Univerza v Ljubljani

Fakulteta za elektrotehniko

Justin Činkelj

Haptični vmesnik, zasnovan v operacijskem sistemu RTLinux

Magistrsko delo

Mentor: prof. dr. Marko Munih

V Ljubljani, julij 2007

Zahvala

Za pomoč se zahvaljujem mentorju prof. dr. Marku Munihu, ki me je usmerjal med delom. Hvaležen sem tudi prof. dr. Tadeju Bajdu za koristne pripombe.

Zahvaljujem se kolegom iz Laboratorija za robotiko in biomedicinsko tehniko. Alešu Bardorferju in Matjažu Mihlju za pomoč pri težavnem začetku dela z Linux/RTLinux operacijskim sistemom. Urošu Maliju, ki mi je predstavil nekatere aplikacije z obstoječim sistemom robota. Janezu Podobniku za koristne razprave o potrebni in nepotrebni funkcionalnosti programske opreme za RTLinux. Janezu Podobniku in Mariu Šikiću za najdene hrošče in pomoč pri odpravi le-teh.

Za posredovano dokumentacijo robota in pojasnila nejasnosti se zahvaljujem Pietu Lammertse, Cedricu Lemaire pa za pomoč pri uporabi orodja CodeWorker.

Za podporo se zahvaljujem tudi celotni družini Makše - Činkelj, Sp. Log in Gantar - Činkelj, Podpeč. Še posebej se želim zahvaliti staršema Milki in Slavku.

Kazalo

Povzetek **1**

Abstract **3**

1. Uvod **5**

1.1 Haptična programska oprema 15

1.2 Cilji naloge 17

2. Obstoječi sistem robota HapticMaster **19**

2.1 Strojna oprema 19

2.1.1 Robotski manipulator 19

2.1.2 Krmilni računalnik 24

2.1.2.1 CIC vmesniška karta 25

2.2 Programska oprema 30

2.2.1 Krmilni program v operacijskem sistemu VxWorks 30

2.2.2 Knjižnica HapticAPI 33

3. Programska oprema za RTLinux **37**

3.1 Zasnova 38

3.2 Nizki modul 40

3.2.1 Gonilniki naprav 41
3.2.1.1 CIC karta 41
3.2.1.2 Aktuator 44
3.2.1.3 Senzor sile 49
3.2.2 Robotski manipulator 49
3.2.3 Model prostostne stopnje 51
3.2.4 Navidezno okolje 55
3.2.5 Haptični objekti 58
3.2.5.1 HapticObject_t 59
3.2.5.2 HapticBlock_t 64
3.2.5.3 HapticSphere_t 64
3.2.5.4 HapticTorus_t 64
3.2.5.5 HapticCylinder_t 65
3.2.5.6 HapticConstantForce_t 65
3.3 Knjižnica ulfeHapticAPI 66
3.3.1 Ročice 66
3.3.1.1 Zgradba ročic 67
3.3.1.2 Uporaba ročic 68
3.3.2 Vmesnik za jezik C 69
3.3.3 Vmesnik za jezik C++ 71
3.3.4 Primera uporabe 73
3.3.4.1 Minimalno haptično okolje 73
3.3.4.2 Prenos obstoječega programa 75
3.4 Visoki modul 77

4. Cpp2c skripta **79**

4.1 Implementacija virtualnih funkcij v jeziku C++ 79

4.2 Zahteve za cpp2c pretvornik 82

4.3 Zasnova cpp2c skript 84

4.4 Razčlenitev vzorčne datoteke in generiranje kode 86

4.5 Koda osnovne (starševske) strukture 88

4.6 Izpeljana struktura 90

4.7 Delovanje generirane C kode 91

5. Zaključek **95**

Literatura **97**

Dodatek A **103**

Dodatek B **107**

Povzetek

Izdelali smo programsko opremo v operacijskem sistemu RTLinux za delo z robotom HapticMaster. Programska oprema omogoča visokonivojsko uporabo robota, kjer se robot obnaša kot haptični vmesnik. Poleg tega je možno tudi nizkonivojsko vodenje robota, kar dovoljuje implementacijo lastnih regulacijskih algoritmov. Knjižnica za visokonivojski haptični način dela je uporabna v Linux ali v Windows operacijskem sistemu.

V uvodu so predstavljene različne haptične naprave ter primeri njihove uporabe. Vodenje haptičnih naprav se mora izvajati v realnem času, kar zahteva uporabo primernega operacijskega sistema. Omenjenih je nekaj pogosteje uporabljenih operacijskih sistemov.

V drugem poglavju je predstavljen obstoječi sistem robota HapticMaster. Ta je sestavljen iz robotskega manipulatorja, krmilnega računalnika in programske opreme, ki se izvaja v VxWorks operacijskem sistemu. Programska oprema krmili gibanje robota v navideznem haptičnem okolju. Lastnosti navideznega okolja nastavljamo s pomočjo knjižnice HapticAPI. Ta komunicira s krmilnikom (strežnikom) preko omrežne povezave, izvaja pa se na ločenem računalniku (odjemalcu) v Windows ali Linux operacijskem sistemu.

Tretje poglavje opisuje realizirano programsko opremo za RTLinux operacijski sistem. Programska oprema je razdeljena v tri dele – nizki modul, visoki modul in knjižnico ulfeHapticAPI. Nizki modul je edini nujno potreben za delo z robotom. Najprej so opisani gonilniki za strojno opremo. Ti so združeni v strukturi/objektu, ki predstavlja celotni robotski manipulator. Robotski manipulator in haptični objekti sestavljajo navidezno okolje. Visoki modul omogoča enostavno dodajanje nove funkcionalnosti (lastni regulacijski algoritem, beleženje podatkov, dinamična haptična okolja itd.) med

delovanjem robota. Knjižnica ulfeHapticAPI omogoča dostop do robota preko omrežja in je združljiva s knjižnico HapticAPI na nivoju izvorne kode.

Četrto poglavje predstavlja pomožno programsko orodje cpp2c. Cpp2c omogoča enostavno implementacijo polimorfičnih objektov v jeziku C na način, ki ga poznamo v jeziku C++. Orodje je realizirano kot skripta za razčlenjevalnik in generator kode codeworker.

V prilogi je opisano orodje srpcgen, ki omogoča izvoz funkcij preko omrežne povezave iz Linux jedra (strežnik) v Linux ali Windows uporabniški prostor (odjemalec). Tudi to orodje je realizirano kot skripta za program codeworker.

Ključne besede: haptični robot, programska oprema, objektno orientirano programiranje, Linux, Windows

Abstract

Software for robot HapticMaster for RTLinux operating system has been developed. High level operation of the robot functioning as a haptic interface is main area of use. Low level control of robot is also possible, permitting implementation of custom control loop algorithms. The library for high level haptic operation of the robot can be called from Linux and Windows operating system.

In the introduction are presented various haptic devices and examples of use. Haptic devices require realtime control, thus a suitable operating system has to be selected. A few often used operating systems are briefly described.

In the second chapter is presented the existing HapticMaster system. It consist of a robot manipulator, control computer and the software. Software is running on VxWorks operating system and controls movement of the robot inside a virtual environment. Properties of the virtual environment are set with a HapticAPI library. The HapticAPI library is executing on a second computer (client) under Windows or Linux operating system. A network connection is used for communication with a control computer (server) .

The implemented software for RTLinux is described in a third chapter. Software consist from three parts: low module, high module and ulfeHapticAPI library. Only low module is indispensable for operation of the robot. First the device drivers are described. They are used in a structure/object, presenting the whole robot manipulator. The robot and haptic objects compose the virtual environment. High module permits simple addition of new functionality (new control loop algorithms, data logging, dynamic haptic environments etc.) while the robot is operating. Use of the robot over network connection is possible via the library ulfeHapticAPI. Programs using ulfeHapticAPI are source level compatible with programs using HapticAPI.

The utility program cpp2c is presented in a forth chapter. Cpp2c allows simple implementation of polymorphic objects in C language using syntax similar to C++ language. The tool is realized as a set of scripts for parsing and code generation with CodeWorker.

In appendix is described the tool srpcgen. Srpcgen permits export of functions in Linux kernel (server) to Linux or Windows user space (client). This tool is realized as a set of scripts for CodeWorker too.

Keywords: haptic robot, software, object oriented programming, Linux, Windows

1.
Uvod

Beseda haptičnost izhaja iz grške besede *hapto*, ki pomeni zmožnost zaznavanja dotika oz. tipanja. Haptičnost obsega informacije o fizičnih lastnostih objektov, kot so masa, vztrajnost, podajnost in grobost površine. Haptičnost lahko občutimo kot kinestetično informacijo (zaznavanje premika ali sile) in kot taktilno informacijo (zaznavanje oblik in teksture).

Haptične naprave uporabljamo za različne naloge na področjih, kot so kirurške simulacije, vaje v medicini, znanstveni prikazi, vmesniki za tele-, mikro- in nanomanipulacijo in različne podporne tehnologije za slepe in slabovidne. Zanimiva skupina so t. i. "force feedback" igralne palice in miške, ki so pojem haptičnosti približale širši publiki. Pomembno področje uporabe haptičnih naprav je rehabilitacija. Pri tem so haptične naprave lahko uporabljene kot merilni sistem za ocenjevanje poteka rehabilitacije pacientov ali kot aktivni rehabilitacijski pripomoček, ki lahko pripomore k boljšemu izidu rehabilitacije oz. razbremeni fizioterapevta napornega fizičnega dela.

Za verodostojno prikazovanje haptičnosti se mora haptična naprava dovolj hitro odzivati. Regulacijska zanka se mora izvajati vsaj s frekvenco 500 Hz, da dobi uporabnik vtis realističnega prikaza [1]. Krmilnik je pogosto realiziran na osebnem računalniku s splošno namenskim operacijskim sistemom, kot sta Windows ali Linux. Tu lahko nastopajo zakasnitve velikosti nekaj 10 ms, kar poslabša kvaliteto haptičnega prikaza. Zaradi zakasnitev pride do zatikajočega se premikanja naprave, ko je potrebno gladko gibanje, in podobnih motečih napak. Pomagamo si lahko tako, da krmilnemu pro-

gramu določimo visoko prioriteto izvajanja in omejimo ali odstranimo opravila, ki bi utegnila motiti njegovo izvajanje. Zanesljivejša rešitev je uporaba razširitev za delo v doslednem realnem času, kot so Windows RT extension, RTLinux in Linux - RTAI, ali pa namenskih operacijskih sistemov za delo v doslednem realnem času npr. VxWorks, Lynx, eCos itd.

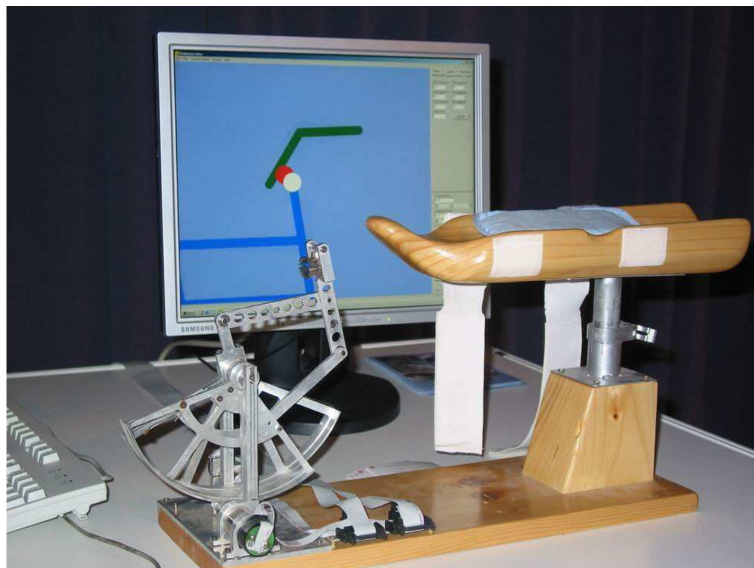
Naprava HIFE ("haptic interface for finger exercise", slika 1.1) je paralelni robotski mehanizem z dvema aktivnima prostostnima stopnjama. Namenjena je razgibavanju prstov na roki. Razvit je bil nabor vaj za razgibavanje prstov in nabor vaj za ocenjevanje funkcionalnih sposobnosti. Naprava je bila preizkušena na skupini pacientov po kapi. Rezultati so potrdili primernost izbora vaj za razgibavanje. Hkrati je prikazana tudi dobra korelacija med M-FIM skalo za ocenjevanje in metodami ocenjevanja, ki jih predlagajo avtorji [2, 3].

V [4, 5] je predstavljen razvoj strojne in programske opreme. Robota vodi osebni računalnik s programom v operacijskem sistemu Windows 2000. Do strojne opreme dostopamo s pomočjo vmesniških kartic za branje enkoderjev in nastavljanje toka motorjev. Za delo v realnem času je poskrbljeno s pomočjo prekinitev s frekvenco 1000 Hz. Prekinitve generira dodatna mikrokrmilniška naprava, priključena na vzporedna vrata. Prekinitve s posredovanjem programskega paketa TVicHW32 [6] prožijo krmilni algoritem. Mikrokrmilnik poleg tega opravlja tudi delo čuvaja časa.

Ena najbolj uporabljenih haptičnih naprav je PHANToM (slika 1.2). To je majhen robot s tremi aktivnimi prostostnimi stopnjami. Na vrhu robota se nahaja še element s tremi pasivnimi rotacijami. Proizvajalec prilaga programsko opremo, ki deluje v Windows 2000 operacijskem sistemu. Razvita je bila tudi programska oprema, ki deluje v RTLinux okolju [7].

V [8] je predstavljena uporaba PHANToM-a za kvantitativno vrednotenje funkcionalnega stanja rok pacientov z različnimi živčno-mišičnimi in nevrološkimi boleznimi. Naloge so zajemale meritve največje sile, ki jo je pacient sposoben generirati, sledenje premikajoči se tarči, gibanje po labirintu in gibanje od točke do točke.

Robotski sistem ARM Guide ("Assisted Rehabilitation and Measurement Guide")



Slika 1.1: HIFE haptična naprava

Slika 1.2: PHANToM

Slika 1.3: ARM Guide

omogoča gibanje gornje extremitete v sagitalni ravnini (slike 1.3). Sistem je prvenstveno namenjen vsiljevanju trajektorij gibanja roki in merjenju pasivnih prispevkov momentov v sklepih. V [9] je prikazano, da lahko robotika pripomore k terapiji in ocenjevanju bolnikov z gibalnimi prizadetostmi po kapi ali poškodbah osrednjega živčnega sistema.

Robotski sistem MIME ("Mirror Image Movement Enabler", slika 1.4) je bil razvit na univerzi Stanford in je namenjen rehabilitaciji gornje extremitete po kapi [10]. Raziskovalci so pokazali, da je lahko robotska rehabilitacija varna in učinkovita. Raziskava temelji na razgibavanju gornje ekstremitete in merjenju območja gibanja sklepov. Terapija je lahko ali avtomatska ali pa jo vodi pacient z neprizadeto roko (telemanipulacija s pomočjo pasivnega robota in ne haptičnega vmesnika). Uporabljen je bil industrijski robot Puma 560. Raziskava je tudi potrdila hipotezo o primernosti uporabe robotov z majhno podajnostjo za rehabilitacijo.

Evropski projekt GENTLE/S je razvil sistem za robotsko terapijo zgornjih ekstremitet (slika 1.5) [11, 12, 13]. Gre za uporabo haptičnega vmesnika za rehabilitacijo gornje

Slika 1.4: Sistem MIME

Slika 1.5: Terapevtski sistem GENTLE/S

ekstremitete pri pacientih po kapi. Voljnost (nezapornost) uporabljenega robota je dosežena z admitančnim vodenjem robota. Pri takem vodenju je pomembna mehanska struktura, ki se nahaja za senzorjem sile. Njene lastnosti odločilno vplivajo na območje dosegljivih mehanskih impedanc in s tem na kvaliteto haptičnega vmesnika. Robotska terapija GENTLE/S temelji na različnih stopnjah pomoči pri preprostih gibih iz točke v točko.

Slika 1.6: Reharob

Sistem Reharob je tako kot GENTLE/S namenjen rehabilitaciji roke pacientov po kapi (slika 1.6) [14]. Uporabljena sta dva industrijska robota, vodena pozicijsko po vnaprej določenih trajektorijah. Senzorji sile v robotskem zapestju so namenjeni implementaciji varnostnih mehanizmov – zaznavanju prevelikih sil in momentov. Senzorji sile so uporabljeni kot del haptičnega vmesnika samo v fazi učenja rehabilitacijske trajektorije. Dinamika haptičnega vmesnika je nizka zaradi načina regulacije. Ta poteka na visokem nivoju pozicijskega vodenja z zunanjo (kaskadno) regulacijo z merjenjem sil/momentov v zapestju robota. Vzrok za tako strukturo vodenja je nezmožnost poseganja v industrijski krmilnik robota.

V [15] je opisan haptični robot na osnovi industrijskega robota Staubli RX90 (slika 1.7). Robot je bil uporabljen za študij gibanja človeške roke v haptičnem okolju. Haptično interakcijo lahko izboljšamo z višjo frekvenco regulacijske zanke in z vključitvijo modela manipulatorja v krmilni algoritem. Krmilnik robota se izvaja v operacijskem sistemu RTLinux s frekvenco 4 kHz. V krmilniku je upoštevan tudi dinamični in statični model manipulatorja [16]. Zaradi nepopolnega poznavanja dinamičnega modela manipula-

Slika 1.7: Robot Staubli RX90, uporabljen kot haptični vmesnik

torja je uporabljeno admitančno vodenje z visokimi ojačenji položajne in hitrostne zanke.

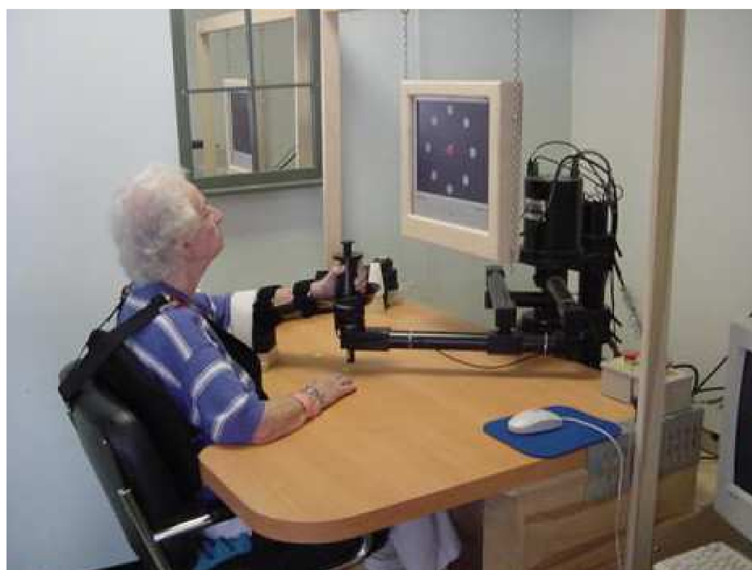
Haptični robot Delta Haptic Device (slika 1.8) temelji na paralelni konfiguraciji robota delta za translacijsko gibanje. Za rotacijsko gibanje je na vrh dodano zapestje na osnovi PARAMAT strukture [17]. Robot omogoča haptično interakcijo v 6 prostostnih stopnjah, dosegljive sile pa gredo do 25 N. Napravo krmilimo s programsko opremo za Windows 2000 in PCI vmesniško karto. Programska oprema omogoča izvajanje dveh asinhronih zank – ena je namenjena grafičnemu, druga pa haptičnemu izrisovanju. Dosegljive so frekvence nad 1 kHz, kar omogoča visoko kvaliteto tudi pri izrisu dinamičnih in upogljivih objektov [18].

Laparoscopic Impulse Engine (slika 1.9) je namenjena simulaciji laparoskopskih in endoskopskih operacij v navideznem okolju [19]. To je haptična krmilna palica z dvema stopnjama prostosti, ki ima na vrhu pritrjeno držalo, kakršno najdemo na ustreznih kirurških orodjih (ročica škarij). Krmilimo jo s PCI ali ISA kartico, ki je v osebnem računalniku z Windows operacijskim sistemom [20].

MIT-MANUS (slika 1.10) je robot za pomoč pri rehabilitaciji ter vrednotenju in dokumentiranju gibanja [21]. MIT-MANUS ima izredno nizko lastno impedanco in je vodljiv s strani pacienta, kar ga tudi že uvršča med haptične vmesnike. Naloge so sledenje krožnici in premikanje po premici pri gibanju iz točke v točko. Za hudo priza-

Slika 1.8: Delta Haptic Device

Slika 1.9: Laparoscopic Impulse Engine



Slika 1.10: Robot MIT-MANUS

dete paciente robot vsiljuje omejitve gibanja. Pacient zato lažje sledi želeni trajektoriji. Robot je sposoben izvajanja planarnih gibov v horizontalni ravnini, zato je učinek rehabilitacije omejen na mišične skupine, ki so odgovorne za gibanje v horizontalni ravnini [22, 23]. Kot odgovor na to slabost je bil razvit dodaten modul za vertikalno gibanje, ki je nameščen na robota ali pa ga uporabljamo samostojno [24].

PHI (pneumatic haptic interface) (slika 1.11) je haptični vmesnik, ki za pogon uporablja pnevmatske aktuatorje. Zgrajen je v obliki eksoskeleta in pokriva delovni prostor roke (ramena in komolca, ne pa tudi zapestja). Krmilnik robota je realiziran na Macintosh delovni postaji. Krmilna zanka se izvaja s frekvenco 500 Hz. Avtorji so med drugim pokazali, da lahko z višjo frekvenco krmilne zanke dosegajo višje trdote navideznih objektov [25, 26, 27].

Sarcos dexterous large arm master je hidravlični haptični eksoskelet z 10 prostostnimi stopnjami. Robot pokriva delovni prostor roke. Na vrhu je nameščeno triprstno prijemalo. Krmili ga delovna postaja z VxWorks operacijskim sistemom. V [28] je bil uporabljen kot vmesnik človek stroj pri sestavljanju v virtualnem okolju. Grafični prikaz se je izvajal na delovni postaji, ki je bila s krmilnikom povezana preko omrežne povezave.

Slika 1.11: Pnevmatični haptični vmesnik PHI

ViSHaRD6 (Virtual Scenario Haptic Rendering Device with 6 actuated DOF) je haptični robot s šestimi prostostnimi stopnjami. Mehanizem in vodenje mehanizma je opisano v [29]. Naprava je zasnovana za delo v navideznem okolju. Pri vodenju je uporabljen poenostavljen dinamični model [30]. V [31] je bil uporabljen kot vmesnik človek stroj za teleoperacijo 7 DOF robotske roke. Avtorji omenjajo oscilacije, do katerih pride zaradi poenostavljenega modela robota.

Pogosta ovira uporabi haptičnih vmesnikov na novih področjih je nedosegljivost primernih haptičnih naprav. Interakcija znotraj velikega delovnega področja zahteva napravo z velikim območjem, večina komercialno dostopnih naprav pa omogoča le majhno območje gibanja. Želja po doseganju dobrih dinamičnih lastnosti narekuje majhne mase segmentov in aktuatorjev, kar omejuje velikost delovnega območja in velikost dosegljivih sil. Mehanizem z vodenimi redundantnimi stopnjami prostosti lahko doseže znatno večje delovno območje, ker se lahko ogiba singularnim legam. ViSHaRD10 (slika 1.12) je hiperredundantna haptična naprava z 10 prostostnimi stopnjami. Spretno delovno območje je valj premera 1.7 m in višine 0.6 m, največja sila pa je 170 N [32].

Slika 1.12: Robot ViSHaRD10

1.1 Haptična programska oprema

Haptična programska oprema je praviloma namenjena delu s strojno opremo posameznega proizvajalca. Iz akademskih krogov prihaja tudi nekaj paketov, namenjenih delu s produkti različnih proizvajalcev. V nadaljevanju je predstavljenih nekaj primerov haptične programske opreme.

CHAI 3D (http://www.chai3d.org) je odprtokodni nabor C++ knjižnic za računalniško vizualizacijo, haptiko in interaktivno realnočasovno simulacijo. Podprtih je nekaj različnih komercialno dostopnih haptičnih naprav (PHANToM od SensAble, DELTA in OMEGA od Force Dimensions ter CUBIC in FREEDOM od MPB Technologies). Poleg navadne grafike je možno uporabljati tudi 3D prikaz (stereo očala). CHAI 3D lahko uporabljamo v Windows operacijskem sistemu, preliminarno pa je podprt tudi Linux.

GiPSi (http://gipsi.case.edu/) je odprtokodno ogrodje, namenjeno razvoju aplikacij za simulacijo kirurških posegov. Za implementacijo je uporabljen C++ jezik. Cilj je

ponuditi intuitiven vmesnik za uporabo dinamičnih modelov organov za haptični in vizualni prikaz. Poudarek je na možnosti uporabe raznovrstnih modelov izračunavanja in na neodvisnosti od modelirnih metod, z namenom omogočiti izmenjavo modelov in algoritmov.

Podjetje SenseGraphics je avtor razvojne platforme H3D API (http://www.sensegraphics.com/, http://www.h3d.org/). H3D API uporablja odprte standarde X3D, OpenGL, XML in STL. Knjižnica je na voljo pod GPL in komercialno licenco. Podprti operacijski sistemi so Windows, Linux in Mac OS. Podprte so haptične naprave podjetja SensAble (PHANToM).

Haptik (http://www.haptiklibrary.org/) je odprtokodna knjižnica, ki nudi HAL ("hardware abstraction layer") za dostop do haptičnih naprav. Tako je možno uporabljati naprave različnih proizvajalcev s poenotenim vmesnikom. Knjižnico lahko uporabljamo s programskim jezikom C++, Matlab, Simulink in Java. Deluje na Windows XP in Linux operacijskem sistemu. Podprte so naprave podjetij SensAble, Force Dimension in MPB Technologies.

Reachin (http://www.reachin.se/) ponuja napravo Reachin Display (slika 1.13). Uporabnik gleda robota preko polprosojnega zrcala, v katerem odseva slika monitorja. Priložena je programska oprema Reachin API. Ta podpira programska jezika Python in VRML (standardna izdaja), v profesionalni izdaji pa tudi C++.

Immersion (http://www.immersion.com) skupaj s svojo strojno opremo CyberForce (haptični eksoskelet, ki poleg delovnega prostora dlani pokriva tudi celotni delovni prostor roke) ponuja tudi VirtualHand SDK. Knjižnica podpira uporabo haptične naprave s Polhemus Fastrak ali Ascension Flock of Birds 3D sledilnim sistemom. Knjižnica omogoča uporabo 3D vizualizacije in detekcije trkov z možnostjo vključitve lastnih, specializiranih algoritmov. Proizvajalec ponuja različico za delo z modeli v CAD programu CATIA.

Handshake (http://www.handshakevr.com) ponuja proSENSE Virtual Touch Toolbox za uporabo komercialnih in lastnih haptičnih naprav. Podpira naprave podjetja SensAble, Quanser, Force Dimensions in MPB. Z uporabe kompenzacije časovne zakasnitve

Slika 1.13: Reachin Display

je možna uporaba haptičnih efektov preko omrežja. Programski paket temelji na programu Matlab s Simulink in Real Time Workshop.

Robote PHANToM proizvajalca Sensable lahko uporabljamo s knjižnico GHOST SDK. Knjižnica omogoča preprosto uporabo 3D haptičnih objektov, brez poznavanja nizkonivojskega izračunavanja sil. Po želji lahko programer tudi sam izračunava silo, s katero haptični objekt deluje na robota oz. operaterja. Sama GHOST knjižnica ne vključuje grafičnega prikaza objektov. Zasnovana je z mislijo na enostavno vključitev popularnih 3D grafičnih paketov. GHOSTGL je dodatek na osnovi GL knjižnice, ki haptičnim objektom doda še grafične lastnosti in jih izriše na zaslon.

Namesto specializirane haptične programske opreme lahko haptično funkcionalnost implementiramo na osnovi klasične robotske programske opreme. V tem primeru pri določanju pozicijske reference upoštevamo izmerjeno silo, oziromo iz pozicije robota določimo želeno kontaktno silo med robotom in operaterjem. Na ta način lahko uporabimo odprtokodni projekt Orocos (http://www.orocos.org/).

1.2 Cilji naloge

Delo ima namen doseči naslednje cilje:

- Pregled obstoječe programske opreme haptičnega vmesnika HapticMaster.
- Implementirati enakovredno rešitev na odprtokodnem operacijskem sistemu RTLinux. To bo omogočilo popoln nadzor nad robotom, tudi na najnižjem nivoju strojne opreme, kar je potrebno pri razvoju lastnih krmilnih in haptičnih algoritmov.
- Poleg haptičnega delovanja naj bo možno še klasično krmiljenje robota.
- Implementacija knjižnice za dostop do haptične funkcionalnosti preko omrežja. Knjižnica naj bo namenjena uporabi v Linux in Windows operacijskem sistemu.
- Haptični programi, ki dostopajo do robota preko omrežja, naj bodo z obstoječimi programi združljivi na nivoju izvorne kode.

2.

Obstoječi sistem robota HapticMaster

2.1 Strojna oprema

2.1.1 Robotski manipulator

Strojna oprema robota HapticMaster je sestavljena iz robotskega manipulatorja (slika 2.1), ki je s signalnim kablom povezan na krmilni računalnik, z močnostnim kablom pa na močnostne ojačevalnike. Močnostni ojačevalniki, enosmerni napajalnik za močnostne ojačevalnike in krmilni računalnik so v skupnem ohišju (slika 2.2). Sistem kot celota potrebuje napajalno napetost 230 V, 50 Hz. Enosmerni napajalnik napaja močnostne ojačevalnike, ti pa napajajo enosmerne servo motorje manipulatorja.

Sistem lahko kadarkoli zaustavimo z varnostnim stikalom (slika 2.3). Varnostno stikalo je vezano na kontaktor, preko katerega pride omrežna napetost na enosmerni napajalnik. Pritisk na varnostno stikalo kontaktor razklene. Posledično se izklopi napajanje močnostnih ojačevalnikov in motorjev. Kontaktor sklenemo z zelenim gumbom za vklop na prednji plošči skupnega ohišja krmilnega računalnika in močnostnih ojačevalnikov. Po pritisku na gumb lahko krmilni računalnik premika robota, tako da ga lahko normalno uporabljamo.

Manipulator ima tri prostostne stopnje (3 DOF). Prvi sklep omogoča translacijo v

Slika 2.1: Manipulator

Slika 2.2: Skupno ohišje krmilnega računalnika in močnostnih ojačevalnikov

vodoravni ravnini, drugi rotacijo okrog navpične osi, tretji pa translacijo v navpični smeri[1]. Zaradi preproste kinematične strukture robot nima kinematičnih sigularnosti,

[1]Oštevilčenje sklepov robota se ne ujema z običajnim načinom številčenja, kjer kot prvi sklep

Slika 2.3: Varnostno stikalo

kar poenostavi programiranje robota. Dosegi posameznih sklepov so zapisani v tabeli 2.1. Z njimi je tudi določen delovni prostor robota, ki je prikazan na sliki 2.4. Delovni prostor manipulatorja približno pokriva delovni prostor človeške roke.

Os	Območje sklepa	enota
1	0.380	m
2	1.010	rad
3	0.457	m

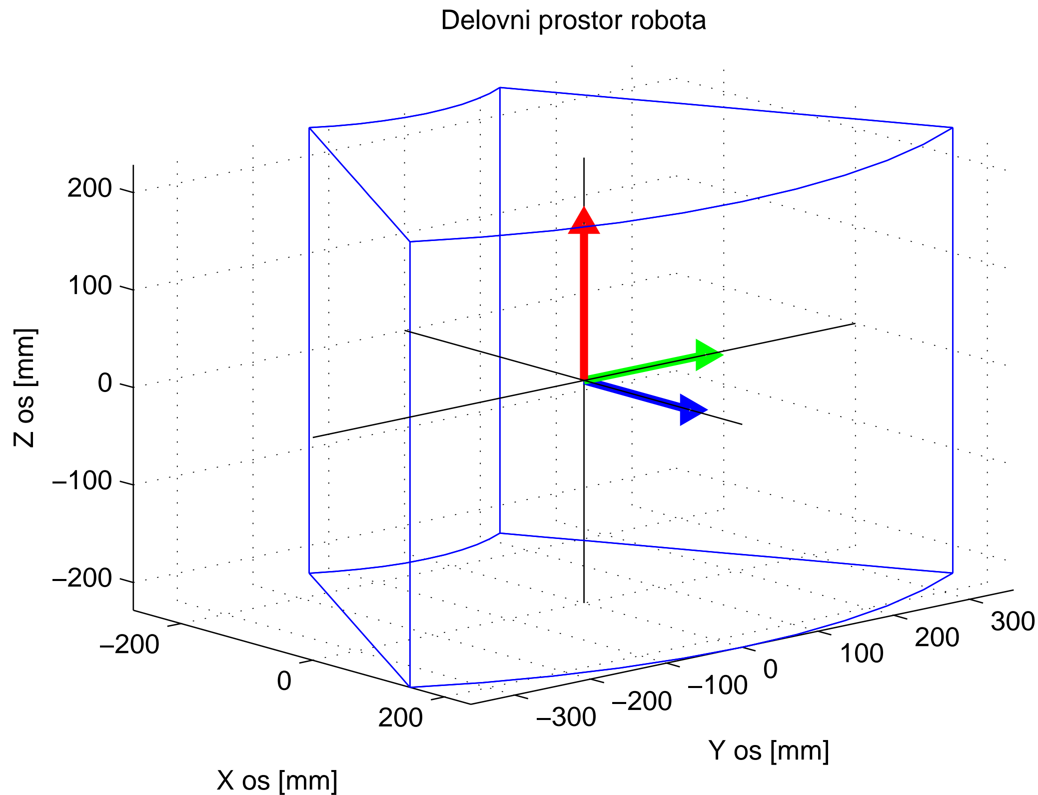
Tabela 2.1: Območje posameznih sklepov

Za pogon so uporabljeni enosmerni servo motorji proizvajalca Parvex iz družine RS (http://www.parvex.com/english/products/product_brochure.htm). Navpično os poganja RS240, preostali dve osi pa RS130. Vsaka os je poleg motorjev opremljena tudi z inkrementalnim enkoderjem in tahometrom[2]. Ločljivosti enkoderjev so naštete v tabeli 2.2.

Servo motorji so krmiljeni tokovno preko servo ojačevalnikov tipa 10A8 (Ad-

označimo tistega, ki je najbližje bazi robota. Zavoljo združljivosti s FCS kodo smo ohranili tudi njihovo oštevilčenje osi. Tako se v izhodiščni legi robota 1. sklep ujema z X osjo svetovnega koordinatnega sistema, 2. z Y osjo in 3. z Z osjo. Uporabljeno oštevilčenje tudi olajša programiranje robota. Prvo os krmili 1. krmilni kanal na 1. vmesniški kartici, drugo os 2. kanal na 1. vmesniški kartici in tretjo os 1. krmilni kanal na 2. vmesniški kartici.)

[2]Motorji družine RS so lahko tovarniško opremljeni z enkoderjem in tahometrom. Verjetno je za motorje tega manipulatorja uporabljena ta opcija.



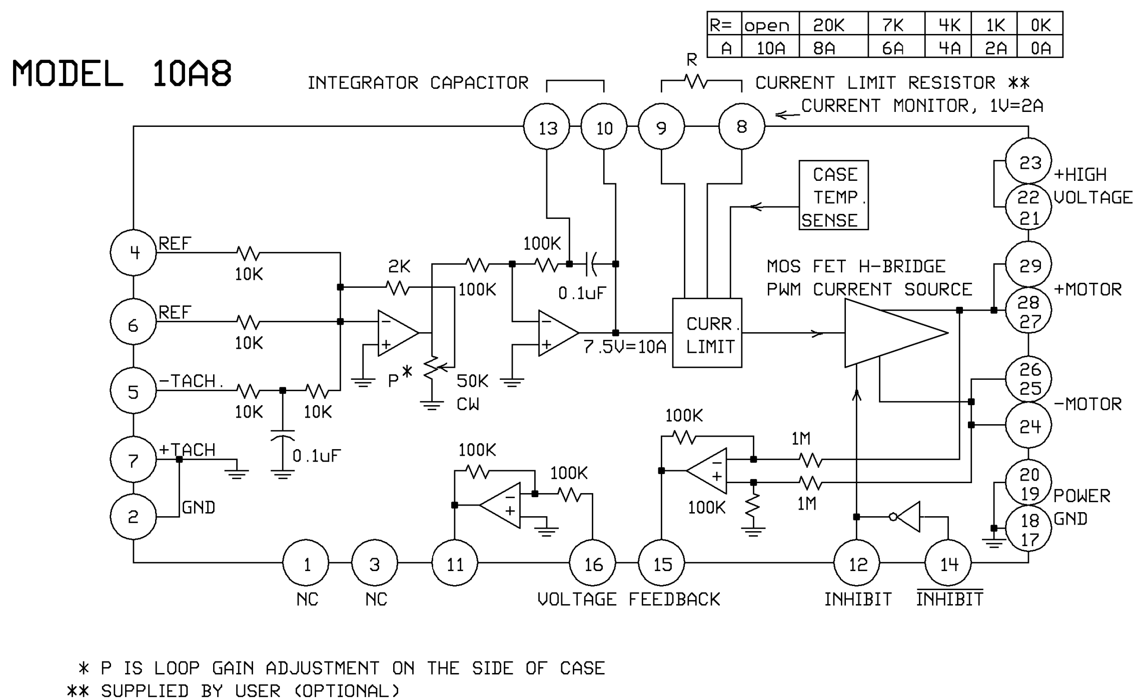
Slika 2.4: Delovni prostor robota HapticMaster

Os	Ločljivost enkoderja	enota
1	-5.080	μm/pulz
2	8.467	μrad/pulz
3	5.080	μm/pulz

Tabela 2.2: Ločljivost enkoderjev

vanced Motion Controls, ZDA, http://www.advancedmotioncontrols.com/download/datasheet/10a8.pdf). Vhod v 10A8 ojačevalnik (glej sliko 2.5) sta referenčna in dejanska hitrost, izhod pa je tokovni PWM signal toka do 10 A (ob konicah, 6 A stalno) in napetosti do 80 V. Izhod neposredno krmili krtačne enosmerne motorje. Ojačevalnik je zaščiten pred prenapetostjo, tokovno preobremenitvijo, pregretjem in kratkim stikom.

Robot uporablja inkrementalne enkoderje, zato mora po vklopu poiskati začetno lego.

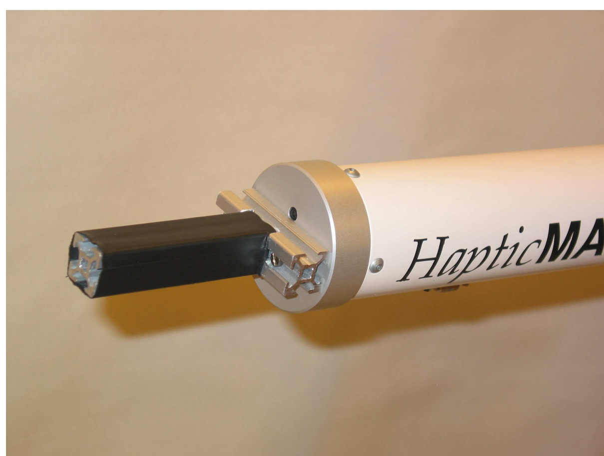


Slika 2.5: Shema 10A8 servo PWM ojačevalnika

Ker enkoderji nimajo vgrajenega indeksnega pulza, je iskanje začetne lege izvedeno s počasnim premikanjem vsakega sklepa, dokler se sklep ne zaleti oz. neha premikati. Ko je motor “zabit”, se pojavi velika razlika med referenčno in dejansko hitrostjo motorja, zato ojačevalnik krmili motor z največjim možnim tokom. Tokovne omejitve so nastavljene na vrednosti, ki dovoljujejo trajno obratovanje z zabitim motorjem brez nevarnosti termičnega uničenja motorja.

Na vrhu robota je 3-dimenzionalni analogni senzor sile (slika 2.6). Ta vsebuje merilno celico z uporovimi lističi. Trije analogni izhodi so priključeni na analogne vhode vmesniških kartic v krmilnem računalniku. Nazivno merilno območje za vsako izmed treh osi je 100 N. Tabela 2.3 podaja konstante za preračun sile iz bitov v N, kot jih je za konkreten senzor s kalibracijo določil proizvajalec.

Na vrh senzorja sile lahko namestimo tudi rotacijski merilni mehanizem s tremi rotacijskimi prostostnimi stopnjami (slika 2.7). Pri našem merilnem mehanizmu so vse tri prostostne stopnje pasivne, obstaja pa tudi tip z eno aktivno in dvema pasivnima prostostnima stopnjama. Koti so merjeni s potenciometri. Tabela 2.4 podaja konstante



Slika 2.6: Ohišje 3 dimenzionalnega senzorja sile na vrhu robota. Na senzor je pritrjena ročica za prijemanje.

Os	Občutljivost	Enota
1	-3.022	mN/bit
2	-2.911	mN/bit
3	2.871	mN/bit

Tabela 2.3: Občutljivost za posamezne osi senzorja sile

za preračun iz bitov v kot zasuka.

Os	Občutljivost	Enota
1	0.193	mRad/bit
2	0.162	mRad/bit
3	-0.165	mRad/bit

Tabela 2.4: Občutljivost za posamezne osi rotacijskega merilnega mehanizma

2.1.2 Krmilni računalnik

Kot krmilni računalnik je uporabljen industrijski PC. Vsebuje 850 MHz Celeron procesor, 128 MB RAM pomnilnika in "flash" trdi disk velikosti 16 MB. Ima priključek za miško (RS-232 vrata), tipkovnico, monitor, 100 MBit/s ethernet omrežje in paralelna vrata. Grafična kartica, integrirana na matični plošči, omogoča najvišjo ločljivost

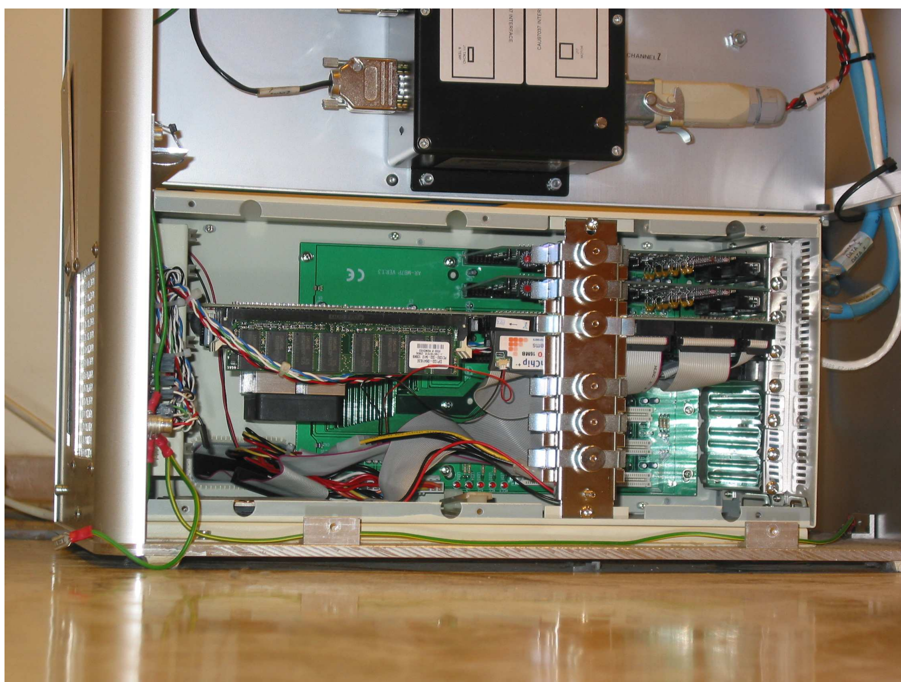
Slika 2.7: Rotacijski merilni mehanizem

800×600 točk. Poleg tega ne omogoča 3D pospeševanja, zato na tem računalniku ne moremo izvajati vizualizacije haptičnega navideznega okolja. Poleg naštetega so na matični plošči še reže PCI in ISA vodila. Računalnik komunicira z manipulatorjem in močnostnimi ojačevalniki preko dveh vmesniških kart (t. i. CIC karti).

2.1.2.1 CIC vmesniška karta

CIC karta je 16-bitna ISA karta in zaseda eno ISA režo. V HapticMaster-ju sta uporabljeni dve takšni karti. Vsaka ima naslednjo funkcionalnost:

- generator prekinitev s frekvenco 5000 Hz in z nastavljivim nivojem prekinitve ("interrupt level"),
- čuvaj časa,
- izbiro osnovnega naslova s stikalom in
- dva neodvisna krmilna kanala za aktuatorje.



Slika 2.8: Krmilni računalnik

En krmilni kanal nudi:

- vhod za kvadraturni signal inkrementalnega enkoderja,
- dva analogna vhoda, uporabljena za priključitev enega kanala analognega senzorja sile in enega kanala merilnega mehanizma,
- analogni izhod, uporabljen za hitrostno krmiljenje motorja,
- digitalni vhod za nadzor statusnih linij močnostnega ojačevalnika,
- digitalni izhod za nadzor omogočitvenih linij ("enable lines") močnostnega ojačevalnika.

Kartici potrebujeta po 16 zlogov I/O naslovnega prostora. Podatki so v obliki 16-bitnih besed. I/O prostor prve kartice (krmili prvo in drugo os), se začne na naslovu 0x8100, naslovni prostor druge kartice (krmili tretjo os) pa na 0x8110. Naslovi so izbrani s stikalom na kartici. En inkrement stikala spremeni osnovni naslov za 16 zlogov (npr. z 0x8100 na 0x8110).

Premik	bralni register	pisalni register
0x0000	analogni vhod 0	diskretni izhod
0x0002	analogni vhod 1	analogni izhod A
0x0004	analogni vhod 2	analogni izhod B
0x0006	analogni vhod 3	nivo prekinitve
0x0008	enkoder A	nastavitev prekinitve
0x000A	enkoder B	
0x000C	diskretni vhod	
0x000E	ID kartice	

Tabela 2.5: Registri CIC karte

Kartica vsebuje bralne in pisalne registre. Naslovni prostor teh registrov se prekriva – vpisovanje na naslov 0x8100 ter branje z naslova 0x8100 se obakrat dogaja na istem naslovu, toda kljub temu dostopamo do dveh različnih registrov. Vrednosti, vpisane na 0x8100, ne moremo kasneje prebrati iz registra – ko beremo z naslova 0x8100, beremo iz vhodnega registra (vpisovali pa smo v izhodni register).

Generator prekinitev s frekvenco 5000 Hz je aktiven samo na kartici z naslovom 0x8100. Namenjen je za proženje krmilnega programa robota v realnem času.

Čuvaj časa je zadolžen za zasilni izklop naprave v primeru programskih napak. Če ga krmilni program ne obnavlja dovolj hitro, mora izklopiti motorje. S tem je zagotovljena varna ustavitev robota v primeru napake v krmilnem programu. Potrebna frekvenca obnavljanja je 500 Hz. Obnovitev dosežemo s preklopom ustreznega bita v digitalnem izhodnem registru.

Enkoderska vhoda vsebujeta vrednosti med 0 in 1023. Uporabnik je odgovoren za detekcijo preliva ("overflow"), ki jo nato uporabi v programski realizaciji absolutnega enkoderja.

Analogne vhode beremo kot 16-bitne predznačene besede (short podatkovni tip)[3]. Kartica uporablja 16-bitni analogno-digitalni pretvornik (ADC). Kartica vzorči vsak kanal z 10 kHz, prebrane vrednosti iz registrov pa so povprečje štirih vzorcev. Koeficient za pretvorbo iz bitov v napetost (V) ni podan. Namesto tega FCS podaja koeficient za pretvorbo iz bitov v silo (N) (za senzor sile) oz. iz bitov v stopinje (za določitev kotov rotacije merilnega mehanizma).

Analogna izhoda sta uporabljena za določanje referenčne hitrosti prostostnih stopenj. Tabela 2.6 podaja konstante za pretvorbo iz bitov v hitrost prostostne stopnje. Pri tem je pomembno, da hitrost 0 m/s zahtevamo z vpisom vrednosti 0x8000 (32768) v register. Vzrok temu je manjši spodrsljaj pri načrtovanju kartice, kar zahteva od programerja dodatno pazljivost.

Os	Koeficient	enota
1	12094	bitov / m/s
2	-7257	bitov / rad/s
3	-24189	bitov / m/s

Tabela 2.6: Pretvorba iz bitov analogne izhodne besede CIC karte v referenčno hitrost

Biti digitalnega vhodnega registra so našteti v tabeli 2.7. Bita 0 in 2 sta enaka 0, če ustrezni močnostni ojačevalnik nima napajanja. Vzrok temu je lahko stikalo za zasilni izklop ali pa je uporabnik pozabil pritisniti gumb za vklop enosmernega napajalnika močnostnih ojačevalnikov. Bita 1 in 3 sta nastavljena, kadar je ustrezen ojačevalnik onemogočen (glej diskretno izhodno besedo, tabela 2.8). Bita 4 in 5 sta namenjena za indeksni signal enkoderjev oz. za končna stikala. HapticMaster nima ne enih ne drugih, zato sta ta dva bita neuporabljena. Bit 7 omogoča preverjanje delovanja čuvaja časa. Kadar ta dovoljuje delovanje motorjev, je bit 7 nastavljen na 1.

[3] Del dokumentacije HapticMaster-ja številka 32 je tudi dokument označen z RHM00044. V tem dokumentu kalibracijski podatki za senzor sile (premik in naklon za preračun iz bitov v N) nakazujejo, da sila 0 N povzroči vrednost analognega vhodnega registra 32768 bitov. Vendar ni tako. Teh 32768 bitov nastopa zato, ker FCS-jeva koda že pri branju analogne vrednosti prišteje 32768 bitov. Kasneje morajo teh 32768 bitov odšteti.

Bit	Maska	Simbolično ime (ang.)
0	0x0001	Not Amplifier Power A
1	0x0002	Amplifier fault A
2	0x0004	Not Amplifier Power B
3	0x0008	Amplifier fault B
4	0x0010	Settle Switch A
5	0x0020	Settle Switch B
6	0x0040	Serial No
7	0x0080	Watchdog OK

Tabela 2.7: Biti v digitalnem vhodnem registru

Bit	Maska	Simbolično ime (ang.)
0	0x0001	Not Amplifier Enable A
1	0x0002	Not Amplifier LR Enable A
2	0x0004	Not Amplifier Enable B
3	0x0008	Not Amplifier LR Enable B
4	0x0010	
5	0x0020	Watchdog
6	0x0040	TXD enable (not used)
7	0x0080	Encoder reset

Tabela 2.8: Biti v digitalnem izhodnem registru

Biti digitalnega izhodnega registra so prikazani v tabeli 2.8. Bita 0 in 1 morata biti oba enaka 0, da močnostni ojačevalnik prvega krmilnega kanala krmili motor. Enako funkcijo imata bita 3 in 4 za drugi krmilni kanal. S preklopom bita 5 dosežemo obnovitev čuvaja časa. Bita 6 in 7 sta oba neuporabljena[4].

[4]Bit 7 ("encoder reset") je bil uporabljen pri simuliranju inkrementalnega enkoderja v primeru priključitve absolutnega enkoderja.

2.2 Programska oprema

Originalna programska oprema robota HapticMaster sestoji iz dveh delov. Aktuatorje robota krmili program v VxWorks realno časovnem operacijskem sistemu. VxWorks v osnovi zagotavlja klasične storitve operacijskih sistemov, kot so dostop do trdega diska oz. do datotek datotečnega sistema na trdem disku, dostop do omrežja preko TCP/IP in UDP/IP protokola, večnitno izvajanje programov itd. V VxWorks je možna tudi zaščita pomnilnika med različnimi programi. Zaščita pomnilnika je samo opcija – za enostavne programe (ali pa za časovno zelo kritične) lahko zaščito pomnilnika izklopimo, tako da prevedeni programi delajo neposredno s pomnilnikom.

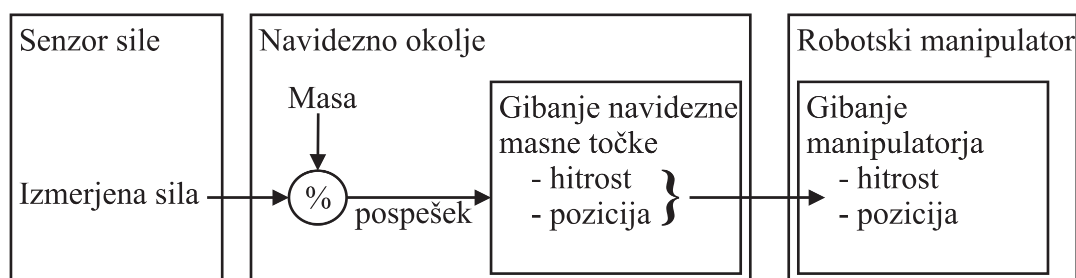
Krmilni program premika vrh robota glede na izmerjeno silo senzorja sile in glede na zahtevano haptično okolje. Haptično okolje je zgrajeno na osnovi zahtev, ki jih krmilnemu programu (strežniku) pošlje odjemalec.

Odjemalec se izvaja na ločenem računalniku. S krmilnim programom komunicira preko omrežne povezave. FCS ponuja C++ knjižnico HapticAPI, ki omogoča komunikacijo s krmilnim programom. HapticAPI lahko uporabljamo v Windows ali Linux operacijskem sistemu.

2.2.1 Krmilni program v operacijskem sistemu VxWorks

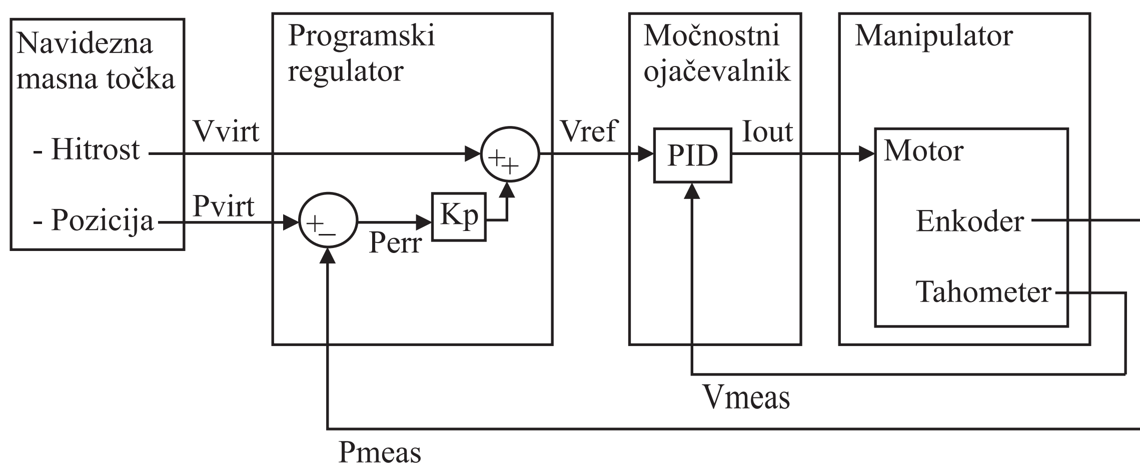
Celoten krmilni program je skupaj z operacijskim sistemom shranjen v eni sami datoteki. Osnova krmilnika je haptični algoritem, v katerem je vrh robota predstavljen kot točka z maso. Blok shema je prikazana na sliki 2.9. Ta masa se giblje zaradi sile, s katero človek pritiska na senzor sile, ter sil, ki se pojavijo v kontaktu s haptičnimi objekti. S tem je določen pospešek mase. Hitrost in pozicijo mase nato dobimo kot prvi in drugi integral pospeška. Kadar vrh robota dosledno sledi tej navidezni masni točki, se robot vede kot haptični vmesnik v podanem haptičnem okolju.

Gibanje robotskega manipulatorja določamo z referenčno hitrostjo močnostnih ojačevalnikov, ki jo vpisujemo v register izhodne analogne vrednosti CIC karte. Blok shemo za izračun reference močnostnih ojačevalnikov prikazuje slika 2.10. Kot vi-



Slika 2.9: Blok shema haptičnega algoritma

dimo, je referenčna hitrost V_{ref} izračunana kot vsota pozicijske napake med pozicijo navidezne masne točke P_{virt} in pozicijo vrha robota P_{meas}, pomnožene s pozicijskim ojačanjem Kp, in hitrostjo navidezne masne točke V_{virt}. Dejanska hitrost aktuatorjev V_{meas} dosledno sledi zahtevani referenčni hitrosti, ker močnostni ojačevalnik določa tok aktuatorja I_{out} preko (analogno realiziranega) PID algoritma. Analogni PID regulator ima kot vhod referenčno hitrost V_{ref} in izmerjeno hitrost V_{meas}. Do napak pride predvsem zaradi analognih napak (npr. napetostni premik in napaka naklona karakteristike) elementov v analogni regulacijski zanki (npr. D/A pretvornika, tahometra, odštevalnika). Zato je uporabljen tudi programski regulator s pozicijsko napako, tako da se vpliv teh napak s časom ne povečuje. Če bi ta člen izpustili, bi celotno haptično okolje počasi lezlo in na koncu "ušlo" iz delovnega prostora robota.



Slika 2.10: Blok shema izračuna referenčne hitrosti robota iz gibanja navidezne masne točke

Krmilni algoritem izrisovanja haptičnega okolja skrbi tudi za varnost. V primeru preve-

like sile na vrhu robota, prevelike pozicijske ali hitrostne napake med navidezno masno točko in vrhom robota bo robot ustavljen. Najpomembnejši del varnosti je omejevanje pozicije masne točke. Ta ne sme uiti izven delovnega prostora robota. Zakaj je to nevarno, najlažje razložimo na primeru napete vzmeti. En konec vzmeti je pripet v sredino delovnega prostora, drugi pa na vrh robota oz. na masno točko. S pritiskanjem na senzor sile se vrh robota in masna točka skupaj premikata do roba delovnega območja. Tu se robot ustavi, masna točka pa se giblje še naprej. Nato nehamo pritiskati na senzor sile. Masna točka se prične vračati, pri čemer prejema v vzmeti shranjeno energijo. Vrh robota medtem čaka na robu delovnega prostora. V trenutku, ko masna točka doseže rob delovnega območja, se bo vrh robota hipoma začel gibati s hitrostjo masne točke. Za roko, ki drži senzor sile, to pomeni nenaden udarec. Zato je potrebno omejiti tudi gibanje masne točke na delovni prostor robota.

Navidezno haptično okolje je sestavljeno iz haptičnih objektov in navidezne, gibajoče se masne točke. Na voljo so objekti, kot so kvader, krogla, valj, stožec, toroid ipd. Osnovni gradnik objektov je stena. Objekti imajo po dve steni, notranjo in zunanjo, zato so lahko tudi votli. Geometrijsko objekte opišemo z njihovo pozicijo, orientacijo in debelino sten. Za nekatere potrebujemo še dodatne geometrijske parametre – npr. toroid potrebuje še notranji in zunanji premer. Posamezno steno opišemo z dvema parametroma – trdoto in koeficientom dušenja. Ko se masna točka nahaja v haptični steni, se celota vede kot dušeno nihalo. Enakovreden mehanski sistem je nihajoča masa na vzmeti ob prisotnosti dušenja (masna točka prispeva maso, haptična stena pa trdoto vzmeti in dušenje).

Haptične objekte moramo pred uporabo ustvariti (v resnici samo vzamemo še neuporabljen objekt iz polja vseh haptičnih objektov posameznega tipa). Nato mu preko omrežne povezave nastavimo haptične in geometrijske lastnosti ter ga omogočimo. Od tu naprej haptični algoritem objekt izrisuje, dokler objekta ne onemogočimo ali izbrišemo.

S haptičnimi objekti delamo izključno preko omrežja. Za komunikacijo je uporabljen CORBA protokol. Implementacijo so napisali v podjetju FCS. CORBA je objektno

orientiran protokol in omogoča izvoz celotnih objektov (tj. podatkov in pripadajočih funkcij) s strani strežnika (krmilni računalnik, kjer se izvaja haptični algoritem) preko omrežne povezave na stran odjemalca. Odjemalec je osebni računalnik, ki generira zahteve za haptično okolje z uporabo knjižnice HapticAPI ter izrisuje grafično predstavitev haptičnega okolja. FCS uporablja CORBO izključno za prenos podatkov (ne uporabljajo objektno orientirane funkcionalnosti protokola). Strežnik in odjemalec morata sama poskrbeti za ustrezno interpretacijo teh podatkov (npr. ugotoviti, da je zahtevana operacija sprememba pozicije, na vrednost [X, Y, Z], za objekt tipa kvader, ki je 5. v polju vseh kvadrov).

2.2.2 Knjižnica HapticAPI

Knjižnica HapticAPI je visokonivojska knjižnica za delo s haptičnimi objekti. Njena uporaba je detajlno opisana v [34] in [35], kjer so tudi podrobno razloženi programski primeri. S strežnikom komunicira preko omrežne povezave. Na voljo je C++ vmesnik. Knjižnico lahko uporabljamo v Windows ali Linux operacijskem sistemu.

Naslednji primer prikazuje minimalen program, ki z uporabo knjižnice HapticAPI inicializira robota ter ustvari haptični objekt kvader. Program je enakovreden prvemu programskemu primeru iz [34], le da je odstranjena koda za grafični prikaz in preverjanje napak (vrnjenih vrednosti posameznih funkcij). Funkcija main() najprej izvede inicializacijo robota tako, da pokliče funkcijo InitHapticMaster(). V tej funkciji dobimo kazalec na objekt tipa HapticMaster s klicem funkcije ConnectToHapticMASTER(). Dobljeni kazalec poleg robota predstavlja tudi celotno navidezno haptično okolje. Nato zahtevamo inicializacijo (iskanje domače lege) robota tako, da mu nastavimo stanje FCSSTATE_INITIALIZED ter počakamo, da to stanje doseže. Naslednji korak je prehod v normalno stanje, tj. ko robot normalno izrisuje haptične objekte. Sledi še odstranitev vseh haptičnih objektov, ki so jih ustvarili in pozabili odstraniti predhodni programi, ter nastavitev navidezne mase na 2 kg.

Funkcija main() lahko nato začne z ustvarjanjem novega haptičnega okolja. V tem enostavnem primeru je le-to sestavljeno iz enega samega kvadra. Tega najprej ustvarimo

s CreateBlock(), nato pa še omogočimo s klicem Enable(). V nakazani while zanki bi lahko bila koda za beleženje podatkov ter grafično 3D vizualizacijo. Ko pritisnemo tipko ESC (ta ima ASCII kodo 27), se zanka prekine. Program nato uniči ustvarjen haptični kvader in zaključi z izvajanjem.

```
#include "HapticAPI.h"
#include "FcsHapticMaster.h"
#include "FcsBlock.h"

#include <windows.h>
#include <conio.h>

CFcsHapticMASTER *pHapticMaster;
CFcsBlock *pBlock;

// Location, Orientation And Size Parameters For The Haptic Block Object
double BlockCenter[3] = {0.0, 0.0, 0.0};
double BlockOrient[3] = {0.0, 0.0, 0.0};
double BlockSize[3] = {0.15, 0.15, 0.15};

// This Function Connects To The HapticMASTER And Initializes It.
int InitHapticMaster(void)
{
    FCSSTATE currentState;

    // Call ConnectToHapticMASTER To Get A Pointer To A
    // CFcsHapticMASTER Object.
    pHapticMaster = ConnectToHapticMASTER ("vmd");

    // Set The HapticMASTER State To Initialised
    pHapticMaster->SetRequestedState ( FCSSTATE_INITIALISED);

    // Wait for The HapticMASTER To Enter The Initialized State
    while (currentState != FCSSTATE_INITIALISED)
    {
        pHapticMaster->GetCurrentState (currentState);
        Sleep(100);
    }

    // Set The HapticMASTER State To NormalForce
    pHapticMaster->SetRequestedState ( FCSSTATE_NORMAL);

    // Wait For The HapticMASTER To Enter The NormalForce State
    while (currentState != FCSSTATE_NORMAL)
```

```
    {
        pHapticMaster->GetCurrentState (currentState);
        Sleep(100);
    }

    // Call pHapticMaster->DeleteAll() To Delete All Haptic Objects
    // Currently Active In The HapticMASTER
    pHapticMaster->DeleteAll();

    // Set The HapticMASTER Inertia To 2.0 Kg.
    pHapticMaster->SetParameter (FCSPRM_INERTIA, 2.0);

    return 0;
}

// Example_01 Main Function
int main(int argc, char** argv)
{
    // Call The Initialize HapticMASTER Function
    InitHapticMaster();

    // If Initiaizing Was OK The CReate A Haptic Block Object
    pBlock = pHapticMaster->CreateBlock(BlockCenter, BlockOrient, BlockSize);
    // The Haptic Block Object Is Disabled By Deafult.
    // Enable The Haptic Block Object
    pBlock->Enable();

    while (getch()!=27)
    {
        // Data logging and 3D visualization
    }

    // Clean Up The Haptic Block Object On The HapticMASTER Side
    pHapticMaster->DeleteBlock(pBlock);

    // Clean Up The Haptic Block Object On The Client Side
    delete pBlock;

    return 0;
}
```

3.
Programska oprema za RTLinux

Programska oprema za VxWorks operacijski sistem omogoča enostavno ustvarjanje haptičnih okolij z uporabo knjižnice HapticAPI. Ko je haptično okolje ustvarjeno, je naloga enostavnih programov samo še spremljanje pozicije vrha robota ter posodabljanje grafičnega prikaza. Kompleksnejši programi morajo poleg tega še shranjevati podatke (npr. beležiti trajektorijo roke) ter spreminjati haptično okolje (npr. občasno premakniti kakšen haptični objekt).

Za beleženje podatkov o človeškem gibanju zadošča vzorčna frekvenca 100 Hz. Vsak klic funkcije iz knjižnice HapticAPI zahteva približno 0.6 ms. Če potrebujemo več različnih spremenljivk (npr. pozicijo, hitrost in silo), moramo izvesti več klicev funkcij in čas izvajanja enega cikla se poveča. K temu moramo dodati še nestabilnost vzorčnega časa, do katere pride zaradi razporejanja opravil v operacijskem sistemu Windows oz. Linux. V Windows so možne zakasnitve nekaj 10 ms, če so razmere neugodne.

V dinamičnem haptičnem okolju je potrebno haptične objekte spreminjati v majhnih korakih, tako da dobi uporabnik občutek zveznih sprememb. Primer uporabe je vodenje pacientove roke s pomočjo haptične vzmeti, ki povezuje pacientovo roko (vrh robota) z željeno referenčno točko. Referenčna točka se premika, tako da dobimo referenčno trajektorijo. Če je čas za izvedbo enega cikla zanke velik, se referenčna točka premika skokovito, pacientova roka pa namesto zveznega povečevanja sile vlečenja čuti neprijetno "cukanje".

Vsaj (in samo) delno lahko te težave omilimo s primerno večnitno zasnovo programa.

Časovno kritična delovna nit bi bila zadolžena samo za beleženje podatkov ali za osveževanje dinamičnega okolja. Grafični prikaz in uporabniški vmesnik bi se izvajala v glavni zanki v časovno nekritični niti. S tem bi časovno kritično opravilo ločili od počasne glavne zanke. Prevelike časovne zakasnitve se še vedno lahko pojavijo zaradi drugih opravil v sistemu (delo z diskom, omrežjem, grafiko, zvokom) in zaradi komunikacije knjižnice HapticAPI preko omrežja. Z uporabo knjižnice HapticAPI ne moremo doseči zanesljivega izvajanja v realnem času.

Obstajajo tudi opravila, ki jih s HapticAPI sploh ne moremo realizirati. Lastne haptične objekte (npr. tunel) bi lahko implementirali s pomočjo haptičnega objekta za prikaz konstantne sile (FcsConstantForce). Glede na pozicijo vrha robota bi spreminjali silo, ki jo povzroča FcsConstantForce. Vendar bi bila kvaliteta takšnih objektov nizka zaradi časovnih zakasnitev pri komunikaciji ter preklapljanja opravil v operacijskem sistemu odjemalca.

Drug primer pa je implementacija robotskih regulacijskih algoritmov, kar je del laboratorijskih vaj pri robotskih predmetih. HapticAPI namenoma skriva nizkonivojske detajle, kot so delo s strojno opremo, regulacijski algoritem ali izračun sile haptičnega objekta. Implementacija lastnih regulacijskih algoritmov preprosto ni mogoča.

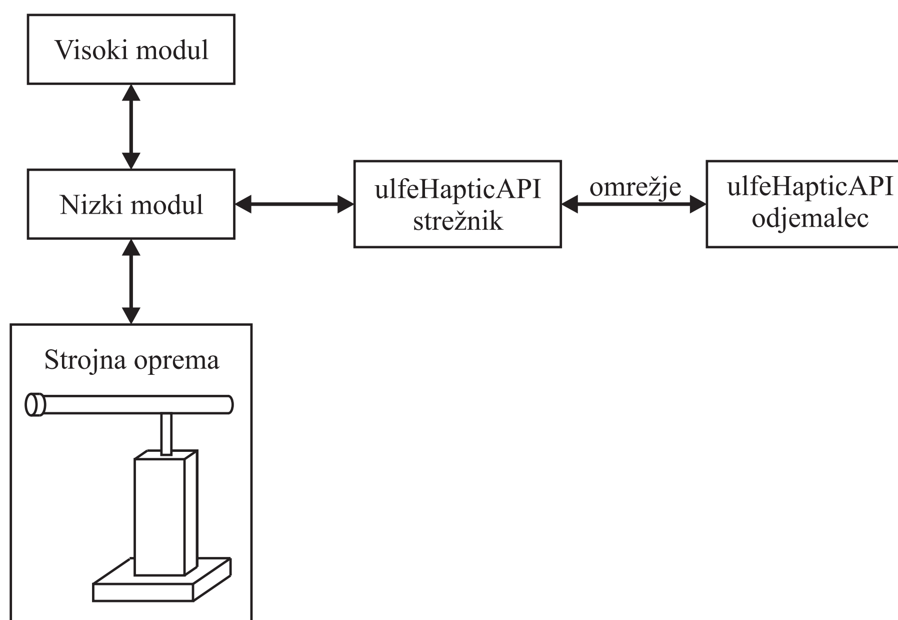
Zato smo se odločili za implementacijo programske opreme za robota HapticMaster v operacijskem sistemu RTLinux. Pri tem smo želeli ohraniti podobnost z obstoječim VxWorks sistemom, hkrati pa dodati možnost implementacije lastnih regulacijskih algoritmov, lastnih haptičnih objektov in beleženja podatkov v realnem času. Ker HapticAPI ponuja preprost in učinkovit vmesnik za uporabo haptičnih objektov, smo implementirali tudi enakovreden nadomestek (knjižnico ulfeHapticAPI).

3.1 Zasnova

Kot vsak robot mora tudi HapticMaster pred začetkom normalnega delovanja ugotoviti svojo lego. Ker so uporabljeni inkrementalni enkoderji, je potrebno poiskati domačo lego ob vsakem zagonu krmilne programske opreme. Iskanje domače lege traja od 60 do

90 sekund, odvisno od lege robota pred začetkom iskanja. Med razvojem programa je običajno stalno preizkušanje programa. Čakati 1 minuto za testiranje tudi najmanjših sprememb postane po nekaj ponovitvah zelo moteče.

Zato smo se odločili, da bo programska oprema realizirana v treh delih (glej sliko 3.1). Prvi del (nizki modul – modul ("Linux kernel module") je izraz za gonilnik v Linux operacijskem sistemu) je stalno naložen. Vsebuje kodo in podatke, ki so nujno potrebni za delo z robotom. To so na primer gonilniki naprav, kinematika robota, varnostne omejitve in trenutna lega robota. V ta modul sodi tudi koda, ki ni nujna za delovanje robota (npr. haptični objekti), potem ko je več ne spreminjamo.



Slika 3.1: Struktura celotne programske opreme. Za delovanje je nujno potreben samo nizki modul.

Drugi del (visoki modul) se nahaja nad nizkim modulom. V njem implementiramo funkcionalnost, ki jo pogosto spreminjamo. Lahko uporablja funkcije, ki smo jih realizirali v nizkem modulu. Namenjen je npr. pošiljanju podatkov za beleženje, testiranju regulacijskih algoritmov ali implementaciji dinamičnih haptičnih okolij.

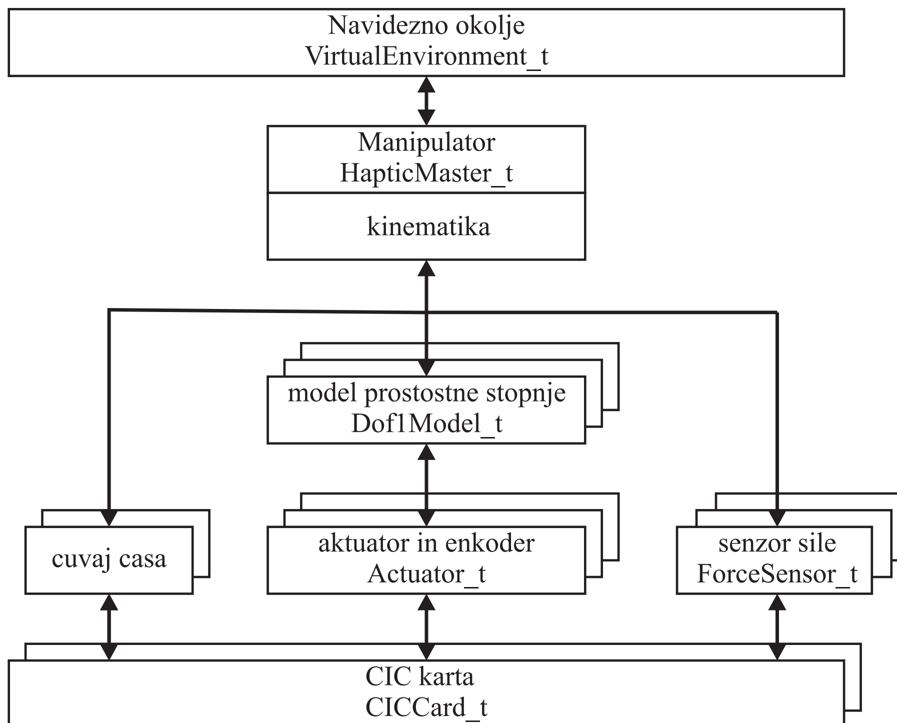
Tretji del je strežnik knjižnice ulfeHapticAPI. Knjižnica ulfeHapticAPI je nadomestilo za FCS-jevo HapticAPI. Strežnik preko omrežne povezave sprejema zahteve odjemalca ter jih posreduje nizkemu modulu. Nizki modul vrne rezultat strežniku, ta pa ga nato

posreduje odjemalcu.

3.2 Nizki modul

Koda za nizki modul je v poddirektoriju lowModule. Datoteka hmLowModule_main.c vsebuje kodo za realno časovno nit, ki se zažene ob vstavitvi modula v Linux jedro. Modul prevedemo z ukazom make, rezultat je binarni modul hmLowModule.o.

Strukturo nizkega modula prikazuje slika 3.2. Na vrhu je navidezno okolje – struktura VirtualEnvironment_t. Vsebuje haptične objekte, med katerimi se giblje navidezna masna točka. Tej v resničnem svetu ustreza vrh robota. Na njeno gibanje vplivajo sile kontakta s haptičnimi objekti in izmerjena sila na vrhu robota.



Slika 3.2: Bločni diagram strukture nizkega modula

Gibanje navidezne masne točke je referenca za gibanje samega robota (struktura HapticMaster_t). Ta med drugim tudi preverja varnost delovanja, npr. ali sta pozicija in hitrost znotraj dovoljenega območja. Preko strukture HapticMaster_t preberemo izhode senzorjev, kot so senzor sile in enkoderji. HapticMaster_t je zadolžen za preračun

med notranjimi in zunanjimi koordinatami, tj. kinematiko robota.

Model prostostne stopnje (Dof1Model_t) uporabljata tako HapticMaster_t kot tudi VirtualEnvironment_t. Vsak od treh modelov prostostne stopnje ustreza enemu sklepu robota. Model prostostne stopnje prejme kot vhod silo, ki deluje na ustrezen sklep. Ta sila je razlika izmerjene sile, s katero deluje na vrh robota človek, in sile haptičnega okolja, s katero delujejo vsi haptični objekti. Iz te sile in vztrajnosti se izračuna gibanje prostostne stopnje. To gibanje je podvrženo omejitvam – pospešek, hitrost in pozicija morajo ostati znotraj varnih mej.

Tako omejeno gibanje navidezne masne točke je referenca za aktuatorje. Ti so predstavljeni skupaj z pripadajočim enkoderjem v strukturi Actuator_t.

Senzor sile vrne izmerjeno silo, izraženo v koordinatnem sistemu vrha.

Čuvaj časa je potrebno periodično obnavljati, saj drugače izklopi robota. Pred vsako obnovitvijo najprej preverimo, če je delovanje robota varno. Možne napake so prevelike hitrosti, prekoračeno varno delovno območje ali prevelika sila na vrhu robota.

Na najnižjem nivoju je gonilnik za obe CIC karti, edinima deloma strojne opreme, s katerima modul neposredno komunicira. Ta gonilnik implementira nabor funkcij, s katerimi dostopamo do funkcionalnosti CIC karte. Poleg tega preprečuje, da bi pomotoma delali z I/O naslovi, ki niso del CIC karte.

3.2.1 Gonilniki naprav

3.2.1.1 CIC karta

V datotekah CICCard.h in CICCard.c je gonilnik za CIC karto. Gonilnik sestoji iz strukture CICCard_t in funkcij za delo s to strukturo. Podrobnosti o sami kartici ter o pomenu registrov so opisani v poglavju 2.1.2.1.

Imena funkcij se začno s predpono cic. Prvi parameter je vedno kazalec na strukturo CICCard_t (CICCard_t *pCIC). Večina funkcij dela z enim od obeh krmilnih kanalov, zato je drugi parameter številka kanala (int chan, vrednost 0 ali 1). Funkcije, ki vplivajo na oba krmilna kanala (npr. čuvaj časa, ki je skupen obema kanaloma), nimajo

parametra chan.

Ker so registri samo bralni oz. samo pisalni, shranjuje CICCard_t vpisano vrednost v register za vsak register, tako da jo lahko kasneje preberemo (branje iz izhodnega registra je simulirano, ker strojna oprema branja izhodnega registra ne omogoča). Shranjene so tudi vhodne vrednosti, prebrane iz vhodnih registrov. Shranjevanje vhodnih vrednosti sicer ni nujno potrebno, ima pa dve prednosti. Prvič, branje in pisanje preko ISA vodila je relativno počasna operacija, zato tako prihranimo nekaj procesorskega časa. Drugič pa tako znotraj enega cikla regulacijske zanke delamo s stabilnimi podatki.

Koristnost stabilnih podatkov ilustrira spodnja koda, kjer preberemo analogni vhod CIC karte. Je nekoliko zavajajoča, ker zaradi časovnega zamika ni nujno, da sta value1 in value2 natanko enaki. To utegne postati težava, če bi npr. obe vrednosti primerjali z operatorjem enakosti (==). Stabilnost podatkov je še bolj pomembna pri razhroščevanju programa. Tedaj lahko med izvedbo obeh zaporednih vrstic poteče poljubno mnogo časa, zato bi obe vrednosti lahko bile povsem različne. Pod takimi pogoji je težko ugotoviti, ali je izhodna vrednost algoritma napačna zaradi napake v algoritmu ali zaradi spreminjajočih se vhodnih vrednosti.

```
int value1, value2;
value1 = cicReadAnalog(pCIC, 0);
value2 = cicReadAnalog(pCIC, 0);
```

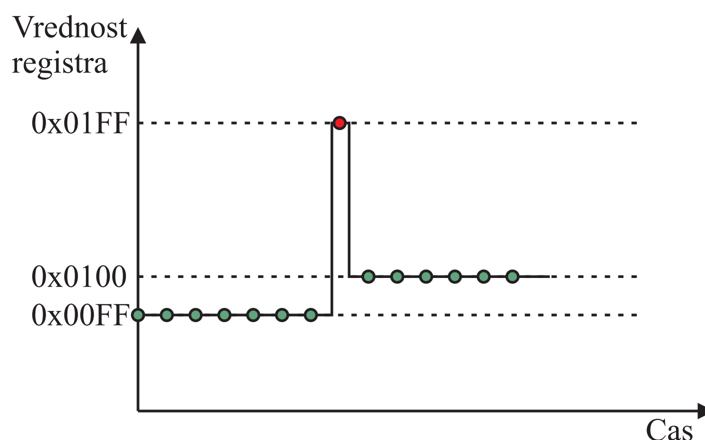
CICCard_t shrani vrednosti izhodnih registrov ob vsakem vpisu v izhodni register. V registre CIC karte se izhodne vrednosti vpišejo takoj. Branje vhodnih registrov iz karte se izvede samo enkrat na cikel regulacijske zanke. Takrat tudi shranimo prebrane vrednosti. Vrednosti vhodnih registrov prebere in shrani funkcija cicReadShadow().

Pri branju vhodnih registrov se pojavi težava z atomičnostjo vsebine samega registra. Pri prehodu vrednosti iz npr. 0x00FF v 0x0100 se lahko zgodi, da je prebrana vrednost sestavljena iz nove in iz stare vrednosti, tako da je rezultat 0x0000 ali pa 0x01FF. Pri branju pozicije enkoderja pomenijo takšni impulzi pozicije tudi impulze hitrosti, ki presežejo največjo dovoljeno hitrost. Ker to pomeni obratovanje zunaj varnega

območja, se posledično izklopijo vsi motorji. To se dogaja naključno med delovanjem, tako da je zelo moteče.

FCS kot rešitev predlaga branje vhodnih registrov takoj po preklopu čuvaja časa. Po preklopu čuvaja časa naj bi bil procesor na CIC karti nekaj časa zaposlen in naj ne bi obnavljal vsebine vhodnih registrov. Žal se ta rešitev ni obnesla.

Problem smo nato zaobšli s pomočjo večkratnega branja registra v enem ciklu regulacijske zanke. Predpostavljamo, da je vsebina registra neveljavna le zelo kratek čas (glede na največjo možno hitrost branja registra), oz. da je večino časa veljavna. To ilustrira slika 3.3. Vsak barvni krožec predstavlja eno prebrano vrednost registra. Večina vrednosti je veljavna (zeleni krožci), samo ob prehodu iz 0x00FF v 0x0100 se pojavi en neveljaven vzorec (rdeč krožec). Neveljavni vzorec izločimo tako, da dvakrat preberemo register. Če sta prebrani vrednosti enaki, smo skoraj gotovo prebrali veljavno vrednost. Če ne, izvedemo naslednje branje registra in upamo, da je vrednost enaka tisti iz prejšnjega branja.



Slika 3.3: Neveljavne vrednosti registra CIC karte

Takšno branje registra je implementirano z makrojem INW_WITH_LOOP(). Makro ponovi branje največ 10-krat. Če ne najde ponovljene vrednosti, vrne zadnjo prebrano vrednost. Ta makro je nato uporabljen v funkciji cicReadShadow(), ki prebere vse vhodne registre.

Izhodno analogno vrednost (referenčno hitrost za aktuatorje) nastavljamo s funkcijo cicWriteReference(). Vhodna referenčna vrednost je predznačeno 16-bitno število. Vre-

dnost parametra 0 bitov ustreza hitrosti 0 m/s (oz. 0 rad/s). Največja referenčna hitrost v bitih je omejena na ±CIC_REF_RANGE bitov. CIC_REF_RANGE ima lahko vrednost med 0 in 32768, trenutno je 8000. Mišljena je kot varovalka pred hudimi napakami v krmilnem programu.

Vrednost enkoderjev preberemo s cicReadRawEncoder() in cicReadEncoder(). Funkcija cicReadRawEncoder() vrne 10-bitno vrednost, tako kot je bila prebrana iz enkoderskega vhoda CIC karte. Funkcija cicReadEncoder() simulira absolutni enkoder. Vrnjena vrednost je predznačeno 32-bitno število. Ničlo simuliranega absolutnega enkoderja nastavimo na trenutno pozicijo s funkcijo cicResetEnkoder().

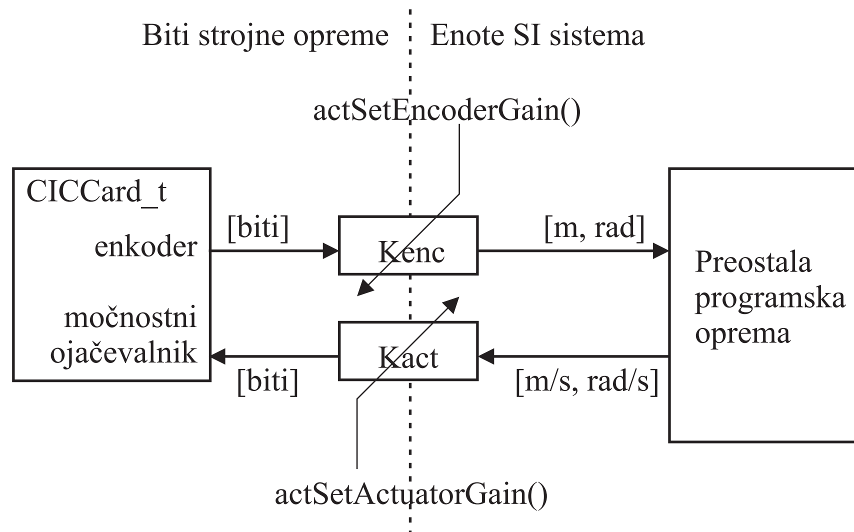
Na vsaki CIC karti je po en čuvaj časa. Ta časovnik je edina strojna zaščita pred programskimi napakami. Če krmilni program preneha obnavljati čuvaja časa, bo ta izklopil napajanje močnostim ojačevalnikom in s tem tudi motorjem. Časovnik obnovimo s preklapljanjem ustreznega bita v izhodni diskretni besedi. To izvede funkcija cicToggleWatchdog().

3.2.1.2 Aktuator

Struktura Actuator_t združuje enkoder kot merilni del in programski regulator za ustrezen močnostni ojačevalnik. Koda za strukturo Actuator_t je v datotekah Actuator.h in Actuator.c. Imena funkcij se začno s predpono act, prvi parameter pa je kazalec na strukturo Actuator_t (Actuator_t *pAct).

Ta struktura opravlja preračun pozicije in hitrosti med biti (s katerimi dela CIC karta) ter enotami SI sistema (s katerim dela vsa ostala programska koda), tako kot to prikazuje slika 3.4. S funkcijo actSetEncoderGain() nastavimo konstanto za preračun iz bitov enkoderja v metre (radiane). Z actSetEncoderRange() nastavimo območje sklepa (razdalja med obema skrajnima legama sklepa, v metrih oz. radianih). Z actSetAmplifierGain() nastavimo konstanto za preračun referenčne hitrosti iz m/s (rad/s) v bite za register CIC karte.

Struktura Actuator_t se lahko nahaja v enem izmed naslednjih stanj:

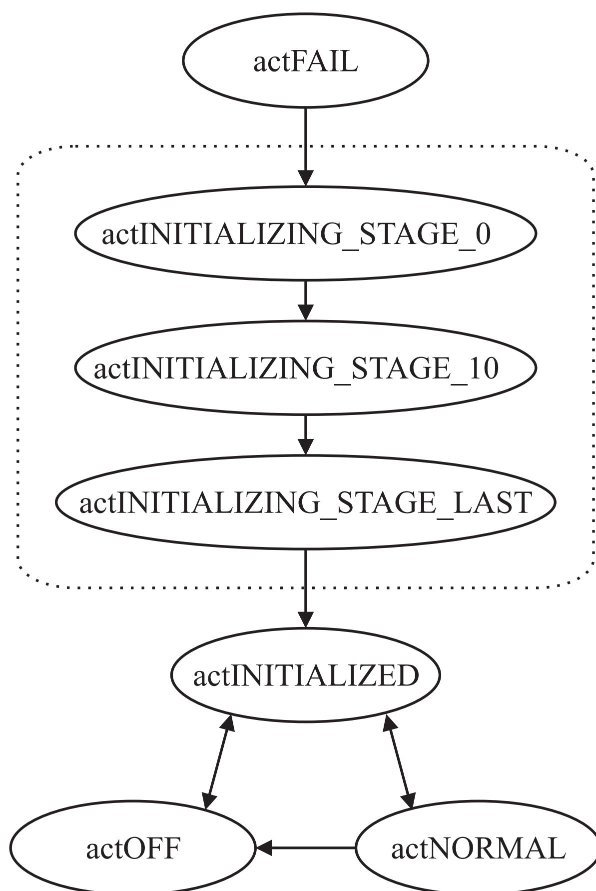


Slika 3.4: Preračun bitov v/iz enot SI sistema

- actFAIL,
- actOFF,
- actINITIALIZING_STAGE_0,
- actINITIALIZING_STAGE_10,
- actINITIALIZING_STAGE_11,
- actINITIALIZING_STAGE_12,
- actINITIALIZING_STAGE_LAST,
- actINITIALIZED,
- actNORMAL.

Prehode med posameznimi stanji podaja slika 3.5. Prehod v novo stanje zahtevamo s klicem funkcije actSetStateMachine(). Na začetku izvajanja programa se nahajamo v stanju actFAIL, kar pomeni ugasnjene motorje in neznano pozicijo sklepa. Naslednji korak je stanje actINITIALIZING* – motorji so vključeni, sklep pa išče domačo pozicijo. Ko je ta najdena, se stanje spremeni v actINITIALIZED, sklep pa ostane

v domači poziciji. V stanju actNORMAL je motor vključen in pozicija sklepa znana. Tedaj lahko programsko vodimo gibanje sklepa. Stanje actOFF pomeni izključene motorje in znano lego sklepa. Iz actOFF lahko pridemo v actNORMAL brez iskanja domače pozicije (torej nekoliko hitreje kot iz actFAIL).

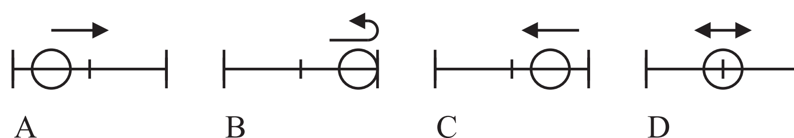


Slika 3.5: Prehodi stanj strukture Actuator_t

Struktura Actuator_t pozna hod svojega sklepa v metrih oz. radianih. Izhodišče koordinatnega sistema posamezne osi je postavljeno v sredino hoda osi[1]. Actuator_t zna poiskati svojo domačo lego. To zahtevamo s periodičnim klicanjem funkcije actSearchHomePosition(). V stanju actINITIALIZING_STAGE_0 se sklep počasi premika v pozitivni smeri (slika 3.6 A, trenutna pozicija je predstavljena s krožcem, celoten hod sklepa pa z vodoravno črto). V nekem trenutku se sklep zaleti in ustavi zaradi fizične omejitve

[1]S tem je določeno izhodišče sklepnega koordinatnega sistema. Tudi izhodišče svetovnega koordinatnega sistema je postavljeno v isto točko. Točka s koordinatami (0, 0, 0) se tako nahaja v sredini dosegljivega delovnega prostora za oba koordinatna sistema.

gibanja sklepa. Sledi prehod v stanje actINITIALIZING_STAGE_10 (10 pomeni 1.0), resetiranje enkoderja in sprememba smeri gibanja (slika 3.6 B). Sklep se zdaj počasi premika v negativni smeri, dokler ne pride do sredine sklepa (slika 3.6 C). Tu drugič resetiramo enkoder. Ta nam od tu naprej sporoča pozicijo sklepa v prej opisanem sklepnem koordinatnem sistemu. Sledi prehod v actINITIALIZING_STAGE_LAST, takoj nato pa v actINITIALIZED. V actINITIALIZED ima aktuator fiksno pozicijsko referenco 0, zato ostane v izhodiščni legi (slika 3.6 D). Stanji actINITIALIZING_STAGE_11 in actINITIALIZING_STAGE_12 sta neuporabljeni[2].

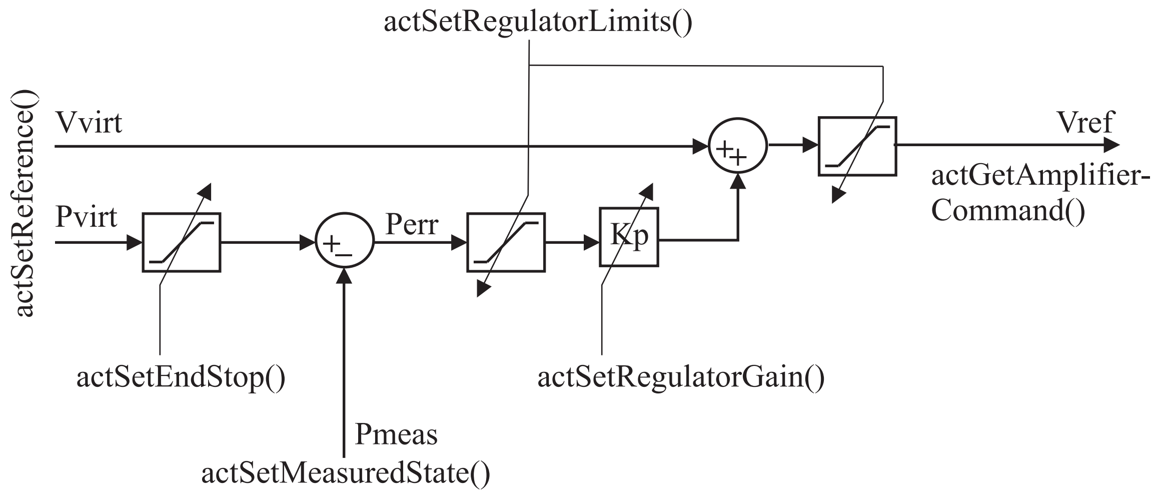


Slika 3.6: Iskanje domače pozicije aktuatorja. A – sklep se začne gibati v neznanem začetnem položaju. B – ko pride do konca območja gibanja, se ustavi (zaleti). C – takrat spremenimo smer gibanja. D – sklep pride do sredine območja gibanja, kjer ga ustavimo.

Programski regulator izračunava referenčno hitrost za CIC karto (Vref) iz referenčne hitrosti aktuatorja (Vvirt), referenčne pozicije aktuatorja (Pvirt) in dejanske pozicije aktuatorja (Pmeas). Algoritem je prikazan na sliki 3.7. Referenca za CIC karto je izračunana kot vsota nastavljene hitrosti ter s pozicijskim ojačenjem Kp pomnožene pozicijske napake (Perr). Referenčne vrednosti aktuatorja nastavljamo s funkcijo actSetReference(), nato pa lahko preberemo izhodno vrednost programskega regulatorja z actGetAmplifierCommand(). Ojačenja regulatorja nastavimo s funkcijo actSetRegulatorGain() ob inicializaciji strukture Actuator_t. Koda predvideva tudi vključitev

[2]Stanji actINITIALIZING_STAGE_11 in actINITIALIZING_STAGE_12 sta bili namenjeni programski kompenzaciji enosmernih analognih premikov izhodne referenčne hitrosti. Ta enosmerni premik povzroči počasno lezenje sklepa, tudi ko je referenčna hitrost nastavljena na 0. Ta premik je pri našem robotu izrazit predvsem v 1. osi, kjer znaša cca. 2 mm/s. V actINITIALIZING_STAGE_11 smo nastavili referenčno hitrost 0 ter počakali 10 sekund, da smo izmerili premik. V actINITIALIZING_STAGE_12 se je sklep vrnil v domačo lego. Nazadnje se je v actINITIALIZING_STAGE_LAST še izračunal popravek za izhodno referenčno hitrost ter izvedel prehod v actINITIALIZED. Z vključitvijo pozicijske povratne zanke v programski regulator strukture Actuator_t je koda za kompenzacijo analognega premika postala nepotrebna in je bila odstranjena.

pospeška v povratno zanko, vendar to ni uporabljeno (ojačanje napake pospeška je 0).



Slika 3.7: Programski regulator aktuatorja

Kot varnostni mehanizem sta največja pozicijska napaka in največja referenčna hitrost omejena. Omejitve nastavimo s funkcijo actSetRegulatorLimits(). Poleg tega je omejeno še pozicijsko območje aktuatorja. Območje omejimo s funkcijo actSetEndStop(), tako da na vsakem robu ostane 20 mm rezerve do konca fizičnega hoda sklepa. Če dejanska pozicija sklepa preseže omejitev ("end stop"), vsilimo referenčno pozicijo za programski regulator enako poziciji omejitve, referenčna hitrost pa je dovoljena samo v smeri proti izhodišču sklepa.

Spremenjene referenčne vrednosti programskega regulatorja se na strojni opremi ne pokažejo takoj. Pošiljanju izračunane referenčne hitrosti za močnostne ojačevalnike je namenjena funkcija actSendCmdToAmplifier(). To kličemo v funkciji actUpdateState(). Funkcija actUpdateState() je namenjena izvajanju operacij, ki se morajo izvesti natanko enkrat na cikel kontrolne zanke[3]. V primeru Actuator_t je to poleg pošiljanja hitrostne reference močnostnemu ojačevalniku še izračun dejanske hitrosti sklepa.

[3]Tudi nekatere druge strukture imajo funkcijo UpdateState(), ki se mora izvesti enkrat in samo enkrat v ciklu kontrolne zanke.

3.2.1.3 Senzor sile

Senzor sile je predstavljen s strukturo ForceSensor_t. Koda se nahaja v datotekah ForceSensor.h in ForceSensor.c. Prvi parameter funkcij je kazalec na strukturo ForceSensor_t (ForceSensor_t *pFs).

Funkcionalnost te strukture je minimalna. Strojna oprema je sestavljena iz uporovih lističev in ojačevalnikov, priključenih na analogne vhode CIC karte. Tako ni potrebno (niti ni mogoče) nobeno izbiranje nastavitev za senzor. Senzor meri vse 3 komponente sile in nobenega navora. Merilno območje je ± 100 N za vse tri osi.

Uporabnik lahko prebere trenutno vrednost izmerjene sile v N s funkcijo fsReadForce3(). Številka 3 v imenu funkcije pomeni, da je vrnjen rezultat vektor dolžine 3. Poleg tega lahko še izberemo ničlo senzorja s funkcijo fsResetOffset(). Ta upošteva trenutno vrednost izmerjene sile kot novo ničlo za vse tri osi.

3.2.2 Robotski manipulator

Celoten robotski manipulator je predstavljen s strukturo HapticMaster_t, katere koda je v datotekah HapticMaster.h in HapticMaster.c. Imena funkcij se začno s predpono hm, prvi parameter je kazalec na strukturo tipa HapticMaster_t (HapticMaster_t *pHm). Ta združuje 2 CIC karti, 3 aktuatorje, 1 tridimenzionalen senzor sile in varnostne omejitve.

HapticMaster_t se lahko nahaja v enem izmed naslednjih stanj:

- hmFAIL,
- hmOFF,
- hmINITIALIZING,
- hmINITIALIZED,
- hmNORMAL.

V stanju hmFAIL so motorji izključeni in robot ne ve, kje se nahaja. V hmOFF so motorji izključeni, pozicija robota pa je znana. V stanju hmINITIALIZING se robot nahaja, kadar išče domačo lego. Ko je domača lega najdena, se stanje spremeni v hmINITIALIZED. Večino časa se robot nahaja v hmNORMAL – pozicija robota je znana, motorji so vključeni in robot se premika v skladu z nastavljeno referenčno hitrostjo.

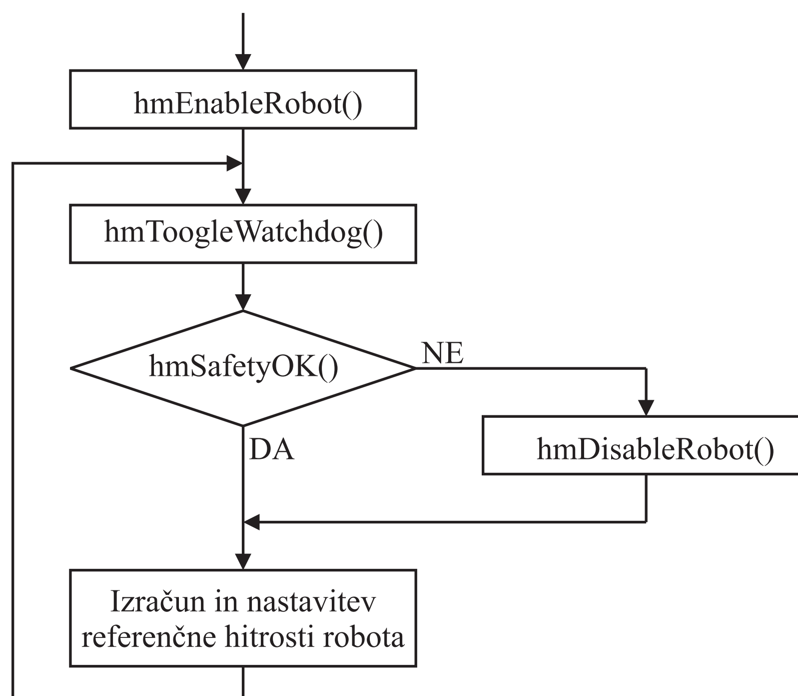
Varnostne omejitve vključujejo hitrost in pozicijo manipulatorja ter izmerjeno silo. V primeru prekoračitve omejitev robota zaustavimo. Kršitev teh omejitev pomeni, da so že prej odpovedali ostali nivoji zaščite (omejevanje pozicije in hitrosti v Actuator_t, omejevanje pozicije, hitrosti in pospeška v Dof1Model_t (glej poglavje 3.2.3), omejevanje sile navideznega okolja (glej poglavje 3.2.4)). Te omejitve so tako samo zadnji nivo zaščite, zato je zaustavitev primeren odziv.

Preverjanje varnostnih omejitev je prikazano na sliki 3.8. Robota (vse motorje – aktuatorje) vklopimo s funkcijo hmEnableRobot(), izklopimo pa s funkcijo hmDisableRobot(). Po vklopu robota moramo redno klicati funkcijo hmToggleWatchdog(), ki obnovi čuvaja časa. Funkcija hmSafetyOK() preverja, ali sta hitrost in pozicija robota ter izmerjena sila v dovoljenih mejah. Funkcija hmToggleWatchdog() obnovi čuvaja časa samo, če hmSafetyOK() ne javi kršitve omejitev. Če hmSafetyOK() javi kršitev omejitev, izklopimo napajanje motorjev (kličemo hmDisableRobot), nehamo obnavljati čuvaja časa, strukturi HapticMaster_t pa nastavimo stanje hmFAIL.

Kadar je na robota priključen merilni mehanizem, lahko kote zasuka po posameznih oseh preberemo s funkcijo hmGetPotmeter3(). Ta prebere vrednosti napetosti na potenciometrih ter jih preračuna v zasuke v radianih.

Pozicijo, hitrost in silo na vrhu robota, izraženo v svetovnem (kartezičnem) koordinatnem sistemu, lahko preberemo s funkcijami hmGetPosition3_WCS(), hmGetVelocity3_WCS in hmGetForce3_WCS. Če končnico WCS ("world coordinate system") zamenjamo z JCS ("joint coordinate system"), dobimo iste vrednosti, izražene v sklepnem koordinatnem sistemu[4]. Referenčno hitrost, s katero naj se giblje vrh robota, nasta-

[4]Izraziti izmerjeno silo senzorja sile v sklepnem koordinatnem sistemu je napačen izraz. Bolj



Slika 3.8: Preverjanje varnostnih omejitev robota

vimo s funkcijama hmSetVelocity3_JCS() in hmSetVelocity3_WCS() – prva uporablja sklepni, druga pa svetovni koordinatni sistem.

Funkcije za preračun pozicije, hitrosti in sile med sklepnim in svetovnim koordinatnim sistemom so zbrane v datotekah Kinematics.h in Kinematics.c. Imena funkcij se začno s predpono hmConvert, sledi ime fizikalne veličine (Position, Velocity ali Force), nato pa še smer pretvorbe (W2JCS – "from world to joint CS", J2WCS – "from joint to world CS"). Funkcija hmConvertPosition3_J2WCS() torej pretvori pozicijo robota iz sklepnega v svetovni koordinatni sistem.

3.2.3 Model prostostne stopnje

Model prostostne stopnje povezuje gibanje navidezne masne kroglice z gibanjem posameznega sklepa. Predstavljen je s strukturo Dof1Model_t, koda pa je v datotekah Dof1Model.h in Dof1Model.c. Imena pripadajočih funkcij imajo predpono dof1, prvi

pravilno bi bilo govoriti o koordinatnem sistemu orodja. Je pa res, da imata oba koordinatna sistema enako usmerjene osi.

parameter pa je kazalec na strukturo tipa Dof1Model_t (Dof1Model_t *pDof).

Celoten robot je predstavljen s tremi modeli prostostne stopnje. Vsak od teh je enodimenzionalno virtualno okolje, ki se ujema z enim sklepom robota. En model prostostne stopnje vsebuje eno enodimenzionalno masno točko. Vrednost mase (oziroma vztrajnostnega momenta za drugo, rotacijsko os) nastavljamo s funkcijo dof1SetInertia(). Za doseg konstantne mase navidezne masne točke celotnega virtualnega okolja sta navidezni masi za oba translacijska sklepa konstantni, vztrajnostni moment za rotacijski sklep pa se spreminja sorazmerno z ročico (razdaljo med osjo rotacije in vrhom robota – navidezno masno točko).

Bistvo haptičnega algoritma je prikazano na sliki 3.9, ustrezne enačbe pa so 3.1 do 3.4. Primer prikazuje delovanje prvega sklepa robota (horizontalni translacijski sklep). Na navidezno masno točko z maso m_{virt} delujeta izmerjena sila človeškega operaterja (F_{meas}) ter navidezna sila virtualnega okolja (F_{virt}). V narisanem primeru operater potiska masno točko v navidezno steno, zato si obe sili nasprotujeta. Rezultanta sile povzroči gibanje masne točke, ki je opisano s pospeškom, hitrostjo in pozicijo (A_{virt}, V_{virt}, P_{virt}). Gibanje navidezne mase točke je referenca za aktuatorje, ki izračunajo izhodno referenčno hitrost za CIC karte (V_{ref}). Dejanski robot ter senzor sile, ki se ga dotika operater, se gibljeta v skladu z referenčno hitrostjo V_{ref}. S tem je sklenjena zanka med resničnim in navideznim haptičnim svetom.

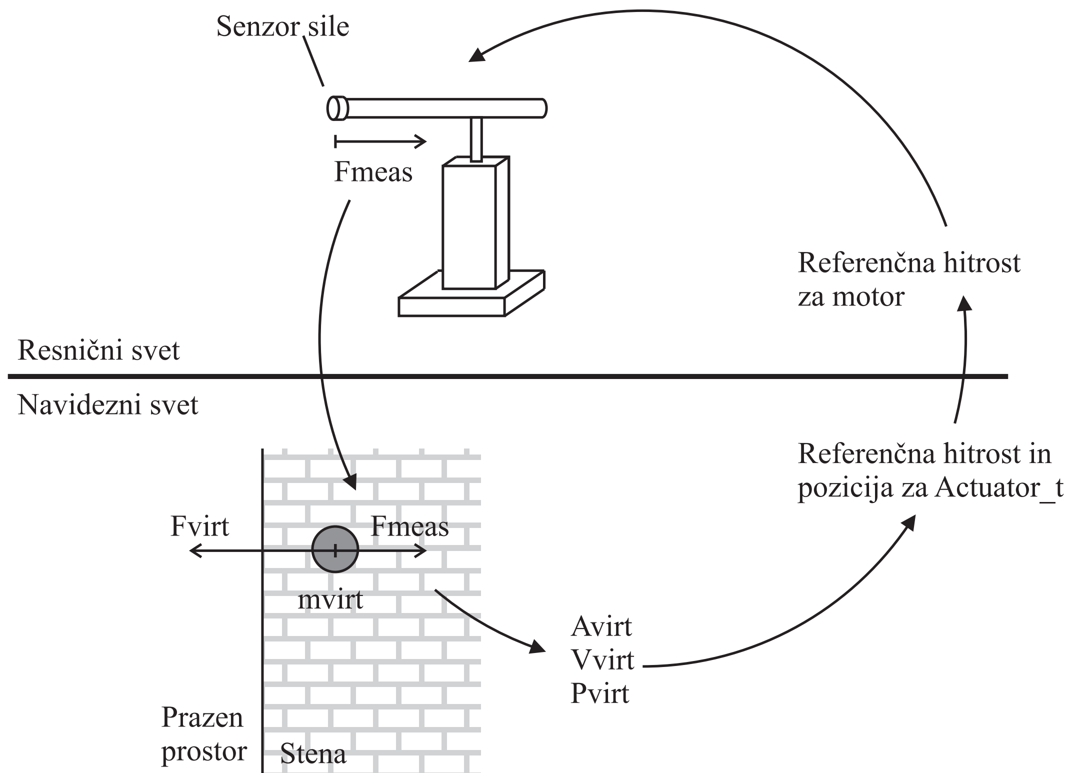
$$A_{virt} = \frac{F_{virt} + F_{meas}}{m_{virt}} \tag{3.1}$$

$$V_{virt} = \int A_{virt}\, dt \tag{3.2}$$

$$P_{virt} = \int V_{virt}\, dt \tag{3.3}$$

$$V_{ref} = f(A_{virt}, V_{virt}, P_{virt}) \tag{3.4}$$

Haptični algoritem izračuna z integracijo pospeška hitrost, z integracijo hitrosti pa pozicijo navidezne masne točke. Model prostostne stopnje omejuje največji pospešek, hitrost in pozicijo navidezne masne točke (glej sliko 3.10). Omejitve nastavimo s funkcijami dof1SetAccLimit(), dof1SetVelLimit() in dof1SetPosLimit(). Integracijo spre-

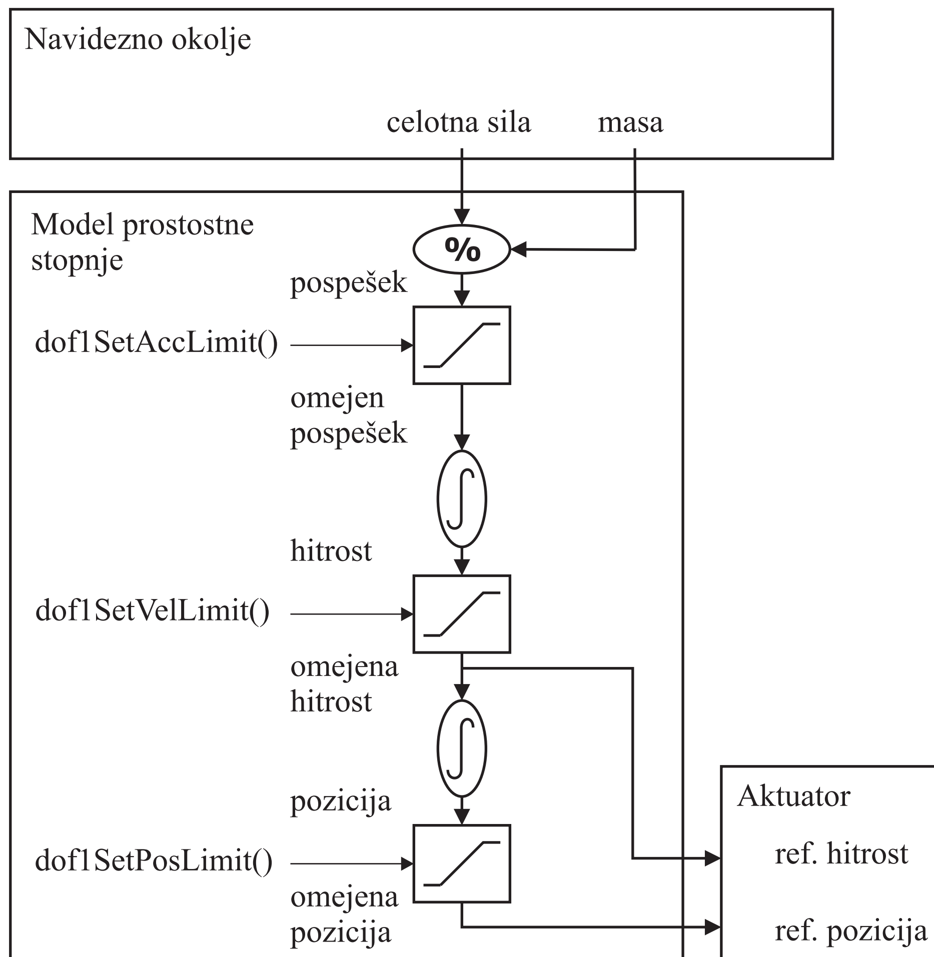


Slika 3.9: Enodimenzionalno haptično okolje za primer prvega sklepa robota

menljivk z upoštevanjem omejitev opravi funkcija dof1IntegrateAndLimit().

Ker se integracija izvaja za vsak sklep posebej, lahko enostavno implementiramo mehko zaustavljanja robota na robovih delovnega prostora. Cilj omejitev je preprečiti navidezni masni točki premik preko določene meje. To dosežemo z zaviranjem s konstantnim pojemkom, če se navidezna masna točka preveč približa meji. Oddaljenost, ko začnemo omejevati gibanje, je poleg pojemka zaviranja (fiksno določen parameter) odvisna še od trenutne hitrosti točke. Pri večji hitrosti moramo začeti omejevati gibanje na večji oddaljenosti, podobno kot se pri večji hitrosti podaljša zavorna pot avtomobila. Potrebno oddaljenost določimo po enačbi 3.5, kjer je v trenutna hitrost, a pojemek zaviranja, l pa oddaljenost, ko je potrebno začeti zavirati. Pojemek zaviranja določimo s funkcijo dof1SetEndStopDeceleration().

$$l = \frac{v^2}{2a} \tag{3.5}$$



Slika 3.10: Iz sile in mase izračunamo pospešek. Z integracijo pospeška dobimo hitrost in pozicijo. Pospešek, hitrost in pozicija so omejeni.

Struktura Dof1Model_t se lahko nahaja v stanjih:

- dof1FAIL,
- dof1OFF,
- dof1NORMAL ali
- dof1VELOCITY_DRIVEN

V stanjih dof1FAIL in dof1OFF navidezna masna točka sledi gibanju dejanskega robota. Tako lahko zamenjamo vloge (tj. da postane masna točka referenca za robota) brez skokovite spremembe hitrosti ali pozicije. V stanju dof1NORMAL se na-

videzna masna točka giblje glede na izmerjeno silo senzorja sile ter navidezno silo haptičnega okolja. Poleg tega je gibanje masne točke referenca za robota. Stanje dof1VELOCITY_DRIVEN je namenjeno implementaciji regulacijskih algoritmov, kjer uporabnik neposredno določa referenčno hitrost za močnostne ojačevalnike, torej za krmiljenje robota na najnižjem nivoju. Pri tem so še vedno aktivne varnostne omejitve za pospešek, hitrost in pozicijo.

3.2.4 Navidezno okolje

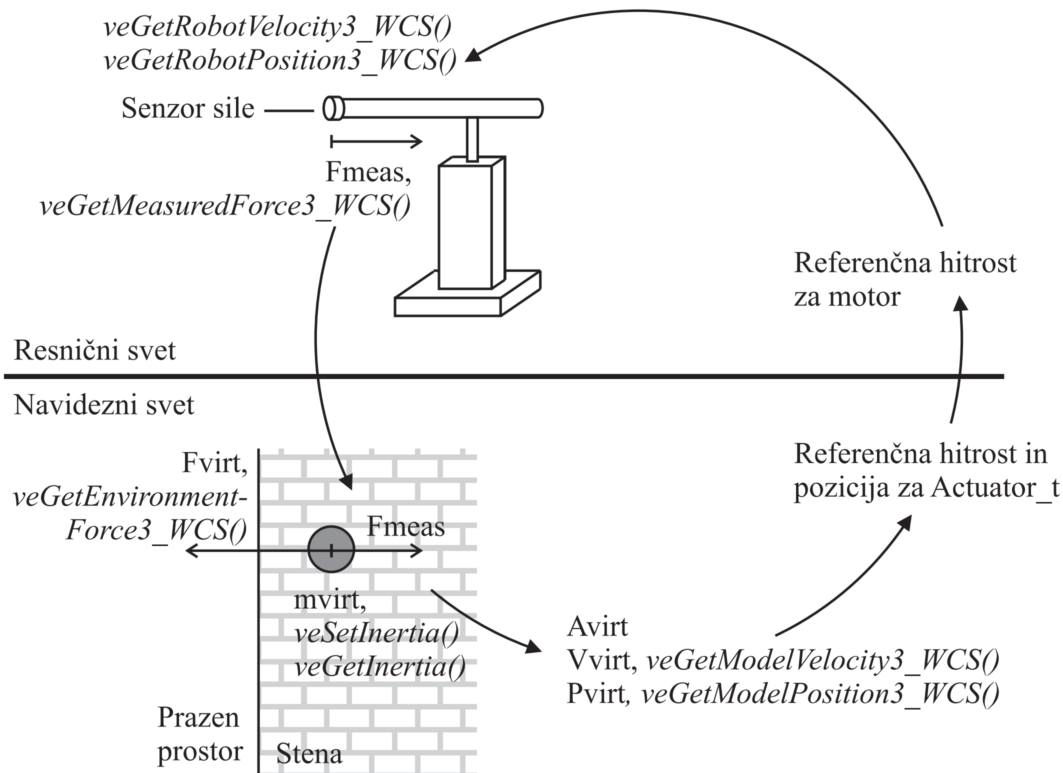
Navidezno okolje je sestavljeno iz navidezne masne točke, ki se giblje v tridimenzionalnem prostoru, in haptičnih objektov. Predstavljeno je s strukturo VirtualEnvironment_t, koda zanjo je v datotekah VirtualEnvironment.h in VirtualEnvironment.c. Funkcije se začno s predpono ve, prvi parameter pa je kazalec na strukturo tipa VirtualEnvironment_t (VirtualEnvironment_t *pVEnv). Struktura vsebuje vrednost navidezne mase, 3 modele prostostne stopnje, robotski manipulator (oz. kazalec na strukturo HapticMaster_t) ter haptične objekte. Struktura VirtualEnvironment_t je funkcionalno enakovredna objektu CFcsHapticMASTER iz FCS-jeve knjižnice HapticAPI[5].

Vrednost navidezne mase lahko nastavimo s funkcijo veSetInertia(), preberemo pa z veGetInertia(). Vrednost mase je podana v kilogramih.

S funkcijo veGetMeasuredForce3_WCS() preberemo izmerjeno silo senzorja sile. Silo, ki jo na navidezno masno točko izvajajo vsi haptični objekti skupaj, izraženo v svetovnem koordinatnem sistemu, preberemo z veGetEnvironmentForce3_WCS(). Pozicijo in hitrost simulirane masne točke dobimo z veGetModelPosition3_WCS() in veGetModelVelocity3_WCS(), pozicijo in hitrost vrha robota pa z veGetRobotPosition3_WCS() in veGetRobotVelocity3_WCS(). Za vse funkcije velja, da z zamenjavo končnice WCS z JCS dobimo veličino, izraženo v sklepnem namesto v svetovnem koordinatnem sistemu. Na sliki 3.11 je prikazana shema haptičnega algoritma in prej omenjene funkcije.

S funkcijami veGetParameter() lahko preberemo X, Y ali Z komponento pozicije ro-

[5]FCS-jev objekt CFcsHapticMASTER poleg robotskega manipulatorja vključuje tudi celotno navidezno okolje. Naš HapticMaster_t ima podobno ime, vendar predstavlja zgolj robotski manipulator.



Slika 3.11: Tridimenzionalno haptično okolje

bota, hitrosti robota, izmerjene sile in orientacije gimbla. Funkcija veGetParameter3() vrne vse tri komponente zahtevane veličine naenkrat kot vektor dolžine 3. Z veSetParameter() lahko nastavimo vrednost mase navidezne masne točke. Te funkcije podvajajo funkcionalnost prej omenjenih, obstajajo pa zaradi združljivosti s FCS-jevo knjižnico HapticAPI (so dostopne preko omrežja s knjižnico ulfeHapticAPI, glej poglavje 3.3).

VirtualEnvironment_t se lahko nahaja v enem izmed naslednjih stanj:

- veUNKNOWN,
- veFAIL,
- veOFF,
- veINITIALIZING,
- veINITIALIZED,
- veNORMAL,

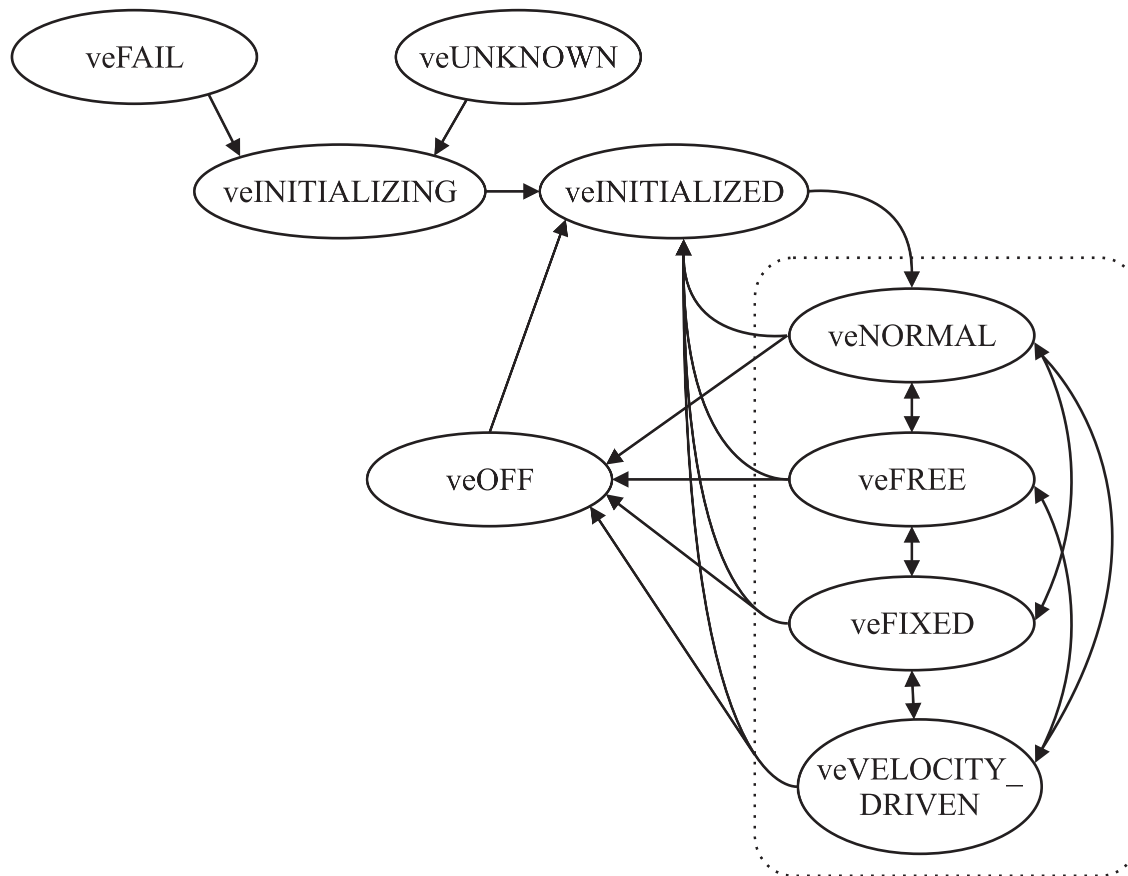
- veFREE,
- veFIXED,
- veVELOCITY_DRIVEN.

Ta stanja so enakovredna stanjem za CFcsHapticMASTER objekt iz FCS-jeve knjižnice HapticAPI. Dovoljeni prehodi med različnimi stanji so prikazani na sliki 3.12. Takoj na začetku se navidezno okolje nahaja v stanju veUNKNOWN[6]. Lega robota ni znana, zato so motorji izključeni. Naslednje stanje je veINITIALIZING, ko vključimo motorje, zato da lahko z nizko hitrostjo (20 mm/s) poiščemo domačo lego robota. Ko je ta znana, ostanemo v njeni bližini - to je stanje veINITIALIZED. Od tu vstopimo v eno izmed štirih stanj, ki so znotraj črtkanega kvadra. To so veNORMAL, veFREE, veFIXED in veVELOCITY_DRIVEN. V teh štirih stanjih je dovoljeno gibanje z normalno delovno hitrostjo. Stanje veNORMAL je namenjeno normalnemu haptičnemu delovanju, ko se navidezna masna točka giblje glede na izmerjeno silo in silo iz navideznega okolja. V veFREE ignoriramo silo navideznega okolja, zato se to obnaša kot prazen prostor. V veFIXED ignoriramo tako izmerjeno silo kot tudi silo navideznega okolja, zato je vrh robota na miru. Stanje veVELOCITY_DRIVEN je namenjeno neposrednemu hitrostnemu krmiljenju robotskega manipulatorja. V veOFF so motorji ugasnjeni, še vedno pa poznamo pozicijo robota. Zato lahko preidemo v veINITIALIZED stanje brez dolgotrajnega iskanja domače lege. Prehod v novo stanje zahtevamo s klicem funkcije veSetStateMachine(). Pred tem lahko s funkcijo veIsNewStateMachineStateAllowed() preverimo, ali je prehod iz trenutnega v novo stanje možen.

Ko se navidezno okolje nahaja v stanju veVELOCITY_DRIVEN, določamo robotskemu manipulatorju hitrostno referenco s funkcijo veSetExternalReferenceVelocity3_JCS(). Hitrost je podana v sklepnem koordinatnem sistemu.

Enkrat na cikel kontrolne zanke moramo klicati funkcijo veUpdateState(). Ta je zadolžena za klicanje UpdateState() funkcij struktur, ki jih navidezno okolje vsebuje. Tako se izračuna sila posameznih haptičnih objektov. Skupno silo vseh haptičnih

[6]Stanje veUNKNOWN obstaja zgolj zaradi združljivosti s FCS-jevo kodo, funkcionalno pa je enakovredno veFAIL.



Slika 3.12: Stanja strukture VirtualEnvironment_t

objektov (tj. celotnega navideznega okolja) preberemo s funkcijo veGetEnvironmentForce3_JCS, to pa nato uporabimo pri izračunu gibanja navidezne masne kroglice. Nazadnje še nastavimo referenčno hitrost robota glede na hitrost navidezne masne kroglice.

Navidezno okolje vsebuje haptične objekte. Ko potrebujemo nov haptični objekt, ga "ustvarimo" (v resnici samo označimo še neuporabljen objekt kot uporabljen) z eno izmed funkcij veCreateHapticBlock(), veCreateHapticSphere() itd. Haptični objekti in njih uporaba so opisani v poglavju 3.2.5

3.2.5 Haptični objekti

Koda za haptične objekte se nahaja v direktoriju lowModule/hapticObject. Vsi objekti strogo posnemajo FCS-jeve haptične objekte. Tako kot v knjižnici HapticAPI je vme-

snik za dostop do objektov tudi tu dostopen preko omrežja z uporabo klicev oddaljenih funkcij ("remote procedure call") in knjižnice ulfeHapticAPI (glej poglavje 3.3).

Na voljo je nespremenljivo število haptičnih objektov posameznega tipa. To število je določeno z makrojem VE_NUM_HAPTIC_OBJECTS (trenutna numerična vrednost je 20). Vsi primerki posameznega tipa so shranjeni v polju v strukturi VirtualEnvironment_t.

Koda se izvaja v Linux jedru, zato lahko uporabljamo samo jezik C. Ta ne pozna pravih objektov, tako kot jih jezik C++. Zato so s stališča programskega jezika C naši haptični objekti samo C strukture (v FCS-jevi HapticAPI pa so pravi C++ objekti). Vendar pa je koncept hierarhije razredov z dedovanjem in virtualnimi funkcijami (dvema glavnima karakteristikama objektno orientiranega programiranja) zelo primeren za implementacijo haptičnih objektov. To je bil zadosten razlog, da smo napisali skripta z imenom "cpp2c".

S pomočjo cpp2c skript definiramo vmesnik objektov (ustreza deklaraciji C++ razreda) v C++ podobnem jeziku. Cpp2c skript nato pretvori vmesnik objekta v vzorčno C kodo – deklaracijo ustrezne C strukture in funkcij za delo s to strukturo. Programer mora nato le še dodati kodo v telo funkcij. Če v vmesnik starševskega objekta dodamo novo člansko spremenljivko ali funkcijo, se to avtomatično odrazi v C kodi za ustrezno C strukturo ter za vse C strukture, ki so izpeljane iz nje. Več o cpp2c skriptih je prikazano v poglavju 4.

3.2.5.1 HapticObject_t

Osnovni haptični objekt je HapticObject_t, iz njega pa so izpeljani ostali haptični objekti – HapticBlock_t, HapticSphere_t, HapticTorus_t, HapticCylinder_t in HapticConstantForce_t. Vmesnik za HapticObject_t je definiran v vzorčni datoteki cw_template_HapticObject.cpp. Generirani datoteki s C kodo sta HapticObject.h in HapticObject.c. Imena funkcij se začno s predpono ho, prvi parameter je kazalec na strukturo tipa HapticObject_t (HapticObject_t *pHObj).

HapticObject_t ima naslednje lastnosti:

- zastavico "omogočen",
- zastavico "v uporabi",
- debelino notranje in zunanje stene,
- trdoto notranje in zunanje stene,
- faktor dušenja notranje in zunanje stene,
- pozicijo,
- orientacijo,
- linearno hitrost,
- kotno hitrost,
- izhodno silo in
- ID številko.

Zastavica "v uporabi" je uporabljena za posnemanje dinamičnega ustvarjanja haptičnih objektov. Trenutno vrednost zastavice preberemo s funkcijo hoIsInUse(), novo vrednost pa vpišemo s hoSetInUse(). Ko zahtevamo nov haptičen objekt, se ustrezna funkcija (npr. veCreateHapticBlock() za objekte tipa HapticBlock_t) sprehodi skozi polje vseh objektov ustreznega tipa, dokler ne najde neuporabljenega primerka. Tega nato označi kot uporabljenega (hoSetInUse(pHObj, TRUE)) ter vrne kazalec nanj. Ko objekta ne potrebujemo več, ga sprostimo s klicem funkcije hoSetInUse(pHObj, FALSE).

Vsak haptičen objekt je lahko omogočen ali onemogočen. Omogočen objekt pomeni, da ga robot haptično izrisuje, tj. da ga lahko otipamo. Funkcija hoEnable() objekt omogoči, hoDisable() pa ga onemogoči. Objekt je lahko omogočen neskončno dolgo ali "duration" milisekund dolgo, pri čemer je "duration" vhodni parameter v funkcijo hoEnable().

Vsak haptični objekt je sestavljen iz dveh sten, notranje in zunanje. Haptično steno poleg debeline opisuje še trdota in faktor dušenja. Haptična stena je osnovni gradnik

haptičnih objektov, in posamezni haptični objekti so sestavljeni iz različno oblikovanih sten.

Postopek izračuna sile, s katero haptični objekt deluje na vrh robota, je prikazan na sliki 3.13, ustrezne enačbe pa so 3.6 do 3.8. Najprej izračunamo, kje se nahaja vrh robota glede na haptični objekt in s kakšno relativno hitrostjo se giblje. To opravi funkcija hoGetLocalCartesianComponents(). Pozicijo vrha robota glede na haptični objekt označimo s P_{robot}, hitrost vrha pa z V_{robot}. Nato preverimo, ali se robot nahaja znotraj haptične stene. Če se, izračunamo globino vdora (razdaljo od vrha robota do roba stene, označeno z d, ter vektor normale stene $\vec{n}$. Velikost sile haptičnega objekta določimo kot prispevek vzmeti (produkt globine vdora d in trdote stene K) ter dušenja (produkt hitrosti $Vrobotn$ in dušenja b). K dušenju prispeva samo komponenta hitrosti v smeri normale stene, zato je smer sile haptičnega objekta enaka smeri normale stene. Tangencialna komponenta hitrosti ne prispeva k sili objekta, zato je drsenje po površini objekta "gladko". Dušenje b je odvisno od faktorja dušenja ζ, trdote vzmeti K in navidezne mase m.

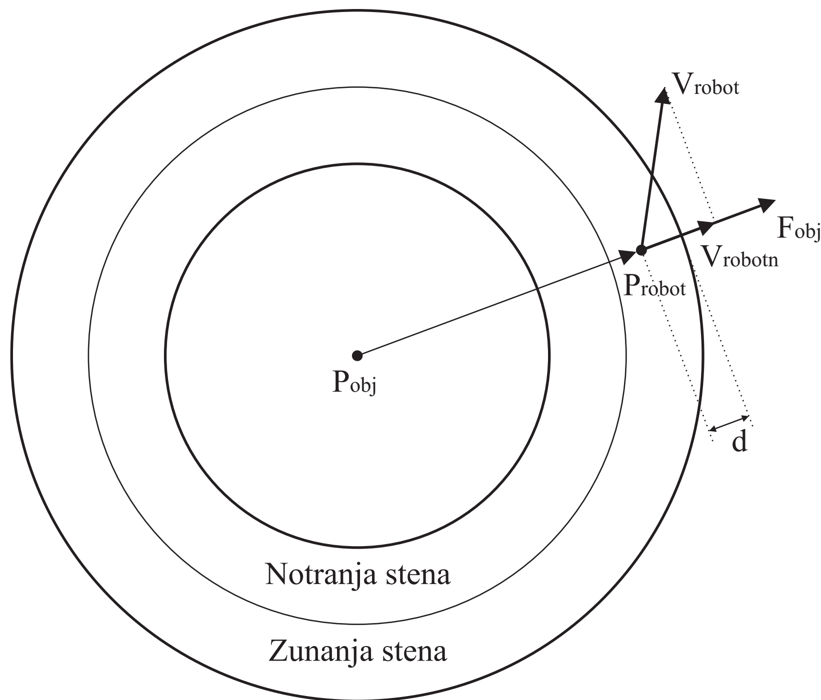
$$b = 2\,\zeta\,\sqrt{k\;m} \tag{3.6}$$

$$Vrobot_n = Vrobot \bullet \vec{n} \tag{3.7}$$

$$Fobj = K\;d + b\;Vrobot_n \tag{3.8}$$

Lastnosti haptičnih objektov beremo s funkcijama hoGetParameter() (za skalarne parametre) in hoGetParameter3() (za tridimenzionalne vektorske parametre). Funkciji za nastavljanje parametrov se imenujeta hoSetParameter() in hoSetParameter3(). HapticObject_t implementira parametre, katerih imena in pomen so našteti v tabeli 3.1.

Haptični objekti imajo tridimenzionalno pozicijo (FCSPRM_POS) in orientacijo (FCSPRM_ORIENTATION). K tema geometrijskima lastnostima sodi še oblika objekta, ki je za različne haptične objekte različna, zato je tudi opisana z različnimi parametri (npr. oblika krogle je opisana z radijem, oblika kvadra pa z dolžinami treh stranic). Parametri za opis oblike so zato deklarirani v izpeljanih razredih.



Slika 3.13: Izračun sile haptičnega objekta

Ime parametra	Pomen	enota	dolžina vektorja
FCSPRM_POS	Pozicija	m	3
FCSPRM_POSX	X koordinata pozicije	m	1
FCSPRM_POSY	Y koordinata pozicije	m	1
FCSPRM_POSZ	Z koordinata pozicije	m	1
FCSPRM_ORIENTATION	Orientacija	rad	3
FCSPRM_VELOCITY	Linearna hitrost	m/s	3
FCSPRM_OMEGA	Kotna hitrost	rad/s	3
FCSPRM_THICKNESSEXT	Debelina zunanje stene	m	1
FCSPRM_THICKNESSINT	Debelina notranje stene	m	1
FCSPRM_SPRINGSTIFFNESSEXT	Trdota zunanje stene	N/m	1
FCSPRM_SPRINGSTIFFNESSINT	Trdota notranje stene	N/m	1
FCSPRM_DAMPINGFACTOREXT	Faktor dušenja zunanje stene		1
FCSPRM_DAMPINGFACTORINT	Faktor dušenja notranje stene		1

Tabela 3.1: Parametri haptičnih objektov

Linearno hitrost (FCSPRM_VELOCITY) in kotno hitrost (FCSPRM_OMEGA) lahko uporabimo za premikanje haptičnih objektov, tj. za implementacijo dinamičnega haptičnega okolja. Premikanje izvajamo v regulacijski zanki s frekvenco 2500 Hz, zato dobimo boljši približek zveznega premikanja, kot če pozicijo in orientacijo posodabljamo preko omrežja.

Preostalih 6 parametrov v tabeli se nanaša na zunanjo (ime parametra se konča z EXT) in notranjo (ime parametra se konča z INT) haptično steno. Določimo lahko debelino stene (FCSPRM_THICKNESS*), trdoto stene (FCSPRM_SPRINGSTIFFNESS*, v enačbi 3.8 označena kot K) in faktor dušenja (FCSPRM_DAMPINGFACTOR*, v enačbi 3.6 označen kot ζ).

Funkcija hoUpdate() se mora izvesti enkrat na cikel kontrolne zanke. Glede na hitrost objekta mora izvesti popravek pozicije in/ali orientacije. Drugo opravilo je izračun sile haptičnega objekta na navidezno masno kroglico. Silo haptičnega objekta preberemo s funkcijo hoGetForce(). Izračunana sila je shranjena, zato večkratni klici hoGetForce() v enem ciklu kontrolne zanke ne zahtevajo ponovitve izračuna.

ID številka je namenjena lažji uporabi haptičnih objektov, ki jih ustvarimo z ulfeHapticAPI (koda se izvaja v uporabniškem prostoru na računalniku za vizualizacijo), v visokem modulu (koda se izvaja v Linux jedru na kontrolnem računalniku). Z ulfeHapticAPI najprej ustvarimo objekt, ga konfiguriramo in mu določimo ID številko. V jedru lahko nato poiščemo ta objekt s funkcijo veGetHapticObjectByID(). Takšna shema izvajanja je smiselna pri implementaciji dinamičnih okolij. Če z ulfeHapticAPI ustvarimo 3 haptične objekte (A, B in C), lahko v visokem modulu premaknemo objekt C tako, da premaknemo tretji objekt v polju vseh objektov. Toda če objekta B več ne potrebujemo, ga ne ustvarimo in objekt C postane drugi po vrsti. Nasprotno pa ID številka ostane enaka. Z uporabo ID številke nam torej ni potrebno poznati vrstnega reda ustvarjanja objektov. Do ID številke dostopamo s funkcijama hoSetID() in hoGetID().

3.2.5.2 HapticBlock_t

Objekt HapticBlock_t predstavlja haptični kvader. Implementiran je v datotekah cw_template_HapticBlock.cpp, HapticBlock.h in HapticBlock.c.

Geometrijsko je opisan z velikostjo (dolžina, širina in višina), ki je shranjena v članski spremenljivki m_v3Size. Ime ustreznega parametra je FSCPRM_SIZE. K funkcijam objekta HapticObject_t dodaja funkcijo hblkSetBaseParameters(), s katero v enem koraku nastavimo pozicijo, orientacijo in velikost kvadra ter debelino, trdoto in faktor dušenja zunanje in notranje stene.

3.2.5.3 HapticSphere_t

Objekt HapticSphere_t predstavlja haptično kroglo. Implementirana je v datotekah cw_template_HapticSphere.cpp, HapticSphere.h in HapticSphere.c.

Geometrijsko je opisana z radijem, ki je shranjen v članski spremenljivki m_dRadius. Ime ustreznega parametra je FSCPRM_RADIUS. K funkcijam objekta HapticObject_t dodaja funkcijo hsphSetBaseParameters(), s katero v enem koraku nastavimo pozicijo in radij krogle ter debelino, trdoto in faktor dušenja zunanje in notranje stene. Od HapticObject_t podeduje tudi lastnost orientacije, vendar ta nima učinka, ker je krogla simetričen objekt.

3.2.5.4 HapticTorus_t

Objekt HapticTorus_t predstavlja haptični obroč (toroid). Implementiran je v datotekah cw_template_HapticTorus.cpp, HapticTorus.h in HapticTorus.c.

Geometrijsko je opisan z notranjim in zunanjim radijem (članski spremenljivki m_dIntRadius in m_dExtRadius). Notranji radij je premer cevi, ki je zvita v krog s premerom zunanjega radija. Ime ustreznih parametrov je FSCPRM_EXTRADIUS in FSCPRM_INTRADIUS. K funkcijam objekta HapticObject_t dodaja funkcijo htorusSetBaseParameters(), s katero v enem koraku nastavimo pozicijo, orientacijo, zunanji in notranji radij obroča ter debelino, trdoto in faktor dušenja zunanje in notranje stene.

3.2.5.5 HapticCylinder_t

Objekt HapticCylinder_t predstavlja haptični valj. Implementiran je v datotekah cw_template_HapticCylinder.cpp, HapticCylinder.h in HapticCylinder.c.

Geometrijsko je opisan z višino in radijem. Poleg tega je lahko odprtina valja zaključena s pokrovom ali pa je odprta. Ustrezne članske spremenljivke so m_dHeight, m_dRadius in m_dLid (m_dLid ima lahko samo dve vrednosti, 1.0 ali 0.0, tj. pokrov je prisoten ali pa ga ni). Ime ustreznih parametrov je FSCPRM_HEIGHT, FSCPRM_RADIUS in FSCPRM_LID. K funkcijam objekta HapticObject_t dodaja funkcijo hcylSetBaseParameters(), s katero v enem koraku nastavimo pozicijo, orientacijo, velikost, višino in premer valja ter debelino, trdoto in faktor dušenja zunanje in notranje stene.

3.2.5.6 HapticConstantForce_t

Objekt HapticConstantForce_t predstavlja konstantno silo. Implementiran je v datotekah cw_template_HapticConstantForce.cpp, HapticConstantForce.h in HapticConstantForce.c.

Edina haptična lastnost je vrednost sile, shranjena v članski spremenljivki m_v3ConstantForce. Ime ustreznega parametra je FSCPRM_FORCE za dostop do vseh treh komponent vektorja, oziroma FSCPRM_FORCEX, FSCPRM_FORCEY in FSCPRM_FORCEZ za dostop do samo ene izmed komponenet. K funkcijam objekta HapticObject_t dodaja funkcijo hconstFSetBaseParameters(), s katero nastavimo želeno vrednost sile.

Objekt lahko uporabimo za implementacijo lastnih haptičnih objektov, npr. tunelov. Tunele lahko matematično opišemo s pomočjo bikubičnih zlepkov. V regulacijski zanki (v visokem modulu) iz pozicije navidezne masne točke določimo silo, s katero tunel deluje na navidezno masno točko. To silo nato samo še vpišemo v objekt tipa HapticConstantForce_t.

3.3 Knjižnica ulfeHapticAPI

Knjižnica ulfeHapticAPI posnema FCS-ovo knjižnico HapticAPI, opisano v [34] in [35]. Z ulfeHapticAPI izvožene funkcije so implementirane v nizkem modulu. To so funkcije za ustvarjanje haptičnih objektov (glej poglavje 3.2.4) in za delo s haptičnimi objekti (glej poglavje 3.2.5). Koda za samo ulfeHapticAPI je v direktoriju k2u_rpc/ulfeHapticAPI.

Knjižnico smo implementirati s pomočjo programa srpcgen (glej Dodatek B). Ta omogoča implementacijo RPC strežnika v Linux jedru (kjer se izvaja koda za haptične objekte), odjemalec pa je lahko v uporabniškem prostoru v Linux ali Windows. Pri tem je Linux naravno okolje odjemalca, za Windows kodo pa so potrebni manjši popravki. Koda za Windows odjemalca v obliki dll knjižnice se nahaja v poddirektoriju msvc_dll, minimalni testni program za to dll knjižnico pa v test_msvc_dll.

Linux knjižnico za odjemalca prevedemo z ukazom "make client", nato pa z "make install_client" namestimo dobljeno binarno datoteko (končnica .so) v direktorij /usr/local/lib, potrebne "header" datoteke (končnica *.h) pa v /usr/local/include. Z ukazoma "make" in "make install" prevedemo in namestimo kodo za strežnik in za odjemalca (to je smiselno samo na krmilnem računalniku robota).

Windows odjemalca prevedemo z Microsoft Visual Studio 6.0. V Windows ni dodatnega koraka za avtomatično namestitev dobljene dll datoteke. Uporabnik jo mora sam prekopirati v direktorij programa, ki dll datoteko potrebuje.

3.3.1 Ročice

Odjemalec kliče s srpcgen izvožene funkcije tako, kot jih bi klicala koda v Linux jedru. Funkcije lahko uporabljajo parametre različnih podatkovnih tipov – osnovne (int, float...), strukture in tudi kazalce. V primeru kazalcev se preko omrežja prenese objekt, na katerega kaže kazalec. Strežnik ta objekt uporabi in ga (morda) tudi spremeni. Nato pošlje odjemalcu spremenjen objekt in odjemalec posodobi izvirni objekt s spremenjeno kopijo.

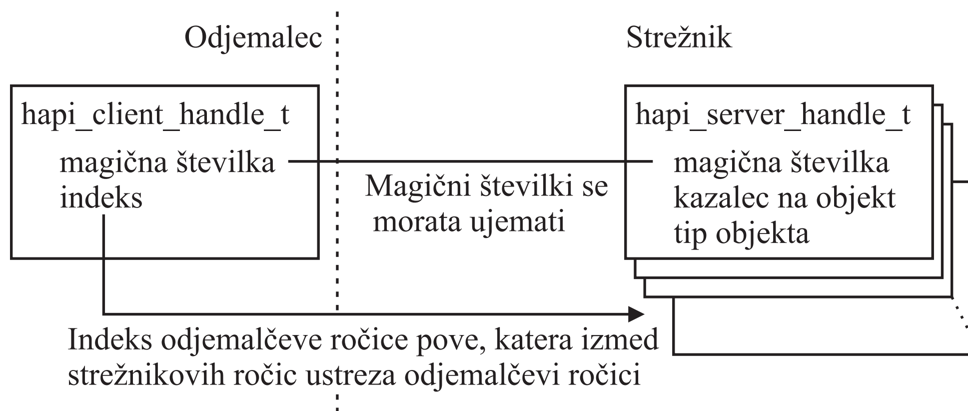
Takšen način dela s kazalci kot parametri je primeren, če se podatki fizično nahajajo na strani odjemalca. V našem primeru pa so podatki (haptični objekti) fizično na strani strežnika. V tem primeru odjemalec ne pošilja strežniku objekta, ki ga strežnik spremeni. Namesto tega mora strežniku samo povedati, kateri objekt naj strežnik spremeni in kako naj ga spremeni.

Najenostavneje bi bilo, če bi odjemalec upošteval, da ima na voljo po 20 haptičnih objektov vsakega tipa in bi nato objekte izbiral s številko med 0 in 19 (kar bi strežnik neposredno uporabil kot indeks v polje objektov). Vendar bi imel v tem primeru strežnik zelo omejen nadzor nad dostopom do haptičnih objektov. Recimo, da hoče odjemalec s funkcijo hoSetBaseParameters() vpisati vse tri komponente velikosti v HapticBlock_t objekt. Pomotoma poda številko, ki ustreza HapticSphere_t objektu. To je očitno narobe. Najhuje pa je, da bo prišlo do poškodbe VFPT kazalca v tistem HapticSphere_t objektu, ki se nahaja za po pomoti podanim HapticSphere_t objektom. Ko bo nizki modul naslednjič uporabil poškodovan objekt, se bo Linux jedro sesulo.

Ročice so standardna rešitev, ki omogočajo nadzorovan dostop do sredstev. Takorekoč vsi operacijski sistemi jih uporabljajo za dostop do datotek, perifernih naprav in podobnih sredstev, kjer moramo slabo napisanim programom preprečiti negativen vpliv na ostale programe in operacijski sistem.

3.3.1.1 Zgradba ročic

V primeru ulfeHapticAPI so ročice dvodelne, kar prikazuje slika 3.14. Odjemalec dobi enostavno ročico, opisano s strukturo hapi_client_handle_t, strežnik pa uporablja nekoliko bolj kompleksen hapi_server_handle_t. Odjemalčeva ročica je sestavljena iz indeksa in magične številke. Strežnikova ročica vsebuje kazalec na haptični objekt, tip haptičnega objekta in magično številko. Koda se nahaja v datotekah ulfeHapticAPI_baseTypes.h (samo deklaracija strukture hapi_client_handle_t) in ulfeHapticAPI_ksvc_kernel_handle.c. Funkcije imajo predpono hapi_handle ("HapticAPI handle").



Slika 3.14: ulfeHapticAPI uporablja dvodelne ročice

3.3.1.2 Uporaba ročic

Strežnik ima vse ročice shranjene v polju. Indeks v odjemalčevi ročici uporabimo kot indeks v to polje, da dobimo ustrezno ročico na strani strežnika. Magično številko uporabimo za preverjanje skladnosti obeh ročic – biti mora enaka v obeh ročicah. Ko odjemalec zahteva nov haptični objekt, ga strežnik poišče (funkcije veCreateHapticBlock() ipd., poglavje 3.2.4). Nato funkcija hapi_handle_alloc() shrani kazalec na objekt v strežnikovo ročico, magično številko nastavi na naključno vrednost[7], tip objekta pa na tip zahtevanega haptičnega objekta (npr. na hht_haptic_block za strukturo HapticBlock_t, predpona htt pomeni "haptic handle type", tj. podatkovni tip, na katerega kaže haptična ročica). Nazadnje funkcija sestavi odjemalčevo ročico iz indeksa stežnikove ročice in magične številke ter vrne odjemalčevo ročico.

Pri uporabi ročic je potrebno iz odjemalčeve ročice določiti kazalec na haptični objekt. Funkcija, ki to opravi, je hapi_handle_h2p() (h2p pomeni "handle to pointer"). Ta preveri ujemanje obeh magičnih številk in ustreznost tipa haptičnega objekta, na katerega kaže strežnikova ročica. Če sta tip in magična številka v redu, vrne kazalec na objekt.

Magična številka v odjemalčevih ročicah je potrebna zato, da odjemalec ne more (namerno ali po nesreči) uganiti kakšne ročice. Brez magične številke bi bila odjemalčeva

[7]Vrednost pravzaprav ni naključna, ampak je vrednost 32-bitnega časovnika, ki šteje s frekvenco procesorja (v našem primeru 850 MHz). Možnost, da bi dvakrat dobili enako magično številko, je zanemarljivo majhna, torej lahko številko obravnavamo kot naključno.

ročica samo indeks v polje strežnikovih ročic. Preverjanje tipa pa prepreči možnost, da bi nad odjemalčevo ročico, ki kaže na objekt enega tipa, klicali funkcije, ki jih smemo uporabiti samo za objekte nekega drugega tipa. V tem primeru bi prišlo do enake poškodbe podatkov kot v prej opisanem (vpis velikosti HapticBlock_t objekta v HapticSphere_t objekt).

3.3.2 Vmesnik za jezik C

Srpcgen lahko izvozi funkcije v jeziku C. Zato smo najprej naredili RPC vmesnik za C. Na strani strežnika je bilo potrebno uvesti dodatno programsko plast, ker mora odjemalec do haptičnih objektov dostopati z ročicami namesto s kazalci. Ta dodatna plast predstavlja večino kode v datoteki srpc_ksvc_ulfeHapticAPI.c.

Poglejmo primer funkcije strežnika, s katero dobimo ročico za celotno virtualno okolje. To ročico potrebujemo, da v virtualnem okolju ustvarimo haptične objekte. Ime funkcije na strani odjemalca je ulfeHapticAPI_GetHapticMaster(), na strani strežnika pa implement_srpc_ksvc_ulfeHapticAPI_ulfeHapticAPI_GetHapticMaster()[8]. Če želimo v Linux jedru (na strani strežnika) dobiti kazalec na VirtualEnvironment_t, pokličemo funkcijo veGetVirtualEnvironment(). Ta ne zahteva nobenega parametra in samo vrne kazalec na edino strukturo VirtualEnvironment_t. Za ulfeHapticAPI pa moramo dobljeni kazalec (skupaj s tipom objekta) še shraniti v strežnikovo ročico in nato vrniti odjemalčevo ročico. Celotna koda je prikazana spodaj.

```
hapi_client_handle_t clnt_hnd;
void* pVe = veGetVirtualEnvironment();
clnt_hnd = hapi_handle_alloc(pVe, hht_virtual_environment);
return clnt_hnd;
```

Nekoliko bolj zapletena je uporaba objektov, na katere kažejo ročice. Ko odjemalec uporabi ročico, se ustrezne funkcije izvajajo v kontekstu niti, ki sprejema podatkovne

[8]Ime funkcije se konča z GetHapticMaster, čeprav se vrnjena ročica nanaša na VirtualEnviroment_t in ne na HapticMaster_t. Potrebno se je spomniti, da je v FCS-jevi kodi objekt CFcsHapticMaster enakovreden VirtualEnvironment_t in da želimo ohraniti združljivost s FCS-jevo HapticAPI. Zato smo prevzeli FCS-jevo poimenovanje.

pakete iz omrežja (podrobnosti o internem delovanju RPC strežnika so v poglavju Dodatek B). Imenujmo to nit kot RPC nit. Izrisovanje haptičnega okolja se izvaja v kontekstu t. i. RT niti ("RT thread"). Kadar RT nit prekine izvajanje RPC niti, lahko pride do nekonsistentnih podatkov. Če RPC nit npr. popravlja vrednost radija haptične krogle (spremenljivka tipa double, velikost 8 zlogov), je možna prekinitev v trenutku, ko 4 zlogi vsebujejo polovico nove vrednosti, preostali 4 pa polovico stare. Rezultat je lahko zelo velika ali zelo majhna, pa tudi neveljavna vrednost.

Zato moramo sinhronizirati delovanje obeh niti. Funkcije (makroji) za sinhronizacijo so v datoteki ulfeHapticAPI_ksvc_util_functions.c. Sinhronizacijo dosežemo z začasnim onemogočenjem prekinitev[9]. Po izvedbi RTLOCK_ACQUIRE so prekinitve onemogočene in RPC nit ima izključujoč dostop do sredstev. V normalno delovanje se vrnemo z RTLOCK_RELEASE.

Primer uporabe sinhronizacije je v funkciji za branje trenutnega stanja strukture VirtualEnvironment_t. Na strani odjemalca je to HapticMaster_GetCurrentState(), na strani strežnika pa implement_srpc_ksvc_ulfeHapticAPI_HapticMaster_GetCurrentState(). Telo strežnikove funkcije je prikazano spodaj. Najprej iz podane odjemalčeve ročice rhHm (rh pomeni "remote handle"[10]) dobimo kazalec na VirtualEnviroment_t. Stanje preberemo s stavkom "*state = veGetStateMachine(pVe);". Ta stavek je zaprt med RTLOCK_ACQUIRE in RTLOCK_RELEASE, zato ima RPC nit izključujoč dostop do strukture *pVe.

```
VirtualEnvironment_t *pVe;
*state = -1;
if( NULL==(pVe = hapi_handle_h2p( rhHm,
                 hht_virtual_environment, hht_virtual_environment )) )
  return -EINVAL;
RTLOCK_ACQUIRE;
{
  *state = veGetStateMachine( pVe );
```

[9]Onemogočenje prekinitev je edini način, da zadržimo izvajanje RT niti. Je pa tudi nevaren, in če pozabimo omogočiti prekinitve, bo sistem "zmrznil".

[10]S stališča strežnika so odjemalčeve ročice na oddaljeni ("remote") strani omrežja.

```
}
RTLOCK_RELEASE;
return 0;
```

Podobno so izvožene še ostale funkcije, ki jih potrebuje odjemalec. To so funkcije za ustvarjanje haptičnih objektov ter za nastavljanje in branje lastnosti haptičnih objektov in samega virtualnega okolja. Dejansko delo opravi ena od funkcij nizkega modula za delo s strukturo VirtualEnvironment_t ali s haptičnimi objekti. Ta funkcija zahteva kot vhodni parameter kazalec na VirtualEnvironment_t oz. na haptični objekt, zato moramo pred klicem funkcije pretvoriti (od odjemalca dobljeno) ročico v kazalec. Poleg tega morajo funkcije za ustvarjanje haptičnih objektov kazalec na ustvarjeni haptični objekt shraniti v ročico, ki jo nato namesto kazalca pošljemo odjemalcu.

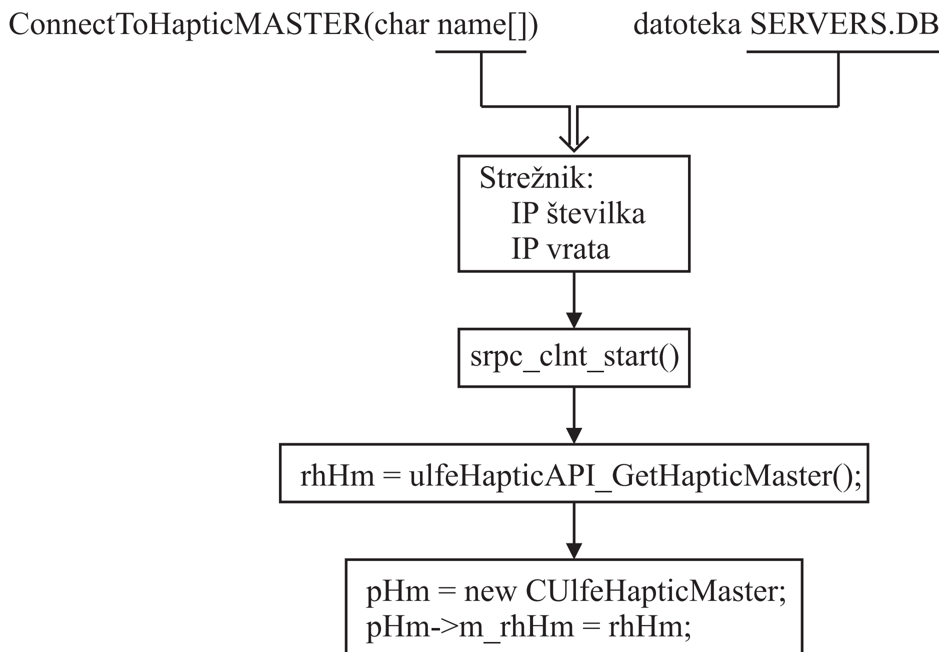
3.3.3 Vmesnik za jezik C++

S prej opisanim vmesnikom za jezik C je na strani odjemalca možno uporabljati vso funkcionalnost haptičnih objektov. Ker želimo združljivost s FCS-jevo HapticAPI, moramo nad vmesnikom za jezik C narediti še vmesnik za jezik C++. S tem se bo tudi poenostavila uporaba knjižnice na strani odjemalca.

C++ vmesnik je implementiran v datoteki ulfeHapticAPI.h. Sestavljajo ga C++ objekti in njihove članske funkcije. C++ objekti vsebuje eno samo člansko spremenljivko. To je ročica, s katero odjemalec dostopa do navideznega okolja ali do haptičnih objektov. Članske funkcije C++ objektov so imenovane enako kot enakovredne članske funkcije C++ objektov iz FSC-jeve HapticAPI.

Navidezno okolje je predstavljeno s C++ objektom CUlfeHapticMaster_t. Ročica do strežnikovega VirtualEnvironment_t je shranjena v člansko spremenljivko m_rhHm. Imamo lahko en primerek tega objekta. Kazalec nanj dobimo s klicem funkcije ConnectToHapticMASTER(), ki je shematično prikazana na sliki 3.15. Tej podamo parameter ime – "name". Ime uporabimo pri iskanju IP številke računalnika, na katerem se izvaja strežnik ulfeHapticAPI. IP številka je zapisana v tekstovni datoteki SERVERS.DB. Vsaka vrstica v SERVERS.DB vsebuje IP številko, številko TCP/IP vrat

in ime. ConnectToHapticMASTER() poišče vrstico, ki ima enako ime, kot je vrednost podanega parametra. Tako ugotovi IP številko in TCP/IP vrata strežnika. Sledi vzpostavitev povezave s strežnikom (klic funkcije srpc_clnt_start(), glej Dodatek B) in pridobitev ročice za navidezno okolje (klic ulfeHapticAPI_GetHapticMaster()). Nato z operatorjem "new" ustvarimo C++ objekt CUlfeHapticMaster, v njegovo člansko spremenljivko m_rhHm shranimo ročico, kazalec nanj pa vrnemo uporabniku.



Slika 3.15: Shematični prikaz funkcije ConnectToHapticMASTER

Z vrnjenim kazalcem lahko uporabljamo članske funkcije C++ objekta CUlfeHapticMaster. Članske funkcije samo kličejo ustrezno funkcijo C vmesnika, kot vidimo v spodnjem primeru funkcije CUlfeHapticMaster::GetCurrentState().

```
inline int GetCurrentState( FCSSTATE& state )
  { return HapticMaster_GetCurrentState( m_rhHm, &state ); }
```

Haptični objekti so predstavljeni s hierarhijo C++ razredov. Osnovni C++ razred je CUlfeHapticObject. Ročica do haptičnega objekta na strani strežnika je shranjena v člansko spremenljivko m_rhObj. CUlfeHapticObject implementira večino članskih funkcij tudi za ostale haptične objekte, ki so izpeljani iz njega. Članske funkcije znova

samo kličejo ustrezno funkcijo C vmesnika, kot vidimo tudi na spodnjem primeru funkcije CUlfeHapticObject::Enable().

```
  virtual int Enable( long duration =0 )
    { return HapticObject_Enable( m_rhObj, duration ); }
```

Ostali haptični objekti so predstavljeni s C++ razredi CUlfeHapticBlock_t, CUlfeHapticSphere_t, CUlfeHapticConstantForce_t, CUlfeHapticTorus_t in CUlfeHapticCylinder_t. Vsi so izpeljani iz CUlfeHapticObject_t. Edina funkcija, ki jo dodajo k CUlfeHapticObject_t, je SetBaseParameters(), ki znova samo kliče ustrezno funkcijo C vmesnika.

Ker se npr. haptični kvader v FCS-jevi knjižnici HapticAPI imenuje CFcsBlock, v naši pa CUlfeHapticBlock_t, je na koncu datoteke ulfeHapticAPI še makro za preslikavo imena CFcsBlock v CUlfeHapticBlock_t. Enako preslikamo tudi imena ostalih haptičnih objektov. Ti makroji poenostavijo prenos obstoječih programov iz HapticAPI v ulfeHapticAPI.

3.3.4 Primera uporabe

3.3.4.1 Minimalno haptično okolje

V direktoriju k2u_rpc/test_ulfeHapticAPI se v datoteki test_ulfeHapticAPI.cpp nahaja program, ki prikazuje uporabo C++ vmesnika v Linux operacijskem sistemu. Koda je kratka, zato je v celoti vključena.

```
#include <stdio.h>
// system wide copy in /usr/local/include
#include "ulfeHapticAPI/ulfeHapticAPI.h"

CUlfeHapticMaster_t * pHm= NULL;
CUlfeHapticBlock_t  *pHBlk=NULL;
CUlfeHapticSphere_t *pHSph=NULL;
CUlfeHapticConstantForce_t *pHConstF=NULL;
```

```
int main(void) {
  pHm = ConnectToHapticMASTER( "vdm" );
  if(!pHm) return -1;

  FCSSTATE state=FCSSTATE_FAIL;
  pHm->GetCurrentState(state);
  if( state != FCSSTATE_NORMAL )
  {
    pHm->SetRequestedState(FCSSTATE_INITIALIZING);
    while( state != FCSSTATE_INITIALIZED )
      pHm->GetCurrentState(state);
    pHm->SetRequestedState(FCSSTATE_NORMAL);
  }
  pHm->DeleteAll();

  TTVec3 pos    = {0.0, 0.0, 0.0};
  TTVec3 orient = {0.0, 0.0, 0.0};
  TTVec3 size   = {0.1, 0.1, 0.1};
  pHBlk = pHm->CreateBlock( pos, orient, size, 5000, 5000, 0.5, 0.5 );
  //pHBlk->Enable( 0 );

  pHSph = pHm->CreateSphere( pos, 0.07, 5000, 5000, 0.5, 0.5 );
  pHSph->Enable( 0 );

  TTVec3 force = {1.5, 0, 0};
  pHConstF = pHm->CreateConstantForce( force );
  //pHConstF->Enable( 0 );

  return 0;
}
```

Najprej vključimo potrebne .h datoteke. Nato deklariramo kazalce na C++ objekte iz knjižnice ulfeHapticAPI. V funkciji main() moramo najprej dobiti kazalec na C++ objekt CUlfeHapticMaster_t. Nato s klici SetRequestedState() zahtevamo FCS-

STATE_NORMAL stanje (FCSSTATE_NORMAL je definiran kot makro, ki se preslika v veNORMAL stanje za VirtualEnvironment_t). Kot zadnji korak inicializacije še pobrišemo vse haptične objekte, ki so jih za seboj pustili drugi programi.

Nato ustvarimo haptični kvader, kroglo in konstantno silo. Če hočemo, da so ti objekti aktivni, moramo še klicati funkcijo Enable(). Za haptični kvader in konstantno silo je le-ta zakomentirana, zato bo robot izrisoval samo kroglo.

3.3.4.2 Prenos obstoječega programa

V direktoriju common/k2u_rpc/fcs-ProgManExamples/Example_01 se nahaja kopija prvega programskega primera za FCS-ovo knjižnico HapticAPI. Ta primer smo izbrali za prikaz prenosa obstoječega programa s FCS-jeve HapticAPI na našo ulfeHapticAPI. Koda programskega primera najprej dobi kazalec na CFcsHapticMASTER objekt, ga inicializira, nato pa še ustvari in inicializira CFcsBlock objekt. Preostanek kode izrisuje grafični prikaz haptičnega okolja s premikajočim se vrhom robota z uporabo OpenGL grafike.

Datoteka s C++ kodo je Example_01.cpp. Originalno in preneseno datoteko smo primerjali s programom diff. Spodaj je prikazan rezultat primerjave. Vrstice iz originalne datoteke se začno z < in vrstice iz prenesene datoteke z >.

```
1,5c1,10
< #include "HapticAPI.h"
< #include "FcsHapticMaster.h"
< #include "FcsBlock.h"
<
< #include <windows.h>
---
> //#include "HapticAPI.h"
> #include "ulfeHapticAPI/ulfeHapticAPI.h" /* ----------- */
> //#include "FcsHapticMaster.h"
> //#include "FcsBlock.h"
>
```

```
> //#include <windows.h>
> #include <unistd.h> /* ----------- */
> #define sleep_ms(a) usleep( (a)*1000 )
> #define Sleep(a) sleep_ms( (a) )
> #define exit(a) _exit( (a) )
314c319,320
<       pHapticMaster->SetForceGetPosition (Force, CurrentPosition);
---
>     //pHapticMaster->SetForceGetPosition (Force, CurrentPosition);
>     pHapticMaster->GetParameter(FCSPRM_POSITION, CurrentPosition);
```

V HapticAPI moramo za vsak tip haptičnega objekta vključiti novo *.h datoteko. V ulfeHapticAPI so vse potrebne *.h datoteke vključene z datoteko ulfeHapticAPI.h.

Datoteka windows.h je specifična za operacijski sistem Windows. Potrebna je zaradi funkcije Sleep(), ki zadrži izvajanje programa za podano število milisekund. V Linux-u je temu namenjena funkcija usleep(), ki zadrži izvajanje programa za podano število mikrosekund. Uporabimo jo v makrojih sleep_ms() in Sleep(), tako da imamo nadomestek za funkcijo Sleep() iz Windows.

Druga Windows specifična funkcija je exit(). Ta je sicer del standardne C knjižnice, vendar ima v Linux-u ime _exit(). Znova je najenostavnejša rešitev uporaba makroja.

V dejanski kodi za delo z robotom moramo popraviti samo vrstico, ki bere pozicijo vrha robota. V FCS-ovi kodi je to funkcija SetForceGetPosition(). Ta lahko poleg branja pozicije izvede še nastavitev sile, s katero bo robot deloval na operaterja[11]. Ker je potrebno samo izvedeti pozicijo robota, jo preberemo s funkcijo GetParameter().

[11] V knjižnici HapticAPI je funkcija SetForceGetPosition() namenjena implementaciji haptičnih okolij, kjer se haptični algoritem izvaja na strani odjemalca. Enakovredno obnašanje bi odjemalec ulfeHapticAPI dosegel z branjem pozicije robota in nastavljanjem sile haptičnega objekta HapticConstantForce_t. Drugače pa je takšnemu delu namenjen visoki modul.

3.4 Visoki modul

Visoki modul je namenjen implementaciji lastnih nizkonivojskih regulacijskih shem, beleženju podatkov, implementaciji dinamičnih haptičnih okolij itd. Skratka funkcionalnost, ki manjka v nizkem modulu in jo želimo napisati, testirati in popravljati brez stalnega čakanja med iskanjem domače lege. Primer kode za visoki modul je datoteka HM_control_main_minimal.c v direktoriju highModule. V HM_control_main_minimal.c so mesta, kamor naj programer doda lastno kodo, označena s komentarjem oblike "/* ADD CODE kratek opis, kaj dodati */".

Bistveni del visokega modula je callback funkcija. V datoteki HM_control_main_minimal.c je tej funkciji ime control_loop(). Funkcija prejme en parameter tipa VirtualEnvironment_t, vrnjena vrednost pa je tipa int. Ob vstavitvi visokega modula mora ta registrirati callback funkcijo pri nizkem modulu. To stori s klicem funkcije hmlowRegisterCallback(). Od tu naprej bo nizki modul periodično klical funkcijo control_loop(). Periodično izvajanje prekinemo, ko visoki modul odstranimo iz Linux jedra.

4. Cpp2c skripta

FCS-jeva knjižnica HapticAPI predstavlja haptično okolje kot nabor tridimenzionalnih objektov z geometrijskimi in s haptičnimi lastnostmi. Na voljo so objekti kvader, krogla, valj ... z lastnostmi, kot so pozicija in orientacija objekta, debelina stene, trdota in faktor dušenja stene itd. Odjemalec dostopa do objektov preko knjižnice HapticAPI, napisane v jeziku C++.

C++ kot objektno orientiran jezik seveda dobro podpira koncept objektov, zato je preprosto definirati osnovni razred (CFcsHapticObject), iz katerega nato izpeljemo ostale razrede (CFcsHapticBlock, CFcsHapticSphere, CFcsHapticCylinder...). Osnovni razred sam po sebi ni uporaben, in primerkov tega razreda niti ne ustvarimo. Nam pa zato ponuja enoten programski vmesnik za dostop do funkcionalnosti izpeljanih razredov. Najpomembnejše so funkcije za nastavljanje geometrijskih in haptičnih parametrov ter za izračun sile, ki jo objekt izvaja na vrh robota. Pri tem je bistveno, da npr. za izračun sile različnih haptičnih objektov vsakič kličemo funkcijo z istim imenom (funkcija Update()), vseeno pa se za vsak razred kliče njegova implementacija funkcije Update(). V C++ so implementaciji takšnega delovanja namenjene virtualne funkcije.

4.1 Implementacija virtualnih funkcij v jeziku C++

Tako idejna zasnova, kot tudi detajlna implementacija virtualnih funkcij je lepo opisana v [36]. Osnovni razred implementira svojo različico funkcije. Če izpeljani razred ne redefinira virtualne funkcije, bo uporabljal implementacijo osnovnega razreda. Če

virtualno funkcijo redefinira, je lahko koda funkcije napisana povsem na novo ali pa vmes kličemo tudi implementacijo osnovnega razreda.

Klici virtualnih funkcij potekajo drugače kot klici navadnih funkcij. Tipičen scenarij uporabe razredov vključuje niz objektov različnih izpeljanih razredov, nad katerimi nato kličemo funkcije. Da si poenostavimo uporabo, kličemo virtualne funkcije preko kazalcev na objekte. Ti kazalci so shranjeni v polje, zato so vsi tipa kazalec na osnovni razred. Prevajalnik ob prevajanju kode v binarni program ne more vedeti, kakšen je natančen tip objekta, na katerega kaže kazalec. Zato tudi ne ve, katero implementacijo virtualne funkcije naj izbere. Odločiti pa se mora med implementacijo osnovnega razreda, implementacijami vseh izpeljanih razredov, implementacijami razredov, izpeljanih iz izpeljanih razredov itn.

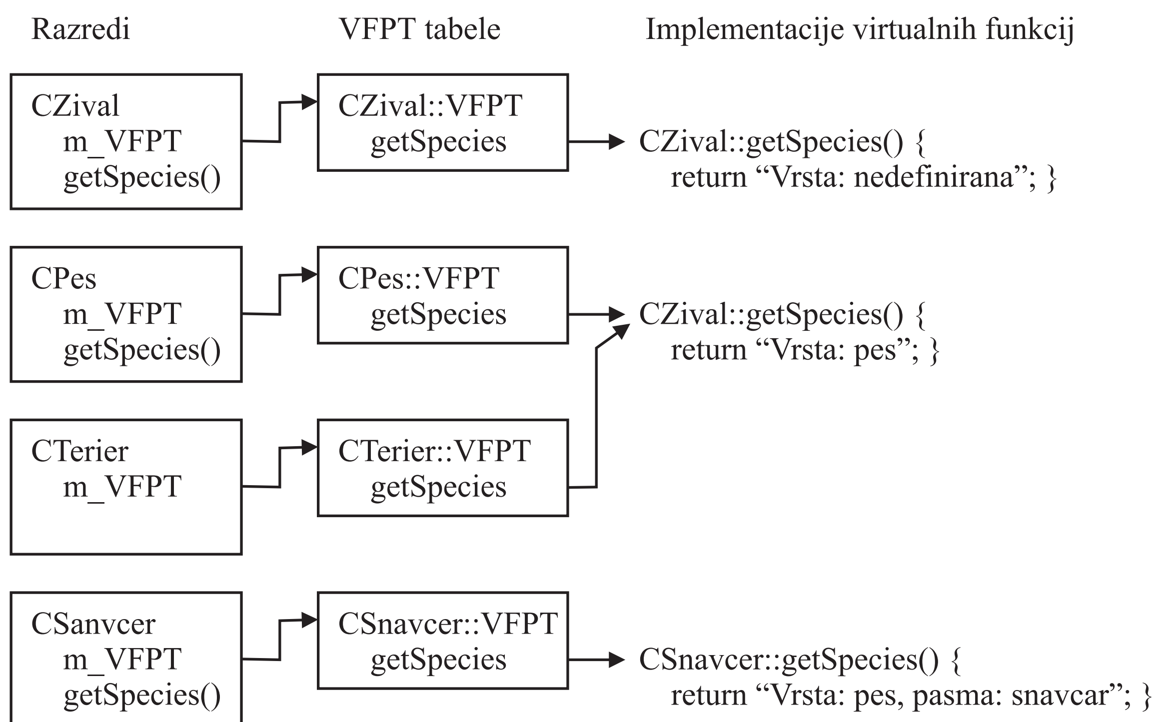
Rešitev je v t. i. poznem povezovanju ("late binding") [36]. Bistvo poznega povezovanja je odlog iskanja ustrezne implementacije virtualne funkcije s faze prevajanja programa na fazo izvajanja programa. Virtualne funkcije se ne kličejo neposredno, temveč preko kazalcev funkcij ("function pointer"), ki so del vsakega objekta. Za izvedbo klica virtualne funkcije mora prevajalnik vedeti samo, kateri kazalec funkcij naj uporabi. Kazalci na virtualne funkcije so shranjeni v tabelo kazalcev virtualnih funkcij ("virtual function pointer table" – VFPT). Vsi objekti, tako tisti od osnovnega razreda kot tudi od izpeljanih razredov, imajo enak vrstni red kazalcev v VFPT tabeli. Zato je določitev ustreznega kazalca enostavna naloga, izvedljiva ob prevajanju programa.

VFPT tabela je lahko velika v primerjavi s preostankom objekta. Za vsako virtualno funkcijo potrebujemo en element. Ker imajo vsi objekti istega razreda enako VFPT tabelo, si jo lahko delijo. Namesto vseh elementov tabele vsak objekt shrani samo kazalec na celotno tabelo. Takšna realizacija je pomnilniško učinkovitejša.

Poglejmo primer, kjer imamo razrede CZival, CPes, CSnavcer in CTerier. CPes je izpeljan iz CZival, CSnacer in CTerier pa iz CPes. Vsi razredi definirajo virtualno funkcijo getSpecies(), s katero preberemo vrsto in pasmo konkretnega osebka. Ta funkcija mora vrniti različen niz za različne razrede, zato mora biti virtualna. V tem primeru smo za CTerier namerno "pozabili" redefinirati funkcijo getSpecies(). CTerier

zato uporablja implementacijo funkcije svojega osnovnega razreda CPes.

Slika 4.1 prikazuje posamezne razrede, njihove VFPT tabele ter implementacije funkcij. CZival, CPes in CSnavcer imajo vsak svojo implementacijo funkcije getSpecies(), zato ustrezen kazalec v VFPT tabeli kaže na razredu lastno implementacijo. CTerier te funkcije ne redefinira. V njegovi VFPT tabeli je vseeno prisoten kazalec za funkcijo getSpecies(). Ta kazalec kaže na implementacijo osnovnega razreda, zato klic funkcije getSpecies() nad objektom tipa CTerier vrne niz “Vrsta: pes”.



Slika 4.1: Primer virtualnih funkcij v C++

Prevajalnik je zadolžen, da vsakemu objektu določi ustrezno VFPT tabelo. To mora storiti, še preden programer kakorkoli uporablja objekt. Primerno mesto je zato konstruktor objekta. Konstruktor je posebna funkcija, ki jo programer uporabi za inicializacijo vrednosti objekta. Prevajalnik vanjo med prevajanjem doda še kodo za nastavitev vrednosti VFPT kazalca na ustrezno VFPT tabelo.

Izpeljani razredi pogosto potrebujejo tudi nove članske spremenljivke in/ali nove virtualne funkcije. Če dodamo nove virtualne funkcije, se VFPT tabela podaljša in naslovi novih virtualnih funkcij se dodajo na konec tabele. Tudi če dodamo nove članske spre-

menljivke, se te dodajo na konec objektov. Tako lahko virtualna funkcija osnovnega razreda deluje nad objekti različnih izpeljanih razredov. Njena koda je nespremenjena (tudi na nivoju zbirne kode, ki jo generira prevajalnik) – uporablja namreč samo začetni del objektov, katerega vsebino definira osnovni razred, preostanek pa ignorira, oziroma se ga niti ne zaveda.

C++ omogoča tudi izpeljavo novega razreda iz več osnovnih razredov. To je t. i. večkratno dedovanje ("multiple inheritance"). Preprosta shema dodajanja kazalcev na nove virtualne funkcije na konec tabele VFPT (in dodajanja novih članskih spremenljivk na konec razreda) v tem primeru več ne zadošča. Recimo, da je razred C izpeljan iz razredov A in B. Če bi bile na začetku objekta tipa C članske spremenljivke osnovnega razreda A, bi virtualne funkcije osnovnega razreda B narobe interpretirala vsebino objekta.

Poleg tega so lahko osnovni razredi tudi sami izpeljani (iz enega ali več osnovnih razredov), in imajo lahko skupne osnovne razrede. To dodatno zaplete prevajalnikovo nalogo implementacije virtualnih funkcij. Poleg pisanja je za programerja težje tudi razumevanje takšne hierarhije objektov. Uporabi večkratnega dedovanja se je zato priporočljivo izogniti ter ga nadomestiti s člansko spremenljivko tipa enega od starševskih razredov [37].

4.2 Zahteve za cpp2c pretvornik

Ker so C++ objekti naraven način za implementacijo haptičnih objektov, jih želimo uporabiti tudi v RTLinux okolju. Vendar pa lahko v RTLinux programih (ki so Linux jedrni moduli) uporabljamo samo C jezik. Zato bomo posnemali C++ objekte in virtualne funkcije v jeziku C. Najprej določimo zahteve za posnemanje C++ objektov – kaj želimo doseči in kako.

Članske spremenljivke razreda lahko namesto v C++ razred shranimo v C strukturo.

Dedovanje članskih spremenljivk realiziramo tako, da na začetek definicije izpeljane strukture prekopiramo definicijo osnovnega razreda. Tako imata obe strukturi is-

toimenske članske spremenljivke na istih mestih. Poleg tega dostopamo do skupnih članskih spremenljivk z enako sintakso za obe strukturi, tako kot pri C++ razredih. Kopiranje definicije članskih spremenljik naj bo avtomatično, tako da se izognemo napakam zaradi nepazljivosti.

Virtualne funkcije realiziramo s pomočjo kazalcev na funkcije. Za vsak tip strukture definiramo svojo tabelo VFPT. Dedovanje virtualnih funkcij realiziramo s kopiranjem definicije tabele VFPT osnovne strukture v tabelo VFPT izpeljane strukture. Kopiranje naj bo avtomatično.

Vsaka struktura ima kot prvi element kazalec na ustrezno tabelo VFPT. Vse strukture istega tipa uporabljajo isto tabelo VFPT.

Želimo avtomatično inicializacijo kazalca na tabelo VFPT. V ta namen bo vsaka struktura imela funkcijo ctor(), ki naj ima enako vlogo kot konstruktor v C++. Koda za nastavitev kazalca na tabelo VFPT v ctor() mora biti generirana avtomatično.

Za primerke struktur, deklariranih v poljubni funkciji, lahko dosežemo avtomatično klicanje funkcije ctor() z definicijo makroja, ki najprej deklarira spremenljivko ustreznega tipa, nato pa nad njo kliče še ustrezno ctor() funkcijo. Žal ne moremo doseči avtomatičnega klicanja funkcije ctor(), kadar strukturo uporabimo kot člansko spremenljivko druge strukture. To se zgodi ravno v primeru haptičnih objektov, kjer VirtualEnvironment_t vsebuje primerke (oz. polja primerkov) posameznih haptičnih objektov.

V C++ kličemo virtualne (in navadne) funkcije objektov kot "objekt.funkcija(parametri)". C ne sprejme takšne sintakse, zato bo prvi parameter funkcij vedno kazalec na objekt, in klic postane "funkcija(&objekt, parametri)".

C++ izvaja pametno pretvorbo kazalcev na objekte pri klicanju funkcij. Imejmo razred A in iz njega izpeljan razred B. Funkcija "void func(A* objA)" sprejema samo kazalce na objekte tipa A. Če pa jo kličemo s kazalcem na objekt tipa B, bo prevajalnik sam ugotovil, da je pretvorba iz B* v A* varna, in kodo pravilno prevedel. Na drugi strani pa bo javil napako, če uporabimo neprimeren podatkovni tip. Preverjanje ustreznosti podatkovnih tipov je pomemben element jezika, ker nas že ob prevajanju programa

opozori na napake.

V C-ju imamo preverjanje tipa parametrov, ne pa tudi pametne pretvorbe tipa kazalcev. Vseeno pa je možno implementirati delno podporo pametni pretvorbi kazalcev. Pri tem nadomestimo preverjanje pravilnosti tipov kazalcev s preverjanjem prisotnosti članskih spremenljivk v obeh podatkovnih tipih.

V C++ imajo funkcije osnovnega in izpeljanega razreda enaka imena, zato jih kličemo z enako sintakso. V C pa mora imeti vsaka funkcija edinstveno ime. Zato bomo pred (kratko) ime funkcije dodali še ime podatkovnega tipa in podčrtaj (_). Tako dobimo dolgo ime funkcije.

Funkcije z dolgim imenom lahko delujejo samo nad enim podatkovnim tipom. Zato kratko ime funkcije definiramo kot makro, ki kliče ustrezno funkcijo z dolgim imenom. Ta makro izvaja tudi pametno pretvorbo tipov, zato ga lahko uporabimo nad različnimi podatkovnimi tipi.

C++ dovoljuje poleg preprostega enojnega dedovanja tudi večkratno dedovanje. Večkratno dedovanje je redko uporabljana lastnost jezika, ki se ji je celo priporočljivo izogniti (mnogokrat je primerneje namesto dedovanja iz razreda X uporabiti novo člansko spremenljivko tipa razreda X – glej [36]). Zato se bomo omejili zgolj na enojno dedovanje.

Poleg virtualnih naj bodo na voljo tudi navadne, nevirtualne funkcije. Pri uporabi se takšne funkcije obnašajo kot navadne C funkcije. Tudi te funkcije se dedujejo. Če izpeljani razred ne redefinira nevirtualne funkcije, bo njegova (avtomatično generirana) implementacija klicala implementacijo starševskega razreda.

4.3 Zasnova cpp2c skript

Skripta za pretvorbo iz C++ v C smo napisali v skriptnem jeziku Codeworker. Datoteke se nahajajo v direktoriju common/cpp2c.

Uporabnik mora najprej v vzorčnih datotekah deklarirati C++ razrede (njihov starševski razred, članske spremenljivke in članske funkcije). Sintaksa sicer ni pov-

sem enaka C++, se pa od nje razlikuje samo v nekaj podrobnostih. Tako na primer ne moremo posnemati C++ ključnih besed "private" in "protected". Z njima omejimo dostop do članskih spremenljivk in funkcij. V C-ju podobnih omejitev ni. Vse članske spremenljivke so povsem dostopne, kar ustreza C++ dostopu "public".

V vzorčni deklaraciji bomo po ključni besedi "public_attribute" našteli članske spremenljivke, po "public_method" pa članske funkcije.

Deklaracija C++ razreda se pretvori v deklaracijo ustrezne strukture – razreda ter v deklaracijo tabele VFPT. Poleg tega generiramo še deklaracije ustreznih (virtualnih) funkcij, makrojev za klicanje virtualnih funkcij ter krnov implementacije virtualnih funkcij.

Deklaracija tabele VFPT vsebuje poleg kazalcev na virtualne funkcije še spremenljivke z imenom "castable_OsnovniRazred" in "castable_const_OsnovniRazred". "OsnovniRazred" je zamenjan z imeni osnovnih razredov (neposrednega in vseh posrednih).

Krni funkcij vsebujejo zaščiteno sekcijo, kamor vpišemo kodo funkcije. Krn funkcije ctor() poleg tega najprej kliče ctor() funkcijo osnovnega razreda ter nastavi VFPT kazalec na VFPT tabelo svojega razreda. Klic funkcije ctor() osnovnega razreda ni v zaščiteni sekciji, zato je programer ne more spreminjati (oziroma so spremembe izbrisane ob naslednjem izvajanju cpp2c skripta).

Kadar redefiniramo funkcijo osnovnega razreda, se v zaščiteno sekcijo funkcije izpeljanega razreda avtomatično vstavi klic na funkcijo osnovnega razreda. To je enako, kot če izpeljani razred funkcije ne redefinira. Programer lahko nato spremeni obnašanje funkcije. Pred klicem funkcije osnovnega razreda lahko spremeni parametre funkcije, po klicu lahko upošteva vrnjeno vrednost ali spremenjene članske spremenljivke, lahko pa klic funkcije osnovnega razreda tudi izpusti.

Za pametno pretvorbo tipov ("type cast") bomo definirali makro CAST. Ta prejme kazalec na spremenljivko ter ime podatkovnega tipa, v katerega želimo pretvoriti kazalec. Če je pB kazalec tipa B*, bo CAST(pB, A) pretvoril pB v kazalec tipa A*. Pravilnost pretvorbe preverimo s pomočjo članskih spremenljivk "castable_*". K nizu "castable_*" prilepimo ime ciljnega tipa (tj. "A"). Če v tabeli VFPT (tisti, ki pri-

pada razredu B) obstaja spremenljivka "castable_A", je razred B izpeljan iz razreda A in je pretvorba dovoljena. V nasprotnem primeru prevajalnik javi napako zaradi uporabe spremenljivke "castable_A" v makru CAST. S tem smo preprečili pretvorbo v nepovezan ("unrelated") podatkovni tip.

Samo pretvorbo tipov izvedemo z eksplicitno pretvorbo podatkovnih tipov preko kazalca void*, tj. "(A*)(void*)pB".

Funkcije s kratkim imenom so nato definirane kot makroji, ki tudi sami uporabljajo makro CAST. Če funkcija s kratkim imenom pričakuje podatkovni tip A*, bo z uporabo makroja CAST(kazalec, A) lahko sprejela tudi kazalce tipa B*. Programer pri tem ne rabi pisati kode za pretvorbo tipa kazalca ob vsakem klicu funkcije.

4.4 Razčlenitev vzorčne datoteke in generiranje kode

Vodilna skripta za pretvorbe vzorčne "C++" datoteke v C datoteko so skripta cpp2c.cws. Ta najprej izvede razčlenitev vhodne datoteke.

Celoten cpp2c razčlenjevalnik je vsebovan v datoteki cpp2c_parser.cwp. V vhodni vzorčni datoteki so dovoljene deklaracije razredov, ki jih želimo pretvoriti v C strukture in ustrezne funkcije, ter uporaba C++ komentarjev.

Deklaracija razreda se začne s "class IME", nato je lahko naštet največ en osnovni razred (" : BASE_CLASS"). Po zavitem oklepaju ("{") sledi označba "public_attribute:" in nato članske spremenljivke razreda.

Vsaka članska spremenljika je deklarirana kot ime podatkovnega tipa, ime spremenljivke ter podpičje (npr. "double RadijKrogle;"). Lahko deklariramo tudi več spremenljivk istega tipa (npr. "double DolzinaX, DolzinaY, DolzinaZ;") ali pa polje spremenljivk (npr. "double Dolzina[3];").

Ko so naštete članske spremenljivke razreda, sledi označba "public_metod:" ter deklaracije članskih funkcij. Deklaracija funkcije sestoji iz podatkovnega tipa vrnjene vrednosti, imena funkcije, oklepaja, deklaracije parametrov, zaklepaja in podpičja. Med zaklepajem in podpičjem je lahko še beseda const, s katero označimo, da funkcija ne

spreminja nobene članske spremenljivke.

Rezultat razčlenitve se shrani v drevesno strukturo. Vsebino drevesne strukture nato uporabimo pri generiranju kode.

Po razčlenitvi vhodne datoteke sledi faza generiranja kode. Če še ne obstajata, ustvarimo za vsak razred datoteki IME.h in IME.c in vanju vpišemo potrebne označene sekcije. Z nizom IME je označeno ime razreda.

Za popravljanje kode v IME.h datoteki so zadolžena skripta cpp2c_expand_h.cwt, za popravljanje datoteke IME.c pa skripta cpp2c_expand_c.cwt. V obeh skriptih uporabljamo nekatere skupne funkcije, te so v skriptih cpp2c_expand.cwt.cws. Koda v skriptih cpp2c_expand_h.cwt v datoteko IME.h vpiše:

- Deklaracijo strukture, ki predstavlja razred IME (struct IME_t).
- Deklaracijo strukture, ki vsebuje VFPT tabele za razred IME (struct IME_t_VFPT_t).
- CTOR_IME makro, ki deklarira in inicializira spremenljivko tipa IME_t.
- Deklaracije funkcij, ki pripadajo razredu IME.
- Implementacije virtualnih funkcij (to so pravzaprav makroji).
- Deklaracije pomožnih funkcij za makro CAST.
- Deklaracije in implementacije tistih funkcij, ki so podedovane od osnovnega razreda in jih trenutni razred (IME) ne redefinira.

Koda v skriptih cpp2c_expand_c.cwt ima naslednje naloge:

- Deklaracijo "tm_Nekaj" makrojev. Z njimi dostopamo do članskih spremenljivk s sintakso tm_Nekaj namesto this->m_Nekaj. Tako je koda krajša, predvsem pa bolj pregledna.
- Implementacijo funkcij, ki so v IME.h samo deklarirane.
- Razveljavitev deklaracije "tm_Nekaj" makrojev.

4.5 Koda osnovne (starševske) strukture

Rezultat generiranja kode si lahko ogledamo na primeru v direktoriju common/cpp2c/zivali. Delovanje cpp2c pretvornika bi lahko razložili tudi na primeru haptičnih objektov, vendar pa so ti že relativno zapleteni. Namesto tega smo raje naredili nov, minimalističen, samostojen primer (program tipa "Hello world"). Ta primer demonstrira samo delovanje in uporabo cpp2c pretvornika.

Osnovnemu razredu je ime Zival_t. Zanj cpp2c najprej ustvari datoteki Zival.c in Zival.h. Datoteka Zival.h se začne z deklaracijo makroja CTOR_Zival_t. Ta makro sprejme ime spremenljivke. Koda, v katero ga razširi C preprocesor, vključuje deklaracijo spremenljivke ter klic funkcije Zival_ctor() nad spremenljivko. Funkcija Zival_ctor() nastavi kazalec VFPT tabele na VFPT tabelo, ki pripada zahtevanemu razredu, ter inicializira članske spremenljivke. To je enakovredno C++ deklaraciji spremenljivke, kjer se "na tiho" kliče privzeti konstruktor objekta.

Naslednji del kode je deklaracija VFPT tabele za strukturo Zival_t. Ta vsebuje kazalce na vse funkcije, ki so v vzorčni datoteki deklarirane kot virtualne. V našem primeru so to funkcije dtor(), getSpecies() in dump(). Nato so deklarirane še pomožne spremenljivke, ki jih uporablja makro CAST. Zival_t nima nobenega osnovnega razreda, zato smemo kazalec na Zival_t pretvoriti samo v kazalec tipa Zival_t* (tj. enako, kot če ga ne pretvorimo). Zato sta deklarirani samo dve pomožni spremenljivki z imenoma castable_Zival_t in castable_const_Zival_t.

Deklaracija same strukture Zival_t se začne s kazalcem na VFPT tabelo, nato sledijo članske spremenljivke strukture. Zival_t ima eno samo spremenljivko m_sName, v katero shranimo ime živali.

Naslednji blok kode so deklaracije funkcij, ki jih razred Zival_t prvič definira, oziroma ki jih razred Zival_t redefinira. Imena vseh se začno z nizom Zival_, s čimer želimo poudariti, da so te funkcije namenjene delu s podatkovnim tipom Zival_t.

Nato sledijo prototipi virtualnih funkcij, ki so bile prvič deklarirane za strukturo Zival_t (npr. virtual_getSpecies()), ter makro, ki omogoča enostavnejše klicanje teh virtualnih

funkcij (npr. getSpecies()).

Po zaključku označene sekcije je še nekaj ročno definiranih makrojev (npr. getName()) za lažje klicanje nevirtualnih funkcij (npr. Zival_getName()).

V Zival.c datoteki je v označeni sekciji najprej vrstica "#define tm_sName (this->m_sName)". Takšen makro je definiran za vsako člansko spremenljivko izbrane strukture (Zival_t ima sicer samo eno). Z uporabo tega makroja je koda članskih funkcij krajša in bolj pregledna. Vsaka članska funkcija namreč kot prvi parameter prejme kazalec objekta, nad katerim opravi operacijo. Ta kazalec ima vedno ime "this", (tako kot v C++). Ker v članski funkciji večinoma uporabljamo članske spremenljivke od *this, se koda skrajša.

Tik pred koncem označene sekcije je makro tm_sName razveljavljen ("undefined"). Tako je doseg makroja omejen na trenutno *.c datoteko. Zato ne more priti do zapletov zaradi večkrat definiranih makrojev, ali pa ker je makro definiran drugače, kot pričakujemo.

V osrednjem delu označene sekcije je najprej definicija globalne spremenljivke tipa Zival_t_VFPT_t z imenom g_Zival_t_VFPT_t. Ta spremenljivka je VFPT tabela za strukturo Zival_t. Deklarirana je kot "const", tako da je ne moremo po nesreči spremeniti. Hkrati z definicijo tudi inicializiramo kazalce funkcij, ki jih vsebuje.

Nato so definicije posameznih (virtualnih in navadnih, nevirtualnih) funkcij. Funkcija Zival_ctor() najprej inicializira kazalec na VFPT tabelo posamezne strukture. V zaščiteni sekciji inicializiramo članske spremenljivke, tako kot bi jih v C++ konstruktorju. Za Zival_t samo vpišemo v spremenljivko m_sName vrednost "noName". Po koncu zaščitene sekcije je še makro RETURN_TDEC_VOID, ki se razširi v navaden "return".

Virtualna funkcija Zival_getSpecies() vrne niz "noSpecies" (makro RETURN_TDEC(x) se razširi v "return x"). Virtualna funkcija Zival_dump() izpiše vsebino strukture na zaslon.

Nevirtualna funkcija Zival_getName() prebere, Zival_setName() pa vpiše ime živali.

4.6 Izpeljana struktura

Struktura Pes_t je izpeljana iz strukture Zival_t (glej datoteko cw_template_Pes.h). Zato podeduje vse članske spremenljivke, virtualne in nevirtualne funkcije, ki smo jih definirali za strukturo Zival_t.

Avtomatično generirana deklaracija strukture Pes_t v datoteki Pes.h se začne z istimi članskimi spremenljivkami kot Zival_t, v istem vrstnem redu. Nove članske spremenljivke so dodane na konec. Enako so za VFPT tabelo Pes_t_VFPT_t najprej kazalci, podedovani od Zival_t, v istem vrstnem redu, šele nato sledijo kazalci novih virtualnih funkcij. V našem primeru sicer nismo dodali nobene nove članske spremenljivke niti nobene nove virtualne funkcije. V cw_template_Pes.h zgolj zahtevamo redefinicijo virtualnih funkcij getSpecies() in dump().

V deklaraciji Pes_t_VFPT_t so zato dodane zgolj članske spremenljivke, ki opisujejo dovoljene pretvorbe kazalcev tipa Pes_t*. To so castable_Zival_t, castable_const_Zival_t, castable_Pes_t in castable_const_Pes_t. Torej lahko Pes_t* z makrojem CAST pretvorimo v Pes_t* (tj. ostane istega tipa) ali v kazalec na razred starša (tj. v Zival_t*).

V Pes.h datoteki so še definicije podedovanih funkcij, ki niso bile redefinirane. Te samo kličejo ustrezno funkcijo osnovnega razreda.

V Pes.c je deklariran in definiran (edini, globalni) primerek strukture Pes_t_VFPT_t. Članska spremenljivka dtor kaže na funkcijo Zival_dtor(), ker Pes_t ne redefinira virtualne funkcije dtor() (zato avtomatično podeduje implementacijo funkcije starševske strukture). Funkciji getSpecies() in dump() sta redefinirani, zato ustrezni članski spremenljivki v Pes_t_VFPT_t vsebujeta naslove funkcij Pes_getSpecies() in Pes_dump() (namesto Zival_getSpecies() in Zival_dump()).

Oglejmo si še implementacije posameznih funkcij. Pes_ctor() najprej kliče Zival_ctor() (enakovredno klicu konstruktorja starševskega razreda v C++). V naslednji vrstici je nastavljena vrednost kazalca na VFPT tabelo na tabelo, ki pripada strukturi Pes_t. Nato sledi še inicializacija članskih spremenljivk ter "return".

Pes_getSpecies() vrne niz, ki opisuje vrsto živali. Pes_dump izpiše na zaslon vse, kar je

shranjeno v posameznem primerku strukture Pes_t. Najprej beseda “Pes”, nato vrsta živali, kot jo vrne virtualna funkcija (makro) getSpecies(), ter nazadnje še ime živali (preberemo vrednost spremenljivke m_sName).

Iz Pes_t sta izpeljani še dve strukturi. To sta Snavcer_t in Terier_t, torej dve različni pasmi iste biološke vrste. Obe redefinirata samo funkcijo getSpecies(). Vrnjen niz je namesto “pes” “pes-snavcer” oz. “pes-terier”, tako da vsebuje poleg imena vrste še ime pasme.

4.7 Delovanje generirane C kode

V datoteki main.c je primer uporabe vseh 4 struktur. Ker je datoteka kratka, je v celoti prikazana spodaj.

```
#include <stdio.h>
#include <stdlib.h>

#include "Zival.h"
#include "Pes.h"
#include "Snavcer.h"
#include "Terier.h"

#define ARR_LEN(a) (sizeof(a)/sizeof(a[0]))

int main()
{
  int ii;
  printf("\n/* ------------------------------------------------ */\n");
  printf("Hello world\n");

  CTOR_Zival_t (ziv1);   CTOR_Zival_t(ziv2);
  CTOR_Pes_t (pes1);     CTOR_Pes_t(pes2);
  CTOR_Snavcer_t (snv1); CTOR_Snavcer_t(snv2);
  CTOR_Terier_t (ter1);  CTOR_Terier_t(ter2);

  Zival_t *pZiv[] = {        &ziv1,        &ziv2,
                      (void*)&pes1, (void*)&pes2,
                      (void*)&snv1, (void*)&snv2,
                      (void*)&ter1, (void*)&ter2 };
  /***************/
```

```
  setName(&snv1, "Crt");
  setName(&ter1, "Ado");

  //const Pes_t c_p1; Pes_t_ctor( (void*)&c_p1 );

  for(ii=0; ii<ARR_LEN(pZiv); ii++)
    dump( pZiv[ii] );

  for(ii=0; ii<ARR_LEN(pZiv); ii++)
    dtor( pZiv[ii] );
  return 0;
}
```

Po vključitvi ustreznih *.h datotek je definiran makro ARR_LEN. Ta vrne število elementov v podanem polju.

Funkcija main() najprej ustvari dva primerka vsake strukture. Uporabljeni so makroji CTOR, ki hkrati deklarirajo in inicializirajo strukturo. "CTOR_Zival_t (ziv1)" tako najprej deklarira spremenljivko tipa Zival_t z imenom ziv1, nato pa jo inicializira s klicem funkcije ctor_Zival_t(&ziv1).

Nato sledi deklaracija polja pZiv, ki vsebuje spremenljivke tipa Zival_t*. Vanj shranimo kazalce na vse prej ustvarjene strukture.

S klicem funkcije setName() nastavimo prvemu snavcerju ime "Crt", prvemu terierju pa "Ado". Preostali psi so brez imena, zato ustrezna spremenljivka ohrani privzeto vrednost "pes_brez_imena".

Najpomembnejši del programa je v for zanki, kjer kličemo virtualno funkcijo (makro) dump() nad vsemi strukturami. Makro "dump(pZiv[ii])" se razširi v klic funkcije virtual_dump (glej datoteko Zival.h, relevantna vrstica je prikazana spodaj). Hkrati se s CAST še preveri ustreznost tipa vhodnega kazalca, ki je nato pretvorjen v tip Zival_t*.

```
#define dump( this ) virtual_dump( CAST(const_Zival_t, this) )
```

Funkcija virtual_dump() kliče nad podanim objektom funkcijo, ki je shranjena v VFPT tabeli, v članski spremenljivki (kazalcu na funkcijo) dump. Ker uporabljajo različne spremenljivke (ziv1, pes1 itd.) različne VFPT tabele, s tem kličemo različne funkcije (Zival_dump(), Pes_dump() itd.).

```
static inline void virtual_dump ( const  Zival_t *this )
{ (this)->m_pVFPT->dump( this ); RETURN_TDEC_VOID;}
```

Delovanje programa sicer najlažje razumemo, če se sprehodimo skozi kodo z razhroščevalnikom. Spodaj pa je izpis testnega programa iz datoteke main.c. Vidimo, da je stavek "dump(pZiv[ii]);" za prva dva kazalca klical implementacijo funkcije dump() za strukturo Zival_t, nato pa sledijo po dva klica funkcije dump() za strukturo Pes_t, Sanvcer_t in Terier_t.

```
[justinc@fokker zivali]$ ./main
/* ----------------------------------------------------- */
Hello world
Zival
Zival
Pes,    pes,    pes_brez_imena
Pes,    pes,    pes_brez_imena
Pes,    pes-snavcer,    Crt
Pes,    pes-snavcer,    pes_brez_imena
Pes,    pes-terier,     Ado
Pes,    pes-terier,     pes_brez_imena
```

5. Zaključek

V uvodu magistrskega delu so predstavljene različne haptične naprave ter primeri haptične programske opreme. Nato sledi opis strojne opreme haptičnega robota HapticMaster ter programske opreme, ki jo k robotu prilaga proizvajalec. Ta se izvaja v VxWorks realno časovnem operacijskem sistemu. Knjižnica za uporabo robota HapticAPI omogoča samo visokonivojsko delo z robotom kot haptično napravo. Implementacija lastnih krmilnih algoritmov, haptičnih algoritmov in uporabo naprave kot klasičnega robota ni možna.

To je bila motivacija za razvoj lastne programske opreme za robota HapticMaster. Izvajanje v doslednem realnem času je zagotovljeno z uporabo operacijskega sistema RTLinux. Uporaba RTLinux-a zahteva programiranje v Linux jedru, kar je ena večjih slabosti. Poleg tega lahko za programiranje v jedru uporabljamo samo jezik C, ne pa tudi C++. Prednost je možnost uporabe vseh programov, ki so na voljo za Linux operacijski sistem, in možnost popolnega nadzora naprave.

Programska oprema je razdeljena v tri dele. To so nizki modul, visoki modul in ulfeHapticAPI. Nizki modul je nujno potreben za delovanje robota. Njegovi osnovni nalogi sta krmiljenje robota in implementacija varnostnih omejitev. Varnostne omejitve vključujejo omejevanje pozicije, hitrosti in pospeška na sprejemljive vrednosti. Če navkljub omejevanju pride do prekoračitve največjih dovoljenih vrednosti, izklopimo motorje robota.

Dodatna naloga nizkega modula je izrisovanje haptičnih objektov. Njihovo število je

omejeno. Haptični objekti so organizirani v hierarhijo objektov in uporabljajo polimorfične funkcije. Jezik C ne podpira objektov, izpeljevanja objektov in polimorfičnih funkcij. Te lastnosti smo zato implementirali s pomočjo pomožnega orodja cpp2c.

Pomožno orodje cpp2c sprejme kot vhod deklaracijo objekta, ki je v C++ podobnem jeziku. Na osnovi deklaracije generira oz. popravlja kodo v izhodnih C-jevskih datotekah. V izhodne C-jevske datoteke mora programer dodati samo jedro funkcij. Cpp2c je realiziran kot skripta za skriptni jezik CodeWorker.

Do haptičnih objektov dostopamo preko omrežne povezave z uporabo knjižnice ulfeHapticAPI. Odjemalec lahko teče na Linux ali Windows operacijskem sistemu. Tako ni nujno programiranje v Linux jedru. Knjižnica je v obliki dinamično povezljive knjižnice (dll knjižnica za Windows, "shared object file" za Linux). Knjižnica za dostop nudi C in C++ vmesnik. C++ vmesnik je zgrajen na osnovi C vmesnika.

Knjižnica uporablja za komunikacijo program srpcgen. Srpcgen ("simple remote procedure call generator") omogoča izvoz C funkcij iz Linux jedra v Linux ali Windows uporabniški prostor. Srpcgen generira iz definicije vmesnika krne kode v jeziku C za strežnik in odjemalca. Generirana koda omogoča odjemalcu klicanje funkcij v Linux jedru z enako sintakso, kot če bi se koda odjemalca izvajala v Linux jedru. Srpcgen je realiziran kot skripa za skriptni jezik CodeWorker.

Visoki modul je namenjen delu z robotom v realnem času. Z njim lahko implementiramo lastne regulacijske algoritme, beležimo podatke ali implementiramo dinamična haptična okolja. Zaženemo ga lahko kadarkoli med delovanjem robota. Pri tem ni potrebno poiskati domače lege robota, ker jo že prej poišče nizki modul. S tem odpravimo nadležno čakanje.

Implementirana programska oprema je s programi, ki uporabljajo originalno knjižnico HapticAPI, združljiva na nivoju izvorne kode. Za prenos na RTLinux so potrebni le manjši popravki (npr. vključitev ene same *.h datoteke namesto večih). Za razliko od originalne programske opreme je možna tudi implementacija dinamičnih haptičnih okolij v doslednem realnem času, kar prepreči zatikanje pri premikanju objektov. Poleg tega lahko implementiramo lastne haptične in krmilne algoritme.

Literatura

[1] Mihelj Matjaž, *Robotsko zaznavanje in umetna inteligenca: haptični sistemi.* Fakulteta za elektrotehniko, Ljubljana, druga izdaja, 2006.

[2] Uroš Mali, *Haptična naprava in navidezno okolje za prst na roki.* Doktorska disertacija, Fakulteta za elektrotehniko, Univerza v Ljubljani, Ljubljana 2006.

[3] Uroš Mali, Nika Goljar, Marko Munih, *Application of Haptic Interface for Finger Exercise.* IEEE Transactions on Neural Systems and Rehabilitation Engineering, zvezek 14, str. 352-360, 2006.

[4] Uroš Mali, *Haptična naprava za vodenje gibanja prsta.* Magisterska naloga, Fakulteta za elektrotehniko, Univerza v Ljubljani, Ljubljana 2003.

[5] Uroš Mali, Marko Munih, *Real-time control of haptic device using personal computer.* Proceedings of IEEE Region 8 EUROCON konference, zvezek 1, str. 401-404, 2003.

[6] EnTech Taiwan, TVicHW32 5.0 - The toolkit for directly accessing hardware from Win32 applications.
URL: http://www.entechtaiwan.com/tvichw32.htm

[7] Aleš Bardorfer, Marko Munih, Anton Zupan and Borut Čeru, *Upper Limb Functional Assessment in Haptic Virtual Environments.* Proceedings of the 2nd Cambridge Workshop on Universal Access and Assistive Technology, Fitzwilliam College, University of Cambridge, Cambridge, United Kingdom, 22.-24. marec 2004.

[8] Aleš Bardorfer, *Haptični vmesnik pri kvantitativnem vrednotenju funkcionalnega stanja gornjih ekstremitet*. Doktorska disertacija, Fakulteta za elektrotehniko, Univerza v Ljubljani, Ljubljana 2003.

[9] D. J. Reinkensmeyer in ostali, *Mechatronic assessment of arm impairment after chronic brain injury*. Technology and and Health Care, Vol. 7, 1999, str. 431-435.

[10] P. C. Shor n ostali, *The effect of robotic-aided therapy on upper extremity joint passive range of motion and pain*, Mounir Makhtari (ur.). Integration of assistive technology in the information age, IOS Press, 2001, str. 79-83.

[11] W. Harwin in ostali, *The GENTLE/S project: a new method of delivering neurorehabilitation*, Č. Marinček (ur.). Integration of assistive technology in the information age, IOS Press, 2001, str. 36-41.

[12] F. Amirabdollahain in ostali, *Error correction movement for machine assisted stroke rehabilitation*, M. Mokhtari (ur.). Integration of assistive technology in the information age, IOS Press, 2001, str. 60-65.

[13] R. Loureir in ostali, *Using haptics technology to deliver motivational therapies in stroke patients*. Concepts and initial pilot studies, EuroHaptics, Birmingham, 2001.

[14] Publikacije na spletni strani projekta Reharob
URL: http://reharob.manuf.bme.hu/publications/

[15] Ponikvar Matija, *Analiza trajektorij gibanja roke v haptičnem okolju*. Doktorska disertacija, Fakulteta za elektrotehniko, Univerza v Ljubljani, Ljubljana 2003.

[16] Ponikvar Matija, *Identification and compliant control of the industrial robot Staubli RX90*. Neobjavljeno tehnično poročilo, 2002.

[17] Helmer P., *3D force-feedback wrist*. MS thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.

[18] Sebastien Grange, Francois Conti, Patrick Helmer, Patrice Rouiller, Charles Baur, *The Delta Haptic Device*. Eurohaptics, Birmingham, England, 2001.

[19] Daniel Thalmann, *Introduction to Virtual Environments*.
URL: http://vrlab.epfl.ch/ thalmann/VR/Intro.pdf

[20] Laparoscopic Impulse Engine, Impulse Engine 2000 Software Development Kit. Release 4.1, Immersion Coorporation, 1999.
URL: www.cs.ubc.ca/labs/spin/publications/related/IESDK.pdf

[21] H. I. Krebs, N. Hogan in ostali, *Robot-aided neuro-rehabilitation*. IEEE Trans. Rehabilitation Engineering, Vol. 6, No. 1, 1998, str. 75-87.

[22] Bruce T. Volpe, Hermano I. Krebs and Neville Hogan, *Is robot-aided sensorimotor training in stroke rehabilitation a realistic option?* Current Opinion in Neurology, zvezek 14, str. 745-752, Lippincott Williams & Wilkins, 2001.

[23] Aisen M. L., Krebs H. I., Hogan N., McDowell F., Volpe B. T., *The effect of robot-assisted therapy and rehabilitative training on motor recovery following stroke*. Archives of Neurology, zvezek 54, str. 443–446, 1997.

[24] Hermano I. Krebs, Mark Ferraro, Stephen P. Buerger, Miranda J. Newbery, Antonio Makiyama, Michael Sandmann, Daniel Lynch, Bruce T. Volpe, and Neville Hogan, *Rehabilitation robotics: pilot trial of a spatial extension for MIT-Manus*. Journal of neuroengineering and rehabilitaion, oktober 2004.

[25] Hurmuzlu Y., Ephanov A., and Stoianovici D., *Effect of a Pneumatically Driven Haptic Interface on the Perceptional Capabilities of Human Operators*. Presence, MIT Press, zvezek 7, številka 3, str. 290-307, 1998.

[26] Richer E., Hurmuzlu Y., *A High Performance Pneumatic Force Actuator System: Part I-Nonlinear Mathematical Model*. ASME Journal of Dynamic Systems Measurement and Control, zvezek 122, številka 3, str. 416-425, 2000.

[27] Richer E., Hurmuzlu Y., *A High Performance Pneumatic Force Actuator System: Part II-Nonlinear Controller Design.* ASME Journal of Dynamic Systems Measurement, and Control, zvezek 122, številka 3, str. 426-434, 2000.

[28] Johm M. Hollerback, Elain C. Cohen, William B. Thompson, Stephen C. Jacobsen, *Rapid virtual prototyping of mechanical assemblies.* Procc. NSF Design and Manufacturing Grantees Conference, pp. 477-478, Albuquerque, 1996.

[29] Ueberle M., Buss M., *Design, control, and evaluation of a new 6 DOF haptic device.* IEEE/RSJ International Conference on Intelligent Robots and System, zvezek 3, str. 2949-2954, 2002.

[30] M. Ueberle, M. Ernst, and M. Buss, *Design of a High Performance 6 DOF Haptic Device.* Proceedings of the 8th Mechatronics Forum International Conference, Enschede, str. 1201-1210, 2002.

[31] Stanczyk B., Buss M., *Development of a Telerobotic System for Exploration of Hazardous environments.* IEEE/RSJ International Conference on Intelligent Robots and System, zvezek 3, str. 2532-2537, 2004.

[32] Ueberle M., Mock N., Buss, M., *VISHARD10, a novel hyper-redundant haptic interface.* Proceedings of 12th International Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (HAPTICS'04), str. 58-65, 2004.

[33] Peer A., Stanczyk B., Buss M., *Haptic telemanipulation with dissimilar kinematics.* Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), str. 3493-3498, 2.-6. avgust 2005.

[34] B. A. Ruiter, *HapticAPI programming manual.* Verzija 1.2, Fokker control systems, 6. avgust 2003.

[35] *HapticAPI reference manual.* Verzija 1.4.0, Fokker control systems, 30. avgust 2004.

[36] Bruce Eckel, Thinking in C++, introduction to standard C++.
URL: http://www.mindviewinc.com/downloads/TICPP-2nd-ed-Vol-one.zip

[37] Bruce Eckel, Thinking in C++, practical programming.
URL: http://www.mindviewinc.com/downloads/TICPP-2nd-ed-Vol-two.zip

Dodatek A
Namestitev

V nadaljevanju je opisan postopek namestitve in zagon opisane programske opreme. Programska oprema se nahaja na priloženem DVD mediju. Vsebina direktorijev je:

red-hat-9

ISO slika instalacije za Red Hat 9 Linux.

rtlinux

Izvorna koda za RTLinux 3.2, izvorna koda ustreznega Linux jedra (2.4.22) in konfiguracijska datoteka za 2.4.22 jedro. Za uspešno prevajanje Real time Linux je potrebno povečati število argumentov programa xargs. Xargs je del paketa findutils, ki je tudi priložen.

codeworker

CodeWorker je uporabljen med prevajanjem programske opreme za HapticMaster.

libnetsocket

Knjižnica libnetsocket za TCP/IP in UDP/IP komunikacijo v Linux in Windows OS.

ulfe-haptic-master

Sama programska oprema za robota HapticMaster v RTLinux operacijskem sistemu.

Velik del ukazov mora izvesti uporabnik root, ker navaden uporabnik nima privilegijev za pisanje v direktorije ali v datoteke.

Najprej namestimo Red Hat 9 Linux. Pri izbiri paketov, ki jih želimo namestiti, moramo izbrati razvojna orodja (gcc in g++ prevajalnik ter ostala C/C++ orodja), gotovo pa hočemo tudi urejevalnik kode (npr. emacs).

Pred namestitvijo RTLinux moramo zamenjati obstoječi xargs program. Namestimo izvorno kodo findutils z ukazom "rpm -i findutils-4.1.7-9.src.rpm". V /usr/src/redhat/SOURCES/findutils-4.1.7 popravimo datoteko /xargs/xargs.c, vrstici 292 in 293. Primerjava originalne in opravljene datoteke s programom diff je spodaj.

```
[root@fokker findutils-4.1.7]# diff xargs/xargs.c.orig xargs/xargs.c
292,293c292,293
<   if (arg_max > 20 * 1024)
<     arg_max = 20 * 1024;
---
>   if (arg_max > 40 * 1024) // modified by justinc, 3.3.2005
>     arg_max = 40 * 1024;
```

Nato prevedemo findutils v skladu z navodili v INSTALL, naredimo kopijo obstoječega xargs (poiščemo z "which xargs") ter zamenjamo obstoječ xargs z novim iz xargs/xargs.

Naslednji korak je namestitev samega RTLinux. Najprej namestimo izvorno kodo primernega Linux jedra (linux-2.4.22.tar.bz2, v direktorij /usr/src/) ter samega RTLinux (rtlinux-3.2-pre3.tar.bz2, tudi v /usr/src/). Nato sledimo navodilom v rtlinux-3.2-pre3/doc/INSTALLATION. Konfiguracijo samega Linux jedra si lahko olajšamo, če začnemo z datoteko na DVD-ju, v direktoriju rtlinux/config-linux-2.4.22-rtlinux-3.2-pre3. To datoteko prekopiramo v /usr/src/linux-2.4.22-rtl3.2-pre3/.config. Ukaz "make menuconfig" bo iz .config prebral staro konfiguracijo, ki jo lahko popravimo.

Naslednji korak je namestitev programa CodeWorker. Datoteko CodeWorker_SRC4_1.zip razširimo v /usr/local/src/. Sam sem uporabljal verzijo 4.1. Ta se na Red Hat 9 ni takoj prevedla, v datoteko CGRuntime.cpp je bilo potrebno v vrstico 46 dodati "# include <stdio.h>". V Makefile je dobro omogočiti uporabo readline

knjižnice, spodaj je popravljen del Makefile datoteke.

```
##----------------------------------------------------------------------
## If you want to take advantage of GNU Readline, uncomment the following
## 2 lines and add the include path to this library
##----------------------------------------------------------------------
LFLAGS  += -lreadline -lcurses
CC      += -DCODEWORKER_GNU_READLINE
# justinc, on RedHat9 is CC ignored (not used for g++)?
CXXFLAGS+= -DCODEWORKER_GNU_READLINE
```

CodeWorker je zdaj preveden. Potrebno je še dodati njegov direktorij v PATH spremenljivko, tako da ga lahko poženemo iz poljubnega direktorija. V /etc/profile in v /root/.bash_profile dodamo "PATH=$PATH:/usr/local/src/CodeWorker4_1". Vrstico dodamo pred "export PATH ...". Zapremo in odpremo novo školjko/terminalsko okno (ali pa izvedemo ". /etc/profile"), da se nova nastavitev upošteva.

Namestiti moramo še knjižnico libnetsocket. Datoteko libnetsocket-0.3.1.tgz razširimo v /usr/local/src in sledimo navodilom v INSTALL datoteki.

Zdaj so izpolnjeni vsi predpogoji za uporabo programske opreme za robota. Datoteko ulfe-haptic-master-0.3.tar.bz2 razširimo v poljuben direktorij, najenostavneje kar v $HOME. Ustvarjen bo nov poddirektorij z imenom fokker z izvorno kodo. **Direktorij, kamor namestimo izvorno kodo, naj bo na ext2/ext3 datotečnem sistemu. Razširitev moramo narediti v Linux, s programom tar. Tako ohranimo t. i. "soft link" povezave, kar omogoča kasnejše popravljanje Makefile ipd.**

V direktoriju fokker prevedemo kodo (kot navaden uporabnik) z "make", nato jo (kot root) namestimo z "make install". Za zagon programa moramo najprej zagnati RTLinux ("rtlinux start"). Nizki modul ter ulfeHapticAPI strežnik zaženemo z "ulfeHapticAPI.sh start".

V poddirektoriju k2u_rpc/test_ulfeHapticAPI/ se nahaja testni program. Datoteko SERVERS.DB moramo popraviti, tako da vsebuje IP številko našega računalnika

(192.168.65.245). Program prevedemo z make, zaženemo pa z “./test_ulfeHapticAPI”. Po zagonu program ustvari 3 haptične objekte (kroglo, kvader in konstantno silo). Omogočena je samo krogla, za preostala objekta je klic funkcije Enable() zakomentiran. Zato lahko otipamo samo kroglo.

S knjižnico ulfeHapticAPI lahko do robota dostopamo tudi preko omrežja. Knjižnico prevedemo (kot navaden uporabnik) z “make client”, namestimo pa (kot root) z “make client_install”, oboje v direktoriju fokker/k2u_rpc/ulfeHapticAPI. Nato znova preizkusimo k2u_rpc/test_ulfeHapticAPI/.

Knjižnico ulfeHapticAPI lahko prevedemo tudi v Windows. Najprej moramo prevesti knjižnico libnetsocket, nato pa še nastaviti Visual Studio 6.0, kar je opisano v Dodatek B. Projektna datoteka knjižnice za MS Visual Studio 6.0 je k2u_rpc/ulfeHapticAPI/msvc_dll/ulfeHapticAPI.dsp. Ko prevedemo kodo, dobimo ulfeHapticAPI.dll. To nato uporabimo v testnem programu v k2u_rpc/ulfeHapticAPI/test_msvc_dll/ direktoriju. Ta testni program samo vzpostavi povezavo z robotom in prebere njegovo pozicijo.

Dodatek B
Srpcgen

V nadaljevanju je vsebina podsklopa naloge z naslovom Srpcgen, preprost RPC generator. Naloga opisuje zgradbo in delovanje orodja srpcgen, ki je bilo uporabljeno za implementacijo knjižnice ulfeHapticAPI. V nalogi so vključeni tudi primeri uporabe.

Izjava

Izjavljam, da sem magistrsko nalogo izdelal samostojno pod vodstvom mentorja prof. dr. Marka Muniha. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Ljubljana, 28.06.2007 Justin Činkelj

Univerza v Ljubljani

Fakulteta za elektrotehniko

Justin Činkelj

Srpcgen, preprost RPC generator

Dodatek B

Kazalo

1. Uvod **109**

2. Obstoječi RPC sistemi **113**

2.1 kORBit 113

2.2 SunRPC 114

2.2.1 Uporaba v Linux jedru 114

2.2.2 Uporaba v Linux uporabniškem prostoru 115

3. Zasnova srpcgen-a **117**

3.1 Zahteve za implementacijo 118

3.1.1 Česa ne bomo implementirali 121

3.2 Podajanje parametrov po vrednosti in po referenci 122

3.2.1 Lokalni klici funkcij 122

3.2.1.1 Podajanje parametrov po vrednosti 122

3.2.1.2 Podajanje parametrov po referenci – kazalci 123

3.2.1.3 Podajanje parametrov po referenci – polja 124

3.2.2 Oddaljeni klici funkcij 125

3.3 Shema delovanja generiranih RPC programov 125

3.3.1 Vsebina RPC paketa 125

3.3.2 Izvedba RPC klica 126

3.4 Jezik za opis vmesnika 127

3.4.1 Kazalci, polja in strukture kot parametri 129

3.5 Potrebna programska oprema 130

3.5.1 Red Hat Linux 9.0 130

3.5.2 Microsoft Windows 2000 130

3.5.3 Codeworker 131

3.5.4 Eclipse 131

3.5.5 Libnetsocket 132

4. Implementacija srpcgen-a **133**

4.1 Razčlenitev definicije vmesnika 134

4.2 Vzorčne datoteke 135

4.3 Razširjanje ciljnih C datotek 137

5. Primeri uporabe **139**

5.1 Definicija vmesnika 139

5.2 Pretvorba definicije vmesnika v krne 140

5.3 Implementacija strežnika 141

5.4 Zagon strežnika 143

5.5 Implementacija odjemalca 145

5.6 Dodajanje novih funkcij 146

5.7 Uporaba novih podatkovnih tipov, kazalcev in polj 148

5.8 Prenos odjemalca v Windows OS 150

5.9 Zasnova knjižnice ulfeHapticAPI 153

5.9.1 Zasnova haptičnih objektov na strani strežnika 153

5.9.2 Izvoz haptičnih objektov k odjemalcu 154

Literatura za dodatek B **157**

1. Uvod

Namen tega dela je opisati delovanje preprostega RPC ("remote procedure call") generatorja, razvitega za lažjo uporabo programske opreme haptičnega robota HapticMASTER.

Originalna programska oprema robota HapticMASTER je sestavljena iz dveh delov. Vodenje v realnem času se izvaja na industrijskem računalniku pod operacijskim sistemom VxWorks (kontrolni računalnik). Celotni krmilni algoritem je skupaj z operacijskim sistemom vsebovan v eni datoteki. Uporabnik ne more vzporedno izvajati lastnih programov (logiranje podatkov, vizualizacija).

Uporabnikom je omogočen samo dostop preko omrežja. Kontrolni računalnik ima pri tem vlogo strežnika. Odjemalec je poljuben računalnik, na katerega namestimo ustrezno programsko opremo. Administratorska aplikacija (FCSExplorer.exe) je uporabna predvsem za nastavljanje nekaterih ključnih parametrov sistema (npr. IP številke kontrolnega računalnika). Za programiranje haptičnih okolij je na voljo knjižnica hapticAPI. Ta omogoča ustvarjanje haptičnih objektov ter spreminjanje njihovih lastnosti (pozicija, velikost, trdota, dušenje ...).

S pomočjo hapticAPI lahko programiramo odjemalce za MS Windows in za Linux operacijski sistem. Programski vmesnik (API – "application programming interface") je definiran v jeziku C++, kar omogoča dobro prenosljivost in enostavno uporabo. Komunikacija med strežnikom in odjemalcem je realizirana s pomočjo klicev oddaljenih funkcij. Po FCS-jevi dokumentaciji traja vsak RPC klic približno 0.6 ms.

HapticAPI lahko uporabimo tudi za beleženje podatkov ("data logging") ali za implementacijo dinamičnih haptičnih okolij, vendar pa moramo upoštevati:

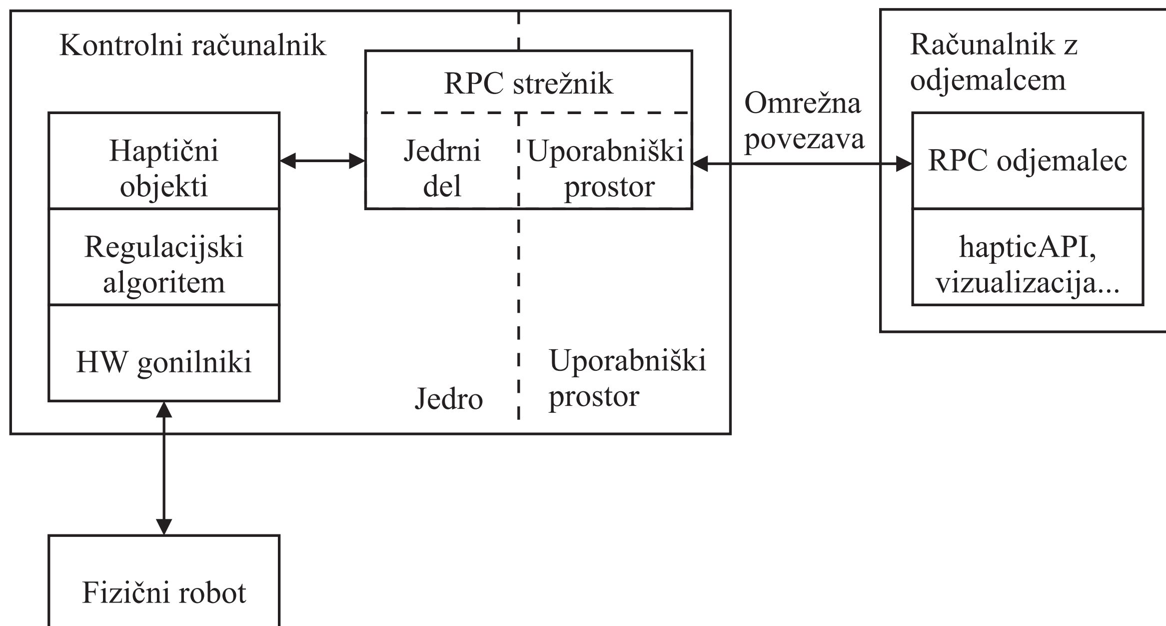
- Hitrost izvajanja RPC klicev je dokaj nepredvidljiva zaradi nepredvidljivosti

mrežne komunikacije.

- Večje število RPC klicev pomeni nižjo frekvenco izvajanja.

Oba faktorja vplivata tako na frekvenco beleženja podatkov kot tudi na kvaliteto dinamičnega okolja. Za vsak primer posebej se moramo odločiti, ali je ta vpliv še znotraj sprejemljivih mej. Druga slabost obstoječega sistema pa je nezmožnost implementiranja lastnih regulacijskih algoritmov. To nam onemogoča implementacijo in testiranje drugačnih haptičnih algoritmov in uporabo robota kot študijskega pripomočka za študente, ki obravnavajo splošne robotske regulacijske sheme.

Zato smo se odločili za realizacijo lastnega krmilnika robota (samo programski del, strojna oprema ostane nespremenjena) na osnovi RTLinux-a [1]. Pri tem smo na eni strani morali velik del programske opreme napisati povsem na novo, na drugi strani pa smo se trudili ohraniti čim boljšo združljivost z obstoječim VxWorks sistemom. Pri tem ne zahtevamo binarne združljivosti, temveč združljivost na nivoju izvorne kode. Edini vmesnik do VxWorks sistema je knjižnica hapticAPI, zato bomo morali implementirati čim bolj podoben vmesnik do RTLinux sistema.



Slika 1.1: Struktura RTLinux sistema

Struktura RTLinux sistema je prikazana na sliki 1.1. Glavni del so gonilniki strojne opreme, haptični regulacijski algoritem in haptični objekti. Vsebovani so v jedrnem gonilniku ("kernel module"). Haptično okolje je sestavljeno iz haptičnih objektov, ki

implementirajo večino funkcij, vključenih v FCS-jev hapticAPI. Te funkcije moramo izvoziti iz Linux jedra preko lokalne mreže do odjemalca in dodati C++ ovitek. Koda odjemalca mora biti uporabna v uporabniškem prostoru v Windows ali Linux OS.

Različni RPC sistemi obstajajo tako za Linux (npr. SunRPC, implementacije protokola CORBA) kot tudi za Windows OS (npr. COM, COM+, DCOM). Vendar pa je večina namenjena komunikaciji s strežnikom v uporabniškem prostoru, mi pa potrebujemo strežnik v jedru. Druga težava je zahteva po prenosljivosti odjemalca v Linux ali v Windows OS. Večina rešitev je izvedenih samo za izbran OS.

Zaradi teh težav smo se odločili za realizacijo lastnega orodja za pomoč pri pisanju RPC programov. Pri tem smo se zgledovali po SunRPC in programu *rpcgen*. Končni program smo poimenovali *srpcgen* ("simple rpcgen"). Ideja uporabe je naslednja:

- Definiramo vmesnik, tj. naštejemo funkcije, ki jih želimo izvoziti.
- *Srpcgen* generira osnovno kodo strežnika in odjemalca (vzpostavitev mrežne povezave ...).
- *Srpcgen* generira krne RPC funkcij za strežnik in za odjemalca.
- Programer ročno popravi krne kode strežnika (samo tiste dele, ki implementirajo dejansko funkcionalnost strežnika).
- Programer uporabi izvožene funkcije v odjemalcu.

Pri tem je poudarek na:

- Enostavni realizaciji minimalnega strežnika in odjemalca.
- Enostavnosti dodajanja novih funkcij. Nove funkcije vpišemo v datoteko z opisom vmesnika, *srpcgen* pa generira krne. Programer doda telo funkcij v kodo strežnika, nato pa lahko funkcije uporablja v odjemalcu.
- Obnašanje RPC funkcij naj bo čim bolj podobno obnašanju enakovrednih lokalnih funkcij.

Sam *srpcgen* je realiziran kot zbirka skript za program *codeworker* [2]. *Codeworker* je odprtokoden program, prosto dostopen na medmrežju. Poleg izvorne kode samega programa je na voljo tudi zajeten priročnik s primeri uporabe in vtič ("plugin") za *elicpse* [3], tako da lahko med pisanjem skript uporabljamo barvno označevanje kode.

Za pošiljanje podatkov preko omrežja smo uporabili knjižnico libnetsocket. Ta omogoča enostavno pošiljanje podatkov med strežnikom in odjemalcem na Linux ali na Windows OS, preko TCP/IP ali UDP/IP protokola. Poleg izvorne kode knjižnice so še primeri uporabe – npr. minimalen “echo” strežnik.

2.
Obstoječi RPC sistemi

Najprej smo nameravali uporabiti kakšnega izmed obstoječih RPC sistemov. Na Windows OS so široko uporabljani Microsoftovi protokoli (COM, DCOM, .NET platforma). Ti protokoli niso pravi RPC sistemi, ker omogočajo izvoz objektov in funkcij. Drugače povedano, podpirajo objektno orientirano programiranje, medtem ko RPC omogoča samo izvoz funkcij – proceduralno programiranje. Teh protokolov ne moremo uporabiti, ker so implementacije specifične za Windows. Generirana koda tako za strežnik kot tudi za odjemalca bi potrebovala veliko ročnega popravljanja, da bi jo lahko prevedli na Linux-u. Poleg tega bi še vedno morali sami implementirati pošiljanje in sprejemanje podatkov preko omrežja.

V nadaljevanju smo si zato raje ogledali RPC sisteme v Linux-u, kjer je praviloma na voljo tudi izvorna koda. Iskali smo sistem, kjer se strežnik ali izvaja v Linux jedru ali pa je prenos strežnika v Linux jedro relativno enostaven. Za odjemalca pa zahtevamo možnost izvajanja v Linux in Windows uporabniškem prostoru (oz. enostaven prenos v Windows OS).

2.1 kORBit

Našo pozornost je najprej pritegnil kORBit [4]. To je implementacija CORBA protokola za Linux, kjer se strežnik izvaja v jedru. Podobno kot DCOM je tudi CORBA objektno orientiran protokol. Originalni ORBit (oz. ORBit2) je uporabljen v GNOME namizju za Linux [5], tako da lahko implementacijo obravnavamo kot kvalitetno napisano. Strežnik na osnovi ORBit2 lahko uporabljamo iz številnih jezikov – poleg popolne podpore C, C++ in Phyton-a je delno podprtih še več drugih jezikov. Jedro je napisano v C-ju in teče pod Linux, Unix in Windows OS.

kORBit nam torej ponuja možnost implementacije RPC strežnika v jedru Linuxa. Vendar pa je celoten projekt uspel zgolj kot "proof of concept", nadaljnji razvoj pa je opuščen [7]. Pri uporabi kORBit-a bi lahko prišlo do težav (nepopoln prenos ORBit-a v jedro, hrošči ali pa zgolj naše nepoznavanje sistema kORBit in CORBA). V tem primeru pač ne moremo pričakovati pomoči avtorjev, tako da bi se morali sami ukvarjati z detajli implementacije kORBit-a. To bi hitro postalo težavno in časovno potratno opravilo, zato tega programa nismo hoteli uporabiti.

2.2 SunRPC

2.2.1 Uporaba v Linux jedru

SunRPC je razvilo podjetje Sun Microsystems za potrebe njihovega porazdeljenega datotečnega sistema (NFS – "network file system"). Ko je NFS postal popularen, se je skupaj s SunRPC razširil na številne platforme, med drugim na Linux in Windows.

Vmesnik do SunRPC programov definiramo v RPC jeziku, ki je razširitev Sunovega XDR ("external data representation") standarda [6]. Definirati vmesnik pomeni predvsem našteti vse funkcije, ki naj jih strežnik izvozi. S pomočjo pomožnega programa *rpcgen* nato ustvarimo večino kode za implementacijo strežnika in za uporabo RPC funkcij v odjemalcu.

Zaradi hitrejšega izvajanja je bil Linux NFS strežnik prenešen iz uporabniškega prostora v jedro (knfsd – "kernel NFS deamon"). Ker je NFS široko uporabljan in takorekoč nepogrešljiv v Linux omrežjih, je to tudi najboljše zagotovilo za stabilnost implementacije.

Primer implementacije SunRPC strežnika v jedru je opisan v [7]. Ker ne moremo uporabiti *rpcgen-a*, moramo za vsako izvoženo funkcijo ročno napisati kodo za razpakiranje ("(de)marshalling") podatkov. Za to pa je potrebno podrobno poznavanje notranjega delovanja SunRPC, kar je tudi največja težava pri uporabi SunRPC v jedru. V [7] je omenjeno, da je bilo potrebno že za implementacijo minimalnega strežnika večkrat uporabiti "vohljanje paketov" ("packet sniffing").

Naslednja pomembna slabost je omejitev števila parametrov RPC funkcij. XDR standard dovoljuje samo en vhodni parameter in samo eno vrnjeno vrednost. Kadar potrebujemo več parametrov, jih moramo shraniti v strukturo, to pa nato uporabimo

kot edini vhodni parameter za RPC funkcijo. Podobno velja v primeru več vrnjenih vrednosti. Najboljša rešitev je definicija nove funkcije ("ovitka"), ki izvaja shranjevanje/nalaganje parametrov v/iz strukture. Vendar pa s tem nastopijo nove slabosti:

- Za (skoraj) vsako funkcijo moramo napisati ovitek. To sicer ni težko, je pa nepotrebno in nadležno opravilo.
- Ob spremembi vmesnika funkcije (dodajanje ali odvzemanje parametrov) moramo popraviti tudi ovitek. V fazi razvoja programa so spremembe vmesnika pogoste. Stalno ročno popravljanje ovitka imamo zato za nesprejemljivo, posebej, ker bi to delo lahko opravil *rpcgen*.

Tretja težava pa je uporaba SunRPC v Windows. Res je, da obstajajo NFS odjemalci za Windows, vendar pa so komercialne implementacije zaprte (na voljo niso niti izvorna koda niti pomožna orodja za generiranje kode). Obstaja tudi odprtokoden NFS odjemalec za Windows, vendar pa je slabo realiziran, zato ni nikoli prišel v resno uporabo. To pomeni, da bi morali napisati lasten SunRPC odjemalec. To bi zahtevalo veliko dela (podrobno preučevanje specifikacij, povratni inženiring obstoječega SunRPC na Linux-u, implementacijo, testiranje). Zato tudi SunRPC ni ustrezna rešitev našega problema.

2.2.2 Uporaba v Linux uporabniškem prostoru

Uporaba SunRPC v Linux jedru je težavna. Nasprotno pa je implementacija minimalnega strežnika in odjemalca enostava. Konkreten primer uporabe je opisan v [8], za podrobnejše razumevanje internega delovanja SunRPC pa lahko preberemo [9]. Pisanje programa je jasno razdeljeno v faze:

- Definiramo vmesnik v RPC/XDR jeziku. Poleg seznama prototipov funkcij moramo določiti še številko programa, ki enolično določa program (ga ločuje od ostalih RPC programov), in verzijo programa.
- *Rpcgen* generira kodo strežnika in krne RPC funkcij odjemalca.
- Implementiramo dejansko funkcionalnost strežnika, tj. telo RPC funkcij.
- RPC funkcije uporabimo v odjemalcu.

- Prevedemo strežnik in odjemalca ter ju zaženemo.

Bistven za enostavnost uporabe je drugi korak, to je avtomatična pretvorba abstraktne definicije vmesnika v programsko kodo. Uporabnik mora nato dodati v strežnik samo kodo, specifično za aplikacijo, ter uporabiti krne funkcij v odjemalcu. Nizkonivojski detajli, kot so številke funkcij, uporabljena verzija RPC protokola . . . , so skriti pred uporabnikom.

Pri pretvorbi prototipov iz definicije vmesnika v C kodo pride do nekaterih sprememb:

- Namesto parametra prejme funkcija kazalec na parameter.
- Namesto dejanske vrnjene vrednosti vrne funkcija kazalec na vrnjeno vrednost. Vrnjena vrednost NULL pomeni napako (npr. zaradi napačne verzije strežnika).
- K imenu funkcije je dodan podčrtaj in številka verzije. To omogoča uporabo različnih verzij strežnika v enem odjemalcu.
- Prvi parameter funkcije je ročica ("handle") do strežnika. Tako lahko odjemalec hkrati uporablja več različnih strežnikov.

Pomemben, a nekoliko skrit del avtomatično generirane kode so XDR rutine. Potrebne so za pakiranje ("marshaling") podatkov pred pošiljanjem preko omrežja. Načeloma se pošilja binarna vsebina objektov, s tem da moramo namesto kazalcev poslati objekte, na katere kažejo kazalci. Poleg samega "(de)marshaling-a" opravljajo še pretvorbo med arhitekturno neodvisnim omrežnim formatom in formatom, ki ga uporablja strežnik oz. odjemalec.

3.
Zasnova srpcgen-a

Med obstoječimi RPC sistemi nismo našli nobenega, ki bi bil primeren za našo aplikacijo. Zato smo se odločili za implementacijo lastnega, preprostega RPC sistema. Med pregledovanjem različnih RPC sistemov smo dobili boljšo predstavo o zahtevanih lastnostih RPC sistema. To so:

- **Enostavnost**

  Sistem mora biti enostaven in preprost za uporabo. Pisanje programov ne sme zahtevati poznavanja detajlov delovanja operacijskega sistema ali omrežnih protokolov, ki niso pomembni za končno aplikacijo.

  Ravno tako ne smemo pričakovati, da bo programer ročno pisal kodo, ki jo lahko generiramo/popravimo avtomatično. Primer je dodajanje novega parametra v prototip obstoječe fukcije. Tak parameter se nato pojavi v deklaraciji in v definiciji funkcije, v kodi strežnika in v kodi odjemalca. Ročno popravljanje bi prehitro vodilo v napake zaradi površnosti.

  Avtomatizacijo dosežemo s formalnim opisom vmesnika v jeziku za opis vmesnika (IDL "interface description language"). To nato pretvorimo s prevajalnikom vmesnika (IDL compiler) v krne kode za strežnik in odjemalca.

- **Učinkovitost**

  Najenostavnejše merilo učinkovitosti RPC sistema je čas, potreben za izvedbo ene RPC funkcije. Glavna omejitev je sama "hitrost" omrežja – čas, ki ga IP paket potrebuje za prenos od strežnika k odjemalcu in nazaj. Delno lahko na hitrost vplivamo z izbiro ustreznega mrežnega protokola. Posebej med "connectionless" (npr. UDP/IP) in "connection-oriented" (npr. TCP/IP) protokoli lahko

pričakujemo znatne razlike [7].

Vpliv porabljenega procesorskega časa je zanemarljiv v primerjavi s časom, potrebnim za mrežno komunikacijo. Nekoliko bolj pazljivi moramo biti pri porabi pomnilnika. Strežnik se bo izvajal v Linux jedru, kjer je sklad omejen na 8 kB. To je dovolj, če pazimo, da na sklad ne shranjujemo nepotrebnih kopij podatkov in da RPC funkcije ne uporabljajo prevelikih struktur.

- **Zanesljivost**

  RPC komunikacija poteka preko v splošnem nezanesljivih povezav. Pri komunikaciji lahko pride do izgube, podvojitve ali pa do sprejema podatkov v napačnem vrstnem redu. Zaradi učinkovitosti lahko izberemo transportni sistem, ki napak ne obdeluje (UDP/IP). V tem primeru moramo obdelavo napak realizirati sami. Če uporabljeni transportni mehanizem sam obdeluje tovrstne napake (TCP/IP), se z njimi ne rabi ukvarjati RPC sistem.

- **Večuporabniškost**

  Resni RPC sistemi omogočajo hkratno uporabo enega strežnika s strani več odjemalcev naenkrat. Strežnik mora biti odporen na slabo nepravilno delovanje odjemalcev. Nepravilno delovanje (ki je lahko tudi namerno) enega od odjemalcev ne sme motiti drugih odjemalcev.

- **Skalabilnost**

  Pri večjih sistemih je zaželena lastnost tudi skalabilnost, se pravi, da strežnik dobro deluje, tudi če izvozimo večje število funkcij ali če ga hkrati uporablja več uporabnikov.

3.1 Zahteve za implementacijo

Glede enostavnosti smo si zadali cilj, da mora biti pisanje minimalnega RPC programa trivialno opravilo, podobno kot pri SunRPC z uporabo *rpcgen-a*. Poznejše dodajanje ali odvzemanje parametrov funkcij v opisu vmesnika se mora avtomatično preslikati v generirano kodo. Koda za "(de)marshalling" parametrov mora biti generirana avtomatično.

Pri hitrosti izvajanja enega RPC klica želimo doseči primerljiv rezultat z originalnim VxWorks sistemom, to je približno 0.6 ms.

Porabo sklada v jedrnem strežniku minimiziramo tako, da parametre izvožene funkcije shranimo v strukturo (ki se ne nahaja na skladu). Funkcije nato uporabljajo kazalec na to strukturo, vse do zadnje, ki prejme vsak posamezen parameter.

Obdelavo napak, do katerih lahko pride pri pošiljanju podatkov preko omrežja, bomo rešili z uporabo TCP/IP protokola. V veliki meri se zanašamo na dejstvo, da so komunikacije v majhnih omrežjih dokaj zanesljive. Komunikacija praviloma deluje zelo dobro (nič izgubljenih paketov) ali pa sploh ne (v primeru iztaknjenega kabla ali nepravilnih nastavitev na nivoju OS). Zato tudi ne bo težav z neustrezno izbranimi časovnimi omejitvami za pošiljanje podatkov.

Bolj previdni moramo biti pri interpretaciji vsebine prejetih podatkovnih paketov. Paket je lahko nesmiseln, kar detektiramo po nepravilni dolžini paketa in/ali po nepravilni vsebini paketa. V tem primeru naj strežnik zahtevo za RPC klic zavrne.

Vmesnik bomo definirali v jeziku, definiranem na osnovi jezika XDR. XDR (uporablja ga SunRPC) je zelo podoben C-ju. Dodali bomo predvsem možnost uporabe več kot enega parametra. Poleg tega naj parametri in vrnjene vrednosti funkcij ne bodo samo kazalci, temveč poljubne vrednosti (zavoljo enostavnosti uporabe).

Strežnik bo implementiran kot modul za Linux jedro, zato bo napisan v jeziku C. Tudi odjemalec bo napisan v jeziku C oziroma C++. Prevajalnik vmesnika mora zato generirati kodo v jeziku C. C++ ne želimo uporabljati iz dveh razlogov:

- Strežnik moramo prevesti kot Linux jedrni modul. V Linux jedru lahko uporabljamo samo C jezik, C++ pa ne.
- C++ bi sicer lahko uporabljali na strani odjemalca, vendar pa bi bili tudi tu močno omejeni. Predvsem ne bi mogli uporabljati objektov zaradi težav z binarno združljivostjo med različnimi C/C++ prevajalniki (objekti z virtualnimi funkcijami imajo dodan skrit "virtual function pointer table" ipd.).

Ker želimo uporabljati odjemalca tudi na Windows OS, mora biti generirana koda odjemalca v standardnem C-ju, oz. biti mora prenosljiva med Linux in Windows OS.

Generirano kodo lahko s stališča avtomatičnega generiranja/popravljanja razdelimo v dva dela. En del so krni RPC funkcij in z njimi povezane podatkovne strukture. Te je potrebno stalno popravljati glede na definicijo vmesnika. Tu sme uporabnik popravljati zgolj telo (tj. implementacijo dejanske funkcionalnosti) izvoženih funkcij za strežnik.

Drugi del kode (npr. inicializacija strežnika in odjemalca) je potrebno generirati samo enkrat, in je prevajalnik vmesnika v kasnejših fazah popravljanja kode ne spreminja. Lahko se zgodi, da je ta koda omejujoča pri izvedbi konkretnega RPC programa. Zato je zaželeno, da ima uporabnik možnost spreminjanja te kode. Tako lahko npr. TCP/IP protokol zamenjamo z UDP/IP protokolom. V tem smislu je ta del avtomatično generirane kode zgolj vzorec ("template"), ki omogoča hitrejšo in lažjo implementacijo minimalnega RPC programa, brez izključitve možnosti uporabe drugačne kode.

Pri SunRPC je vrnjena vrednost RPC funkcije vedno kazalec. V primeru napake (npr. klic RPC funkcije, ki ni implementirana v strežniku) je nastavljen na NULL vrednost. To zahteva preverjanje uspešnosti izvršitve po vsakem klicu, kar je nesprejemljivo dodatno delo. Namesto tega želimo, da so RPC funkcije predstavljene povsem transparentno. Morebitne napake naj krn vsake RPC funkcije na strani odjemalca preusmeri v splošno funkcijo za obdelavo napak, katere telo lahko uporabnik popravi po lastnih željah.

Rpcgen omogoča definicije novih podatkovnih tipov v datoteki z definicijo vmesnika. Te definicije se pretvorijo v C-jevsko definicijo tipov v generiranih *.h datotekah. V našem primeru so nekateri podatkovni tipi že definirani v drugih C-jevskih datotekah (npr. podatkovni tip za vektorje in matrike v matematični knjižnici). Če bi *srpcgen* dodal še eno definicijo obstoječih tipov, bi morali popravljati obstoječe C-jevske datoteke. Zato bo v definiciji vmesnika možno samo deklarirati nove podatkovne tipe. Dejansko definicijo bomo morali napisati v C-jevski kodi. Tako je tudi lažje vključiti RPC funkcije v obstoječi projekt, kjer so tipi spremenljivk že definirani v stari C kodi, in je *srpcgen* edini, ki teh tipov ne pozna.

Poleg enostavnih podatkovnih tipov (int, long, double ...) želimo imeti tudi možnost uporabe kazalcev, polj in struktur. SunRPC rešuje pošiljanje poljubnih podatkovnih tipov z uporabo t. i. XDR rutin, ki so lahko generirane avtomatično z *rpcgen-om* ali pa ročno. Mi smo nekoliko omejili raznolikost tipov podatkov, ki jih smemo uporabljati v RPC funkcijah, zato da smo poenostavili implementacijo. Kazalci smejo biti samo enojni, polja morajo biti fiksne dolžine, strukture pa morajo biti "preproste" – ne smejo vsebovati kazalcev). Več o tem je v poglavju 3.2.

3.1.1 Česa ne bomo implementirali

Večuporabniške podpore ne nameravamo realizirati. V prvi vrsti je, vsaj za enostavne programe, niti ne potrebujemo. Zato je bolje, da je tudi ne implementiramo. S tem se ognemo skušnjavi, da bi napisali znatno količino kode, ki bi ostala slabo testirana. V takšni kodi pogosto ostanejo skriti hrošči, ki jih (z veliko truda) odkrijemo šele med uporabo končnega strežnika/odjemalca. Funkcionalno bogati, a slabo implementirani programi so lahko neuporabni zaradi hroščev. S skromnimi, a dobro delujočimi programi pa lahko nasprotno uporabniku celo dodatno prihranimo čas, ker lažje in hitreje oceni, ali je program primeren za njegov namen.

SunRPC omogoča tudi številne "napredne" načine uporabe, npr. "broadcast" pošiljanje, avtentikacijo in paketno pošiljanje ("batching"). Paketno pošiljanje omogoča hitrejše izvajanje RPC odjemalca. V tem načinu odjemalec ne čaka na odgovor strežnika, ki potrdi uspešnost izvedbe. Z odpravo čakanja na odgovor strežnika prihranimo čas. Poleg tega pa lahko na strani odjemalca še združimo več zaporednih RPC klicev ter jih pošljemo v enem samem podatkovnem paketu, kar znova prihrani čas. Takšen način izvajanja bi bil sicer zanimiv za našo aplikacijo, vendar pa ne omogoča vračanja informacije o uspehu izvršitve. Večina funkcij v FCS-jevi hapticAPI vrne kodo napake, tako da moramo zaradi ohranitve združljivosti čakati na odgovor strežnika.

RPC strežniki, implementirani na osnovi SunRPC, uporabljajo program portmapper za vzpostavitev povezave med strežnikom in odjemalcem [10]. RPC strežnik ne posluša na fiksnih IP vratih, temveč na poljubnih. Uporabljena vrata nato registrira pri lokalnem portmapper-ju. Odjemalec nato najprej vpraša portmapper za vrata, kjer posluša strežnik, in šele nato neposredno komunicira s strežnikom. Ta rešitev omogoča uporabo večjega števila strežnikov, ker ne potrebuje vsak svojih, fiksnih vrat.

V našem primeru bo tekel samo en strežnik (ali največ nekaj strežnikov, če bi hoteli izvoziti še kakšne funkcije, brez spreminjanja že implementiranega strežnika). Uporaba portmapper-ja bi vnesla samo nepotrebno kompleksnost. Portmapper zato ne bo uporabljen. Strežnik bo poslušal na fiksnih vratih, ki jih podamo ob zagonu strežnika kot parameter v ukazni vrstici.

3.2 Podajanje parametrov po vrednosti in po referenci

V jeziku C lahko klicana funkcija spremeni vhodne parametre. Če je takšen parameter podan po referenci, je sprememba vidna tudi v klicoči funkcijo. Eden od ciljev implementacije RPC funkcij je čim boljše posnemanje lokalnih funkcij. Zato si poglejmo načine podajanja parametrov na primeru lokalnih funkcij, tako da bomo vedeli, k čemu težiti pri implementaciji RPC funkcij.

3.2.1 Lokalni klici funkcij

Izvedba lokalnega klica funkcije v splošnem vključuje:

- Shranjevanje parametrov funkcije na sklad. To shranjevanje izvede klicoča funkcija.
- Izvršitev klicane funkcije. Funkcija uporabi parametre in generira vrnjeno vrednost. Poleg tega lahko povzroči še druge “stranske učinke”, npr. spremeni vhodne parametre.
- Klicana funkcija shrani vrnjeno vrednost na sklad.
- Klicana funkcija se neha izvajati, nadzor se prenese nazaj na klicočo funkcijo.
- Klicoča funkcija odstrani parametre s sklada[1] in nadaljuje z izvajanjem.

Klicoča funkcija lahko vidi spremembe parametrov, ki jih je naredila klicana funkcija, ali pa tudi ne. To je odvisno od tega, ali so parametri podani po referenci ali po vrednosti.

3.2.1.1 Podajanje parametrov po vrednosti

Poglejmo na primeru, kaj se dogaja pri klicu funkcije. Osredotočimo se na C jezik. Naj funkcija *main* kliče *add_two_numbers*.

```
double add_two_numbers(double num1, double num2)
{
    return num1 + num2;
```

[1]Parametre lahko počisti tudi klicana funkcija (razlika med cdecl in stdcall “calling convention”).

```
}

int main(void)
{
    double number1 = 10.0, number2 = 20.0;
    double sum;

    sum = add_two_numbers(number1, number2);
    return 0;
}
```

Prevajalnik generira kodo, ki najprej potisne oba parametra (*number1* in *number2*) na sklad. Nato se izvede klic funkcije *add_two_numbers*. Ta ne dostopa do originalnih spremenljivk, temveč do njunih kopij *num1* in *num2* (ki pa imata enaki vrednosti kot *number1* in *number2* – 10.0 in 20.0). Da je *num1* samo kopija od *number1*, najlažje preverimo z razhroščevalnikom – vrednosti spremenljivk sta isti, naslova pa različna. Ob zaključku klica funkcije *add_two_numbers* se oba parametra odstranita s sklada, vrnjena vrednost pa se vpiše v spremenljivko *sum*.

Ker je parameter *num1* samo kopija *number1*, lahko *add_two_numbers* spremeni vrednost *num1*, pa to ne vpliva na vrednost *number1*. Temu rečemo podajanje parametrov po vrednosti ("parameter passing by value").

3.2.1.2 Podajanje parametrov po referenci – kazalci

Pogosto pa moramo spremeniti tudi vhodne parametre, in to tako, da so spremembe vidne tudi v klicoči funkciji. Tipičen primer je vračanje več kot ene vrednosti – sintaksa C-ja nas omejuje na največ eno vrnjeno vrednost, zato moramo ostale podati kot parametre. V ta namen imamo na voljo kazalce. V spodnjem primeru sta obe vrnjeni vrednosti *product* in *quotient* podani kot parametra. Funkcija *multiply_divide_two_numbers* prejme naslova od *product* in *quotient*, zato je vpis izračunanih vrednosti v **pProd* in **pQuot* viden tudi v klicoči funkciji *main*. Takemu podajanju parametrov rečemo podajanje parametrov po referenci ("parameter passing by reference").

```
void multiply_divide_two_numbers(double num1, double num2,
                                 double *pProd, double *pQuot)
{
    *pProd = num1 * num2;
    *pQuot = num1 / num2;
```

```
    return;
}

int main(void)
{
    double number1 = 10.0, number2 = 20.0;
    double product, quotient;

    multiply_divide_two_numbers(number1, number2, &product, &quotient);
    return 0;
}
```

3.2.1.3 Podajanje parametrov po referenci – polja

Drug pogost primer podajanja po referenci je podajanje polj. V C-ju prejme klicana funkcija namesto (kopije) vsebine polja naslov polja (tj. naslov prvega elementa polja). Vse spremembe v vhodnem polju vidi tudi klicoča funkcija. Če *add_three_numbers* po nesreči vpiše spremenjene vrednosti v elemente polja *num*, vidi klicoča funkcija *main* te spremembe v polju *number*.

```
double add_three_numbers(double num[3])
{
    return num[0] + num[1] + num[2];
}

int main(void)
{
    double number[3] = {10.0, 20.0, 30.0};
    double sum;

    sum = add_three_numbers(number);
    return 0;
}
```

Takšni vpisi se lahko zgodijo tudi po nesreči, iz neprevidnosti. Pred njimi se lahko delno zaščitimo z uporabo ključne besede *const*. V tem primeru nas bo prevajalnik opozoril, če poizkusimo vpisovati v polje. Še vedno pa lahko takšno (napačno) kodo prevede, in v končnem programu bo klicoča funkcija videla spremembe v polju. Polja so zato vedno vhodni in izhodni parametri.

3.2.2 Oddaljeni klici funkcij

Pri lokalnih klicih funkcij se po referenci prenašajo kazalci in polja. Klicana funkcija lahko spremeni vrednost teh parametrov, in sprememba bo vidna tudi v klicoči funkciji. Pred spremembo se lahko zaščitimo z uporabo besede "const". Vendar pa uporaba besede "const" ne prepreči spremembe parametra, zagotovi nam samo opozorilo s strani prevajalnika, da je naša koda morda napačna. Klicana funkcija lahko vseeno spremeni parameter.

To smo upoštevali tudi pri *srpcgen-u*. Če je parameter funkcije kazalec ali polje, lahko strežnik spremeni njegovo vsebino, in po zaključku RPC klica bo tudi odjemalec videl spremenjene parametre. Uporaba besede "const" na to nima vpliva.

3.3 Shema delovanja generiranih RPC programov

Pri RPC klicih shranimo parametre namesto na sklad v podatkovno strukturo (RPC paket), ki jo pošljemo preko omrežja. V paket moramo poleg tega shraniti še informacijo o funkciji, ki naj jo strežnik izvede nad poslanimi parametri. Zato je vsaki RPC funkciji dodeljena unikatna številka. Strežnik nam nato vrne vrnjeno vrednost funkcije in morebitne ostale pomembne podatke (npr. nove vrednosti parametrov, ki jih je spremenila klicana funkcija – štejejo samo podatki, podani "po referenci", glej poglavje 3.2).

3.3.1 Vsebina RPC paketa

RPC paketi se začno z glavo, temu pa sledijo vhodni ali izhodni podatki. Spodaj je prikazana definicija glave. *m_ulInLength* in *m_ulOutLength* sta dolžini vhodnih in izhodnih podatkov.

Verzija SRPC protokola je shranjena v *m_ulSRPCVersion*. V *m_ulProgID* je shranjena številka programa, ki "identificira" program – zahtevan strežnik, *m_ulProgVersion* pa vsebuje zahtevano verzijo programa. V dejanski implementaciji je uporaba teh treh števil minimalna, pomembno je samo ujemanje med odjemalcem in strežnikom. V primeru, da se npr. ne ujema "identifikacijska" številka programa (*m_ulProgID*), bo strežnik zapisal napako v *m_lError*.

Najpomembnejše polje je *m_ulProcedure*. Strežniku pove, katero RPC funkcijo zahteva

odjemalec.

Na koncu so še vhodni (če gre RPC paket v smeri od odjemalca k strežniku) ali izhodi (če gre RPC paket v povratni smeri, od strežnika k odjemalcu) podatki. Dolžina tega polja je zapisana v *m_ulInLength* oz. *m_ulOutLength*. Vhodni podatki vsebujejo parametre, s katerimi je odjemalec klical krn RPC funkcije. Izhodni podatki vsebujejo poleg vrnjene vrednosti RPC funkcije še parametre, ki jih je spremenila klicana funkcija.

3.3.2 Izvedba RPC klica

Slika 3.1 prikazuje shemo delovanja. Puščice prikazujejo pošiljanje parametrov oz. RPC paketa s parametri. Puščice so dvosmerne – v smeri od odjemalca k strežniku potuje vhoden RPC paket, v smer od strežnika k odjemalcu pa izhodni oz. povratni RPC paket.

Odjemalec uporablja RPC funkcije tako, da kliče krne RPC funkcij. Klicanje krnov RPC funkcij je povsem enako klicanju lokalnih funkcij. Nobene posebne vrnjene vrednosti ali dodatni parametri funkcije niso rezervirani za obdelavo napak.

Krn vsake RPC funkcije shrani parametre v podatkovno strukturo. Na začetku strukture je še glava, ki ima enako obliko za vse funkcije. V njej je najpomembnejša številka funkcije, s pomočjo katere strežnik ugotovi, katero funkcijo zahteva odjemalec.

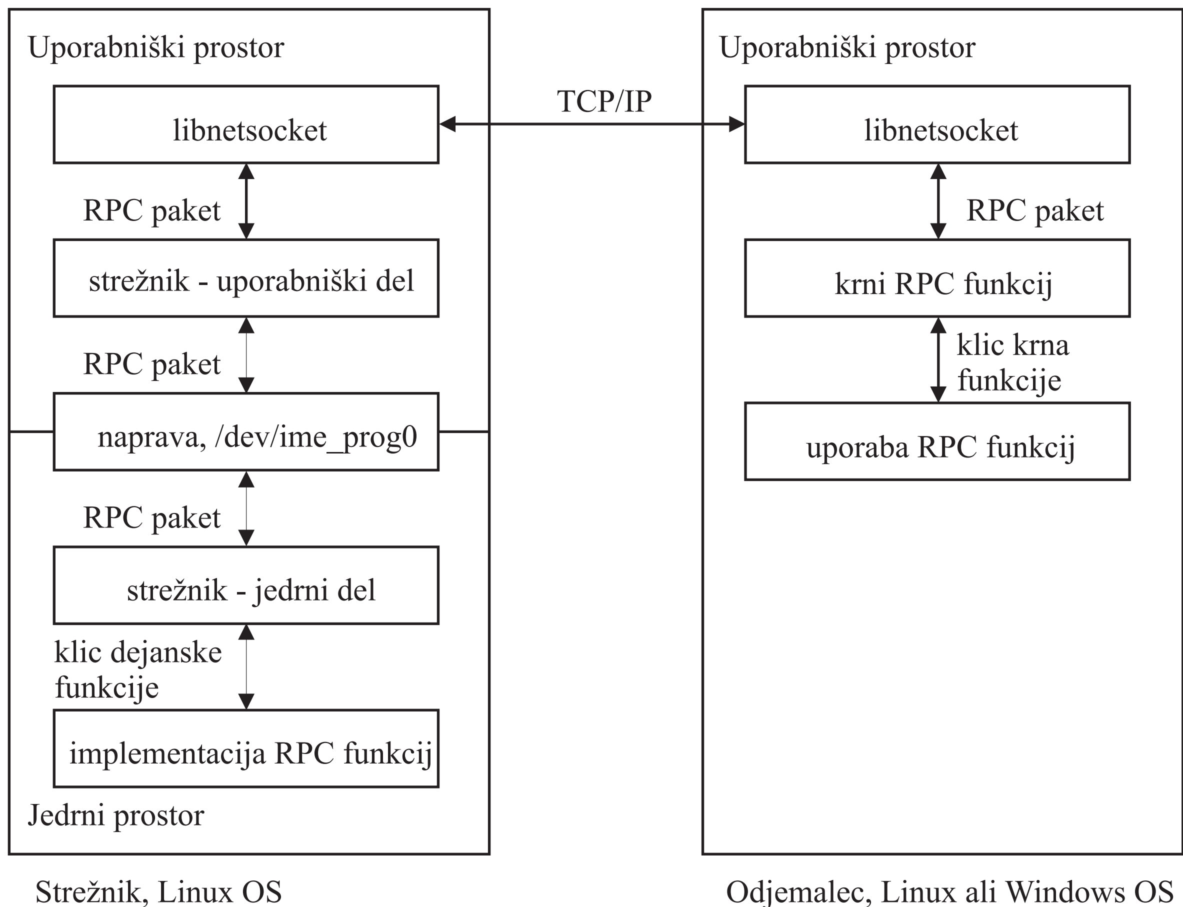
Krn nato pošlje podatkovno strukturo (ki ji zdaj lahko rečemo RPC paket) preko omrežja z uporabo knjižnice libnetsocket. Koda za pošiljanje in sprejemanje podatkov preko omrežja ter za obdelavo napak pri mrežni komunikaciji je skupna za vse funkcije. Po oddaji paketa odjemalec čaka na odgovor strežnika.

Na drugi strani strežnik prejme paket. Sprejem se zgodi v uporabniškem prostoru, kjer teče uporabniški del strežnika ("user space daemon"). Ta sprejeti paket pošlje jedrnemu strežniku preko naprave ("/dev/ime_prog0").

Jedrni strežnik prejme paket in preveri njegovo vsebino (ujemanje verzij, ujemanje dejanske dolžine podatkov s pričakovano dolžino podatkov za zahtevano RPC funkcijo). Nato odpakira ("demarshaling") posamezne parametre iz RPC paketa ter z njimi kliče implementacijo RPC funkcije.

Implementacija RPC funkcije opravi svoje delo. Pri tem generira vrnjeno vrednost in morda tudi spremeni vhodne parametre.

Nato se začne povratni del RPC klica. Jedrni strežnik shrani vrnjeno vrednost in



Slika 3.1: Shema delovanja RPC klica

spremenjene parametre v RPC paket. Ta nato potuje k strežniku v uporabniškem prostoru, ki ga preko omrežja pošlje odjemalcu. Odjemalec prejme RPC paket in ga preda krnu klicane RPC funkcije. Krn najprej prepiše svoje, originalne parametre, s spremenjenimi parametri iz RPC paketa. Nato še vrne vrnjeno vrednost iz RPC paketa ter s tem zaključi izvajanje RPC klica.

3.4 Jezik za opis vmesnika

Srpcgen-ov jezik za opis vmesnika ("interface description language" – IDL) je podoben tistemu pri SunRPC in *rpcgen-u*, ta pa je podoben C-ju. Sintakso bomo najlažje razložili na primeru. Napišimo definicijo vmesnika RPC programa, ki lahko sprejme in nato vrne število. Celoten vmesnik strežnik – odjemalec sestavljata samo dve funkciji:

```
void setNumber( double num );
```

```
double getNumber( void );
```

Vmesnik definiramo v datoteki “example1.x”. Uporabljena je končnica .x, tako kot pri *rpcgen-u*. Uporabljamo lahko C ali C++ komentarje, tj. “/* komentar v vec vrsticah */” ali “// komentar do konca vrstice”.

```
/* file example1.x */
program example1 {
  version example1_version
  {

    void setNumber( double num ) =1;
    double getNumber( void );

  } = 0x001; // program version 0.01
} = 1; // program number 1
```

Na začetku povemo, da je ime programa example1. Sledi definicija verzije programa, nato pa prototipi funkcij s številko funkcije. Na koncu definiramo še “identifikacijsko” številko programa in verzijo programa.

Vsak program mora uporabljati drugačno številko programa. Verzijo programa pa povečamo, če spremenimo “application binary interface” – ABI, npr. prototipu funkcije dodamo, odvzamemo ali spremenimo vrstni red parametrov, ali pa če spremenimo številke funkcij. Odjemalec in strežnik morata biti prevedena z isto številko programa in verzijo programa, v nasprotnem primeru pride do napake.

Številke funkcij se začnejo z 1 (ali več), številka 0 pa je rezervirana, tako kot pri SunRPC. Mogoče je določiti samo številko prve funkcije, vsaki naslednji funkciji pa je potem dodeljena za 1 višja številka od predhodne. To je drugače kot pri SunRPC, kjer je potrebno ročno napisati številko za vsako funkcijo. Zapis števila je lahko v desetiški ali v šestnajstiški obliki.

Prototipi funkcij uporabljajo podobno sintakso kot jezik C. Osnovni podatkovni tipi parametrov ter vrnjenih vrednosti so:

- void
- char
- short

- int
- long
- float in
- double.

Uporabljamo lahko tudi ključne besede const/volatile[2] ter signed/unsigned.

3.4.1 Kazalci, polja in strukture kot parametri

V prejšnjem primeru so uporabljeni samo enostavni podatkovni tipi. V kompleksnejših programih uporabljamo tudi kazalce, polja, strukture, oštevilčene (enumerirane, "enumerated") tipe, "tipe", definirane z "#define" itd. Srpcgen dovoljuje tudi takšne tipe, vendar z določenimi omejitvami:

- Kazalci so lahko samo enojni (tj. samo ena *).
- Polja morajo biti fiksne dolžine, lahko pa so večdimenzionalna.
- Strukture naj ne bi vsebovale kazalcev. Če jih, se prenesejo k strežniku kot kazalci in ne kot kopije objektov, tako da strežnik ne more uporabiti objekta, na katerega kaže kazalec (objekt je ostal na strani odjemalca). Strukture lahko vsebujejo polja ali druge strukture.

Poleg tega moramo na začetku example1.x datoteke deklarirati imena novih tipov. Poglejmo primer, kjer deklariramo ime za polje treh double števil (TTVec3 – vektor treh števil), za "oštevilčen" podatkovni tip (enumOperation_t) in za strukturo (SResult_t), nato pa dodamo še funkcijo math_operation().

```
/* file example1.x */
typedef_array TTVec3;
```

[2]Besede const ne moremo uporabiti. V tem pogledu *srpcgen* sam sicer deluje, vendar pa prevajalnik (gcc, verzija 3.2.2) ne sprejme unije 2 struktur, ki imata const člane (napaka "copy assignment operator not allowed in union"). Rešitev bi bila drugačna definicija paketov. Namesto da imamo en podatkovni tip za vhodni in izhodni paket, ki ima na začetku glavo, nato pa unijo vhodnega in izhodnega paketa, uporabimo dva podatkovna paketa (en vhodni, drugi izhodni), oba pa imata na začetku glavo. Tako ne bi bilo potrebno uporabljati unije, da bi prihranili prostor.

```
typedef_simple enumOperation_t;
typedef_simple SResult_t;

program example1
{
  version example1_version
  {

    void setNumber( double num ) =1;
    double getNumber( );
    double add_two_numbers( double num1, double num2 );
    int math_operation( TTVec3 vec, enumOperation_t oper, SResult_t *pRes );

  } = 0x001; // program version 0.01
} = 1; // program number
```

Srpcgen bo uspešno sprejel zgornjo definicijo vmesnika. Vedeti pa moramo, da z besedama "typedef_array" in "typedef_simple" nove tipe samo "deklariramo". Za polja moramo uporabiti "typedef_array", za strukture, oštevilčene tipe ipd. pa "typedef_simple". Razlika v generirani C kodi je v načinu kopiranja podatkov – polj ne moremo kopirati z operatorjem prirejanja (=), strukture in oštevilčene tipe pa lahko.

3.5 Potrebna programska oprema

3.5.1 Red Hat Linux 9.0

V prvi vrsti je naša programska oprema namenjena izvajanju na operacijsken sistemu Linux. Uporabljen je bil Red Hat Linux 9.0. Originalno jedro smo zamenjali z jedrom verzije 2.4.22, dodan je bil še "patch" za real time Linux (rtLinuxFree) verzije 3.2-pre3. C prevajalnik je gcc verzije 3.2.2, tj. originalen prevajalnik, vključen v distribucijo Red Hat 9.0.

3.5.2 Microsoft Windows 2000

Za kodo odjemalca zahtevamo tudi možnost izvajanja v Windows OS. Sami smo uporabili Windows 2000, kodo pa smo prevajali z Visual Studio 6.0. Težav ne bi smelo biti tudi pri uporabi na Windows XP oz. na drugih združljivih sistemih.

3.5.3 Codeworker

Za avtomatično generiranje in popravljanje kode smo uporabili orodje *codeworker*, prosto dostopno na [2] v obliki izvorne kode. Orodje lahko poganjamo v Linux ali v Windows OS. Orodje uporablja lasten skriptni jezik. Za barvno označevanje skrip je priporočeno uporabiti vtič ("plugin") za Eclipse integrirano razvojno okolje (IDE – "integrated development environment"). Poleg programa je na voljo še obsežna dokumentacija in primeri uporabe.

Codeworker je namenjen slovnični razčlenitvi in avtomatičnemu generiranju programske kode. Razčleni lahko poljuben jezik, moramo pa najprej definirati pravila tega jezika, oz. napisati razčlenjevalnik ("parser"). Rezultat razčlenitve se shrani v drevesno strukturo, ki jo nato uporabimo pri generiranju kode.

Naslednji korak je avtomatično generiranje in popravljanje kode. Pri tem lahko generiramo celotno izhodno datoteko izključno s *codeworker-jem* ali pa popravljamo samo dele obstoječe datoteke. Za našo aplikacijo je primeren drugi način. Tu *codeworker* vstavlja generirano kodo na točno določena mesta v izhodni datoteki (t. i. označene sekcije). Morebitne spremembe vsebine označene sekcije so uničene ob naslednjem zagonu *codeworker-ja*. Da se temu ognemo, je lahko v vsaki označeni sekciji zaščitena sekcij, katere vsebina ostane nespremenjena.

3.5.4 Eclipse

Eclipse (v širšem pomenu) je odprtokodna platforma za razvoj programskih orodij [3]. Pogosto pa se uporablja za poimenovanje manjšega dela celotnega projekta, tj. za Eclipse SDK. Ta je v osnovi namenjen predvsem pisanju JAVA aplikacij, lahko pa ga razširimo z dodatnimi komponentami in ga uporabimo za razvoj programov v drugih jezikih. Tako z uporabo C/C++ razvojnih komponent dobimo C/C++ integrirano razvojno okolje. Platforma je s konceptom dodajanja komponent dovolj fleksibilna, da je bila uporabljena celo za razvoj aplikacij na programiranju povsem tujih področjih, npr. v bančništvu, medicini in raziskovanju vesolja.

Mi smo eclipse uporabljali izključno kot urejevalnik programske kode pri pisanju codeworker skript. Da omogočimo barvno označevanje kode, moramo namestiti še codeworker vtič, ki ga dobimo na [2].

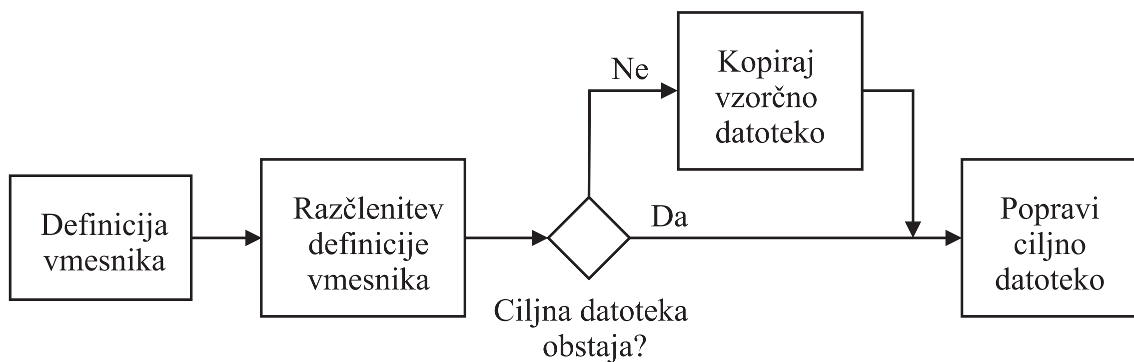
3.5.5 Libnetsocket

Za pošiljanje podatkov preko omrežja smo uporabili knjižnico libnetsocket (verzija 0.3.1, avtor Aleš Bardorfer, Fakulteta za elektrotehniko, Laboratorij za robotiko in biomedicinsko tehniko). Ta knjižnica omogoča enostavno pošiljanje in sprejemanje podatkov preko omrežne povezave. Podprta sta TCP/IP in UDP/IP protokola. Uporabljamo jo lahko na Linux ali Windows OS. Poleg izvorne kode so priloženi še kratki programi, ki ilustrirajo uporabo knjižnice. Ti so dovolj enostavni, da dodatna dokumentacija ni potrebna.

4. Implementacija srpcgen-a

Delovanje *srpcgen-a* je razdeljeno v 3 faze, kot prikazuje slika 4.1:

- Najprej se razčleni datoteka z definicijo vmesnika. Rezultat razčlenitve se shrani v drevesno strukturo.
- Če je *srpcgen* klican prvič, je potrebno prekopirati vzorčne datoteke (C koda, Makefile, skript za zagon strežnika) v ciljne datoteke. V vzorčnih datotekah je že pripravljen tisti del kode, ki ni odvisen od definicije vmesnika. Takšna koda je npr. zagon strežnika in vzpostavitev omrežne povezave.
- Rezultat razčlenitve uporabimo za spreminjanje (ciljnih) C datotek.



Slika 4.1: Shema delovanja *srpcgen-a*

Sam *srpcgen* je implementiran kot zbirka skript za *codeworker*. Datoteke so:

- srpcgen.cws, t. i. vodilni skript v terminologiji *codeworker-ja*.
- srpcgen_parser.cwp, vsebuje kodo, potrebno za razčlenitev definicije vmesnika.

- srpcgen_expand_functions.cwt vsebuje funkcije, ki jih uporabljajo druge srpcgen_expand_*.cwt skripte.
- srpcgen_expand_PROG_sh.cwt popravlja skript za zagon strežnika (ta ima ime PROG.sh, PROG se zamenja z imenom RPC programa).
- Datoteke srpcgen_expand_*.cwt. Vsak od teh skript popravlja eno od generiranih datotek s C kodo.

4.1 Razčlenitev definicije vmesnika

Vsa koda, potrebna za razčlenitev definicije vmesnika, je v skripti srpcgen_parser.cwp. Za podroben pregled vsebine bi bilo potrebno poznati sintakso *codeworker-jevih* skript, zato bomo to izpustili in podali zgolj grob opis delovanja. Potek razčlenitve je naslednji:

- Najprej povemo, da datoteka z definicijo vmesnika vsebuje C++ tip komentarjev.
- Sledi branje deklaracij novih podatkovnih tipov (typedef_array in typedef_simple).
- Nato sledi deklaracija RPC programa. Ta vključuje:
  - ime programa,
  - ime verzije programa,
  - ene ali več deklaracij RPC funkcij,
  - številko verzije programa in
  - številko programa.

Deklaracija RPC funkcij je sestavljena iz:

- vrnjene vrednosti,
- imena funkcije,
- nič ali več parametrov ter
- številke funkcije. Če je izpuščena, se uporabi za 1 povečana številka predhodne funkcije.

Razčlenjevalnik k vsakemu parametru funkcije pripiše, ali je samo vhoden (podajanje parametra po vrednosti) ali je tudi izhoden (podajanje parametra po referenci).

Srpcgen izvede po zaključku razčlenitve še minimalistično preverjanje rezultata. Za vsako funkcijo preveri, ali je v njej kakšen parameter – kazalec, ki ni enojen; dvojni in večkratni kazalci so namreč prepovedani. Rezultat razčlenitve je shranjen v drevesno strukturo, ki se uporabi pri razširjanju ("expanding") vzorčnih datotek.

4.2 Vzorčne datoteke

Vzorčne datoteke so v poddirektoriju c_sources. Pri kopiranju se niz "PROG" zamenja z dejanskim imenom RPC programa, kot je definiran v definiciji vmesnika. Poleg tega se v preimenovanih datotekah tudi zamenja niz "PROG" z dejanskim imenom programa. Vzorčne datokete imajo že vstavljene t.i. zaščitene sekcije, kamor lahko *srpcgen* (natančneje *codeworker*) vstavlja avtomatično generirano kodo. Spisek vzorčnih datotek (in vsebina njihovih kopij – potem ko le-te popravi "srpcgen") je:

- Makefile, to je Makefile za GNU/Linux make. Ukaz "make all" naredi 3 izvršljive datoteke.
  - srpc_ksvc_ime_programa.o je jedrni strežnik, ki implementira dejansko funkcionalnost strežnika.
  - srpc_usvc_ime_programa je del strežnika v uporabniškem prostoru.
  - srpc_clnt_ime_programa je minimalni, testni odjemalec.
- PROG.sh, skript za zagon strežnika. Tu lahko uporabnik nastavi nova TCP/IP vrata uporabniškega strežnika (USER_SERVER_TCP_PORT), če privzeta vrata že uporablja drug program.
- Vzorčne datoteke s C kodo:
  - srpc_PROG.h, datoteka z definicijami podatkovnih tipov, ki jih potrebujeta tako strežnik kot tudi odjemalec. Tu je definicija glave RPC paketa (*srpc_packet_head_t*), ki je na začetku vsakega RPC paketa. Sledijo številke verzije *srpc* protokola, programa, verzije programa in številke posameznih RPC funkcij. Na koncu je za vsako RPC funkcijo še definirana struktura vhodih in struktura izhodnih parametrov, ter ustrezen RPC paket. Slednji

ima na začetku glavo, nato pa sledi ali struktura vhodnih parametrov (če gre paket od odjemalca k strežniku) ali pa struktura izhodnih parametrov (če gre paket od strežnika k odjemalcu – povratni RPC paket).

- srpc_clnt.h in srpc_clnt.cc, datoteki s kodo odjemalca (clnt pomeni "client"), ki se ne spreminja z definicijo vmesnika. To je npr. vzpostavitev in prekinitev povezave odjemalec – strežnik (funkciji *srpc_clnt_start* in *srpc_clnt_stop*) in pošiljanje RPC paketa strežniku (*srpc_clnt_rpc_call*).
- srpc_clnt_PROG.h in srpc_clnt_PROG.cc, datoteki s kodo odjemalca, ki se spreminja z definicijo vmesnika. V .h datoteki so deklaracije krnov RPC funkcij. V .cc datoteki pa je implementacija RPC krnov. Vsak krn shrani vhodne parametre v RPC paket, ki ga nato pošlje strežniku. Na koncu prepiše originalne parametre funkcije s spremenjenimi vrednostmi (kot jih vrne strežnik v povratnem RPC paketu) ter vrne vrnjeno vrednost.

  Poleg tega je na začetku še vzorčna *main* funkcija, ki demonstrira zagon odjemalca in jo lahko uporabimo za testiranje RPC funkcij.
- srpc_svc.h, datoteka z definicijami konstant, ki jih uporabljata oba dela strežnika (svc pomeni "service"). Tu sta pomembni predvsem ime in glavna ("major") številka naprave, preko katere poteka komunikacija uporabniški prostor – jedro. RPC paket se pošlje v jedro preko IOCTL klica.
- srpc_ksvc_PROG.c, datoteka s kodo jedrnega strežnika. V funkciji *srpc_ksvc_start* je najbolj pomembna registracija naprave – ta pove Linux OS, da je za delovanje naprave z izbrano glavno številko odgovoren naš modul. Funkcija *srpc_ksvc_stop* ima ravno nasprotno vlogo in jo moramo klicati, preden modul odstranimo iz jedra.

  Pri funkcijah naprave je najpomembnejša *srpc_ksvc_device_ioctl_rpc_call_dispatch*, ki glede na številko funkcije v RPC paketu kliče implementacije posameznih RPC funkcij. Poleg tega sta tu še *srpc_ksvc_device_open* in *srpc_ksvc_device_release*. Del strežnika v uporabniškem prostoru jih kliče ob začetku svojega izvajanja (ko odpre napravo) oziroma ob koncu izvajanja (ko zapre napravo). Z njima jedrni strežnik nadzoruje, ali je v uporabi, oziroma detektira, kdaj je odjemalec prekinil povezavo.
- srpc_usvc_PROG.cc, del strežnika v uporabniškem prostoru. Najprej s funkcijo *srpc_usvc_start* odpremo napravo za komunikacijo z jedrnim strežnikom

ter TCP/IP "socket". Nato čakamo, da se odjemalec poveže. Ko je povezava vzpostavljena, se v *main* funkciji vsak prejet RPC paket pošlje v jedro. Jedro nato vrne povratni RPC paket, ki ga pošljemo odjemalcu.

V primeru napake pri TCP/IP komunikaciji se uporabniški del strežnika avtomatično ponovno zažene.

4.3 Razširjanje ciljnih C datotek

Vsako izmed C datotek razširja drug *codeworker* skript. Imena teh skript se začno s srpcgen_expand_srpc. Najpomembnejše (in tudi najdaljše) so:

- srpcgen_expand_srpc_PROG_h.cwt. Tu je najpomembnejša koda, ki generira definicije struktur s parametri za vsako RPC funkcijo.
- srpcgen_expand_srpc_clnt_PROG_cc.cwt. Ta skript generira kodo za odjemalčeve krne.
- srpcgen_expand_srpc_ksvc_PROG_c.cwt. Tu je pomembna predvsem:
  - Implementacija klicanja prave RPC funkcije glede na številko RPC funkcije v RPC paketu – *expand_ksvc_switch_procedure*
  - Implementacija (krnov) RPC funkcij – *expand_ksvc_stubs_implementation.* Ti izvajajo "demarshaling" podatkov/parametrov, klic dejanske implementacije RPC funkcije ter shranjevanje izhodnih parametrov v povratni RPC paket.

5. Primeri uporabe

Primer uporabe se začne s programom tipa "Hello world" (podpoglavja 5.1 do 5.5). Ko ta deluje, predstavlja večino dela samo še dodajanje novih funkcij (podpoglavje 5.6). Nekoliko zahtevnejši programi potrebujejo kompleksnejše podatkovne tipe (polja, kazalci, strukture). Uporaba le-teh je opisana v 5.7. Podpoglavje 5.8 prikazuje prenos odjemalca v Windows OS.

Podpoglavje 5.9 opisuje implementacijo knjižnice ulfeHapticAPI z uporabo *srpcgen-a*. UlfeHapticAPI je aplikacija, zaradi katere smo se odločili za razvoj *srpcgen-a*.

Posebej v prvem poizkusu programiranja se hitro zmotimo, sporočila o napakah pa so pogosto nerazumljiva. Zato je v primerih uporabe namerno prikazanih tudi nekaj napak ter povezanih sporočil o napakah.

Nekateri primeri kode in izpisov s konzole vsebujejo vrstice, ki so daljše od širine strani. V teh primerih smo vrstice oštevilčili.

5.1 Definicija vmesnika

Za prvo nalogo si zadajmo izvedbo strežnika, ki lahko sprejme in nato tudi vrne število. Celoten vmesnik strežnik – odjemalec bosta zato sestavljali samo dve funkciji.

```
void setNumber( double num );
double getNumber( void );
```

Vmesnik definiramo v datoteki example1.x. Uporabljena je končnica .x, tako kot pri rpcgen. Uporabljamo lahko C ali C++ komentarje, tj. "/* komentar v več vrsticah */" ali "// komentar do konca vrstice". Celoten opis vmesnika je potem:

```
/* file example1.x */
program example1 {
  version example1_version
  {

    void setNumber( double num ) =1;
    double getNumber( void );

  } = 0x001; // program version 0.01
} = 1; // program number 1
```

Na začetku povemo, da je ime programa example1. Sledi definicija verzije programa, nato pa prototipi funkcij s številko funkcije. Na koncu definiramo še "identifikacijsko" številko programa in verzijo programa. Vsak program mora uporabljati drugačno številko programa.

Verzijo programa povečamo, če spremenimo "application binary interface – ABI", npr. prototipu funkcije dodamo, odvzamemo ali spremenimo vrstni red parametrov, ali pa če spremenimo številke funkcij. Odjemalec in strežnik morata biti prevedena z isto številko programa in verzijo programa, v nasprotnem primeru pride do napake.

Številke funkcij se začnejo z 1 (ali več), številka 0 pa je rezervirana, tako kot pri SunRPC. Dovoljeno je določiti samo številko prve funkcije, nato je vsaki naslednji funkciji dodeljena za 1 višja številka od predhodne. To je drugače kot pri SunRPC, kjer je potrebno ročno napisati številko za vsako funkcijo. Zapis števila je lahko v desetiški ali v šestnajstiški obliki.

Prototipi funkcij uporabljajo podobno sintakso kot jezik C. Parametri so lahko poleg osnovnih tipov (void, char, short, int, long, float in double) tudi kazalci, polja ali strukture; več o tem je v 3.4.1 in 5.7.

5.2 Pretvorba definicije vmesnika v krne

Datoteko "example1.x" je potrebno pretvoriti v C kodo. Predpostavljamo, da je program codeworker že pravilno nameščen. V nasprotnem primeru ga dobimo na [2], skupaj z navodili za namestitev in dokumentacijo. Zaženemo

```
codeworker srpcgen.cws example1.x -I ../../srpcgen/ --varexist
```

Datoteka srpcgen.cws je t. i. vodilni skript, "-I ../../srpcgen/" pa je pot do direktorija s srpcgen.cws. Srpcgen v prvi fazi prebere example1.x, nato pa generira datoteke v jeziku C, vzorčen Makefile in vzorčen skript za zagon in ustavitev strežnika. Celoten spisek na novo ustvarjenih datotek je:

- example1.sh, skript za zagon strežnika,
- Makefile, makefile za Linux make ter
- datoteke s C kodo:
  - srpc_example1.h,
  - srpc_clnt.cc,
  - srpc_clnt.h,
  - srpc_clnt_example1.cc,
  - srpc_clnt_example1.h,
  - srpc_ksvc_example1.c,
  - srpc_svc.h in
  - srpc_usvc_example1.cc.

5.3 Implementacija strežnika

Da dejansko implementiramo strežnik, moramo popraviti kodo v srpc_ksvc_example1.c. Če poizkusimo prevesti celotno kodo z ukazom make, dobimo naslednji izpis:

```
1 [justinc@fokker example1]$ make
2 gcc -MD -D__KERNEL__ -Wall -Wstrict-prototypes -Wno-trigraphs -fno-s
  trict-aliasing -fno-common -pipe -mpreferred-stack-boundary=2 -march
  =i686 -DMODULE -DMODVERSIONS -include /usr/src/linux/include/linux/m
  odversions.h -D_LOOSE_KERNEL_NAMES -O2 -I/usr/src/linux/include -c s
  rpc_ksvc_example1.c -o srpc_ksvc_example1.o
3 srpc_ksvc_example1.c: In function ‘implement_srpc_ksvc_example1_setN
  umber’:
4 srpc_ksvc_example1.c:136: warning: implicit declaration of function
  ‘setNumber’
5 srpc_ksvc_example1.c: In function ‘implement_srpc_ksvc_example1_getN
  umber’:
```

```
6 srpc_ksvc_example1.c:144: warning: implicit declaration of function
  ‘getNumber’
7 g++ -MD -Wall -lnetsocket srpc_usvc_example1.cc -o srpc_usvc_example1
8 g++ -MD -Wall -lnetsocket srpc_clnt.cc srpc_clnt_example1.cc -o srpc
  _clnt_example1
```

Ker še nismo implementirali funkcij v strežniku, smo dobili opozorili glede nedeklariranih (tj. manjkajočih, oz. implicitno deklariranih) funkcij *setNumber* (datoteka srpc_ksvc_example1.c, vrstica 136) in *getNumber* (datoteka srpc_ksvc_example1.c, vrstica 144). Vrstici z napako datoteki srpc_ksvc_example1.c najdemo v izpisu prevajalnika, potem ko zaženemo make. Del kode v srpc_ksvc_example1.c, ki je “kriv” za težave, je izpisan spodaj.

```
//##markup##"ksvc_stubs_implementation"
//##begin##"ksvc_stubs_implementation"
static void implement_srpc_ksvc_example1_setNumber(double num) {
//##protect##"implement_srpc_ksvc_example1_setNumber"
/***** ENABLE COPY-PASTE FROM *.x FILE *****/
  setNumber( num );
//##protect##"implement_srpc_ksvc_example1_setNumber"
}

static double implement_srpc_ksvc_example1_getNumber(void) {
//##protect##"implement_srpc_ksvc_example1_getNumber"
/***** ENABLE COPY-PASTE FROM *.x FILE *****/
  return getNumber(  );
//##protect##"implement_srpc_ksvc_example1_getNumber"
}
```

Neobičajni komentarji oblike “//##...” označujejo dele kode, ki jih ureja srpcgen oz. codeworker. Predvsem moramo vedeti, da so vse spremembe v označeni sekciji (tj. koda med “//##begin##” in “//##end##”) izgubljene, ko naslednjič zaženemo codeworker/srpcgen. Izjema je le koda v zaščiteni sekciji (tj. koda med prvim “//##protect##” in drugim “//##protect##” v eni označeni sekciji).

Sedaj lahko ali

- definiramo funkciji *setNumber* in *getNumber* kot samostojni funkciji, v samostojni .c datoteki ali pa
- zamenjamo klica teh funkcij s kodo, ki bi jo sicer uporabili v obeh funkcijah.

Prvi način je pravilnejši, ker omogoča lažje testiranje kode strežnika[1]. Drugi način je enostavnejši, zato bomo uporabili to različico. Zato kodo popravimo tako kot spodaj:

```
static double s_dStoredNumber=0;

//##markup##"ksvc_stubs_implementation"
//##begin##"ksvc_stubs_implementation"
static void implement_srpc_ksvc_example1_setNumber(double num) {
//##protect##"implement_srpc_ksvc_example1_setNumber"
  s_dStoredNumber = num;
//##protect##"implement_srpc_ksvc_example1_setNumber"
}

static double implement_srpc_ksvc_example1_getNumber(void) {
//##protect##"implement_srpc_ksvc_example1_getNumber"
  return s_dStoredNumber;
//##protect##"implement_srpc_ksvc_example1_getNumber"
}
```

Make bi moral zdaj prevesti program brez opozoril ali napak.

5.4 Zagon strežnika

Za zagon strežnika uporabimo skript example1.sh. Za začetek je v njem pomembna predvsem vrstica

```
USER_SERVER_TCP_PORT=1234
```

Z njo določimo TCP/IP vrata, kjer posluša strežnik (tisti del, ki se izvaja v uporabniškem prostoru – koda v datoteki srpc_usvc_example1.cc). Privzeta vrata 1234 bodo povsem primerna. Vrata lahko spremenimo, če bo na istem računalniku hkrati teklo več različnih strežnikov. Zdaj zaženemo strežnik z ukazom

[1] Koda izvoženih RPC funkcij za strežnik je v samostojni datoteki. Če odjemalca (ki je program v uporabniškem prostoru) povežemo "linkamo" s to kodo namesto s kodo odjemalčevih krnov RPC funkcij, se RPC funkcije spremene v lokalne in lahko jih preprosto razhroščujemo. Težava pa je, da so RPC funkcije namenjene izvajanju v jedru. Verjetno bomo uporabljali funkcije ali funkcionalnost, ki je na voljo samo v jedru (v nasprotnem primeru je bolje implementirati strežnik v uporabniškem prostoru), tako da implementiranih RPC funkcij v uporabniškem prostoru ne moremo prevesti, izvajati ali pa oboje.

```
 1  [justinc@fokker example1]$ ./example1.sh start
 2  USER_SERVER_FILE      = srpc_usvc_example1
 3  USER_SERVER_HOST_NAME = fokker.robonet
 4  USER_SERVER_TCP_PORT  = 1234
 5  USER_SERVER_CMD       = ./srpc_usvc_example1 fokker.robonet 12
    34
 6  Starting srpc_ksvc_example1.o: Warning: loading /home/justinc/
    srpc_ksvc_example1.o will taint the kernel: no license
 7    See http://www.tux.org/lkml/#export-tainted for information
    about tainted modules
 8  Module srpc_ksvc_example1 loaded, with warnings
 9  /dev/example10 major number should be:  254
10  Charackter device file /dev/example10 does note exist jet.
11  Create it AS ROOT !: Password:
12  ls -l /dev/example10:
13  crw-rw-rw-    1 root     root     254,   0 Apr 13 09:03 /dev/e
    xample10
14  [justinc@fokker example1]$
```

Ker strežnik zaganjamo prvič, nas skript opozori, da naprava dev/example10 še ne obstaja. Da jo ustvarimo, moramo najprej vnesti root geslo. Naprava dev/example10 je t. i. znakovna naprava. Strežnik jo uporablja za komunikacijo med jedrom in uporabniškim prostorom. Naprava ima privzeto glavno številko ("major number") 254 in pomožno številko 0 ("minor number"). Če bi potrebovali hkratno izvajanje več strežnikov, bi morali različni strežniki uporabljati različne glavne številke. Več o tem najdemo v [11], poglavje 3. V vrstici 9 vidimo, da program pričakuje glavno številko naprave 254. V vrstici 13 pa z izpisom ukaza "ls" vidimo dejansko glavno številko naprave (znova 254) - ta se mora ujemati s pričakovano.

Izvajanje strežnika lahko preverimo z "lsmod" in "ps afx" ukazoma. Tu je primer izpisa:

```
[justinc@fokker example1]$ lsmod | grep srpc_ksvc_example1
srpc_ksvc_example1      6344   1
[justinc@fokker example1]$ ps afx | grep srpc_usvc_example1
 2063 pts/2    S      0:00  |           \_ grep srpc_usvc_example1
 2055 pts/2    S      0:00  \_ ./srpc_usvc_example1 fokker.robonet 1234
```

Skript example1.sh preusmeri stdout in strerr programa srpc_usvc_example1 v datoteki srpc_usvc_example1.log in srpc_usvc_example1.err. Če so kakšne težave z komunikacijo preko mreže, si lahko ogledamo ti dve datoteki.

5.5 Implementacija odjemalca

Popraviti moramo še kodo odjemalca. Odpremo datoteko srpc_clnt_example1.cc in si ogledamo funkcijo *main*.

```
int main(int argc, char* argv[]) {
  char *hostname;
  unsigned short port;
  int ret=0;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s <hostname> <port>\n", argv[0]);
    return(-1);
  }

  hostname = argv[1];
  port = atoi(argv[2]);

  if( (ret=srpc_clnt_start(hostname, port)) )
    return ret;

  int sum=-1;
  int cnt=0;
  while( 1 )
  {
    // in .x file is "int sum3(int a, int b, int c)=?;"
    //sum = add3(1,5,cnt);
#ifdef DEBUG_SRPC_CLNT
    printf( "add3(1,5,%d)=%d\n", cnt, sum);
#endif // DEBUG_SRPC_CLNT
    if(++cnt == CNT_MAX) break;
  }

  // Close the TCP/IP connection
  srpc_clnt_stop(); //delete s_pLink;
  return(0);
}
```

Odjemalec pričakuje dva parametra v ukazni vrstici, prvi je ime računalnika, kjer se izvaja strežnik (npr. fokker.robonet), drugi pa so TCP/IP vrata, na katerih posluša strežnik (1234). Vzorčna koda najprej odpre povezavo do strežnika s klicem *srpc_clnt_start(hostname, port)*. Nato se v while zanki izvajajo dejanski RPC klici. Na koncu zapremo povezavo do strežnika s klicem *srpc_clnt_stop*.

Zdaj lahko popravimo telo while zanke. Najprej shranimo številko v strežnik, nato pa jo preberemo in izpišemo.

```
  int cnt=0;
  while( 1 )
  {
    double number=0;
    setNumber( cnt*10 );
    number = getNumber();
    printf( "number = %f\n", number );
    if(++cnt == CNT_MAX) break;
  }
```

Ponovno prevedemo program in zaženemo odjemalca. Na konzolo se izpiše:

```
[justinc@fokker example1]$ ./srpc_clnt_example1 fokker.robonet 1234
number = 0.000000
number = 10.000000
number = 20.000000
number = 30.000000
number = 40.000000
[justinc@fokker example1]$
```

S tem smo dobili delujoč strežnik in odjemalec. Delovanje odjemalca lahko preverimo še s kakšnega drugega računalnika. Tam bomo verjetno morali še enkrat prevesti odjemalca, drugače pa ne bi smelo biti razlik.

5.6 Dodajanje novih funkcij

Za vsako novo funkcijo moramo najprej vnesti njen prototip v *.x file. Dodajmo funkcijo, ki vrne vsoto dveh števil. Popravljen example1.x izgleda kot:

```
program example1 {
  version example1_version
  {

    void setNumber( double num ) =1;
    double getNumber( );
    double add_two_numbers( double num1, double num2 );

  } = 0x001; // program version 0.01
} = 1; // program number
```

Da bo nova funkcija prisotna tudi v C kodi, moramo ponovno zagnati srpcgen:

```
codeworker srpcgen.cws example1.x -I ../../srpcgen/ --varexist
```

Ker zdaj datoteke s C kodo že obstajajo, srpcgen spremeni samo označene dele datotek, pri čemer ostanejo zaščitene sekcije nespremenjene. Znova moramo popraviti novo funkcijo *implement_srpc_ksvc_example1_add_two_numbers* v datoteki srpc_ksvc_example1.c.

```
1  static double implement_srpc_ksvc_example1_add_two_numbers(dou
   ble num1, double num2) {
2  //##protect##"implement_srpc_ksvc_example1_add_two_numbers"
3    return num1 + num2;
4  //##protect##"implement_srpc_ksvc_example1_add_two_numbers"
5  }
```

V odjemalcu moramo nato samo še uporabiti funkcijo *add_two_numbers*. V while zanko (funkcija *main*, datoteka srpc_clnt_example1.x) dodamo

```
1  printf( " %g = add_two_numbers(%g, %g)\n", add_two_numbers(100
   .0, cnt), 100.0, (double)cnt );
```

Prevedemo celoten program, nato ponovno zaženemo ("restart") strežnik in odjemalca. Pričakovan je naslednji izpis na konzolo:

```
 1  [justinc@fokker example1]$ make 1>/dev/null
 2  [justinc@fokker example1]$ ./example1.sh restart
 3  Stopping srpc_usvc_example1: ./example1.sh: line 104:  2318 Te
    rminated              nohup $USER_SERVER_CMD >$USER_SERVER_LOG
     2>$USER_SERVER_ERR
 4
 5  Stopping srpc_ksvc_example1:
 6  USER_SERVER_FILE      = srpc_usvc_example1
 7  USER_SERVER_HOST_NAME = fokker.robonet
 8  USER_SERVER_TCP_PORT  = 1234
 9  USER_SERVER_CMD       = ./srpc_usvc_example1 fokker.robonet 12
    34
10  Starting srpc_ksvc_example1.o: Warning: loading /home/justinc/
    srpc_ksvc_example1.o will taint the kernel: no license
11    See http://www.tux.org/lkml/#export-tainted for information
    about tainted modules
```

```
12  Module srpc_ksvc_example1 loaded, with warnings
13  /dev/example10 major number should be:  254
14  ls -l /dev/example10:
15  crw-rw-rw-    1 root     root     254,   0 Apr 13 09:03 /dev/e
    xample10
16  [justinc@fokker example1]$ ./srpc_clnt_example1 fokker 1234
17  number = 0.000000
18  100 = add_two_numbers(100, 0)
19  number = 10.000000
20  101 = add_two_numbers(100, 1)
21  number = 20.000000
22  102 = add_two_numbers(100, 2)
23  number = 30.000000
24  103 = add_two_numbers(100, 3)
25  number = 40.000000
26  104 = add_two_numbers(100, 4)
27  [justinc@fokker example1]$
```

5.7 Uporaba novih podatkovnih tipov, kazalcev in polj

Do zdaj smo uporabljali samo enostavne podatkovne tipe. V kompleksnejših programih uporabljamo tudi kazalce, polja, strukture, oštevilčene (enumerirane, "enumerated") tipe, "tipe" definirane z "#define" itd. Srpcgen dovoljuje tudi takšne tipe, vendar z določenimi omejitvami:

- Kazalci so lahko samo enojni (tj. samo ena *).
- Polja morajo biti fiksne dolžine, lahko pa so večdimenzionalna.
- Strukture ne smejo vsebovati kazalcev. Lahko pa vsebujejo polja ali druge strukture.

Na začetku datoteke example1.x moramo deklarirati imena novih tipov. Poglejmo primer, kjer deklariramo ime za polje treh double števil (TTVec3 – vektor treh števil), za "oštevilčen" podatkovni tip (enumOperation_t) in za strukturo (SResult_t), nato pa dodamo še funkcijo *math_operation*.

```
1  typedef_array TTVec3;
2  typedef_simple enumOperation_t;
3  typedef_simple SResult_t;
4
```

```
 5  program example1
 6  {
 7    version example1_version
 8    {
 9
10      void setNumber( double num ) =1;
11      double getNumber( );
12      double add_two_numbers( double num1, double num2 );
13      int math_operation( TTVec3 vec, enumOperation_t oper, SRes
    ult_t *pRes );
14
15    } = 0x001; // program version 0.01
16  } = 1; // program number
```

Srpcgen bo uspešno generiral kodo na osnovi zgornje definicije. Vendar pa moramo dodati še dejanske definicijo novih tipov za C prevajalnik. Ker te definicije potrebujeta tako strežnik kot tudi odjemalec, je primerno mesto datoteka srpc_example1.h. Definicije dodajmo takoj za koncem označene sekcije "begin_file_included".

```
//##end##"begin_file_included"

typedef double TTVec3[3];
typedef enum enumOperation_t
{
  enumOperation_SUM     = 1,
  enumOperation_PRODUCT = 2
} enumOperation_t;
typedef struct SResult_t
{
  double result;
} SResult_t;

#ifndef max
```

Napišemo še dejansko implementacijo funkcije *math_operation* za strežnik, se pravi, da popravimo funkcijo *implement_srpc_ksvc_example1_math_operation*. Ta bo glede na vrednost parametra *oper* v *pRes* vpisala vsoto ali pa produkt elementov *vec*. V primeru uspešne izvršitve vrne funkcija 0, v primeru napake (*oper* ima neznano vrednost) pa negativno število.

```
 1  static int implement_srpc_ksvc_example1_math_operation(TTVec3
    vec, enumOperation_t oper, SResult_t *pRes)
```

```
 2 {
 3 //##protect##"implement_srpc_ksvc_example1_math_operation"
 4   switch( oper )
 5   {
 6   case enumOperation_SUM:
 7     pRes->value = vec[0] + vec[1] + vec[2];
 8     return 0;
 9   case enumOperation_PRODUCT:
10     pRes->value = vec[0] * vec[1] * vec[2];
11     return 0;
12   default:
13     pRes->value = 0;
14     return -1;
15   }
16 //##protect##"implement_srpc_ksvc_example1_math_operation"
17 }
```

Testna koda za odjemalca je:

```
TTVec3 vec = {0, 2, 3};
vec[0] = cnt;
SResult_t result;
printf("vec = {%g, %g, %g}\n", vec[0], vec[1], vec[2] );
math_operation( vec, enumOperation_SUM, &result );
printf("  sum = %g\n", result.value);
math_operation( vec, enumOperation_PRODUCT, &result );
printf("  product = %g\n", result.value);
```

5.8 Prenos odjemalca v Windows OS

Poizkusimo prevesti obstoječega odjemalca za uporabo v Windows OS. Predpostavimo, da smo libnetsocket že prevedli in da se knjižnica nahaja v direktoriju *U:/libnetsocket/libnetsocket-0.3.1*. Zaženemo Visual Studio 6.0 in izberemo "New/Win32 console application". Projekt naj ima ime "win32_client", nahaja pa naj se v direktoriju *example1/win32_client*. Kliknemo "Next", izberemo "A Hello world application", nato pa "Finish".

Najprej moramo popraviti nekatere nastavitve. Gremo v meni *Project/Settings*, kjer:

- V zavihku *C/C++*, kategorija *code generation* nastavimo *struct member alignment* na 4 byte (namesto privzetih 8).

- V zavihku *C/C++*, kategorija *Preprocessor* dodamo k *Additional include directories* še direktorija *U:/libnetsocket/libnetsocket-0.3.1* in *../* (vmes vstavimo vejico).

- V zavihku *C/C++*, kategorija *Precompiled headers* pa izberemo "Not using precompiled headers" ali pa "Automatic use of precompiled headers".

- V zavihku *Link*, kategorija *General* dodamo k *Object/Library modules* datoteki *libnetsocket.lib* in *Ws2_32.lib*

- V zavihku *Link*, kategorija *Input* dodamo pod *Additional Library Path* še *U:/libnetsocket/libnetsocket-0.3.1/Debug*

Kliknemo OK. Program bi se moral zdaj uspešno prevesti in zagnati.

V projekt moramo dodati še odjemalčevi datoteki srpc_clnt.cc in srpc_clnt_example1.cc. Ker Visual Studio sprejme samo datoteke s končnico .c ali .cpp, moramo za obe datoteki ali:

- narediti t. i. "soft link" (ukaz *ln -s srpc_clnt.cc srpc_clnt.cpp*) z novo končnico .cpp na originalno datoteko ali pa

- spremeniti imena originalnih datotek in popraviti Linux Makefile.

Sam sem izbral prvo možnost. Je pa druga možnost nekoliko boljša. Če namreč z Visual Studio urejevalnikom shranimo datoteko (ki je v resnici "soft link"), se lahko zgodi, da bo ustvarjena nova kopija datoteke, ki ima ime "soft link-a", povezava pa bo zaradi tega prekinjena.

Nato dodamo srpc_clnt.cpp in srpc_clnt_example1.cpp v projekt (meni *Project/Add to project/Files*). Ker datoteka srpc_clnt_example1.cc že vsebuje funkcijo *main*, moramo zakomentirati funkcijo *main*, ki je v win32_client.cpp.

Nato poizkusimo prevesti program. Dobimo naslednjo napako:

```
1  srpc_clnt_example1.cpp
2  u:\fokker\k2u_rpc\examples-srpcgen\example1\srpc_clnt.h(11) :
   fatal error C1083: Cannot open include file: 'net/tcpsocket.h'
   : No such file or directory
```

Težava je v tem, da se "include" datoteke knjižnice v Linux-u nahajajo v standardnih poteh, v poddirektoriju net/, v Windows pa ne. Prevajalnik zato ne more najti datoteke "net/tcpsocket.h". Odpremo datoteko srpc_clnt.h in popravimo vrstico

```
#include "net/tcpsocket.h"
```

v

```
#ifdef WIN32 // Windows
#  include "tcpsocket.h"
#else // Linux
#  include "net/tcpsocket.h"
#endif
```

Pri naslednjem poizkusu prevajanja dobimo napako *definition of dllimport function not allowed* za vsako RPC funkcijo. Do tega pride zaradi definicije makroja *FUNCTION_IMP_EXP* v datoteki srpc_clnt_example1.h. Vrednost tega makroja namreč predpostavlja, da je odjemalec v Windows sestavljen iz (1) dll knjižnice, ki jo nato uporablja (2) končni program. Ker je naš primer enostavnejši, je potrebno samo spremeniti vrstico

```
#    define FUNCTION_IMP_EXP __declspec(dllimport)
```

v

```
//#    define FUNCTION_IMP_EXP __declspec(dllimport)
#    define FUNCTION_IMP_EXP
```

Makro *FUNCTION_IMP_EXP* je zdaj prazen in nima nobenega vpliva.

Program bi se zdaj moral uspešno prevesti. Naslednji korak je zagon programa v razhroščevalniku. Ob zagonu moramo podati 2 argumenta – ime računalnik s strežnikom in TPC/IP vrata, kjer strežnik posluša. Zato gremo še v meni *Project/settings*, in pod *Debug/Program arguments* vnesemo "fokker 1234". Nato zaženemo program (tipka F11). Na zaslon moramo dobiti enak izpis kot pri Linux odjemalcu.

5.9 Zasnova knjižnice ulfeHapticAPI

5.9.1 Zasnova haptičnih objektov na strani strežnika

Knjižnica ulfeHapticAPI omogoča uporabo haptičnih objektov v jedrnem prostoru strežnika iz uporabniškega prostora odjemalca. Poglejmo si najprej zasnovo haptičnih objektov na strani strežnika.

Različne vrste haptičnih objektov je smiselno organizirati v hirearhijo C++ razredov. Najosnovnejši razred HapticObject_t ima lastnosti, kot so pozicija, orientacija, debelina notranje in zunanje stene, trdota notranje in zunanje stene itd. Izpeljani objekti dodajo svoje specifične lastnosti. Krogla (HapticSphere_t) ima lastnost radij, kvader (HapticBlock_t) pa dolžino, širino in višino. Funkcije *HapticSphere_SetBaseParameters*, *HapticBlock_SetBaseParameters* itd. omogočajo nastavitev vseh bistvenih lastnosti haptičnega objekta v enem koraku.

Do lastnosti objektov ne dostopamo neposredno, temveč preko nabora funkcij. Primeri funkcij so *hoGetParameter*, *hoSetParameter* in *hoGetForce*; predpona *ho* pomeni "haptic object". Številne funkcije so virtualne (polimorfične). Takšna je npr. funkcija *hoGetForce*, ki izračuna silo, s katero haptični objekt deluje na vrh robota, pa tudi *hoGetParameter* ter *hoSetParameter*, ki morata upoštevati dodane lastnosti izpeljanih objektov.

V Linux jedru ne moremo uporabljati jezika C++. Zato smo C++ razrede "simulirali" v jeziku C. Vsak haptični razred je predstavljen s C-jevsko strukturo ter s funkcijami, ki operirajo nad to strukturo. Na začetku strukture je kazalec na tabelo virtualnih funkcij. Vsak razred ima svojo tabelo virtualnih funkcij, ki si jo delijo vsi haptični objekti tega razreda. Izpeljani razredi dodajo na konec starševskega objekta nove članske spremenljivke. Nove virtualne funkcije dodamo na konec tabele virtualnih funkcij starševskega razreda. Obstoječe funkcije starševskega razreda pa prekrijemo (redefiniramo, angl. "override") z zamenjavo ustreznega elementa v tabeli virtualnih funkcij.

Tako je funkcija *hoGetParameter* različna za kvader (*HapticBlock_hoGetParameter*) in za kroglo(*HapticSphere_hoGetParameter*), čeprav jo za oba objekta kličemo z enako sintakso (z *hoGetParameter*). *HoGetParameter* je v resnici makro, ki kliče funkcijo *virtual_hoGetParameter*, ta pa nato kliče dejansko implementacijo (*HapticBlock_hoGetParameter* ali *HapticSphere_hoGetParameter*) preko kazalca v tabeli vir-

tualnih funkcij. Vse funkcije prejmejo kot prvi parameter kazalec na haptični objekt, nad katerim naj se klicana funkcija izvede.

Zgornja pravila zagotavljajo možnost uporabe funkcij starševskega razreda nad objekti izpeljanega razreda. Funkcije starševskega razreda bodo namreč uporabljale samo tiste dele objekta, ki so del definicije starševskega razreda, preostanek objekta pa bo ignoriran. Če hočemo uporabiti tudi podatke iz preostanka objekta, starševsko funkcijo prekrijemo.

Omeniti moramo še posebnost "družine" funkcij *SetBaseParameters*. Njihov prototip je drugačen za vsak razred (krogla ima en dodaten parameter – radij, kvader pa tri – dolžina, širina in višina), zato ne moremo uporabiti dedovanja in virtualnih funkcij.

5.9.2 Izvoz haptičnih objektov k odjemalcu

Tako definiran programski vmesnik do haptičnih objektov moramo prenesti na stran odjemalca. Vse potrebne funkcije (razen *SetBaseParameters*, ki je drugačna za vsak razred) so virtualne že na strani strežnika. Zato moramo zanje izvoziti samo virtualno funkcijo osnovnega starševskega razreda. *SetBaseParameters* moramo izvoziti za vsak razred posebej.

Vse funkcije pričakujejo kot parameter tudi kazalec na ciljni objekt/strukturo. Tu pa se pojavi možnost, da odjemalec pošlje strežniku neveljaven kazalec, zaradi česar se strežnik zruši. Zato naj odjemalec dostopa do haptičnih objektov preko ročic (angl. "handles"). Vsaka ročica vsebuje poleg indeksa haptičnega objekta še magično število in tip haptičnega objekta, kar omogoča strežniku preverjanje konsistentnosti ročice.

Odjemalec mora najprej vzpostaviti povezavo s strežnikom z *srpc_clnt_start*, nato pa dobi ročico do celotnega virtualnega okolja s klicem *ulfeHapticAPI_GetHapticMaster*. Ročice do posameznih objektov dobimo s klici *HapticMaster_CreateSphere*, *HapticMaster_CreateBlock* ipd.

Do zdaj opisan vmesnik uporablja izključno jezik C. Za C++ vmesnik moramo napisati še minimalen ovitek za vsak razred. Vsak objekt na strani odjemalca shrani samo ročico do objekta na strani strežnika, ki jo nato uporabi pri klicih C-jevskih funkcij. Večina funkcij C++ razredov je trivialnih, kar lepo ilustrira koda, potrebna za razred CUlfeHapticObject_t.

```
class CUlfeHapticObject_t
{
public:
  HAPI_HANDLE m_rhObj;
public:
  CUlfeHapticObject_t()
    { m_rhObj.ind = HAPI_HANDLE_INVALID_INDEX; }
  ~CUlfeHapticObject_t()
    { m_rhObj.ind = HAPI_HANDLE_INVALID_INDEX; }
  virtual int Enable( long duration =0 )
    { return HapticObject_Enable( m_rhObj, duration ); }
  virtual int Disable()
    { return HapticObject_Disable( m_rhObj ); }
  virtual int ClearParameters()
    { return HapticObject_ClearParameters( m_rhObj ); }
  virtual int SetParameter( FCSPARAMETER type, double val)
    { return HapticObject_SetParameter ( m_rhObj, type, val); }
  virtual int SetParameter( FCSPARAMETER type, TTVec3 val)
    { return HapticObject_SetParameter3( m_rhObj, type, val); }
  virtual int GetParameter( FCSPARAMETER type, double &val)
    { return HapticObject_GetParameter ( m_rhObj, type, &val); }
  virtual int GetParameter( FCSPARAMETER type, TTVec3 val)
    { return HapticObject_GetParameter3( m_rhObj, type, val); }
};
```

Nekoliko daljše so le funkcije za kreiranje objektov, ki preverijo veljavnost vrnjenih ročic, preden ustvarijo nov C++ haptični objekt. Kot primer je tu koda za nov haptični blok.

```
  inline CUlfeHapticBlock_t* CreateBlock(
        TTVec3 center, TTVec3 orient, TTVec3 size,
        double extSpringStiffness=1000, double intSpringStiffness=1000,
        double extDampingFactor=1.0,    double intDampingFactor=1.0,
        double extThickness=0.01,       double intThickness=0.01 )
    {
      HAPI_HANDLE rhObj;
      rhObj = HapticMaster_CreateBlock( m_rhHm, center, orient, size,
        extSpringStiffness, intSpringStiffness,
        extDampingFactor,   intDampingFactor,
        extThickness,       intThickness );
      if( rhObj.ind == HAPI_HANDLE_INVALID_INDEX )
        return NULL;
      CUlfeHapticBlock_t *pObj = new CUlfeHapticBlock_t;
      if( pObj ) pObj->m_rhObj = rhObj;
```

```
  return pObj;
}
```

Literatura za dodatek B

[1] RTLinuxFree
http://www.rtlinuxfree.com/

[2] CodeWorker, A universal parsing tool & a source code generator.
http://codeworker.free.fr/

[3] Eclipse IDE.
http://www.eclipse.org/

[4] KORBit: A Kernel Space CORBA ORB.
http://home.case.edu/~ajr9/documents_and_writings/cs423/KORBit/KORBit.html

[5] ORBit2.
http://www.gnome.org/projects/ORBit2/

[6] R. Srinivasan, Sun Microsystems, Inc. RFC 1832: External Data Representation Standard.

[7] Mark McLoughlin, Corba in the Kernel?
http://www.csn.ul.ie/~mark/fyp/fypfinal.html

[8] Rpcgen Programming Guide.
http://www.crypto.com/courses/fall05/cse380/rpcgen1.pdf

[9] Remote Procedure Call Programming Guide.
http://www.crypto.com/courses/fall05/cse380/rpcguide.pdf

[10] Remote Procedure Call and the portmapper daemon.
http://www.bga.org/ lessem/psyc5112/usail/network/services/portmapper.html

[11] Alessandro Rubini, Jonathan Corbet: Linux device drivers. O'Reilly Media, Inc., junij 2001, druga izdaja, ISBN 0-596-00008-1