Report - Project Prim

Automatic generation code for Lego EV3 Mindstorm

Thiago Matheus Bruno da Silva

Télécom Paristech Student

Supervisor:

Etienne Borde

1 Abstract

In a world connected as today, we have a progressive number of events occurring on the same time. For example, a system of train transportation or even the air space of airplanes. Controlling all the routes on the same time it's very difficult, especially when each year we need more and more ways of transportation. It's up to engineerings to know new methods to solve this problem.

On this project we're going to modelize it with two robots Lego Ev3 Mindstorm into a single track. This one will have a restricted zone inside between a red and yellow mark, which we are going to call critical section. This zone will only let one robot once and if the other one try to enter should wait outside it. The robots should run on the track following the black line on it. To realize this project was necessary have knowledge of tcp/ip connection, for monitorate both of robots, concurrent programming for to control the flux of critical section and make possible create two TCP/IP connections at the same time and PID control algorithm to follow the black line.

2 Configuration

The brick of Lego Mindstorm EV3 don't come with a OS that makes possible do *ssh connection*. The software we used was the ev3dev, and you can find where install him following the steps on this site:

https://www.ev3dev.org/

There you can also chose the language you are going to use. We used C to code the project below. After you install the ev3dev you have to install the libraries in it. To make this follow the head on the github's suppository:

https://github.com/in4lio/ev3dev-c

To compile the code you have just to link lev3dev-c to the object. You have some examples on the readme. To help, the project's Makefile below:

```
CC = gcc
EV3_LIBS = -lev3dev-c
PTHREAD_LIB = -lpthread
CFLAGS = -c -g
WARNINGS = -Wextra
EXECS_EV3 = light.exe sensor.exe motor.exe uart.exe client1_test.exe
client2_test.exe stop_motor.exe PID_test.exe
```

```
 ------------------------------------------------

ev3: $(EXECS_EV3)

%.exe: %.o
	$(CC) $< $(EV3_LIBS) -o $@

%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

server_test.exe: server_test.o
	$(CC) $< -o $@

server_test.o: server_test.c
	$(CC) $(CFLAGS) $< -o $@

client2_test.exe: client2_test.o
	$(CC) $< -o $@

client2_test.o: client2_test.c
	$(CC) $(CFLAGS) $< -o $@

client1_test.exe: client1_test.o
	$(CC) $< -o $@

client1_test.o: client1_test.c
	$(CC) $(CFLAGS) $< -o $@

endian.exe: endian_test.o
	$(CC) $< -o $@

endian_test.o: endian_test.c
	$(CC) $(CFLAGS) $< -o $@

# ------------------------------------------------

main_client: main_client.o client.o PID.o
	$(CC) $^ $(EV3_LIBS) -o $@

main_client.o: main_client.c client.h PID.h
	$(CC) $(CFLAGS) $< -o $@
```

```
%.o: %.c %.h
	$(CC) $(CFLAGS) $< -o $@

# ------------------------------------------------

main_server: main_server.o server.o
	$(CC) $^ $(PTHREAD_LIB) -o $@

main_server.o: main_server.c server.h

server.o: server.c server.h

# ------------------------------------------------

thread_test: thread_test.o
	$(CC) $^ $(PTHREAD_LIB) -o $@

thread_test.o: thread_test.c


# ------------------------------------------------

server_test: server_test.o

server_test.o: server_test.c

# ------------------------------------------------
.PHONY: ssh clean

ssh:
	ssh robot@ev3dev.local

clean:
	rm -rf *.exe *.o main_client main_server
```

3 Knowledges

3.1 Introduction

To understand a bit better of this project we are going to take a quick breath of each study area that was used to realize this project. These areas were the mentioned before: concurrent programming, socket programming, PID controller.

3.2 Concurrent programming

Concurrent programming is very useful when we have a couple of tasks running and competing for the same thing. A good remark should be made to difference parallel and concurrent programming. The first one, happens when two or more tasks are running on the same time, on a multi-core computer (it's impossible to occur in a single core computer) but without competing. The other one, it's about overlappings process not necessary happening at the same instant.

In our project we used Concurrent programming, because the two robots was competing for the same track and, also, to establish the TCP/IP connection with two client robots at the same time. To implement the concurrent programming we could use a lot o libraries, however as we made the project on C language we had to use the POSIX library.

3.3 Socket Programming

Socket programming it's a way of connecting two (or more) machines to transfer data with each other. We have two types of sockets: stream sockets and datagram sockets.

The first one uses the TCP protocol and it's the most commonly protocol used on internet. It's a two way communication system that guarantees that you will receive your data package (or just package). When the sending socket sends a package for the receiving one, the second socket sends back another packet to signalize the first socket that the message arrived well. As long as this step is completed he can resends another information. It's good to emphasize that a package will be sent just if the previously one has arrived, otherwise the sending socket must resend the information until arrives.

The second one, uses the UDP protocol, which it's a little bit different from the stream types. Here, we have no longer a back-and-forth communication system. The receiving socket doesn't alert the sending if the package arrived and there is no need to resend data. Therefore, if the data does not arrives it will be lost. So what's the purpose of using a protocol like that? When a data is lost, we don't have to resend and wait it arrives and that's a very useful thing. Normally, UDP protocol is used on video streaming for

example. When you are watching a video on YouTube would be very annoying, when the connection is bad, to wait everything load on the same order. But what really happens is that video jumps some seconds forward and this don't prejudice the comprehension of the movie, because it's too fast for our brain.

On this project, we used the TCP protocol because we have to make sure that data will arrive. Lost data, would interfere on the alert of critical section for example. If the information that someone is already there is lost would prejudice all the system.

3.4 PID controller

A PID controller it's a control loop feedback mechanism used to control a system. The progressive-integral-derivative controller, as it's known too, has this name because it's controller system works with a error value, calculated as a difference between setpoint and a measured value, which is corrected by applying a proportional, integral and derivative on it.

The controller is very easy to be implemented especility on this project that doesn't have any delay and the measurement is not so fast to prejudice the calculation. He was used to make the robot follow the black line on the track using a light sensor that reads the luminosity on the ground.

4 Implementation

4.1 Introduction

The main problem of this project is to let just one robot, once, through the critical section. To realize this idea, was needed a communication between the robots, that isn't possible to be made directly. For this reason, was needed to create a TCP/IP connection with a PC which he would be the server and each of the robots a client. Below, I will explain the code used on server and both clients.

4.2 Server

As previously said, the server it's the only way of communication between the two robots. He runs on the terminal of the own PC, differently from client that will be explained after. To enable two connections simultaneously with two clients was needed to utilize two threads, as can be seen below.

```
main_server.c
```

```c
#include "server.h"
#include <stdbool.h>
#include <pthread.h>

int main(void)
{
	pthread_t car1_id, car2_id;
	int sock1_server, sock2_server, status;
	char *hello = "Hello from server";
	struct message *message_red1, *message_red2;
	strcpy(message_red1->car_ip,  "192.168.0.105");
	strcpy(message_red2->car_ip, " ");

	// Make connection with car 1
	sock1_server = establish_server_connection(message_red1->car_ip,
PORT1);

	// Make connection with car 2
	sock2_server = establish_server_connection(message_red2->car_ip,
PORT2);

	// Creating the threads
	status = pthread_create( &car1_id, NULL, thread_send_recv, (void *)
message_red1 );
	if( status != 0 )
	{
		perror("pthread_create");
		exit(EXIT_FAILURE);
	}

	status = pthread_create( &car2_id, NULL, thread_send_recv, (void *)
message_red2 );
	if( status != 0)
	{
		perror("pthread_create");
		exit(EXIT_FAILURE);
	}

	close(sock2_server);
	close(sock1_server);
	return 0;
```

```
}
```

On the code, we can see that two connections are established firstly. In other words, we can say that the *establish_server_connection()* function only goes until the acceptation step and then the program can wait for the moment to send and receive data.

After establishing the connection we have to create the threads, whose will run the function *thread_send_recv(),* that is responsible for sending and receiving data.

server.h

```h
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdbool.h>
#include <pthread.h>
#define PORT1 4444
#define PORT2 8080

struct message
{
      bool flag;
      int  port;
      char *car_ip;
};

// Creates the socket. Makes the Bind. Accept connection from client.
// Returns the socket server
int establish_server_connection(char *client_ip, int port);

// Function to be used for the threads. Change data with client
void *thread_send_recv(void *message_arg);
```

4.2.1 Establishing connection

server.c

```c
#include "server.h"

typedef struct sockaddr Sockaddr;
typedef struct in_addr In_addr;
typedef struct sockaddr_in Sockaddr_in;

int establish_server_connection(char *client_ip, int port, int *status)
{
	int sock_server, opt =1;
	Sockaddr_in address;

	// Socket creation
	if( (sock_server = socket( AF_INET, SOCK_STREAM, 0)) < 0)
	{
		perror("socket");
		exit(EXIT_FAILURE);
	}

	// Avoid bind error: address already in use
	if( setsockopt( sock_server, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt) ) )
	{
		perror("setsockspot");
		exit(EXIT_FAILURE);
	}

	address.sin_family = AF_INET;
	address.sin_addr.s_addr = inet_addr(client_ip);
	address.sin_port = htons(port);

	// Bind: Assign adress to the socket
	*status = bind(sock_server, (const struct sockaddr *)&address,
sizeof(address));
	printf("%d \n", *status);
	if( *status < 0 )
	{
		perror("bind");
		exit(EXIT_FAILURE);
```

```
	}

	*status = listen(sock_server, 3);
	if( *status < 0 )
	{
		perror("listen");
		exit(EXIT_FAILURE);
	}
	else
		printf("Waiting connection from a client...\n");

	// Accept Connection
	Sockaddr foreignAddr;
	int cli_len = sizeof(address);

	*status = accept(sock_server, (Sockaddr *)&address, &cli_len);
	if( *status < 0 )
	{
	  printf("Acceptation failed with status %s\n", strerror(errno));
	  exit(EXIT_FAILURE);
	}

	printf("connection made \n");
	return sock_server;
}
```

Right above, we can see the function *establish_server_connection()*, which receives as argument the client's IP, because each robot has a different one, and the port used to connect, that must be different for each connection made.

The function returns the socket number created by the *socket()* function, because it's necessary to close the socket on the end of *main_server.c*.

4.2.2 Threaded function

```
                                server.c
void *thread_send_recv(void *message_arg)
{
	// variables flag's indicates if it's possible to enter on the
critical
```

```
	// section
	struct message *message_red = (struct message*) message_arg;
	struct message  *message_cs;
	message_cs = (struct message *) malloc(sizeof(struct message));
	message_cs->flag = true;
	message_cs->port = 0;
	bool *buffer;
	buffer = (bool *) malloc(sizeof(bool));

	pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

	printf("%d \n", message_cs->flag);

	while(true)
	{
		// red_flag = true if the car entered in the cs and false if
exited
		// read too from which car the signals cames from
		read(message_red->status, (void *) buffer, sizeof(bool));
		message_red->flag = (bool *) buffer;

		pthread_mutex_lock(&lock);
		if( message_red->flag == true && message_cs->flag == true )
		{
			message_cs->flag = !message_cs->flag;
			message_cs->port = message_red->port;
			printf("%d \n", message_cs->flag);
		}
		if( message_red->flag == false && message_cs->flag == false )
		{
			message_cs->flag = !message_cs->flag;
			message_cs->port = 0;
			printf("%d \n", message_cs->flag);
		}
		pthread_mutex_unlock(&lock);

		send(message_red->status, (const void *) message_cs,
sizeof(struct message), 0);

	}
	free(buffer);
}
```

The function runned for the threads is responsible for transfering data and also to control the flux on the critical section. One thread can communicate only with one client. She receives a struct pointer as argument to make possible receive more than one parameter, because thread function must be of type: *void func(void arg).* The argument is a struct created called *struct Message* that has a flag of type *bool*, which is true if the respective robot passed through the red mark, and a *string* variable with the IP of the robot. The IP of each one, is defined before on the file *main_server.c*, however the *bool* variable of the struct will serve to receive the message from the client.

Firstly, was necessary to create a new variable of the same struct type, just to cast the argument received by the function that must be of type *void**. After we have to create our shared variable *message_cs*, that is responsible to keep the critical section flag, which is true if there is someone in there or false otherwise, and the IP of the respective robot on the said zone.

Another necessity of using threads, besides control the which robot is in the critical section, was the problem of creating two connections on the same time. When two connections are created, the place you call the first *accept()* function matters a lot. The second calling, for the second client naturally, can only transfer data after the first one. This is a big problem, because the clients can't communicate with server independently. Therefore, if the second robot has arrived on the critical section the server will be just updated if the first had sent any message. The solution found, was create a infinity loop making each thread receive so many messages from each client so fast that would be impossible to have a delay between both. So, normally, the variable buffer is receiving a *false* message until any car pass through the red mark and is *true* while someone is in her.

After receiving the client's message, we enter in a lock to change our shared variable. As was mentioned before, the flag of *message_red* is always false because the client is always sending messages until arrives on the critical section. The flag of *message_cs* is always true until someone enter on the section. So, the first condition on the code will just happen some robot enter on the zone and then the *message_cs*'s flag is changed to false and the IP of the respective robot is saved. Note, that this flag will only be true again when the robot inside leaves the section and then his IP is erased from the variable.

Exiting the lock, the function is almost finished. A message is sent to the respective client's thread with a flag warning if there is someone of critical section and his port.

4.3 Client

The client is the program that will run inside each robot. Both of them have the same code (except for the PORT number), which are responsible his trajectory, The PID. Also, as already said, they have a connection with the server that warn them if someone is on the critical section. The first part of main's code can be seen right below.

main_client.c

```c
#include "client.h"
#include "PID.h"
#include <unistd.h>

int main(void)
{

// PID variables ------------------------------------------------------

      float value0, value1, value2, target, error = 0, prop, integral = 0,
derivate = 0, PID_value;
      float last_error = 0, kp = 0.4, ki = 0, kd = 0, sum, ks = 0.1;
      int duty_cycle, duty_cycle_sp;
      POOL_T sock_col_sensor;
// --------------------------------------------------------------------

// TCP variables ------------------------------------------------------

      int status, valread, sock_client, port;
      port = PORT;
      struct message *message_cs;
      message_cs = (struct message *) malloc(sizeof(struct message));
      message_cs->port = 0;
      bool *red_flag;
      red_flag = (bool *) malloc(sizeof(bool));
      *red_flag = false;

// --------------------------------------------------------------------

      //Change this depending on the client
      char car_ip[14] = "192.168.0.104";

      // Setting a value for the target
      target = 60;

      // Making connection with server
      sock_client = establish_client_connection(car_ip);

      // Initialize the brick
      init_brick();
```

```
    // Find and set color sensor
    sock_col_sensor = search_set_color_sensor();

    int count_r = 0, count_g = 0, cont = 0;
    float left_motor_speed = 0, right_motor_speed = 0;
```

The first step is set everything we need to be used on the rest of the code. We began declaring robot's IP that is the same for both robots, the only thing that changes it's the port number. Then, we set our target we are going to use on the PID controller, which will be explained with more details on the next part.

The second step it's establish the connection with *establish_client_connection()* function that has all the proceedings to establish a TCP connection, but not exchange data, like it was made on the server code. After that, we just need to initialize the brick and set our sensor. On this function *search_set_color_sensor()* we have to search for the color sensor components which have a lot of differents types of modes like intensity of light, rgb, etc…

We are going to talk which type was chosen right below on the PID's part.

4.3.1 PID controller

The PID is the most important part of client's program. As mentioned before, he controls the car through the black line. Firstly, the the PID code was put on inside a function of a header called *PID.c.* However, after some tests the function didn't react well on the *main* and was necessary to split the code. Right below rest of the code on *main* related to PID control.

```
                            main_client.c
    while(true)
    {

// PID code ------------------------------------------------------------
            value0 = sensor_get_value0( sock_col_sensor, COLOR_RGB_RAW);
            value1 = sensor_get_value1( sock_col_sensor, COLOR_RGB_RAW);
            value2 = sensor_get_value2( sock_col_sensor, COLOR_RGB_RAW);
            sum = value0 + value1 + value2;
```

```
            // Error
            error = target - ks*sum;

            // Proportional
            prop = error;

            // Integral
            integral += error;

            // Derivate
            derivate = error - last_error;

            PID_value = prop*kp + integral*ki + derivate*kd;

            last_error = error;

            // Makes the car come back to the line
            left_motor_speed = SPEED_LINEAR - PID_value;
            right_motor_speed = SPEED_LINEAR + PID_value;
            tacho_set_duty_cycle_sp( MOTOR_LEFT, left_motor_speed);
            tacho_set_duty_cycle_sp( MOTOR_RIGHT, right_motor_speed);
            tacho_run_direct(MOTOR_BOTH);

            if( value0 >= 200 && value1 <= 40 && value2 <= 30 )
            {
                  *red_flag = true;
                  printf("RED \n");
                  count_r++;
            }
            if( value0 >= 200 && value1 > 150 && value2 <= 40 )
            {
                  printf("YELLOW \n");
                  *red_flag = false;
                  count_g++;
            }

            // Report values
            if( count_r == 1 || count_g == 1 || count_r == 2 || count_g ==
2 )
            {
                  printf("RED: %.2f \n", value0);
```

```
                printf("GREEN: %.2f \n", value1);
                printf("BLUE: %.2f \n", value2);
                printf("error: %.2f\n", error);
                printf("integral*ki: %.2f\n", integral*ki);
                printf("derivate*kd: %.2f\n", derivate*kd);
                printf("PID value: %.2f\n", PID_value);
                printf("Motor left speed: %.2f \t right speed: %.2f \n",
left_motor_speed, right_motor_speed);
            }


// PID code ending ----------------------------------------------------------

// Transfer data code ------------------------------------------------

            // Warn (or not) server if red mark was found (entered on
critical section)
            send( sock_client, (const void *) red_flag, sizeof(red_flag), 0
);

            // Receive if it's possible to enter on critical section
            read( sock_client, (void *) message_cs, sizeof(struct message)
);

            printf("%d %d %d \n\n", message_cs->port, *red_flag,
message_cs->flag);
            // the car will only stop if there is someone on the critcal
section
            //  AND it's not him (check the adress) AND  just if he arrives
            //  on red mark, otherwise he can continue to run
            if( message_cs->flag == false && *red_flag == true && port ==
message_cs->port )
                tacho_stop(MOTOR_BOTH);
// Transfer data code ending
-------------------------------------------------------------

      }

      printf("Red counters: %d \n Green counters: %d \n", count_r,
count_g);
      free(red_flag);
      free(message_cs);
```

```
        close(sock_client);
        return 0;

}
```

As already said, the PID it's one way of controlling in control theory. And, as every control algorithm we need a infinity loop to make it.

In our case, on the beginning of the loop we have three types of incoming data. The red, green and blue types, due to the RGB mode that must be chosen.

A big problem during the project was the mode of the color sensor. On the beginning was chosen the intensity of light mode that gives you numbers from 0 to 100. It's quite good to be used on the implementation of PID, because this data type doesn't has a lot of oscillations. However, this data type it's not very good to differentiate types of colors, like the red mark on the trail. Therefore, it became impossible to chose this mode for the sensor.

A good strategy to solve that was to use the RGB which gives you three outputs of data: red, green and blue. The data's value goes from 0 to 1025 what is a lot more than before. Now, with three outputs we can distinguish each color with more facility. Even with more values was still difficult to do the PID. I tried a lot of times to use one of the values but the PID doesn't work very well. After watch the display a lot of times, I figured out that the values arrived up to 300 maximum. Therefore, the sum of them arrived until 900 and if we divide it by ten until ninety. Now, we have a output that goes from 0 to ninety which is much better than before. The new output was so better that was only necessary to put the proportional constant to regulate the PID. Also, we hadn't had great oscillations making perfect for the project.

Right after, we send red mark warn to server that will use the received data on *thread_send_recv()* function. Then, we receive a data *structure message* with the information if there is someone on the critical section and his port. The last step is to check if someone is already on critical zone receiving a message of type *struct message*. The car will only stops if someone is already on the critical zone and if he arrived on the same time on the red mark, otherwise there is no need to stop, because when one is going out of the zone the other is coming.

5 Conclusion

Was a little bit difficult to implement all the project especially when I tried to mix all the different parts of the project together. I wasn't expecting either problems with sensor measurement. At the end, everything was working except for the flux inside the critical section, due to a lot of problems during the project.

Was enhanced experience make this project because I could learn a bit of concurrent programming, control theory and network connections and use everything on the same time.

6 References

- http://docs.ev3dev.org/projects/lego-linux-drivers/en/ev3dev-jessie/index.html#
- http://in4lio.github.io/ev3dev-c/index.html
- https://www.ev3dev.org/
- https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf
- https://www.geeksforgeeks.org/socket-programming-cc/
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html