

CONTROL OF A MULTI-ROBOT SYSTEM USING VISUAL SENSORS

- Mihai Veliche – Romania, Iasi, 2011- SplitResearch.org -

The purpose of the project is to obtain the sensorial input information necessary to control a robot system by using visual sensors. To do so we will need to acquire basic obstacle and distance information. For this each robot will have a camera mounted on it, but to get the information we will need to take two pictures from the same position. So each camera will be attached to a servomotor that will slide the camera horizontally. In this way will get a kind of stereo image that can be use determine the distance to obstacles. The main idea is to extract the features from the two images using the SURF algorithm (Speeded Up Robust Features), compare the offset of each common feature and establish a depth map. Also the recognition of glyphs is applied for recognizing certain landmarks.

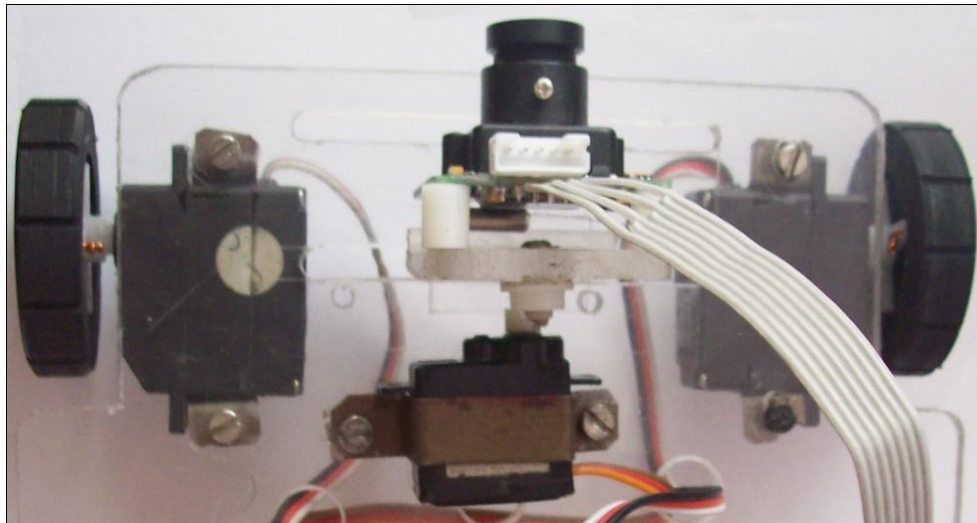


Fig. Camera moving system

The experiment requires the development of two mobile platforms; each equipped with a visual sensor, in this case a LinkSprite JPEG TTL Camera. The camera communicates at TTL level, so we can connect it directly to the Uart interface of the FlyPort module. Also, two continuous rotation servo motors will assure the traction, while a normal servomotor will be used to move the camera. In the center of the design lays the FlyPort Wi-Fi module, produce by OpenPicus.

The hardware design is described in the following image:

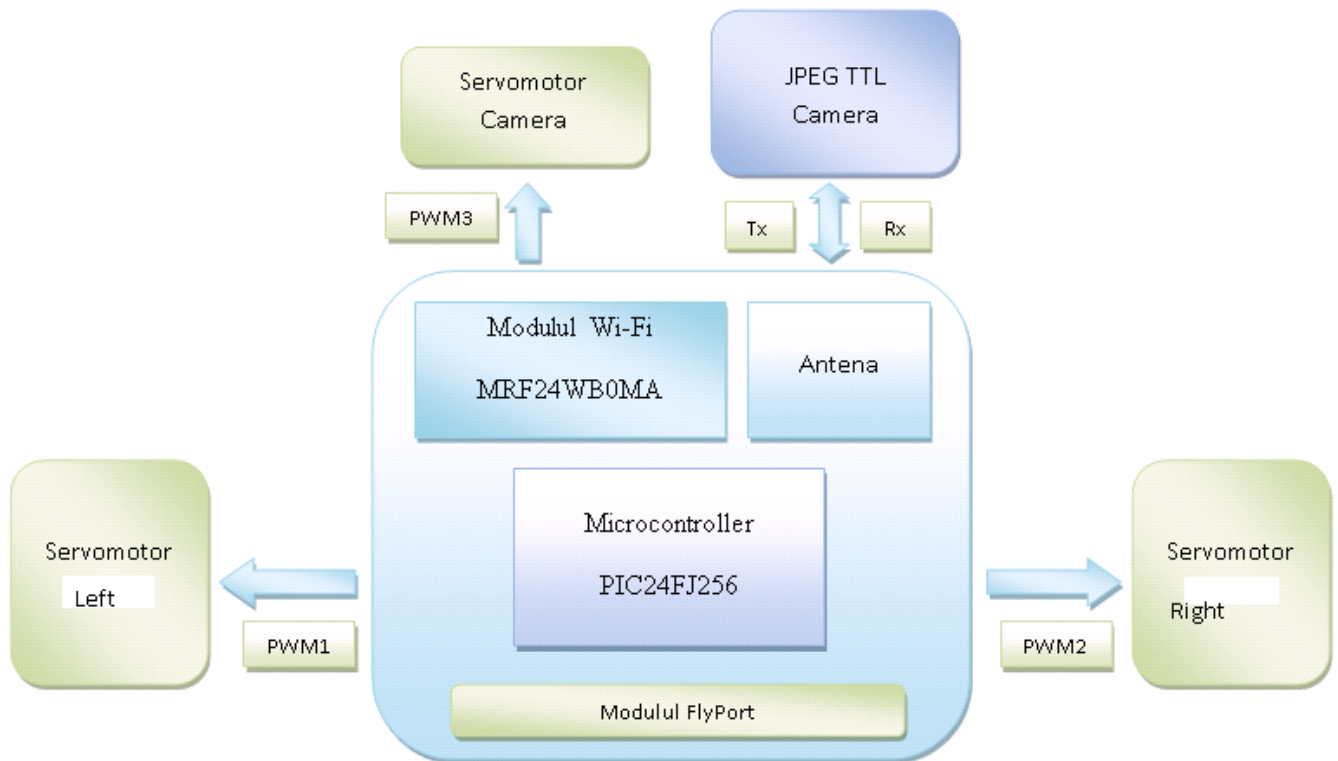


Fig. General hardware model

JPEG TTL CAMERA



Fig. LS-Y201 Camera

The connection pins:

Name	Description
TXD (Out pin)	Transmission pin. Connected to the Rx pin of the microcontroller.
RXD (In pin)	Reception pin. Connected to the TX pin of the microcontroller
+5V	VCC (the datasheet stats that the camera can operate at 3.3V, but for the best of operations use 5V)
GND	Ground

COMMUNICATION PROTOCOL

To succeed in capturing and downloading an image from the camera we need to send specific command on the UART interface. If a command was received and executed by the camera a response will be sent back.

RESET

Command (HEX)	Response(HEX)
56 00 26 00	76 00 26 00

CAPTURE IMAGE

Command (HEX)	Response(HEX)
56 00 36 01 00	76 00 36 00 00

When this command is send the camera takes a picture and saves it in the camera memory.

GET THE SIZE OF THE CAPTURED IMAGE

Command (HEX)	Response(HEX)
56 00 34 01 00	76 00 34 00 04 00 00 XH XL

After the Capture Image command has been sent, we need to find out the size of the image to be able to download it from the camera memory. So we will send the GetSize command and from the response we'll be able to extract the size of the image – XH XL.

READ IMAGE

Command (HEX)	Response(HEX)
56 00 32 0C 00 0A 00 00	76 00 32 00 00 FF D8
MH ML 00 00 KH KL 00 0A	76 00 32 00 00 FF D9

Reading an image from the camera memory is done in chunks. So this command must be sent until the whole image is downloaded.

MH ML – represent the address from which we are reading.

KH KL – represent the size of the chunk.

We can place din command in a while loop until the address becomes bigger then the size of the image (read with the GetSize command), or until we receive the bytes FF-D9, which mark the end of a jpeg file.

SET IMAGE SIZE

Command (HEX)	Response(HEX)
56 00 31 05 04 01 00 19 11 (320*240)	76 00 31 00 00
56 00 31 05 04 01 00 19 00 (640*480)	76 00 31 00 00
56 00 31 05 04 01 00 19 22 (160*120)	76 00 31 00 00

SET BAUD RATE

Command (HEX)	Response(HEX)
56 00 24 03 01 XX XX	76 00 24 00
XX XX	Baud Rate
AE C8	9600
56 E4	19200
2A F2	38400
1C 4C	57600
0D A6	115200

APPLICATION

The applications for both robots are identical, except for the addresses of each module. The third device will be a laptop with Wi-Fi capabilities, which will run the desktop application. This application will only look for these two addresses or the host name.

NOTE: For the best of the operations disable the DHCP Server in the TCP configuration wizard, because the first module that will go online will provide the other one with an IP address, usually 192.168.1.2. And since one module can only assign one IP, when we connect the third device an IP conflict will occur.

To connect all these devices on to one network we can configure on the laptop a new ad-hoc network. Then for each FlyPort module we must set the SSID Name to be the same as the created network. Now, after the *WFConnect (WF_DEFAULT)* command, all the devices will be part of one network.

Once a connection is made, we will initialize each FlyPort module to behave as a TCP Server, using the following lines:

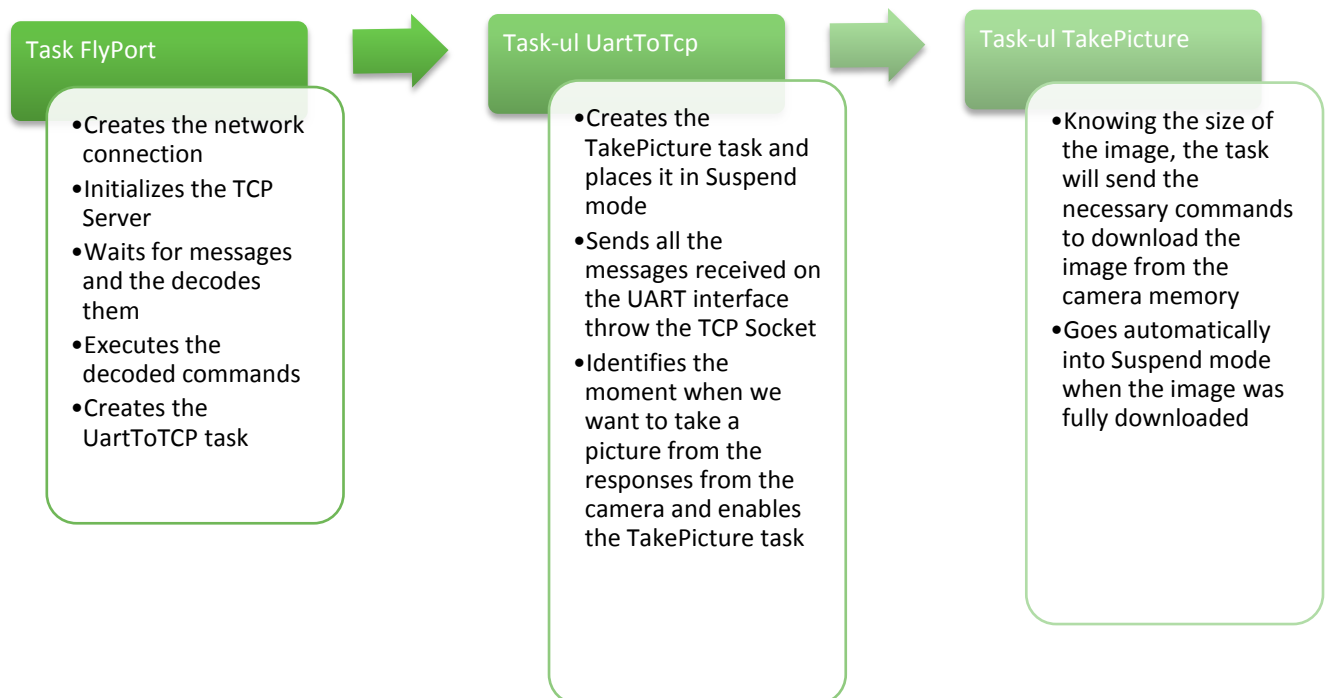
```
TCP_SOCKET socket;  
socket = TCPServerOpen("3001");
```

To initialize a new socket we must provide a port number. I assigned to Robot1 the port 3001 and to Robot2 3002; this was done only for visualization purposes, and don't matter if the ports are the same because they will be created on different modules and the desktop application will create two different sockets. These will later be used in the reception and transmission of data. The desktop application is responsible for connecting to each module and for monitoring the connection.

The main objectives of the hardware application are:

- Receiving and decoding commands from the TCP Socket and then executing them.
- Establishing a UartToTCP Bridge, for communication with the serial camera
- Download an image from the camera.

To establish these objectives I have created two more tasks inside the FreeRTOS environment, beside the default FlyPort task created by the OpenPicus IDE.



The **FlyPort task** runs an infinite loop in which all the messages received on the TCP socket are decoded and the specific command is executed.

```

while(1)
{
    if(TCPRxLen(socket) !=0)
    {
        TCPBuffSize = TCPRxLen(socket);
        TCPRead(socket, TCPBuff, TCPBuffSize+1);

        decode(TCPBuff,TCPBuffSize);

        TCPFlush(socket);
        vTaskDelay(20);
    }
}

```

Also it creates **the UartToTCP task**. This task will run an infinite loop that will send all the data from the UART interface on the TCP socket.

```

while(1)
{
    if (UARTBufferSize(1) != 0)
    {
        vTaskDelay(delay);
        char data[256];
        int size = UARTBufferSize(1);
        UARTRead(1,data, size);

        leng = TCPWrite(socket,data,size);
    }
}

```

To be able to capture images the task will create the task TakePicture and will put it in suspend mode. Also in the while loop will put the following code

```

//if we receive a GetSize Command Response from
//the Camera then we start the TakePictureTask
if((data[0]==0x76) &&
    (data[1]==0x00) &&
    (data[2]==0x34) &&
    (data[3]==0x00) &&
    (data[4]==0x04))
{
    sizeH=data[7];
    sizeL=data[8];

    i=0;
    address=0;
    picture_size = ((sizeH & 0x00FF)<<8) | (sizeL & 0x00FF);
    //task resume
    vTaskResume(hTakePicTask);
}

```

This means that the task will verify each message received from the camera and when it recognizes a response for a GetSize command will save the size and resume the Take Picture task.

The **TakePicture task** will run a while loop until the size read earlier will be smaller than the incrementing address. After that the task will auto suspend itself.

```
while(1)
{
    if(address < picture_size){
        for(i=0; i<8; i++) UARTWriteCh(1,READ_CMD[i]);
        UARTWriteCh(1,address>>8);
        UARTWriteCh(1,address & 0x00FF);
        UARTWriteCh(1,0x00);
        UARTWriteCh(1,0x00);
        UARTWriteCh(1,0x00); //SH--|
        UARTWriteCh(1,0x40); //SL--|
        UARTWriteCh(1,0x00);
        UARTWriteCh(1,0x0A);
        address = address + 64;
        vTaskDelay(200 / portTICK_RATE_MS);
    }
    else
    {
        vTaskSuspend(NULL);
    }
}
```

In this loop the Read Picture command will be sent until the whole picture is downloaded. Also the variable used in this task must be reset before activation.

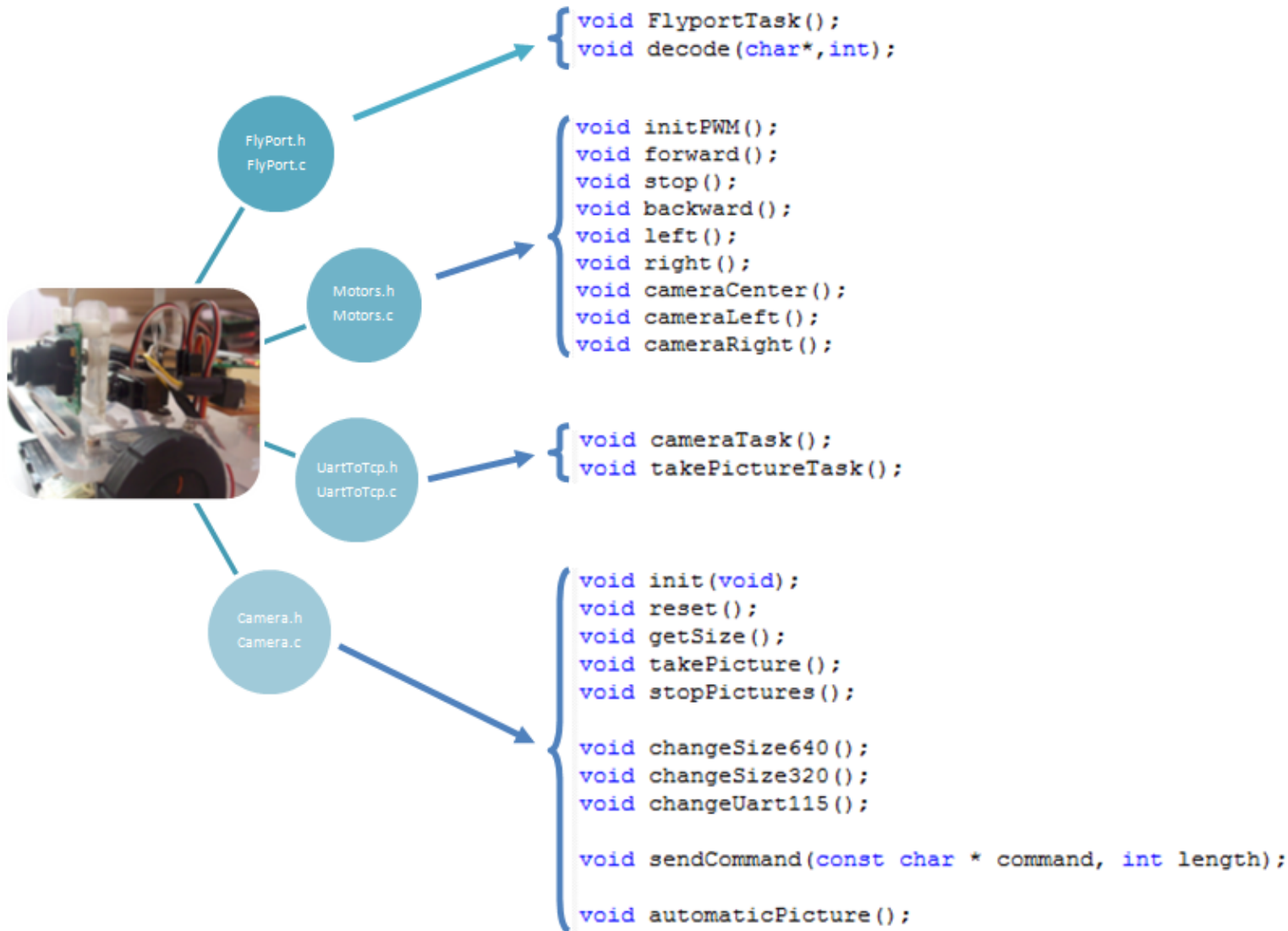


Fig. Hardware application

THE DESKTOP APPLICATION

The desktop applications main tasks are:

- Automatic connection to the TCP Servers on the modules
- If a connection is lost, the application will try to reconnect
- Simultaneous control of the robots
- The messages being received/transmitted will be saved in Lists and handled in the order that they arrived.

- Receiving the raw jpeg data and saving the image to a file
- Image processing

The application was written in C# so I used Background Workers instead of Threads. (Same thing... used a lot of them :D).The following graphic shows the background workers used to maintain the connection with the devices:

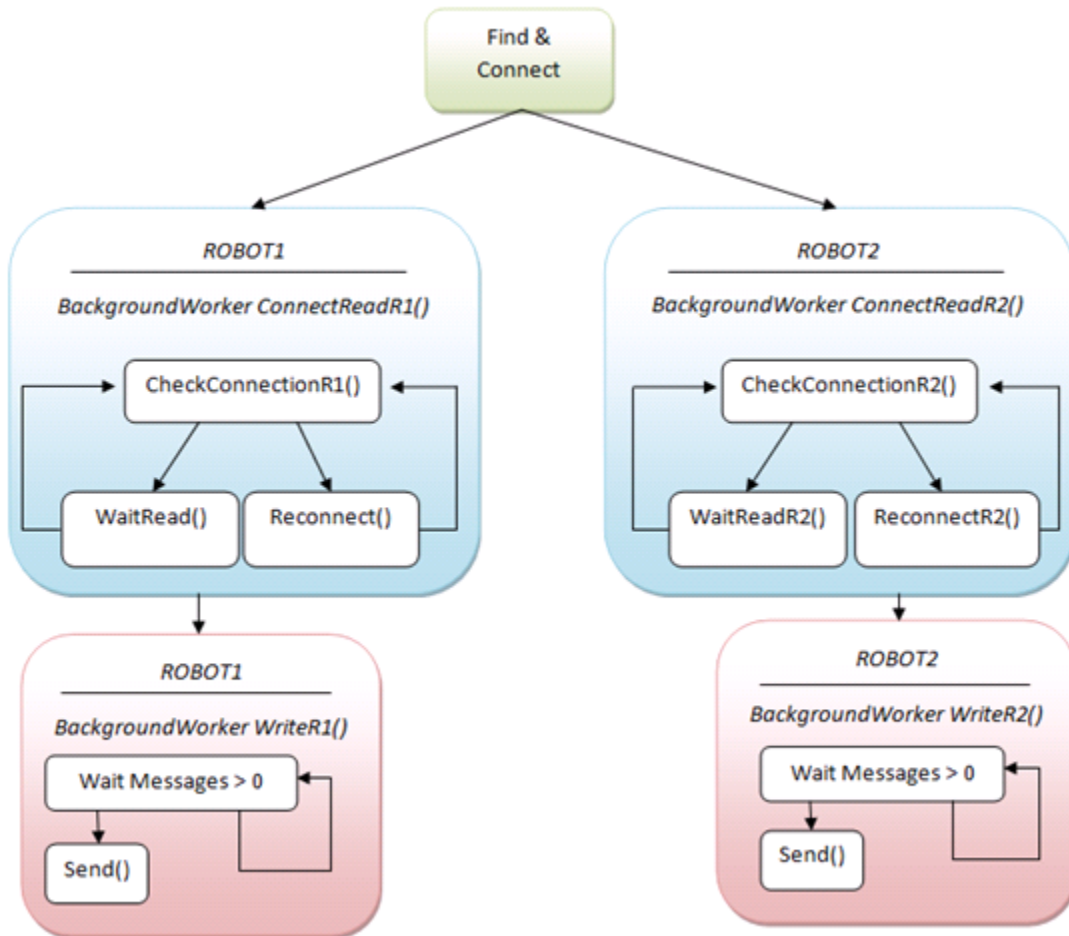


Fig. Background Workers used to Connect/Write/Reconnect

There is the option of manual controlling the robots by entering the command in a textbox. The commands are:

- Directional: w –forward, a – left, d – right, x back, s – stop;
- Camera: 4 – reset, 5 – GetSize, 6 – Capture image, 7 –stop capturing images, 8 – Automatic capture;
- Camera position: 1 – left, 2 – central, 3 – right;

- Setting commands: i – change baud rate, o – change image size to 320 x 240 , p – change image size to 640 x 480;

TAKING A PICTURE

To take a picture we can send the GetSize command to the robot or the Automatic capture command, which will be interpreted by the robot by resetting the camera, taking the picture and sending the GetSize command which will trigger the TakePicture task.

The commands to get the picture from the memory of the camera will be managed by the hardware application. So the desktop application will receive the raw jpeg file. We must monitor the received messages and look for the FF-D9 bytes that mark the end of a jpeg file. When we receive them we know that we have all the information in the receive stack to build the jpeg file.

The messages inside the stack look something like this:

76-00-32-00-00-FF-D8-...raw bytes (size of chunk)...-00-00-00-76-00-32-00-00

To create the image we just create a file with the extension .jpeg and write into it only the raw bytes converted to bits.



Fig. Example of downloaded image

CAPTURE STEREO IMAGES

To capture stereo images we must take two pictures, but first we must change the position of the camera.

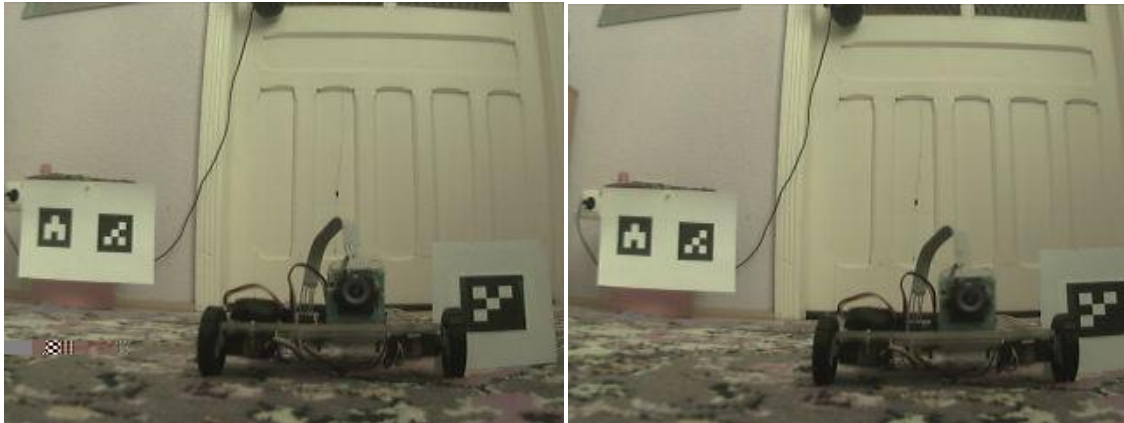


Fig. Example of stereo images

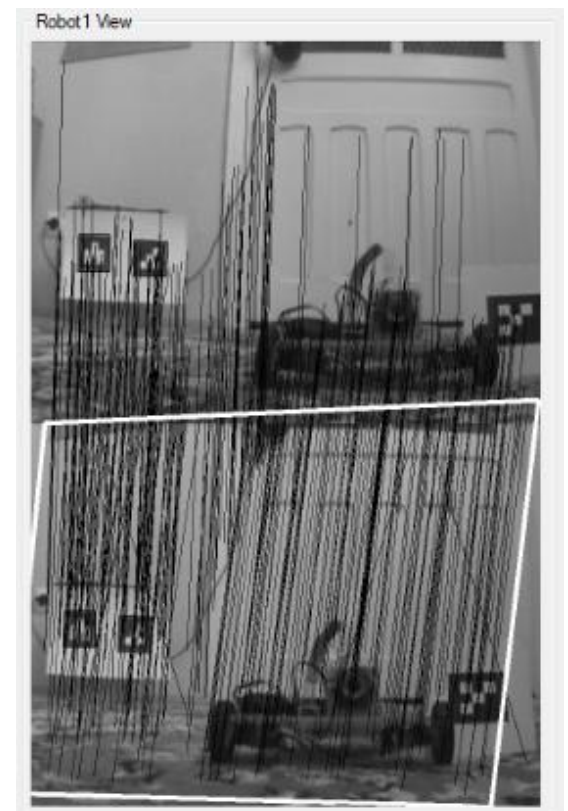
APPLY THE SURF ALGORITHM

The SURF algorithm is used to find point correspondences between two images of the same scene. It is implemented in the OpenCV libraries but I used the Emgu wrapper for .NET.

Thus we can draw lines between the two images to link the matching features.

It can be observed that the objects that are closer will have a bigger offset than the features that are further. (Observe the angle of the lines).

Using these information's we can calculate the offset for each of the matched feature and we can build a depth map by drawing circles of different color intensity in the points found to be features. I did this by creating several intervals and then assign each point to the one that fit its offset.



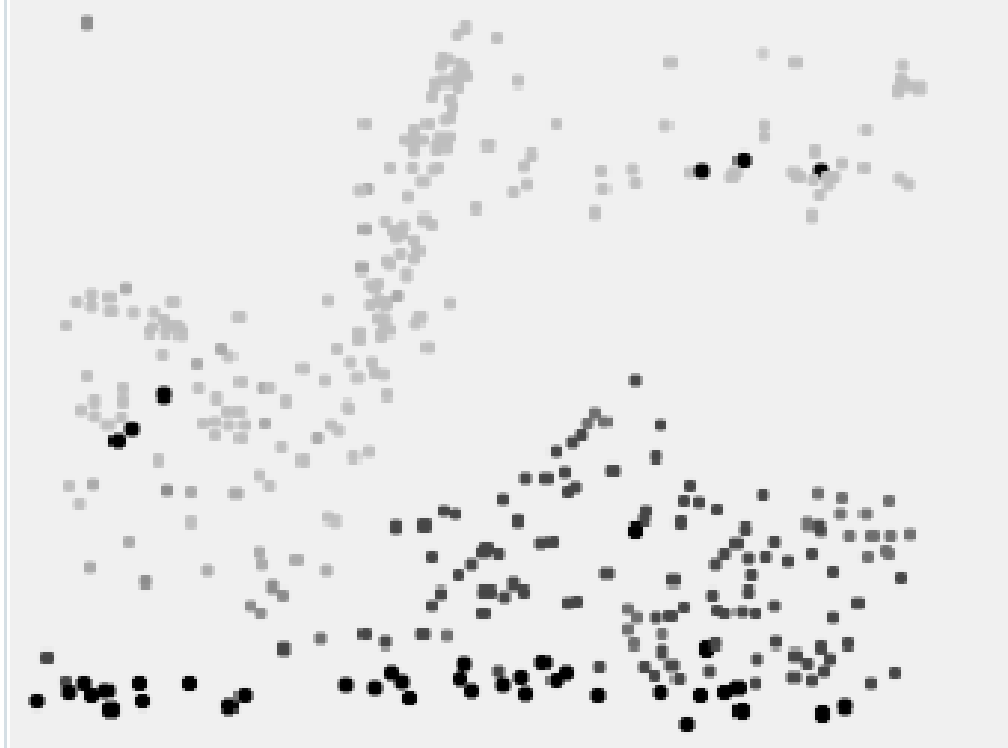


Fig. Depth Map for the above pictures

The darkest points represent the features that are closer.

We are limited to only the found features and sometimes these are not very precise. This could be managed if we eliminate the features that have an offset that is too big.

GLYPH RECOGNITION

All glyphs are represented with a square grid divided equally to the same number of rows and columns. Each cell of the grid is filled with either black or white color. The first and the last row/column of each glyph contain only black cells, which creates a black border around each glyph.



To perform these operations I used the AForge GRATF (Glyph Recognition and Tracking Framework) library. The library can be used in robotics applications for example, where glyphs may serve as commands or directions to robots. However, most popular application of optical glyph recognition is augmented reality.

Not only that we can find glyphs in an image but also we can match them against a database.

After I applied the SURF algorithm I tried to find all the glyphs in the images (if there are any). Then I separated the points (found with the SURF algorithm) that are inside the area of a found glyph (marked them with red) and then I calculated the average offset for these points.

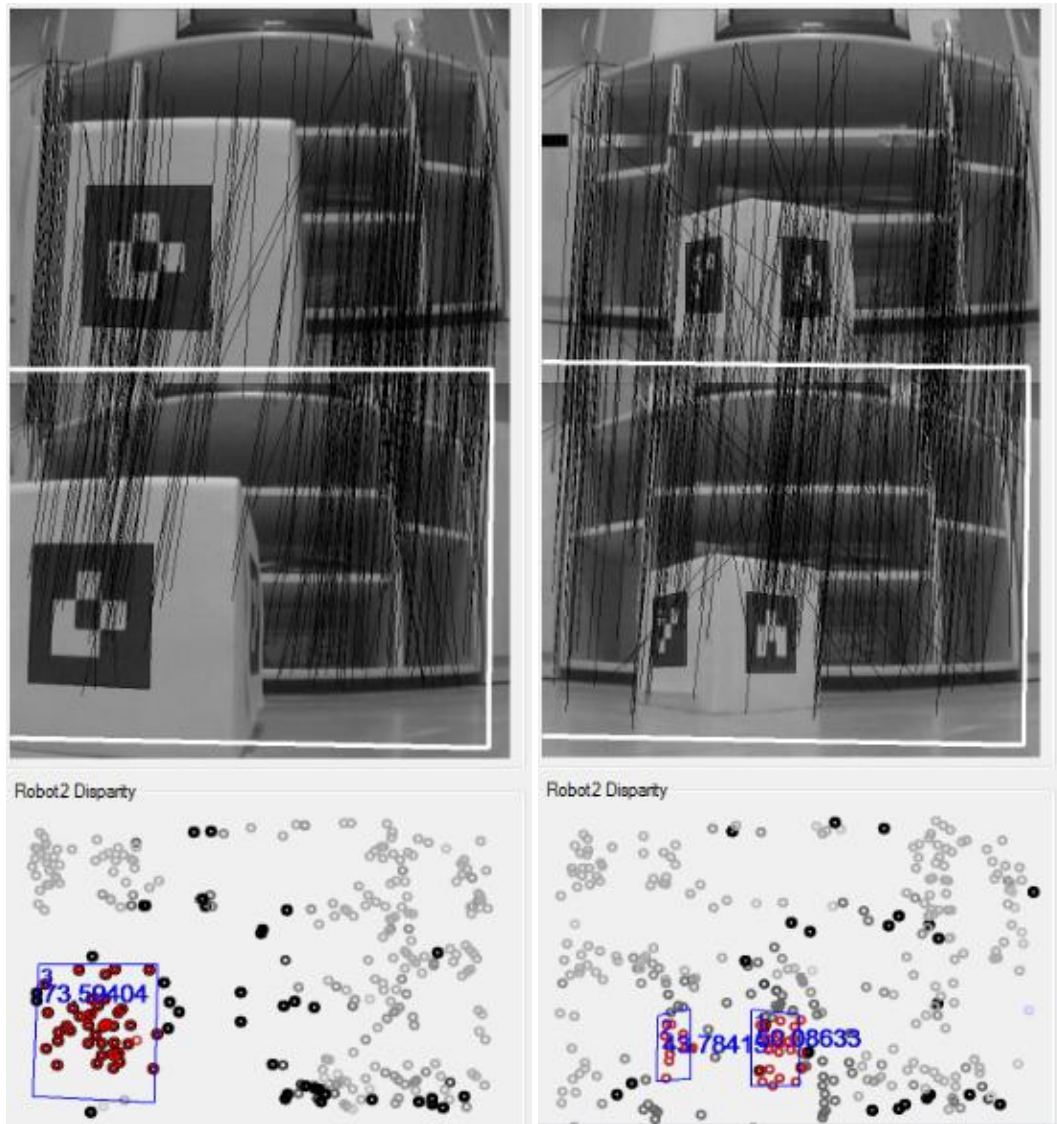
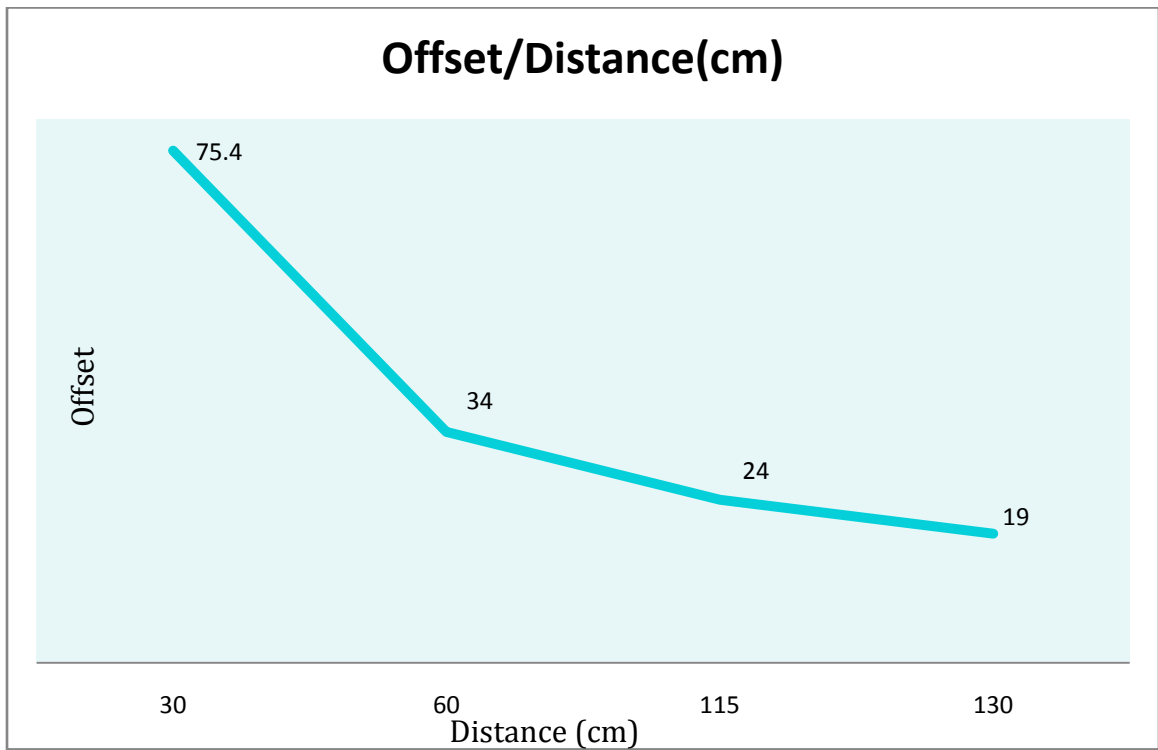


Fig. Found Glyphs and calculated offset

Through experimental measurements we can find for each offset an approximated distance:



The trend is clear: bigger offset means closer distance. If we incorporate this in the program we have a primitive way of avoiding obstacle or navigate to a target.

I would like to thank the OpenPicus Team for supporting this project.

The logo for OpenPicus, featuring the word "open" in a blue, lowercase, sans-serif font, followed by a stylized blue figure of a person with arms raised, and then the word "icus" in the same blue, lowercase, sans-serif font.

Also thanks to the developers and supporters of the other open source projects that I used:

