

Projektni Zadatak

Marko Pušković RA 230/2021
Nemanja Milošević RA 200/2021
Aleksa Stanković RA 219/2021

ROS-less Routines

Operativni sistemi u realnom vremenu

Departman za računarstvo
i automatiku
Odsek za računarsku tehniku
i računarske komunikacije

Mentori:
dr Miloš Subotić
prof. dr Miroslav Popović
Milorad Trifunović

1 Uvod

1.1 Ideja projekta

Ideja ovog projekta je blia dizajnirati i implementirati drajversku aplikaciju koja čita unapred definisane rutinske fajlove, leksički i semantički ih obradjuje, nakon čega upravlja robotskom rukom preko drajvera koji šalju signal ispunе na odredjene GPIO pinove na Raspberry Pi 2.

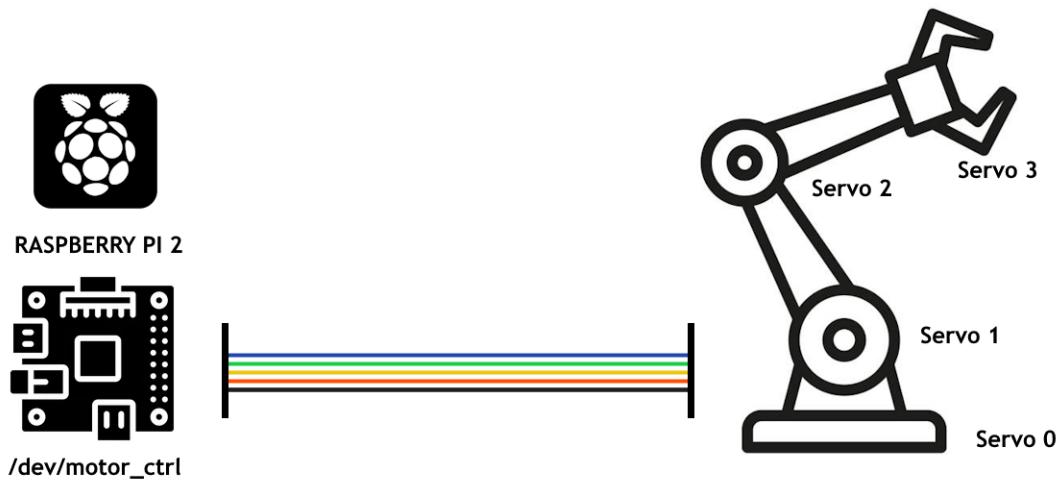


Figure 1: Šematski prikaz rada sistema

1.2 Ciljevi

Glavni ciljevi projekta uključuju:

- Čitanje prethodno definisanih rutinskih datoteka.
- Prevodjenje rutinskih komandi u komandnu strukturu.
- Vremenska interpolacija signala faktora ispunе servo motora.
- Upis bafera u Linux drajver.

2 Prepreke u implementaciji

2.1 Rukovanje greškama

Jedan od znatnijih izazova pri implementaciji je bilo pitanje rukovanja greškama na robustan način. Greške i nepredvidivi slučajevi se mogu desiti u bilo kom delu rada sistema tako da je potrebno praviti ekstenzivne provere funkcionalnosti aplikacije.

Primeri rukovanja greškama:

- Neispravan format unosa sa komandne linije.
- Prilikom otvaranja drajverske datoteke.
- Pri parsiranju rutine.
- Tokom pisanja bafera u drajver.

3 Leksička analiza rutinskih datoteka

3.1 MOVEITCMD sintaksa

Format u kom se rutinske datotke nalaze se zove MOVEITCMD. Ovaj format služi za opis načina komandovanja robotskom rukom. Komande su odvojene novim linijama, što znači da se dve komande ne mogu naći u istoj liniji.

Tipovi podataka koji se pojavljuju u datoteci:

- Broj - definisan kao označen decimalni broj (tipa *double* u ovoj implementaciji).
- Vektor - definisan kao uredjena n-torka brojeva. 2D vektor se predstavlja u formatu [*broj* < *broj* >], dok se 3D vektor predstavlja u format [*broj* < *broj* > < *broj* >].
- Entitet - definiše se kao element iz skupa {HAND, ARM}

MOVEITCMD format definiše sledeće komande:

- **%include <file>** - ova komanda umeće sadržaj falja *<file>* na njeno mesto, čime se postiže modularnost upravljačke rutine.
- **use <entitet>** - ova komanda postavlja entitet kao upravljački objekat (aktuator). Sve naredne naredbe će uticati na ovaj entitet sve dok se aktuator ne promeni.
- **go <vektor>** - ova komanda zadaje odredišne koordinate upravljačkom objektu, odnosno pokreće aktuator.
- **wait <broj>** - ova komanda zaustavlja ceo upravljački sistem na vreme koje je definisano brojem (u sekundama).
- Moguće je definisati promenljive za vektore u rutinskoj datoteci tako što joj se zada ime i vrednost izmedju kojih se nalazi znak jednakosti. Primer: **home = [0 0 0]**
- Linije koje počinju tarabom (#) se interpretiraju kao komentar.

Datoteke koje počinju donjom crtom (_) se uglavnom ne koriste kao ulazna tačka rutine nego ozačavaju da se ona koristi kao modul u drugim datotekama.

3.2 Proces analize

Leksička analiza datoteke se sastoji iz niza procesa koji osiguravaju da se ona uspešno parsira u skup tokena, odnosno komandi, kao i da vodi računa o sintakšičkim greškama i obavesti korisnika o istim. Kao prvo, program potražuje od operativnog sistema pristup rutinskoj datoteci za čitanje, nakon čega se datoteka čita liniju po liniju, i kako po definiciji sintakse, svaka linija se smatra naredbom, program se vodi činjenicom da se svaka linija može interpretirati kao jedna komanda, te je tako i obradjuje.

Postoji 7 slučajeva koji definišu ispravnu komandu:

Ako linija počinje sa ”%**include**”, program predpostavlja da se radi o *include* komandi, odnosno proverava da li je datoteka koja je prosledjena kao parametar validna, i ukoliko jeste, program se poziva rekursivno nad tom datotekom.

Ukoliko linija počinje sa ”**go**”, pretpostavljamo da se radi o *go* komandi, tako da uzimamo vrednost promenljive paramatra ove komande kao koordinate njenog odredišta.

Istom analogijom se procesuira i ”**use**” komanda. Entitet koji je prosledjen uz ovu naredbu se koristi kao aktuator za sve naredne komande.

Ako linija na početku sadrži ”**wait**”, radi se o komandi čekanja, tako da očekujemo da je njen parametar broj sekundi za koje je potrebno zaustaviti proces.

Napomene: Ukoliko linija počinje tarabom (#) ili karakterom nove linije (\n), linija se kompletно preskače i prelazi se na sledeću. Ovaj proces se ponavlja sve dok se ne dodje do kraja datoteke ili ako se naidje na grešku u sintaksi datoteke.

Tip naredbe	Očekivani parametar
include	file
go	vector
use	entity
wait	seconds

Table 1: Očekivani parametar za tipove naredbi

3.3 Generisanje tokena

Glavni cilj leksičke analize jeste da generiše skup tokena koji proizilaze iz tekstualne reprezentacije datoteke. Radi jednostavnosti procesa, skup tokena se može uprostiti tako da svaki token predstavlja jednu komandu koja ima **tip**, **entitet** i **vektor**.

- Tip - predstavlja o kojoj komandi se radi. U ovoj implementaciji može biti **GO** ili **WAIT** ukoliko je naredba pokreta ili čekanja respektabilno.
- Entitet - govori koji deo robotske ruke je aktuator za komandu u kojoj se nalazi. Ovaj podatak predstavlja skup servo motora koje je potrebno pokretati. Prilikom komande čekanja, ovaj parametar se ignoriše.
- Vektor - sadrži uredjenu n-torku brojava. Ako se radi o naredbi pokreta, vektor se interpretira kao odredišne koordinate servo motora. Ukoliko je reč o naredbi čekanja, prvi broj u skupu sadrži vreme koje je potrebno čekati u sekundama.

4 Semantička obrada

4.1 Analiza tokena

Nakon uspešno završenog procesa generisanja tokena, potrebno ih je analizirati i nagovestiti programu kako da ih izvrši. Sam proces analize nije toliko složen. Sastoji se od prolaska kroz sve komande, od prve do poslednje, nakon čega program određuje u kom stanju sistem treba da se nalazi.

U slučaju da je tip komandnog tokena naredba čekanja, sistem ulazi u stanje čekanja na vreme određeno parametrima komandnog tokena. Ako je tip komandnog tokena naredba pokreta, sistem beleži parametarski vektor kao novo odredište niza faktora ispune i prelazi u stanje interpolacije. Tokeni se analiziraju sve dok se ne dostigne poslednji token ili dok ne dodje do greške prilikom parsiranja.

5 Programska implementacija

5.1 Hjjerarhija projekta

Programsko rešenje ovog projekta je implementirano u programskom jeziku C na platformi Raspberry Pi 2. Operativni sistem je Raspbian, što je distribucija bazirana na Debian sistemu. Procesorska arhitektura je ARM sa 32-bitnim instrukcijama.

Struktura projekata je sledeća:

- ROS/
 - arm_and_chassis_ws/ - Sadrži rutinske fajlove koji određuju putanju robotske ruke. Puna putanja direktorijum je: **ROS/arm_and_chassis_ws/src/common_teleop/routines/s3a/**
- SW/
 - Driver/ - Sadrži sav izvorni kôd koji je zadužen za rad drajvera, koji se ubacuje direktno u kernelski prostor.
 - App/ - Sadrži datoteke aplikacije za rad sa rutinama.
 - * **rosless_routine.c**
 - * **parser.h**
 - * **parser.c**

5.2 Datoteka parser.h

Datoteka **parser.h** predstavlja datoteku zaglavljiva koja implementira funkcionalnosti parsiranja rutinskih fajlova. Ona definiše enumeracije za tipove komandi kao i za entitete.

U ovoj datoteci se takođe definišu dve značajne strukture:

- Vektor:

```
typedef struct Vector
{
    double values[3];
    uint8_t length;
} Vector;
```

- Naredba:

```

typedef struct Command
{
    uint8_t type;
    uint8_t entity;
    Vector vector;
} Command;

```

5.3 Datoteka parser.c

U ovoj datoteci su definisani prototipi funkcija iz datoteke **parser.h**

U te funkcije spadaju:

- Glavna funkcija parsiranja rutiske datoteke:

```
CommandNode* parse_routine(const char *file_path, uint8_t *success)
```

Ova komanda vraća pokazivač na prvi element liste svih naredbi koje su se nalazile u datoteci specificiranoj **file_path** parametrom. Referenca **success** nam govori da li je parsiranje bilo uspešno. Postupci koje ova funkcija izvršava su sledeći:

- Čitanje datoteke liniju po liniju.

```
while (fgets(line, sizeof(line), file))
```

- Detektovanje ključnih reči u sintaksi preko komande **strncmp()**. Primer detektovanja *go* naredbe:

```

Command command;
command.type = GO;
command.entity = current_entity;
command.vector = variables[i].vector;

// Add the command to the list of commands
if (!add_command(command))
{
    *success = 0;
    printf("[ERROR] - Failed - to - add - command\n");
    return NULL;
}

```

- Primer detektovanja naredbe čekanja:

```

Command command;
command.type = WAIT;
command.vector.length = 1;
command.vector.values[0] = seconds;

// Add the command to the list of commands
if (!add_command(command))
{
    *success = 0;
    printf("[ERROR] - Failed - to - add - command\n");
    return NULL;
}

```

Kod naredbe čekanja, prvi broj u vektoru se interpretira kao broj sekundi koje je potrebno čekati pa je stoga dužina vektora jednaka jedinici.

- Proces dodavanja nove promenljive, odnosno ažuriranja vrednosti stare:

```

for (uint8_t i = 0; i < num_variables; i++)
{
    if (strcmp( variables [ i ].name, new_variable.name) == 0)
    {
        existing_variable_index = i;
        break;
    }
}

if (existing_variable_index != -1)
{
    variables [ existing_variable_index ] = new_variable;
}
else
{
    variables [ num_variables++ ] = new_variable;
}

```

5.4 Datoteka rosless_routine.c

Ova datoteka je ulazna tačka celog programa. U njoj je implementirana kompletna logika rada aplikacije koja uključuje izvršavanje naredbi i komunikaciju sa drajverom.

Kao prvo, imamo par konstanti definisanih kao predprocesorske direktive.

```

 $\text{\texttt{//}}$ 
#define SPEED HALF_PI
#define FREQUENCY 50
#define T (1.0/FREQUENCY)
#define SPEED_PER_T ((SPEED)*T)
 $\text{\texttt{//}}$ 

```

Brzina je postavljena na $\frac{\pi \text{ rad}}{2 \text{ s}}$, što je ekvivalentno 180° u roku od 2 sekunde.

Frekvencina označava koliko često ćemo osvežavati servo motore odnosno slati im PWM signale u hercima. T predstavlja periodu, a SPEED_PER_T označava put koji je potrebno preći u jednoj periodi da bi se ispoštovala goredefinisana brzina.

Imamo dve funkcije za konvertovanje iz faktora ispune (duty) u radiane i obrnuto.

```

 $\text{\texttt{// Function to convert angle in radians to duty cycle}$ 
uint16_t angle_to_duty(double radians) {
    if (radians < -HALF_PI) radians = -HALF_PI;
    else if (radians > HALF_PI) radians = HALF_PI;
    return (uint16_t)(500 + 400 * radians / HALF_PI);
}

 $\text{\texttt{// Function to convert duty cycle to angle in radians}}$ 
double duty_to_angle(uint16_t duty) {
    return (double)(duty - 500) * HALF_PI / 400;
}

```

Funckija za ispis komande koja se koristi prilikom otklanjanja greški u kôdu:

```
void printCommand(Command command) {
    printf("Command: ");
    if (command.type == GO) {
        printf("GO");
        if (command.entity == ARM) printf("ARM");
        else if (command.entity == HAND) printf("HAND");

        printf("[");
        for (int i = 0; i < command.vector.length; i++) {
            printf("%f", command.vector.values[i]);
            if (i < command.vector.length - 1) printf(", ");
        }
        printf("]\n");
    }
    else if (command.type == WAIT) {
        printf("WAIT");
        printf("%f\n", command.vector.values[0]);
    }
}
```

Provera ispravnosti unosa sa komandne linije:

```
// Check if the file path is provided as a command line argument
if (argc < 2) {
    printf("Usage: %s <file_path>\n", argv[0]);
    return 1;
}
```

Inicijalizacija niza faktora ispune:

```
uint16_t duties[MOTOR_CLTR_N_SERVO] = {500, 500, 500, 500};
```

Pozivanje parserske funkcije u sklopu glavnog programa:

```
uint8_t success;
// Parse the routine from the file
CommandNode *commands = parse_routine(argv[1], &success);
// Check if the routine is parsed successfully
if (!success) {
    printf("[ERROR] - Failed - to - parse - routine\n");
    return 1;
}
```

Čekanje se postiže jednostavnim pozivom komande **usleep()** koja kao parametar zahteva broj mikrosekunda koje program treba da čeka.

Realizacija naredbe pokreta:

```
double s0, s1, s2, s3, s0_target, s1_target, s2_target, s3_target; // Alternatively we
// could have converted the angles to duty cycles during the parsing which would only
// take 2 bytes each

// Convert the duty cycles to angles
s0 = duty_to_angle(duties[0]);
s1 = duty_to_angle(duties[1]);
s2 = duty_to_angle(duties[2]);
s3 = duty_to_angle(duties[3]);

// If the command is for the arm, then set the target angles
if (current->command.entity == ARM) {
    s0_target = current->command.vector.values[0];
    s1_target = current->command.vector.values[1];
```

```

    s2_target = current->command.vector.values[2];
    s3_target = s3;
}
// If the command is for the hand, then set the target angle for the hand
else if (current->command.entity == HAND) {
    s0_target = s0;
    s1_target = s1;
    s2_target = s2;
    s3_target = current->command.vector.values[0];
}

```

Konverzija u signal faktora ispune i prosledjivanje drajveru:

```

duties[0] = angle_to_duty(s0);
duties[1] = angle_to_duty(s1);
duties[2] = angle_to_duty(s2);
duties[3] = angle_to_duty(s3);

printf("duties[0] == %d, duties[1] == %d, duties[2] == %d, duties[3] == %d\n", duties[0],
       duties[1], duties[2], duties[3]);

// Write the duty cycles to the device file
r = write(fd, (char*)&duties, sizeof(duties));
// Check if the write operation is successful
if (r != sizeof(duties)) {
    fprintf(stderr, "[ERROR] - write - went - wrong!\n");
    return 4;
}

```

Oslobadjanje dinamički alocirane memorije:

```

CommandNode *next;
current = commands;
// Free the memory allocated for the commands
while (current != NULL) {
    next = current->next;
    free(current);
    current = next;
}

```

5.5 Ograničenja aplikacije

Aplikacija svojstveno poseduje nekolicinu ograničenja na koje je potrebno paziti. Iako je skup komandi dinamički realizovan kao struktura spregnute liste, ostali skupovi kao na primer skup promenljivih su realizovani kao statički nizovi, konkretno u slučaju promenljivih niz je definisan sa 256 članova, što znači da ne možemo imati više od 2^8 različitih promenljivih. Broj sekundi u narebi čekanja je u predstavi tipa *double* što znači da vreme čekanja programa ne može preći $5.698 \cdot 10^{294}$ godina.

6 Hardver

Kako bi pripremili hardversku stranu projekta, potrebno je povezati GPIO pinove na portove robotske ruke. Raspberry Pi 2 se preko jumper-a povezuje na Arduino koji služi za stabilizaciju signala koji dolazi od strane Raspberry Pi-a. Svaka žica osim crne (GND) predstavlja jedan od četiri servo motora. Signal se kasnije rastavlja na portove koji su direktno povezani na servo motore robotske ruke. (Slika 3.)

```

pi@raspberrypi:~ $ pinout
File Edit Tabs Help
pi@raspberrypi:~ $ pinout
-----+
0000000000000000 J8
1000000000000000 | USB
| USB
| USB
| USB
Pi Model 2B V1.1
| SoC |
| D | | S | | C | | S | | A | Net
| I | | I | | I | | V |
pwr | HDMI | | | | | | | |
-----+
Revision : a02082
SoC : BCM2837
RAM : 1024MB
Storage : MicroSD
USB ports : 4 (excluding power)
Ethernet ports : 1
Wi-fi : True
Bluetooth : True
Camera ports (CSI) : 1
Display ports (DSI) : 1

J8:
 3V3 (1) (2) 5V
GPIO02 (3) (4) 5V
GPIO03 (5) (6) GND
GPIO04 (7) (8) GPIO14
GND (9) (10) GPIO05
GPIO17 (11) (12) GPIO18
GPIO27 (13) (14) GND
GPIO22 (15) (16) GPIO23
 3V3 (17) (18) GPIO24
GPIO10 (19) (20) GND
GPIO09 (21) (22) GPIO25
GPIO11 (23) (24) GPIO08
GND (25) (26) GPIO07
GPIO00 (27) (28) GPIO01
GPIO05 (29) (30) GND
GPIO06 (31) (32) GPIO12
GPIO13 (33) (34) GND
GPIO19 (35) (36) GPIO16
GPIO26 (37) (38) GPIO20
GND (39) (40) GPIO21

For further information, please refer to https://pinout.xyz/
pi@raspberrypi:~ $

```



Figure 2: GPIO pinovi

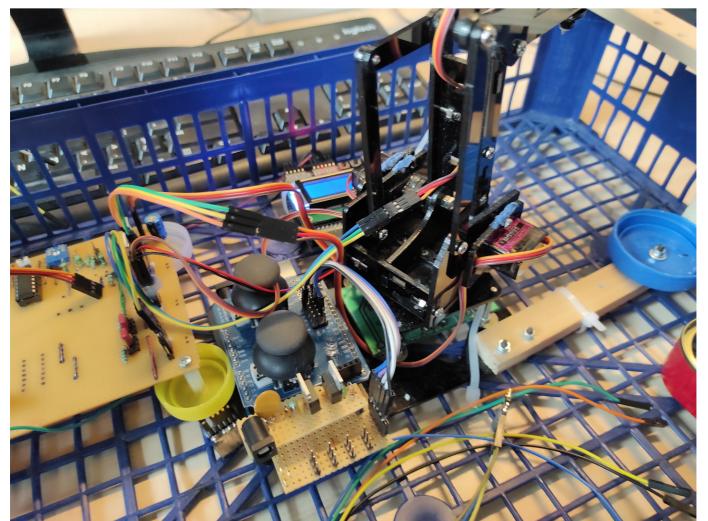
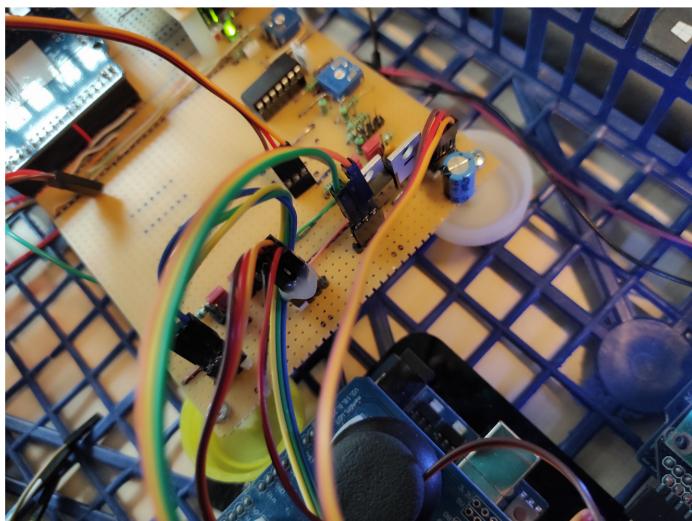


Figure 3: Portovi povezivanja servo motora robotske ruke

7 Kompajliranje i pokretanje aplikacije

Kako bi počeli proces komajliranja, potrebno je pozicionirati se u **ROS/arm_and_chasis_ws** direktorijum.

Kompajliranje se vrši pomoću skripte **tmuxer.py**.

- **./tmuxer.py build_drv**: Inicijalizuje drajversku datoteku, kreira izvršne datoteke za upravljač drajvera kao i za rutinsku aplikaciju.

Pokretanje:

- **./tmuxer.py run_drv**: Pokreće okruženje za rad aplikacije. (Pokretati samo jedanput)

Nakon pokretanja, komandna linija za upravljanje aplikacijom će se pojaviti u gornjem desnom uglu terminala.

Napomena: Budući da je sistem komajliranja i pokretanja realizvan preko terminalskog multipleksera (tmux), potrebno je u bilo koji tmux prozor ukucati **tmux_exit** kako bi završili proces.

8 Zaključak

Ovaj projekat je imao za cilj kreiranje robustne drajverske aplikacije koristeći jezik C, Linux drajvere i hardversku spregu. Uspešna implementacija ovog projekta ne samo da pokazuje praktičnu primenu koncepta upravljanja, već i naglašava važnost bezbednog i pouzdanog rukovanja hardverskim sistemima. Pažljivim razmatranjem potencijalnih problema i implementacijom efikasnih rešenja, stvorili smo otporan program sposoban da kontroliše fizičku robotsku ruku. Uvidi stečeni iz ovog projekta doprinose ne samo oblasti hardverskog upravljanja već i širem razumevanju kernelskog prostora i rada sa drajverima na efikasan način iz stvarnog sveta.

```
#include <stdio.h>

int main() {
    printf("Hvala - na - pažnji !");
    return 0;
}
```