednodušená deklarace třídy `AStatus` 32
3.18 Deklarace struktury `bEvent` 33
3.19 Zjednodušená deklarace třídy `CIOReader` 34
3.20 Zjednodušená deklarace třídy `MainWidget` 37
3.21 Zjednodušená deklarace třídy `HorizontWidget` 40
3.22 Zjednodušená deklarace třídy `CIOReader` 41
3.23 Zjednodušená deklarace třídy `CIOReader` aplikace `map` 48
3.24 Zjednodušená deklarace třídy `coordConverter` 48
3.25 Deklarace třídy `dataCoordSystem` 49
3.26 Zjednodušená deklarace třídy `mapObject` 51
3.27 Zjednodušená deklarace třídy `mapObjectTree` 52
3.28 Zjednodušená deklarace struktury `mapContainer` 52
3.29 Zjednodušená deklarace třídy `mapManager` 53
3.30 Zjednodušená deklarace struktury `imgCoords` 55
3.31 Zjednodušená deklarace třídy `mapWidget` 56
3.32 Deklarace výčtového typu `mapPathStyle` 58
3.33 Zjednodušená deklarace struktury `gridLine` 60
3.34 Zjednodušená deklarace třídy `lineSmoother` 61
3.35 Zjednodušená deklarace struktury `container` 62
3.36 Příklad spuštění svou instancí aplikace `datalog` 65
4.1 Script pro vizualizaci a záznam letových dat. 67
4.2 Script pro přehrání letových dat. 68
A.1 Příkazy pro zkompilování systému PILOT ze zdrojových kódů. I

A.2 Příklad spuštění systému PILOT - generátor dat. I
A.3 Příklad spuštění systému PILOT. I

Reference

[1] Trolltech, *Online Reference Documentation* [online] 2009 [cit. 2009-10-05]
URL: http://doc.trolltech.com/

[2] Stephen Prata, *Mistrovství v C++*, 3. aktualizované zadání, Computer Press 2007, ISBN: 978-80-251-1749-1

[3] Gnuplot, *The official homepage* [online] 2009 [cit. 2009-12-05]
URL: http://www.gnuplot.info/

[4] Projekt RAMA, *Otevřený řídicí systém pro bezpilotní létající prostředky* [online] 2009 [cit. 2009-06-05]
URL: http://rtime.felk.cvut.cz/helicopter/

[5] Qwt, *Qt Widgets for Technical Applications* [online] 2009 [cit. 2009-11-05]
URL: http://qwt.sourceforge.net/

[6] Wikipedia, *Mercator projection* [online] 2009 [cit. 2009-03-05]
URL: http://en.wikipedia.org/wiki/Mercator_projection

[7] Wikipedia, *Mona Lisa by Leonardo da Vinci* [online] 2009 [cit. 2009-17-05]
URL: http://en.wikipedia.org/wiki/Mona_lisa

[8] Wikipedia, *Web Map Service* [online] 2009 [cit. 2009-09-05]
URL: http://en.wikipedia.org/wiki/Web_Map_Service

[9] The Open Geospatial Consortium, Inc., *The official homepage* [online] 2009 [cit. 2009-08-05]
URL: http://www.opengeospatial.org

[10] European Petroleum Survey Group, *The official homepage* [online] 2009 [cit. 2009-10-05]
URL: http://www.epsg.org

[11] The GNU General Public License, *The official homepage* [online] 2009 [cit. 2009-15-05]
URL: www.gnu.org/licenses/

[12] Petr Svoboda, *The PILOT system official homepage* [online] 2009 [cit. 2009-17-05]
URL: http://www.poppik.net/pilot/

A Obsah CD

Obsah přiloženého CD:

- `diplomovaprace_2009_svobop10.pdf` Diplomová prací v elektronické podobě.
- `pilot_bin443.tar.gz` Zkompilovaný systém PILOT pro verzi Qt 4.4.3 a vyšší.
- `pilot_bin450.tar.gz` Zkompilovaný systém PILOT pro verzi Qt 4.5.0 a vyšší.
- `pilot_src.tar.gz` Zdrojové kódy systému PILOT.

Příkazy pro zkompilování systému PILOT ze zdrojových kódů jsou na výpisu A.1.

```
tar -xf pilot_src.tar.gz
cd tools/
make prepare
make
```

Výpis A.1: Příkazy pro zkompilování systému PILOT ze zdrojových kódů.

Příkazy pro spuštění systému PILOT jsou na výpisech A.2 a A.3. Nejprve je třeba zapnou umělý generátor a následně se na něj připojit.

```
tar -xf pilot_bin450.tar.gz
cd data_generator/
./data_generator
```

Výpis A.2: Příklad spuštění systému PILOT - generátor dat.

```
./start data.txt
```

Výpis A.3: Příklad spuštění systému PILOT.

B Verze použitého softwaru

Verze kompilátoru a knihovny, používaných při vývoji systému PILOT

```
Qt Creator v1.0.0. (compiled from source codes)
libqt4-dev 4.5.0-0ubuntu4.1
qt4-qmake 4.5.0-0ubuntu4.1
libqtgui4 4.5.0-ubuntu4.1
g++ 4:4.3.3-1ubuntu-1
gcc-4.3-base 4.3.3-5ubuntu4
build-essential 11.4
```

Počítačové sestavy, na kterých byl systém PILOT testován (zkompilován a spuštěn):

- IBM ThinkPad x41, CPU Intel Pentium M 1.5GHz, 1.5GB RAM, OS Ubuntu 9.04
- Intel Little Falls 2, CPU Intel Atom 330 1.6 GHz, 1GB RAM, OS Ubuntu 8.10
- HP Compaq nc6400, OS Debian Lenny

Obsah

1 Úvod **6**

2 Koncepce **7**
2.1 Popis síťového komunikačního protokolu telemetrie modelu vrtulníku . . 8
2.1.1 Kódování dat v těle zprávy 9
2.2 Popis meziprocesní textové komunikace 9
2.3 Výběr programovacího jazyka a grafické knihovny 11

3 Programové moduly **12**
3.1 Aplikace `datalog` 12
3.1.1 Třída `configReader` 14
3.1.2 Třída `ramaTcpClient` 18
3.1.3 Třída `ramaFifoBuffer` 20
3.1.4 Třída `dataParser` 22
3.2 Aplikace `graph` 25
3.2.1 Třída `GraphObject` 26
3.2.2 Třída `GScaler` 30
3.2.3 Třída `AStatus` 32
3.2.4 Třída `CIOReader` 33
3.2.5 Třída `MainWidget` 36
3.3 Aplikace `horizont` 39
3.3.1 Třída `HorizontWidget` 39
3.3.2 Třída `CIOReader` 41
3.3.3 Třída `MainWidget` 42
3.4 Aplikace `map` 43
3.4.1 Souřadné systémy používané aplikací `map` 44
3.4.2 Třída `CIOReader` 48
3.4.3 Třída `coordConverter` 48
3.4.4 Třída `affineConverter` 49
3.4.5 Třída `dataCoordSystem` 49
3.4.6 Třída `geographyCoords` 50
3.4.7 Třída `mapObject` 50
3.4.8 Třída `mapObjectTree` 51
3.4.9 Třída `mapManager` 53
3.4.10 Třída `mapWidget` 56
3.4.11 Třída `lineSmoother` 61
3.4.12 Třída `Widget` 63
3.4.13 Třída `goToDialog` 64
3.5 Pomocné utility `concater`, `atee` a `gct` 65
3.5.1 Utilita `concater` 65
3.5.2 Utilita `atee` 65
3.5.3 Utilita `gct` 66

4 Použití a testování celého systému **67**
4.1 Použití 67
4.2 Testování 68
4.2.1 Testování jednotlivých komponent 68

5 Závěr **70**

Seznam obrázků **71**

Seznam tabulek **72**

Seznam výpisů **73**

Přílohy **I**

A Obsah CD **I**

B Verze použitého softwaru **II**