

Proyecto Final

Robot recolector de residuos

Guillermo Campelo
Juan Ignacio Goñi
Diego Nul

Tutor: Prof. Juan Miguel Santos

Instituto Tecnológico de Buenos Aires
Departamento de Informática

27 de septiembre de 2010

Resumen

Palabras clave: *Robot, recolector, residuos, robótica, comportamientos, visión, MR-2FA, L298, FR304, HX5010, GP2D120, BC327, SRF05, CNY70, motor, servo, telémetro, daisy chain, RS232, subsumption, Webots, OpenCV, detección, objetos, contornos, hsv*

Índice

1. Introducción	12
2. Hardware	13
2.1. Introducción	13
2.2. Ideas de implementación	13
2.2.1. Locomoción	13
2.2.2. Sensado del entorno	13
2.2.3. Controlador	14
2.2.4. Método de recolección	15
2.3. Actuadores	15
2.3.1. Motores de continua	15
2.3.1.1. Características	15
2.3.1.2. Circuito de control	16
2.3.1.3. Rutinas de control	18
2.3.2. Servo motores	18
2.3.2.1. Circuito de control	18
2.3.2.2. Rutinas de control	19
2.4. Sensado	19
2.4.1. Telémetros infrarrojos	19
2.4.1.1. Características	19
2.4.1.2. Circuito de control	19
2.4.1.3. Rutinas de control	21
2.4.2. Sensor de distancia por ultrasonido	21
2.4.2.1. Características	21
2.4.2.2. Circuito de control	21
2.4.2.3. Rutinas de control	22
2.4.3. Sensor reflectivo de piso	23
2.4.3.1. Características	23
2.4.3.2. Circuito de control	23
2.4.3.3. Rutinas de control	23
2.4.4. Encoders	24
2.4.4.1. Características	24
2.4.4.2. Circuito de control	25
2.4.4.3. Rutinas de control	25
2.4.5. Sensado de la batería	25
2.4.6. Consumo del motor	25
2.4.6.1. Pulsador u otro dispositivo disparador	26
2.4.6.2. Rutinas de control	27
2.5. Controladores	27
2.5.1. Netbook	27
2.5.2. Microcontrolador	27
2.5.2.1. Características	27
2.5.2.2. Módulos internos	28
2.5.2.3. Programación del firmware	29
2.6. Comunicación	30
2.6.1. Conectividad entre módulos	30
2.6.2. Protocolo de comunicación	30
2.6.2.1. Comandos comunes	32

2.6.2.2. Comandos específicos	32
2.7. Placas controladoras	35
2.7.1. Placa genérica	36
2.7.1.1. Características principales	36
2.7.1.2. Módulo de comunicación	37
2.7.1.3. Alimentación de la placa	37
2.7.1.4. Configuración	38
2.7.1.5. Esquemático	38
2.7.1.6. Circuito	38
2.7.1.7. Código básico	40
2.7.1.8. Posibles extensiones	40
2.7.2. Placa controladora de motores DC	40
2.7.2.1. Características principales	40
2.7.2.2. Comunicación	41
2.7.2.3. Alimentación de la placa	41
2.7.2.4. Configuración	41
2.7.2.5. Esquemático	42
2.7.2.6. Circuito	42
2.7.2.7. Código básico	45
2.7.2.8. Posibles extensiones	45
2.7.3. Placas de sensado	45
2.7.3.1. Características principales	45
2.7.3.2. Módulo de comunicación	46
2.7.3.3. Alimentación de la placa	46
2.7.3.4. Configuración	46
2.7.3.5. Esquemático	46
2.7.3.6. Circuito	48
2.7.3.7. Código básico	48
2.7.3.8. Posibles extensiones	49
2.7.4. Placa controladora de servo motores	49
2.7.4.1. Características principales	49
2.7.4.2. Módulo de comunicación	50
2.7.4.3. Alimentación de la placa	50
2.7.4.4. Configuración	50
2.7.4.5. Esquemático	50
2.7.4.6. Circuito	52
2.7.4.7. Código básico	52
2.7.4.8. Posibles extensiones	52
2.8. Armado del prototipo	52
2.8.1. Diseño	53
2.8.2. Desarme	54
3. Arquitectura de Comportamientos	57
3.1. Introducción	57
3.2. Investigaciones previas	58
3.2.1. Desarrollo de comportamientos no triviales en robots reales : Robot recolector de basura - Stefano Nolfi [17]	58
3.2.2. Arquitectura de control para un Robot Autónomo Móvil - Neves And Oliveira [16]	59

3.2.3.	Path Planning usando Algoritmos Genéticos - Salvatore Candido [5]	60
3.2.4.	Navegación predictiva de un robot autónomo - Foka And Trahanias [9]	60
3.2.5.	Algoritmo de navegación y evitamiento de obstáculos en un entorno desconocido - Clark Et. al [18]	61
3.3.	Arquitectura propuesta	62
3.4.	Comportamientos e implementaciones	64
3.4.1.	Wandering	65
3.4.1.1.	Detalle del comportamiento	65
3.4.1.2.	Implementación del comportamiento	66
3.4.2.	Enfocar Basura	67
3.4.2.1.	Detalle del comportamiento	69
3.4.2.2.	Implementación del comportamiento	69
3.4.3.	Ir a Basura	71
3.4.3.1.	Detalle del comportamiento	71
3.4.3.2.	Implementación del comportamiento	72
3.4.4.	Recolectar Basura	73
3.4.4.1.	Detalle del comportamiento	74
3.4.4.2.	Implementación del comportamiento	74
3.4.5.	Ir a zona de descarga de basura	74
3.4.5.1.	Buscar línea	75
3.4.5.1.1.	Detalle del comportamiento	75
3.4.5.1.2.	Implementación del comportamiento	75
3.4.5.2.	Entrar a línea	77
3.4.5.2.1.	Detalle del comportamiento	77
3.4.5.2.2.	Implementación del comportamiento	77
3.4.5.3.	Seguir línea	79
3.4.5.3.1.	Detalle del comportamiento	79
3.4.5.3.2.	Implementación del comportamiento	79
3.4.6.	Descargar Basura	80
3.4.6.1.	Detalle del comportamiento	80
3.4.6.2.	Implementación del comportamiento	80
3.4.7.	Ir a base de recarga de batería	80
3.4.8.	Cargar Batería	80
3.4.8.1.	Detalle del comportamiento	80
3.4.8.2.	Implementación del comportamiento	81
3.4.9.	Evitar Obstáculos	81
3.4.9.1.	Detalle del comportamiento	81
3.4.9.2.	Implementación del comportamiento	81
3.4.10.	Salir de situaciones no deseadas	82
3.4.10.1.	Detalle del comportamiento	82
3.4.10.2.	Implementación del comportamiento	83
3.5.	Odometría	84
3.5.1.	Problemas con la odometría	85
3.6.	Interfaces con hardware y módulo de reconocimiento de objetos	87
3.6.1.	Interfaz con hardware	87
3.6.2.	Interfaz con módulo de reconocimiento de objetos usando Visión	87
3.7.	Resultados obtenidos	89

3.7.1.	Distribución y progreso de comportamientos	90
3.7.2.	Tiempos en <i>caa</i> y <i>cai</i> de comportamientos	93
3.7.3.	Transiciones entre comportamientos	94
3.7.4.	Arena recorrida	95
3.8.	Conclusión	97
3.8.1.	Posibles extensiones	100
4.	Visión	101
4.1.	Introducción	101
4.2.	Trabajos previos	101
4.2.1.	Mobile Field Robot with Vision-Based Detection of Volunteer Potato Plants in a Corn Crop [22]	101
4.2.2.	Review of shape representation and description techniques [24]	103
4.2.3.	Sidewalk Following Using Color Histograms [19]	105
4.2.4.	Skin Detection using HSV color space [21]	106
4.2.5.	Line Detection and Lane Following for an Autonomous Mobile Robot [1]	106
4.3.	Algoritmo de detección	107
4.3.1.	Estructura general	107
4.3.2.	Detección de color	108
4.3.3.	Threshold	109
4.3.4.	Operaciones morfológicas	114
4.3.4.1.	Dilatación	114
4.3.4.2.	Erosión	115
4.3.5.	Detección de contornos	115
4.3.5.1.	Algoritmo	117
4.3.5.2.	Representación	118
4.3.5.3.	Polígonos	119
4.3.6.	Filtros	120
4.3.6.1.	Filtro de área	120
4.3.6.2.	Filtro de área por zona	121
4.3.6.3.	Filtro de perímetro	121
4.3.6.4.	Filtro de excentricidad	121
4.3.6.5.	Filtro de circularidad	122
4.3.6.6.	Filtro de área rectangular	122
4.3.6.7.	Filtro de elipse	122
4.3.6.8.	Filtro de vaso	122
4.3.6.9.	Filtro de histograma	122
4.3.7.	Reconocimiento de vasos	123
4.3.8.	Reconocimiento de colillas	123
4.3.9.	Reconocimiento de platos	123
4.3.10.	Centroide	123
4.4.	Sistema de predicción	123
4.5.	Focalización	127
4.6.	Resultados	128
4.6.1.	Parámetros de medición	128
4.6.2.	Ánálisis	130
4.7.	Conclusión	132
4.7.1.	Posibles extensiones	133

5. Conclusiones	135
6. Agradecimientos	136
A. Protocolo de comunicación	138
A.1. Formato del paquete	138
A.2. Posibles comandos	139
A.2.1. INIT	139
A.2.2. RESET	140
A.2.3. PING	140
A.2.4. ERROR	140
A.3. Comandos específicos	141
A.4. MAIN CONTROLLER	141
A.5. DC MOTOR	141
A.5.1. SET DIRECTION	141
A.5.2. SET DC SPEED	142
A.5.3. SET ENCODER	142
A.5.4. GET ENCODER	142
A.5.5. RESET ENCODER	143
A.5.6. SET ENCODER TO STOP	143
A.5.7. GET ENCODER TO STOP	143
A.5.8. DONT STOP	144
A.5.9. MOTOR CONSUMPTION	144
A.5.10. MOTOR STRESS ALARM	144
A.5.11. MOTOR SHUT DOWN ALARM	145
A.5.12. GET DC SPEED	145
A.6. SERVO MOTOR	145
A.6.1. SET POSITION	145
A.6.2. SET ALL POSITIONS	146
A.6.3. GET POSITION	146
A.6.4. GET ALL POSITIONS	146
A.6.5. SET SERVO SPEED	147
A.6.6. SET ALL SPEEDS	147
A.6.7. GET SERVO SPEED	147
A.6.8. GET ALL SPEEDS	148
A.6.9. FREE SERVO	148
A.6.10. FREE ALL SERVOS	148
A.6.11. GET STATUS	149
A.6.12. ALARM ON STATE	149
A.6.13. SWITCH ALARM	150
A.7. DISTANCE SENSOR	150
A.7.1. ON DISTANCE SENSOR	150
A.7.2. OFF DISTANCE SENSOR	150
A.7.3. SET DISTANCE SENSORS MASK	151
A.7.4. GET DISTANCE SENSORS MASK	151
A.7.5. GET VALUE	151
A.7.6. GET ONE VALUE	152
A.7.7. ALARM ON STATE	152
A.7.8. SWITCH ALARM	153
A.8. BATTERY CONTROLLER	153

A.8.1. ENABLE	153
A.8.2. DISABLE	154
A.8.3. GET BATTERY VALUE	154
A.8.4. BATTERY FULL ALARM	154
A.8.5. SET BATTERY EMPTY VALUE	154
A.8.6. BATTERY EMPTY ALARM	155
A.8.7. SET FULL BATTERY VALUE	155
A.9. TRASH BIN	155
A.9.1. GET TRASH BIN VALUE	156
A.9.2. BIN FULL ALARM	156
A.9.3. SET FULL BIN VALUE	156
A.10. Codificación de los valores enviados	157
A.11. Ejemplos	157
A.11.1. Ejemplo 1	157
A.11.2. Ejemplo 2	157
B. Código fuente	158
B.1. Placa genérica	158
B.2. Archivo <i>protocolo.c</i>	160
B.3. Placa controladora de motor de dc	163
B.4. Placa controladora de sensores	172
B.5. Placa controladora de servo motores	184
C. Costo del prototipo	191
D. Implementación de arquitectura de software del controlador	192
D.1. Comportamientos	194
D.2. Dispositivos del robot	196
D.3. Reconocimiento de objetos	198
E. Implementación del protocolo en PC	200
E.1. Packet Server	200
E.2. Packets	200
E.3. Handlers	201
E.4. Modificación del protocolo	202

Índice de cuadros

1.	Características del motor Ignis MR-2FA.	16
2.	Tabla de verdad para el control del driver <i>L298</i>	17
3.	Características del servo HX5010.	18
4.	Características del sensor de distancia infrarrojo GP2D120.	20
5.	Características del sensor de ultrasonido SRF05.	21
6.	Tensión de la batería y la tensión de salida en el divisor.	26
7.	Tabla comparativa para el consumo del motor.	26
8.	Pines de programación en circuito con <i>ICD2</i>	29
9.	Conexionado entre placas en modo Link	30
10.	Conexionado entre placa y la PC	31
11.	Formato y header del paquete de datos	31
12.	ID para cada grupo según su función.	32
13.	Comandos comunes a todos los controladores.	32
14.	Comandos específicos al <i>DC MOTOR</i>	33
15.	Comandos específicos al <i>SERVO MOTOR</i>	34
16.	Comandos específicos al <i>DISTANCE SENSOR</i>	35
17.	Comandos específicos al <i>BATTERY CONTROLLER</i>	36
18.	Comandos específicos al <i>TRASH BIN</i>	36
19.	Alimentación de la lógica	37
20.	Pines de alimentación del motor.	41
21.	Pines del header de comunicación con el motor.	42
22.	Asignación de pesos para evitar obstáculos	81
23.	Parámetros utilizados para las simulaciones	89
24.	Resultados sobre steps en los cuales los comportamientos están activos	92
25.	Resultados sobre steps que los comportamientos están continuamente activos	93
26.	Resultados sobre steps que los comportamientos están continuamente inactivos	93
27.	Transiciones entre comportamientos	94
28.	Transiciones desde <i>i</i> hacia <i>j</i> en porcentajes	95
29.	Transiciones desde <i>j</i> hacia <i>i</i> en porcentajes	95
30.	Porcentaje de arena cubierta	96
31.	Performance del algoritmo de visión	132
32.	Performance con sistema de predicción configuración 1	132
33.	Performance con sistema de predicción configuración 2	133
34.	Performance promedio de las colillas	133
35.	Performance promedio de los vasos	134
36.	Performance promedio de los platos	134
37.	Formato y encabezado del paquete de datos	138
38.	Paquete de datos del ejemplo 1	157
39.	Paquete de datos del ejemplo 2	157
40.	Lista de materiales y costos (en pesos a marzo de 2010).	191

Índice de figuras

1.	Vista lateral y frontal del motor Ignis MR-2FA.	16
2.	Diagrama interno del driver L298.	17
3.	Diagrama de tiempos del sensor GP2D120.	20
4.	Voltaje según la distancia al objeto del telémetro GP2D120.	20
5.	Ángulo de apertura según la distancia del telémetro GP2D120. .	20
6.	Haz ultrasónico del sensor SRF05.	22
7.	Diagrama de tiempos del sensor SRF05.	22
8.	Medidas en milímetros del sensor CNY70.	23
9.	Principio de funcionamiento reflectivo del sensor CNY70.	24
10.	Corriente en el colector según la distancia del sensor CNY70. .	24
11.	Divisor de tensión para el sensado de la batería.	25
12.	Diagrama del microcontrolador PIC16F88.	28
13.	Módulos internos del microcontrolador PIC16F88.	28
14.	Diagrama general del método Daisy-Chain	30
15.	Conectores RJ11 (6P4C) y DB9.	37
16.	Bornera de alimentación.	37
17.	Microcontrolador y headers	38
18.	Comunicación, switch de modo y conectores de entrada y salida .	38
19.	Fuente de alimentación	39
20.	Máscara de componentes de la placa genérica.	39
21.	Capas superior e inferior de la placa genérica.	39
22.	Microcontrolador y header de programación.	42
23.	Comunicación, switch de modo y conectores de entrada y salida.	43
24.	Driver y header de conexión con el motor.	43
25.	Divisor de tensión para el voltaje de referencia.	43
26.	Fuente de alimentación de la lógica y motor.	43
27.	Máscara de componentes de la placa controladora de motores DC.	44
28.	Capas superior e inferior de la placa controladora de motores DC.	44
29.	Microcontrolador y conector de programación.	47
30.	Comunicación, llaves de modo y conectores de entrada y salida. .	47
31.	Puertos de conexión para los sensores y header de <i>pull-up</i>	47
32.	Fuente de alimentación de la lógica.	47
33.	Máscara de componentes de la placa controladora de sensores. .	48
34.	Capas superior e inferior de la placa controladora de sensores. .	49
35.	Microcontrolador y header de programación.	51
36.	Comunicación, llaves de modo y conectores de entrada y salida. .	51
37.	Puertos de conexión para los servo motores y de uso general. .	51
38.	Fuente de alimentación extendida para la lógica y servo motores.	51
39.	Máscara de componentes de la placa controladora de servo motores.	52
40.	Capas superior e inferior de la placa controladora de servo motores.	53
41.	Diagrama de desarme exterior.	55
42.	Diagrama de desarme frontal.	55
43.	Diagrama de desarme interior.	56
44.	Robot Khepera con el módulo Gripper	58
45.	Robot E-puck	59
46.	Algoritmo propuesto por Clark Et. al	61
47.	Esquema de comportamiento	62

48.	Ejemplo de la arquitectura Subsumption	63
49.	Arquitectura de comportamientos implementada	65
50.	Diagrama de posición de la cámara	66
51.	Zona vista por la cámara	67
52.	Primera aproximación de “Ir a la basura”	68
53.	Segunda aproximación de “Ir a la basura”	68
54.	Inconveniente con la primera aproximación de “Ir a la basura” . .	68
55.	Área vista por la cámara y distancias y ángulos	70
56.	Caso particular de distancia hacia una basura	71
57.	Descomposición de figura 56	71
58.	Distancias a pixeles en imagen	72
59.	Distintos δ_f	73
60.	Conversión de coordenadas de imagen de cámara	73
61.	Etapas de recolección de basura	74
62.	Arena de simulación y ejes de coordenadas	76
63.	Disposición de sensores de piso	76
64.	Possibles estados iniciales de “Entrar a la línea”	78
65.	Possible estado final luego del primer giro del comportamiento “Entrar a la línea”	79
66.	Centro del robot sobre la línea y posibles giros	79
67.	Red neuronal entre los sensores de distancia y los motores . . .	82
68.	Error Sistemático Máximo a lo largo de las iteraciones	85
69.	Arena que utilizamos para las simulaciones	90
70.	Distribución de los tiempos de los comportamientos en la simulación	90
71.	Evolución de los comportamientos a lo largo de la simulación .	91
72.	Progreso del comportamiento <i>Recolectar Basura</i>	92
73.	Área visualizada y área no visualizada a lo largo de la simulación	96
74.	Técnicas de descripción de figuras	105
75.	Etapas del algoritmo de visión	108
76.	Conversión RGB-HSV	109
77.	Espacio de color HSV	110
78.	Filtro de color con distintos rangos	110
79.	Distintos valores de umbral	112
80.	Comparación threshold con filtro de color	112
81.	Eliminación de ruido por threshold	113
82.	Resultados de aplicar thresholding	113
83.	Operación de dilatación	114
84.	Operación de erosión	115
85.	Efecto de la operación de dilatación	116
86.	Eliminación de ruido mediante dilatación	116
87.	Ejemplo de erosión	117
88.	Tipos de conectividad entre pixeles	118
89.	Ejemplo de segmentación en imágenes binarias	118
90.	Ejemplo de codificación de un contorno utilizando los códigos de Freeman	119
91.	Matriz para la codificación de Freeman	119
92.	Aproximación de contornos mediante polígonos	120
93.	Algoritmo de aproximación de polígonos	121
94.	Ejemplo de descriptores de contorno	124
95.	Ventaneo	129

96.	Ejemplo de ventana	129
97.	Arquitectura de software: principales componentes	193
98.	Arquitectura de software: comportamientos	195
99.	Arquitectura de software: api de robot	197
100.	Arquitectura de software: módulo de reconocimiento	199
101.	Diagrama de clases: Servidor de paquetes e interface	201
102.	Diagrama de clases: Estructura general	202
103.	Diagrama de clases: Estructura paquetes 1	203
104.	Diagrama de clases: Estructura paquetes 2	204
105.	Diagrama de clases: Handlers 1	205
106.	Diagrama de clases: Handlers 2	206

1. Introducción

El tratamiento de los residuos es uno de los problemas principales que aquejan a las grandes ciudades. Entre otros problemas, la basura no recolectada puede dañar al medio ambiente de diferentes maneras. Recipientes abiertos como vasos o botellas pueden acumular agua de lluvia favoreciendo la proliferación de mosquitos que pueden causar enfermedades como el dengue. Los residuos no recolectados pueden ser acarreados por los caudales de lluvia perjudicando los sistemas de desagües y contaminando los ríos y mares donde finalmente desembocan. Esta problemática es tratada por distintos organismos y sectores de la sociedad, tanto gubernamentales como organizaciones sin fines de lucro. El objetivo de este trabajo es contribuir a esta área desde el punto de vista de la robótica.

Con este propósito, diseñamos un prototipo de robot móvil capaz de reconocer y recolectar residuos en un ambiente dinámico como lo es la terraza de nuestra universidad. Este robot, navega a través de dicho entorno en búsqueda de elementos residuales que son detectados mediante una cámara a bordo. El robot cuenta además, con un sistema de comportamientos que le permite recorrer el entorno eficazmente y recargar su batería cuando sea necesario, esquivando los obstáculos que se le pudieran presentar en el camino. El reconocimiento de residuos se logra a través de un sistema de visión que realiza el procesamiento de las imágenes capturadas por la cámara en busca de colillas de cigarrillos, vasos y platos. Para realizar esto, equipamos al robot con una serie de sensores y dispositivos de procesamiento que permiten la ejecución de estos algoritmos y rutinas programadas para dicha labor.

Este manuscrito lo organizamos de la siguiente forma: en la sección 2 explicamos detalladamente las características físicas del robot, desde su composición hasta su ensamblaje. En la sección 3 mostramos la implementación y arquitectura de los comportamientos del robot que definen la manera en que éste interactúa con el entorno. En la sección 4 detallamos la implementación del algoritmo de procesamiento de imágenes utilizado para el reconocimiento de residuos. La sección 5 concluye el trabajo.

2. Hardware

2.1. Introducción

En esta sección detallamos las características físicas del robot en cuanto a dispositivos que lo componen, decisiones de diseño y armado del prototipo.

Este apartado se divide de la siguiente forma: en la sección 2.2 exhibimos las principales ideas de implementación, en las secciones 2.3 y 2.4 describimos los actuadores y sensores utilizados, en la sección 2.5 mostramos los dispositivos de control elegidos, en la sección 2.6 detallamos el método de comunicación diseñado para la comunicación entre los controladores de cada periférico, en la sección 2.7 exhibimos el diseño de las placas controladoras realizadas y en la sección 2.8 explicamos el armado del prototipo.

2.2. Ideas de implementación

Evaluamos distintos factores para la elección de cada dispositivo o módulo presente en el robot. Desde la forma de desplazamiento hasta el método de sensado, pasando por el controlador, captura de imágenes y método de recolección de residuos. Estos son los factores que explicamos y detallamos en este apartado.

2.2.1. Locomoción

Un robot móvil se caracteriza por su capacidad de desplazarse por el entorno. Es por esto que la locomoción es un factor importante que tuvimos que resolver.

La forma que usaría el robot para desplazarse, si sería mediante ruedas, orugas o patas nos iba a determinar las necesidades de control, equilibrio, capacidad de coordinación, velocidad y hasta la autonomía.

El uso de orugas era una excelente opción como base para soporte del peso y proveía buen equilibrio, pero también implicaba un sistema de actuación más complejo desde el punto de vista mecánico respecto al uso de ruedas. Por otra parte, la coordinación de las patas y el equilibrio del cuerpo implicaba una complejidad mecánica y de control, como así también un consumo mayor de energía respecto a las otras posibilidades.

El uso de dos ruedas de tracción y con al menos una tercera rueda de tipo castor para proveer equilibrio fue nuestra elección de diseño.

En la sección 2.3 explicamos detalladamente la elección de los actuadores para esta tarea.

2.2.2. Sensado del entorno

El sensado del ambiente englobaba varios puntos que debíamos tener en cuenta a la hora de elegir la cantidad, forma, rango, método y costo de los sensores. En principio los requerimientos eran claros, necesitábamos poder detectar objetos en el ambiente, evitar obstáculos y realizar mediciones internas al robot, como eran el nivel de batería, consumo y posición de los motores.

La captura de imágenes debía ser realizada por medio de una cámara de video o webcam. Ésta proveía de la principal fuente de datos para el proceso de reconocimiento de residuos explicado en el documento. Las distintas características de la cámara como su resolución, refresco, nivel de ruido, mejoras de la imagen y su velocidad de respuesta nos determinarían la elección de la misma.

La detección de obstáculos implicaba sensores con características diferentes. Principalmente necesitábamos sensores de distancia con la apertura necesaria para cubrir lo mejor posible, el perímetro del robot, otorgando un rango de detección lo suficientemente holgado que permitiera una adecuada velocidad de respuesta al controlador principal para poder evitar objetos desconocidos o identificados como obstáculos.

Entre los distintos tipos de sensores de distancia estaban los de ultrasonido, telémetros infrarrojos y scanners láser. Los sensores de distancia por ultrasonido tienen un rango aproximado entre 2 centímetros y 4 metros en la zona de detección y un ángulo de apertura variable según la distancia al objetivo. El tiempo de espera que debemos tener entre disparo y disparo del sensor por posibles rebotes del pulso de sonido es relativamente elevado, lo que no lo hace el más adecuado para colocarlo a lo largo del perímetro del robot. En contraparte los telémetros infrarrojos no poseen esta limitación en cuanto a los rebotes. Pero los rangos efectivos de detección de objetos no cubrían completamente nuestras necesidades.

Debido a las características de cada sensor decidimos combinarlos para aprovechar los beneficios y suplir las falencias de un tipo con otro el tipo. Un anillo de 8 telémetros distribuidos de forma tal que haya una mayor resolución en la zona frontal del robot contra la zona trasera y lateral y un único sensor de distancia por ultrasonido al frente para captar objetos a mayor distancia fue nuestra elección.

El uso de un scanner láser hubiera sido ideal para un análisis topográfico del terreno, aunque ésto no era un requerimiento de nuestro proyecto y el elevado costo se escapaba de nuestra escala.

La detección de una línea en el suelo la resolvimos mediante el uso de sensores ópticos reflectivos apuntando hacia el piso de manera tal que permitieran diferenciar entre distintos niveles de reflexión y por ende, diferenciar una línea del piso.

En la sección 2.4 explicamos con más detalle cada uno de los sensores elegidos.

2.2.3. Controlador

Teníamos especificaciones que nos generaban la necesidad de ejercer control sobre los distintos dispositivos presentes en el robot, ya sea para configurar o obtener la información que proveen los sensores, para controlar la velocidad de las ruedas o para comunicar las distintas partes. No existía una única forma para realizar esto.

Experiencias previas nos daban la idea de tener un único controlador que mantuviera el control sobre cada uno de los dispositivos en nuestro robot. Esto generaba grandes exigencias de hardware y una alta complejidad de diseño a nivel software para coordinar y mantener todo en funcionamiento. Otra opción era la existencia de pequeños controladores distribuidos con menor capacidad de procesamiento, destinados a pocas o una única tarea, de diseño simple y con una menor complejidad a nivel software. Esta solución combinada con una buena modularización y una comunicación acorde, generaba a nuestra visión y según nuestras necesidades, la mejor opción y por ende, fue la que elegimos.

En la sección 2.5 explicamos detalladamente la elección de los controladores. En la sección 2.6 explicamos todo lo referente a la comunicación entre módulos

y en la sección 2.7 el diseño y construcción de las placas.

Como la captura de imágenes la realizaríamos por medio de una cámara de video, webcam o algún otro tipo de cámara integrada y todo el procesamiento de imágenes requiere grandes capacidades de cálculo, resolvimos por simplicidad el uso de una computadora preferentemente de arquitectura x86 para facilitar la compatibilidad a nivel software. De manera conjunta decidimos el uso de una webcam con puerto USB como principal dispositivo de captura de imágenes. Por poseer una batería propia de gran capacidad, facilidad de uso, disponibilidad de sistema operativo, entorno de desarrollo, costos y soporte a nivel repuestos, elegimos una netbook como controlador principal del robot.

2.2.4. Método de recolección

La necesidad de recolección de basura del robot nos sugería la existencia de un mecanismo que permitiera tomar objetos y llevarlos al recipiente interior para la futura descarga. Entre las opciones que tuvimos en cuenta la existencia de un brazo mecánico con los suficientes grados de libertad para realizar la tarea, idea que rechazamos por las dificultades de control y construcción que sólo el brazo generaban.

Otra opción fue un mecanismo de aspirado de los residuos hacia el interior, pero esta solución no nos permitía diferenciar entre lo que recolectábamos, juntar colillas de cigarrillos, vasos o botellas y generaba gran consumo de energía.

Finalmente elegimos una solución que constaba de una pala que se extendía, recogía y contraía para dejar los residuos en el recipiente. Aunque por falta de tiempo el mecanismo no fue implementado para la presentación y armado físico del prototipo. Aún así consideramos su existencia durante todo el desarrollo del proyecto y diseñamos los mecanismos para ejercer el control tanto desde los comportamientos del robot, como también a nivel protocolo de comunicación y placas controladoras para los actuadores físicos.

2.3. Actuadores

La principal forma en que el robot puede interactuar activamente con el ambiente que lo rodea son los motores y cada tarea que debíamos realizar requería de actuadores acordes. Estas cuestiones son las que analizamos en este apartado.

2.3.1. Motores de continua

Para la tracción principal de las ruedas necesitábamos motores que tuvieran el torque suficiente para mover el robot, pero que pudieramos medir y controlar la velocidad era la principal necesidad. Para esta tarea utilizamos motores de continua con caja reductora y encoder. Con dos de estos motoreductores podíamos garantizar la velocidad necesaria en cada una de las ruedas.

2.3.1.1. Características

Los motores que elegimos son de la marca Ignis¹ modelo *MR-2FA* con características expresadas en el cuadro 40. Están provistos de una caja reductora, poseen un encoder de 4 estados por vuelta en el eje del motor y un sensor de

¹<http://www.ignis.com.ar>

Característica	Unidad	Mínimo	Nominal	Máximo
Tensión	V	8	9	12
Corriente	A	0.6	1.2	2.4
Velocidad	RPM	1	60	60
Aceleración	$1/s^2$	0.1	0.1	0.5
Torque	kgf*cm	0	1.2	6.4

Cuadro 1: Características del motor Ignis MR-2FA.

efecto de campo para determinar una vuelta en la salida de la caja reductora. En la figura 1 mostramos las dimensiones exteriores en centímetros del motor.

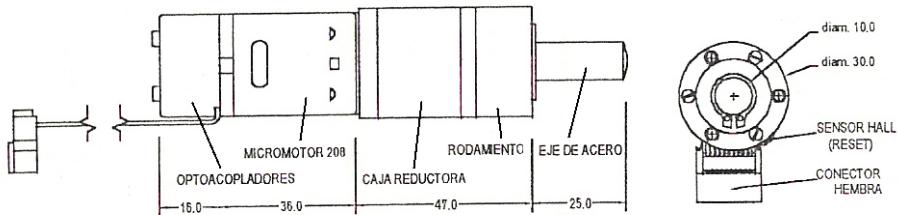


Figura 1: Vista lateral y frontal del motor Ignis MR-2FA.

Una ventaja que encontramos en este modelo es que ya trae el encoder integrado aunque su resolución podría haber sido mayor. Los encoders los explicamos más en detalle en la sección 2.4.4. La relación de la caja reductora es de 94:1.

2.3.1.2. Circuito de control

Para alimentar y poder controlar los motores elegimos el driver *L298* de la marca ST². Internamente tiene dos puentes H puenteables y puede soportar hasta 4A. Ésto lo logramos teniendo las salidas 1 y 2 puenteadas con las salidas 4 y 3 respectivamente, como mostramos en el diagrama de la figura 24.

La principal función del driver es proveer de la corriente y voltaje necesarios para el funcionamiento del motor, pero la configuración del puente H nos dió la posibilidad de, con una lógica simple, determinar el sentido de la corriente y potencia que recibía el motor.

Este integrado admite puentear ambas salidas, aumentando así, la máxima corriente que puede circular por el motor. Para hacer esto, conectamos las salidas *Out1* y *Out4* por un lado y por el otro las salidas *Out2* y *Out3*. De igual forma los pines de habilitación *EnA* y *EnB*, luego la entrada *In1* con la *In4* por un lado y por el otro la *In2* con la *In3*. De esta forma, lo controlamos con sólo 3 cables, uno de habilitación y otros 2 de *Input* que determinan la polarización de los transistores internos y por ende, el sentido de giro del motor. En el cuadro 2 mostramos la tabla de verdad para los pines de control, donde H es estado alto, L estado bajo y X cualquier estado. En la figura 2 mostramos el diagrama interno del integrado.

²<http://www.st.com>

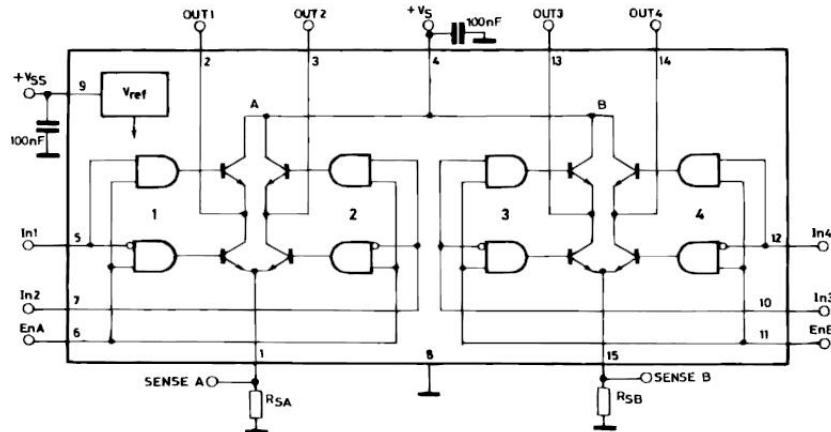


Figura 2: Diagrama interno del driver L298.

Enable	Input 1	Input 2	Función
H	H	L	Sentido horario
H	L	H	Sentido anti-horario
H	L	L	Motor frenado
H	H	H	Motor frenado
L	X	X	Motor libre

Cuadro 2: Tabla de verdad para el control del driver *L298*.

Para determinar la potencia que recibirá el motor usamos el módulo de *PWM* del microcontrolador, que explicamos más en detalle en la sección 2.5.2.2. Variando el ancho del pulso sobre el pin de habilitación del driver determinamos la cantidad de tiempo que el motor recibe tensión, lo cual se traduce en la potencia que éste tiene para realizar el movimiento. Cuanto mayor es el tiempo en estado alto del pulso, mayor la potencia.

Para contrarrestar la corriente negativa en las salidas del driver usamos los diodos *FR304* que cumplen con las especificaciones del driver con 150ns de tiempo de recuperación y una corriente de 3A.

El consumo del motor lo medimos mediante los pines de sensado en el driver, conectados a masa por una resistencia y al módulo de *ADC* del microcontrolador, que explicamos en la sección 2.5.2.2. Conociendo el valor de la resistencia y el valor leído por el módulo de *ADC*, pudemos determinar cuánta corriente circula por la resistencia y por ende la corriente consumida por el motor.

Para controlar la velocidad del motor usamos uno de las dos salidas del encoder que trae. Conectada como entrada para el módulo de *Timer* del microcontrolador, en configuración de contador, que explicamos más en detalle en la sección 2.5.2.2. Usando otro *Timer* para tener una base de tiempo fija y con el valor del contador pudimos determinar y controlar la velocidad de las ruedas. Dependiendo el tamaño de las mismas será la velocidad de final del robot.

En la sección 2.7.2 explicamos el desarrollo de la placa controladora de estos motores.

Característica	Unidad	Valor
Torque	kg	6.5
Velocidad	segundos/grado	$\frac{0.16}{60}$
Voltaje	V	4.8 a 6
Delay máximo	μs	4
Dimensiones	mm	40x20x38
Peso	g	39

Cuadro 3: Características del servo HX5010.

2.3.1.3. Rutinas de control

Desde el punto de vista del código, tuvimos que desarrollar las rutinas necesarias para el manejo de los motores según las instrucciones del controlador principal.

Configuramos a uno de los timers internos del microcontrolador para que genere una interrupción cada $6.25ms$, la cual usamos para realizar chequeos y ejercer control sobre el motor. Verificamos el consumo del motor para evitar sobrecargar el circuito y los motores ante un posible atasco de las ruedas. También actualizamos el acumulado de vueltas realizadas por el motor para la odometría.

Cada $200ms$ (32 interrupciones) tomamos la cantidad de cuentas del encoder y corregimos la velocidad de giro del motor ajustando el ancho del pulso generado por el PWM. Luego borramos el contador y esperamos otros $200ms$.

2.3.2. Servo motores

Para el movimiento de las partes del módulo de recolección, una cámara con paneo y giro o un sensor de ultrasonido colocado en la parte superior haciendo las veces de radar, pensamos en el uso de servo motores. La principal característica de estos actuadores es que sólo debemos indicar el ángulo al que queremos que esté el eje del motor y éste se coloca automáticamente. El ángulo de trabajo va desde 0° a 180° y algunos llegan hasta los 200° .

Aunque no fue implementado el ningún mecanismo que requiriera el uso de estos motores, explicamos en este apartado el trabajo realizado en torno a este tipo de actuadores. En la sección 2.7.4 explicamos el diseño de las placas que los controlan.

El servo de prueba que utilizamos para el desarrollo es el modelo *HX5010* de la marca Hextronik³ con características que expresamos en el cuadro 3

2.3.2.1. Circuito de control

La alimentación y consumo depende del modelo específico, variando también el torque que posee el servo.

No es necesario el uso de un driver para manejarlos, simplemente con la alimentación y una señal con el ángulo es suficiente. La forma de comunicar el ángulo varía entre los distintos servos y fabricantes. Hay servos analógicos y servos digitales. En los primeros la posición se determina mediante un voltaje que varía según cierto rango y si es digital, se setea mediante el ancho de un

³<http://www.hextronik.com/>

pulso que tiene un tiempo mínimo y máximo para mapear los ángulos mínimo y máximo respectivamente.

Dentro del modo de uso, podemos hacer que queden sueltos o que se queden fijos en cierta posición indicando, de forma continua, el valor del ángulo requerido. La frecuencia a la que debemos setear la posición depende del modelo.

2.3.2.2. Rutinas de control

Debido a que pensamos usar servos digitales y en una primera etapa del diseño se especificó el requerimiento de 3 servos, decidimos realizar la misma función pero por software.

Usamos el timer de 16bits del microcontrolador configurado con el clock interno como medida del tiempo para crear 5 salidas con pulsos que varían su ancho en forma independiente cada una. Definimos un ancho mínimo y máximo, pudiendo configurar pasos intermedios de 1° (aproximadamente $69.4\mu s$).

2.4. Sensado

En este apartado explicamos detalladamente cada uno de los sensores que utilizamos para realizar tanto las mediciones externas como las internas al robot. Analizamos las ventajas de cada uno, problemas que encontramos y sus soluciones.

En la sección 2.7.3 explicamos el diseño y construcción de las placas que controlan todos los sensores del robot.

2.4.1. Telémetros infrarrojos

El principio de funcionamiento de estos sensores es mediante un haz de luz infrarroja que es emitido hacia el objetivo, el cual es reflejado y captado a través de un lente por un sensor de posición relativa en el interior del sensor. En base a esta medición calculamos la distancia entre el sensor y el objeto reflectivo que se encuentra frente a él.

2.4.1.1. Características

Los telémetros infrarrojos que elegimos son de la marca Sharp⁴, modelo GP2D120. En el cuadro 4 detallamos los valores característicos del modelo.

Este tipo de sensores tiene un retardo de aproximadamente $43.1ms$ durante el cual la lectura que realizamos no es confiable y luego las nuevas lecturas se hacen en ventanas de aproximadamente el mismo tiempo. En la figura 3 mostramos el diagrama de tiempos.

2.4.1.2. Circuito de control

No necesitábamos un driver para manejar los telémetros, pero decidimos usar un transistor para poder habilitarlos o no, de otra manera el sensor estaba tomando mediciones continuamente provocando un consumo de batería innecesario. De esta forma sólo se enciende cuando se va a utilizar.

⁴<http://sharp-world.com/products/device>

Característica	Unidad	Valor
Rango máximo	cm	30
Rango mínimo	cm	4
Tensión para la máxima distancia	V	1.95
Tensión para la mínima distancia	V	2.55
Tensión de alimentación	V	5
Consumo máximo	mA	50

Cuadro 4: Características del sensor de distancia infrarrojo GP2D120.

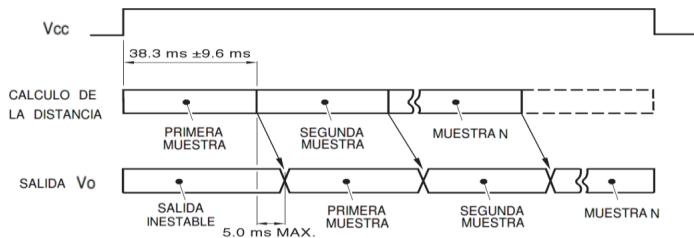


Figura 3: Diagrama de tiempos del sensor GP2D120.

El transistor que utilizamos es un conmutador y amplificador de uso general, el *BC327* y por ser de tipo PNP se excita con un estado bajo, por lo que la lógica de conmutación está negada.

El tipo de salida de este modelo de telémetros es analógica y están conectadas al módulo *ADC* del microcontrolador. En la figura 4 mostramos el cuadro de conversión entre voltaje de salida y distancia al objeto, y en la figura 5 mostramos el ángulo de apertura de la zona de detección según la distancia al objetivo.

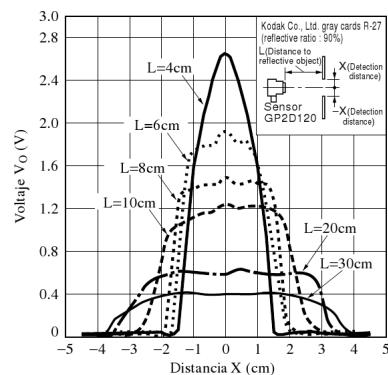
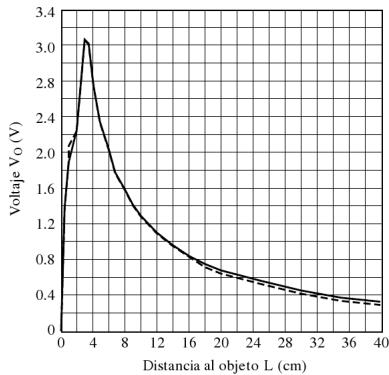


Figura 4: Voltaje según la distancia al objeto del telémetro GP2D120.

Figura 5: Ángulo de apertura según la distancia del telémetro GP2D120.

2.4.1.3. Rutinas de control

Controlar los telémetros es relativamente sencillo. Usamos el timer de 16bits del microcontrolador configurado con el clock interno para determinar el tiempo en el cual debíamos tomar las muestras con el ADC. Para filtrar el ruido de cada muestra individual, la distancia al objetivo es obtenida mediante el promedio de una repetición de mediciones.

Para hacer esto desarrollamos una pequeña máquina de estados que controla y maneja los tiempos para tomar las muestras que explicamos más en detalle en la sección 2.7.3.7.

2.4.2. Sensor de distancia por ultrasonido

Estos sensores de distancia se basan en la velocidad del sonido para calcular la distancia al objetivo. Generan un tren de 8 pulsos ultrasónicos y luego se espera como respuesta, el mismo tren de pulsos que debería haber rebotado contra el objetivo. En base a la diferencia de tiempo entre la emisión del tren de pulsos y la respuesta, se calcula la distancia a la que se encuentra el objetivo.

2.4.2.1. Características

El sensor de distancia por ultrasonido que elegimos es el modelo *SRF05* de la marca Devantech Ltd⁵. Esta versión mejorada del modelo *SRF04*, aumenta el rango de detección y mejora el modo de control y lectura de los datos, permitiendo hacerlo mediante un único pin.

La distancia medida mediante el tren de pulsos es codificada linealmente en el ancho de un pulso que varía de $100\mu s$ a $25ms$. Si dentro del rango de detección no se encuentra ningún objeto, el pulso tendrá un ancho de $30ms$.

En el cuadro 5 detallamos las características del sensor *SRF05* y en la figura 6 mostramos el haz ultrasónico del sensor.

Característica	Unidad	Valor
Tensión de alimentación	V	5
Corriente	mA	4
Frecuencia de trabajo	KHz	40
Rango máximo	cm	400
Rango mínimo	cm	1.7
Duración mínima del pulso de disparo	μs	10
Duración del pulso eco de salida	μs	100 - 25000
Tiempo mínimo de espera entre mediciones	ms	50
Dimensiones	mm	43x23x40

Cuadro 5: Características del sensor de ultrasonido SRF05.

2.4.2.2. Circuito de control

No necesitamos de un driver para manejar al sensor ya que lo conectamos directo a los 5 volts de la placa. El pin de *Mode* lo dejamos en estado bajo para

⁵<http://www.robot-electronics.co.uk/>

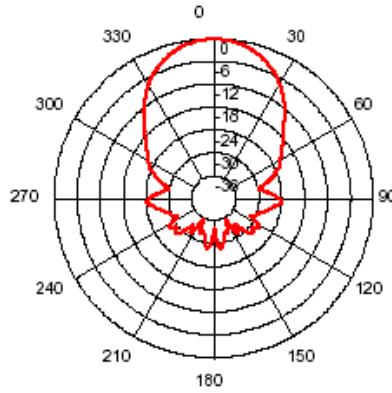


Figura 6: Haz ultrasónico del sensor SRF05.

indicar que debe funcionar bajo el nuevo modo y no en compatibilidad con el *SRF04*.

En el pin de *TRIGGER* sólo generamos un pulso de al menos $10\mu s$ para desencadenar la lectura de la distancia al objetivo. El sensor nos asegura que no generará el pulso de respuesta hasta pasados los $700\mu s$ desde pasado el pulso de trigger. En la figura 7 mostramos el diagrama de tiempos.

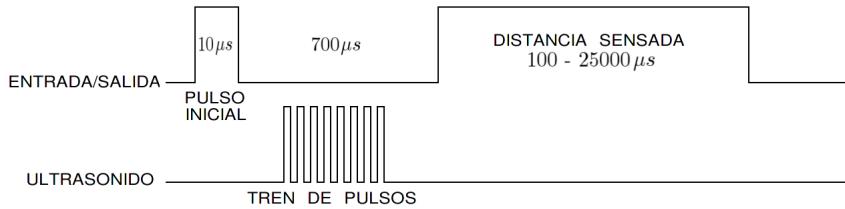


Figura 7: Diagrama de tiempos del sensor SRF05.

Para evitar que el rebote de otros sensores o sensados anteriores influya en la lectura, se debe esperar un mínimo de $50ms$ antes de generar otra medición.

2.4.2.3. Rutinas de control

Usamos uno de los pines con interrupción externa en el que conectamos el pin de *TRIGGER* y el timer de 16bits del microcontrolador configurado con el clock interno.

Para realizar la medición, generamos un pulso de $15\mu s$ para asegurarnos el disparo del sensor y cambiamos el modo del pin a entrada con interrupción ante un flanco ascendente. Cuando la interrupción es activada significa que comienza al pulso con la distancia codificada en su ancho, por lo que tomamos una muestra del timer y configuramos al pin para que genere ahora una interrupción ante un flanco descendente. Cuando la interrupción es nuevamente activada, será porque terminó el pulso con la medición, por lo que sólo debemos hacer la resta entre el valor actual del timer y la muestra que tomamos al principio para conocer la distancia a la que se encuentra el objetivo.

Un tiempo obtenido mayor a los $25ms$ indica que no se detectó ningún objeto dentro del rango del sensor.

2.4.3. Sensor reflectivo de piso

Estos sensores ópticos reflectivos emiten luz infrarroja y captan el nivel de luz reflejada sobre la superficie a sensar como mostramos en la figura 9. En la figura 8 mostramos las dimensiones del sensor. La intensidad de luz captada depende de la distancia al objetivo y del color y nivel de reflectividad de la superficie. Es por esto que usamos estos sensores para identificar una línea en el piso.

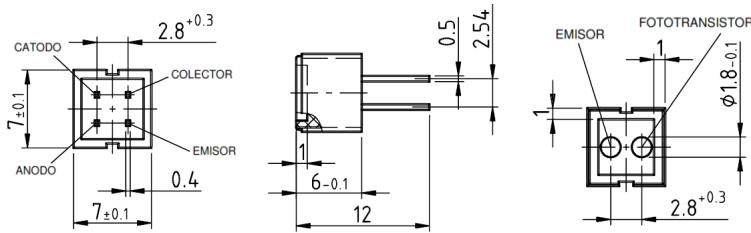


Figura 8: Medidas en milímetros del sensor CNY70.

2.4.3.1. Características

Los sensores que elegimos son del modelo *CNY70* de la marca Vishay Semiconductor⁶.

El rango efectivo de sensado ronda los $3mm$ de distancia aunque, con un incremento en la corriente que circula por el emisor se puede llegar a una distancia mayor a la recomendada por el fabricante y que nos permita un uso más acorde al proyecto. El emisor soporta un pulso de hasta $3A$ por un tiempo menor o igual a $10\mu s$.

En la figura 10 mostramos la corriente que circula por el colector del fototransistor en base a la distancia al objeto medido.

2.4.3.2. Circuito de control

De igual forma que en los telémetros utilizamos un transistor para comutar la tensión de alimentación en el sensor. Esto nos dió la posibilidad de encenderlo y apagarlo a la hora de tomar las muestras de la luz reflejada por piso o la línea. Nuevamente la lógica de habilitación es invertida por tratarse de un transistor *BC327*.

Agregamos una resistencia para limitar la corriente que circulaba por el emisor y otra como pull-up en el emisor del fototransistor que a su vez, conectamos al módulo de *ADC* del microcontrolador para efectuar las mediciones.

2.4.3.3. Rutinas de control

El código que desarrollamos para obtener las muestras de estos sensores es sencillo, simplemente debemos habilitar el transistor que alimenta al sensor con

⁶<http://www.vishay.com/>

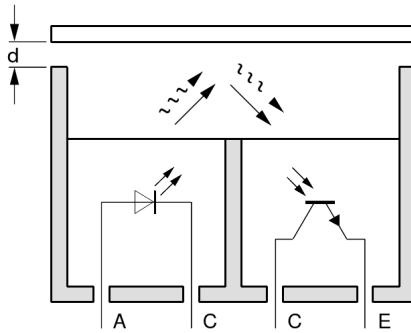


Figura 9: Principio de funcionamiento reflectivo del sensor CNY70.

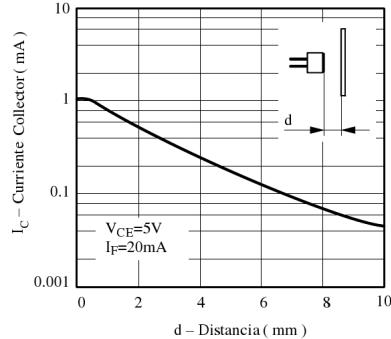


Figura 10: Corriente en el colector según la distancia del sensor CNY70.

un estado lógico bajo, tomar al menos 4 muestras y promediarlas para tener un valor adecuado del nivel de luz reflejado por la superficie. Luego deshabilitamos el sensor y enviamos el valor.

Debido al circuito que armamos con un nivel alto de reflexión leemos un valor bajo en el conversor analógico digital y con un nivel bajo de luz, un valor alto.

2.4.4. Encoders

Los encoders son sensores que convierten una posición lineal o angular en señal eléctrica o tren de pulsos. Pueden determinar una posición de forma absoluta o simplemente informar que hubo un movimiento. El método de sensado y la resolución del ángulo de giro que detectan varía según el modelo.

2.4.4.1. Características

Los motores *MR-2FA* tienen encoders de cuadratura conectados al eje, previo a la caja reductora. Estos encoders son de tipo fotoeléctricos, están dispuestos a 135° uno del otro y marcan 4 estados por cada vuelta del motor. El motivo de ésto es para poder conocer el sentido de giro midiendo la secuencia de estados de cada encoder. Adicionalmente el motor cuenta con un sensor de efecto Hall el cual nos permite detectar una revolución completa en el eje de salida de la caja reductora.

Como conocemos el sentido de giro del motor, pues lo determinamos con el puente H, necesitamos sólo una de las dos salidas del encoder para conocer y controlar la velocidad a la que gira el rotor del motor.

A la tensión máxima los motores superan las 320 cuentas por segundo, el mínimo y máximo recomendables son, para que podamos mantener un giro constante, 60 y 300 cuentas por segundo respectivamente. Estos cálculos son usando uno de los dos fototransistores del encoder.

2.4.4.2. Circuito de control

Conectamos una resistencia pull-up a 5V para la salida de sensado de los fototransistores y del sensor de efecto Hall. También incluimos un switch doble inversor para elegir cuál de los dos encoders usar. El punto común del switch está conectado al pin de entrada de clock externo de uno de los timers del microcontrolador.

La alimentación de los encoders es directa y permanecen encendidos en todo momento.

2.4.4.3. Rutinas de control

Como explicamos en la sección 2.3.1.3, en cada interrupción del timer actualizamos el histórico de cuentas del motor adicionando o restando el último valor del contador.

También comparamos contra las cuentas esperadas por intervalo de tiempo que fueron determinadas desde el controlador principal para poder determinar si debemos incrementar o disminuir la potencia del motor y por ende la velocidad de las ruedas.

2.4.5. Sensado de la batería

El sensado del nivel de tensión en la batería lo hacemos mediante un divisor de tensión entre los polos de la batería. La salida del divisor es sensada de igual forma que los otros sensores, mediante el módulo de *ADC* del microcontrolador. En la figura 11 mostramos el diagrama del divisor de tensión.

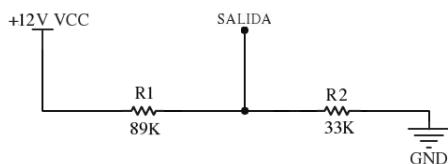


Figura 11: Divisor de tensión para el sensado de la batería.

En el cuadro 6 mostramos las posibles tensiones en la batería y la tensión de salida en el divisor. También incluimos el valor aproximado para el *ADC* con tensión de referencia a 5V que leería la salida del divisor. El rango de voltajes que analizamos es teniendo en cuenta la posibilidad de efectuar mediciones durante la carga de la batería y sabiendo que con una tensión menor a 5V la lógica del prototipo que construimos comenzaría a fallar.

La conexión es simple, los cables de entrada del divisor los conectamos a los polos de la batería y la salida al *ADC*. Luego sólo debemos realizar las lecturas en el *ADC* y promediarlas para poder realizar los cálculos de la tensión en la batería.

2.4.6. Consumo del motor

El consumo de los motores de corriente continua lo medimos leyendo el pin de sensado que se encuentra en el puente H que alimenta al motor. Lo que medimos con el módulo de *ADC* del microcontrolador es la tensión en este pin.

Batería (V)	Salida (V)	Valor en el ADC
16	4.327	886
15	4.057	831
14	3.786	776
13	3.516	720
12	3.245	665
11	2.975	609
10	2.704	554
9	2.434	499
8	2.163	443
7	1.893	388
6	1.623	332
5	1.352	277

Cuadro 6: Tensión de la batería y la tensión de salida en el divisor.

Ésta depende de la caída de tensión en la resistencia conectada a masa y de la corriente que circula por el motor. Conociendo el valor de la resistencia podemos calcular el consumo del motor. En el cuadro 7 comparamos el consumo en el motor, el voltaje sensado y la lectura en el ADC.

Tensión (V)	Consumo (A)	Valor en el ADC
0	0	0
0.09	0.19	18
0.18	0.38	36
0.27	0.57	55
0.36	0.76	73
0.45	0.95	92
0.54	1.14	110
0.63	1.34	129
0.72	1.53	147
0.81	1.72	165
0.90	1.91	184
0.99	2.10	202
1.08	2.29	221
1.17	2.48	239

Cuadro 7: Tabla comparativa para el consumo del motor.

2.4.6.1. Pulsador u otro dispositivo disparador

Además de los sensores que describimos, nos hace falta sensar switches que monitorean diversos estados del robot. Por ejemplo saber cuándo una parte de algún mecanismo llega a cierto punto o detectar finales de carrera de un servo o tornillo sin fin. También pueden ser otro tipo de sensores que generen un cambio de estado en el pin de sensado que se conecta al microcontrolador.

Agregamos la posibilidad de usarlos en las distintas placas como explicamos

en detalle en las secciones 2.7.3 y 2.7.4.

2.4.6.2. Rutinas de control

La lectura en el estado de los pulsadores puede ser bajo demanda con sólo leer el estado del pin en el que están conectados o puede ser ante una interrupción por cambio de estado. Estas cuestiones las tuvimos en cuenta a la hora de diseñar las placas.

2.5. Controladores

Todas las funciones del robot las debíamos controlar mediante algún tipo de dispositivo. Decidimos utilizar una netbook y microcontroladores para esta tarea. Estas cuestiones son las que explicamos en este capítulo.

2.5.1. Netbook

Elegimos como controlador principal la netbook *EeePC 1005-HA* de la marca Asus⁷. Como características principales cuenta con un procesador Intel⁸ Atom N280 1.66 GHz, 1GB DDR-2, un disco de 250GB, una pantalla de 10 pulgadas y una batería de 48Wh que nos da una autonomía de aproximadamente 8 horas. Cuenta con una cámara integrada de 0.3MP, placa de red inalámbrica y ethernet, placa de sonido y 3 puertos USB. Pesa aproximadamente 1.27Kg y sus dimensiones son 26.2 x 17.8 x 3.7 centímetros.

Usamos Ubuntu⁹ como sistema operativo y programamos tanto la lógica de comportamientos como la captura y análisis de imágenes en C/C++.

2.5.2. Microcontrolador

Para el control de la velocidad de los motores, lectura de los encoders y los sensores de distancia usamos un microcontrolador. En nuestro diseño contemplamos la existencia de varios módulos con una o pocas funciones simples que se comunicaran con el controlador principal, que para nuestro prototipo la netbook. La razón por la cual lo armamos así fue para simplificar cada placa controladora a nivel software y hardware.

Explicamos la comunicación entre los distintos controladores en la sección 2.6.

2.5.2.1. Características

El microcontrolador que elegimos para realizar las tareas de control, configuración y comunicación a bajo nivel es el *PIC16F88* de Microchip¹⁰. Cuenta con una arquitectura de memoria del tipo Harvard, con una memoria *FLASH* para 4096 instrucciones de programa, una memoria *RAM* de 368 bytes y una memoria *EEPROM* de 256 bytes. Tiene un set de instrucciones básicas reducido y todas con el mismo tiempo de ejecución. En este apartado nombramos algunos

⁷<http://www.asus.com/>

⁸<http://www.intel.com/>

⁹<http://www.ubuntu.com/>

¹⁰<http://www.microchip.com/>

de los principales periféricos incluidos en el microcontrolador y la utilidad dentro del proyecto que encontramos para ellos. Utilizamos un cristal externo de 20MHz como clock.

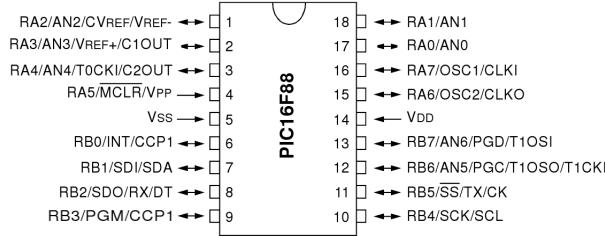


Figura 12: Diagrama del microcontrolador PIC16F88.

El microcontrolador tiene 2 puertos de 8 entradas y salidas cada uno de tipo TTL y CMOS. Como mostramos en la figura 12 cada pin se encuentra multiplexado con uno o más unidades internas.

2.5.2.2. Módulos internos

Internamente el microcontrolador tiene una serie de unidades que proveen funciones extras y que utilizamos para lograr cumplir con las necesidades de nuestro proyecto. En la figura 13 mostramos los distintos módulos internos del microcontrolador.

Cuenta con 3 timers, 2 de 8bits (*TIMER0* y *TIMER2*) y 1 de 16bits (*TIMER1*). Podemos configurarlos para que tomen al clock del microcontrolador o que tomen una fuente externa de clock. También podemos aplicarles demultiplicadores que generan un clock de menor frecuencia al que se usa como entrada. Pueden ser etapas previas o posteriores al timer y nos dan gran flexibilidad de uso.

El *TIMER0* lo utilizamos para hacer control del tiempo en nuestros códigos. Conectados con el clock principal y configurados para que generen una interrupción al hacer overflow, obtenemos una medida del paso del tiempo.

El *TIMER1* configurado como fuente externa a la salida del encoder, lo usamos como contador de pasos para medir la velocidad del motor. También lo usamos como medición del tiempo para hacer las lecturas de los sensores. Elegimos a este timer ya que al ser de 16bits posee un mayor rango de valores y

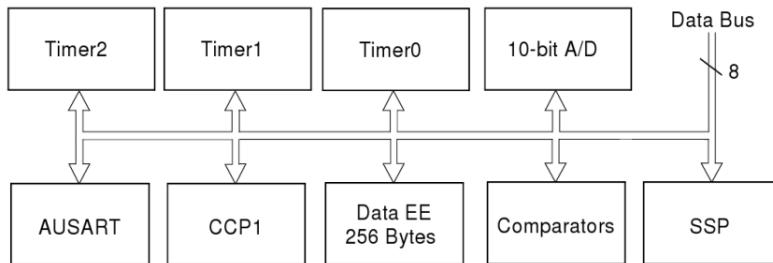


Figura 13: Módulos internos del microcontrolador PIC16F88.

Pines	Señal
1 y 2	MCLR
3 y 4	5V VCC
5 y 6	GND
7 y 8	PGD (Data)
9 y 10	PGC (Clock)

Cuadro 8: Pines de programación en circuito con *ICD2*.

por lo tanto, podíamos medir un mayor lapso de tiempo o cuentas del encoder en cada caso.

El *TIMER2* lo usamos en conjunto con el módulo de *PWM* para determinar el ancho del pulso que habilita al puente H que provee de energía a los motores.

El módulo conversor analógico digital nos dió la posibilidad de medir tensiones analógicas como por ejemplo las salidas de los sensores, tensión en la batería o el consumo de los motores. Tiene 7 canales o pines distintos y podemos configurarlo para que genere un valor de 8 o 10bits. También podemos determinar si se debe usar el valor de *Vcc* y *GND* como referencia o podemos proveer de forma externa de los voltajes de referencia para generar un rango de voltajes diferente y aumentar o disminuir así la resolución del conversor.

El módulo de *PWM* nos provee la posibilidad de generar pulsos continuos de un ancho determinado. En nuestro proyecto lo utilizamos como habilitación del puente H que alimenta a los motores variando así la potencia y por lo tanto, la velocidad final de las ruedas. Se utiliza en conjunto con el *TIMER2* fijando el prescaler y postscaler para determinar el ancho del estado alto y del estado bajo de los pulsos.

Gracias al módulo de *AUSART* podemos realizar la comunicación entre los distintos microcontroladores por hardware. Este periférico nos provee una comunicación sincrónica o asincrónica dependiendo de la configuración. Creamos la red de *Daisy-Chain* sobre el protocolo de RS-232.

El microcontrolador dispone de otros periféricos como un comparador *CCP* y comunicación sincrónica *SPI*. Utilizamos los pines de uso general para realizar otras funciones como habilitación, conmutación o las señales de *PWM* por software para determinar la posición de los servo motores, por ejemplo.

2.5.2.3. Programación del firmware

Para crear el firmware de cada microcontrolador usamos al IDE de programación *Microchip MPLAB*¹¹. Usamos el lenguaje C y lo compilamos con *CCS PCM V4.023*¹². Usamos el programador *ICD2* de la empresa *Microchip* para descargar código compilado al microcontrolador. Este programador, a su vez, nos dió la posibilidad de depurar el código en más de una oportunidad.

En el cuadro 8 detallamos el conector de programación en circuito, con el cual establecemos la interface con el programador *ICD2* para cargar el firmware en el microcontrolador o para depurarlo.

¹¹<http://www.microchip.com/>

¹²<http://www.ccsinfo.com/>

2.6. Comunicación

La comunicación interna entre los controladores de cada periférico y el controlador principal, donde concentraremos la lógica de los comportamientos, era vital. Para poder tomar las decisiones adecuadas la telemetría debía tener toda la información requerida, la frecuencia suficiente para que las reacciones sean lo más dinámicas y rápidas posibles y poseer la menor cantidad de errores para evitar retransmisiones.

En esta sección analizamos todo lo referente al medio de transmisión, protocolo y comandos que forman parte de la comunicación interna entre módulos.

2.6.1. Conectividad entre módulos

Para establecer el canal de comunicaciones decidimos emplear una configuración basada en el método *Daisy-Chain*¹³ entre los controladores. Creando un anillo donde cada nodo de la cadena se comunica con su vecino retransmitiendo cada paquete hasta su destinatario como mostramos en la figura 14.

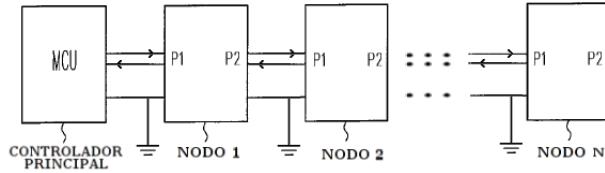


Figura 14: Diagrama general del método Daisy-Chain

La configuración que elegimos para realizar la comunicación fue una velocidad de 115200 baudios, 8 bits, con 1 bit de parada, sin bit de paridad y sin control de flujo. Logramos una gran velocidad de respuesta a los comandos de esta forma.

En los cuadros 9 y 10 especificamos el conexionado entre las placas y contra el controlador principal.

Función	Conector RJ11	Conector RJ11	Función
Serial RX	2	2	Serial TX
Serial TX	3	3	Serial RX
No conectado	4	4	No conectado
GND	5	5	GND

Cuadro 9: Conexionado entre placas en modo Link

Como terminación de la cadena, debemos colocar una ficha nula que interconecte *TX* con *RX* o cambiar el switch de configuración de *LINK* a *LAST*.

2.6.2. Protocolo de comunicación

El protocolo de comunicación está formado por paquetes que tienen un formato específico y representan un pedido de información o comando que debe

¹³Patente US20090316836A1

Función	Conektor RJ11	Conektor DB9	Función
Serial RX	2	3	Serial TX
Serial TX	3	2	Serial RX
No conectado	4	4	Shield
GND	5	5	GND

Cuadro 10: Conexionado entre placa y la PC

ser ejecutado en el destino.

El paquete consta de un encabezado común con datos que identifican al emisor y receptor del paquete, el comando a enviar y posibles datos extras que sean requeridos. Todos los paquetes tienen una respuesta obligatoria de confirmación de recepción. Cuando el paquete requiere una respuesta con datos, la confirmación va acompañada de la información requerida. En el cuadro 11 mostramos la estructura interna de un paquete típico.

LARGO	DESTINO	ORIGEN	COMANDO	DATO	XOR
-------	---------	--------	---------	------	-----

Cuadro 11: Formato y header del paquete de datos

Tanto los paquetes de envío de datos como los de respuesta tienen el mismo formato y comparten el valor en el campo de comando.

El campo *LARGO* indica el tamaño en bytes del paquete que viene a continuación. Consta de 1 byte. Esto es necesario porque el campo *DATO* es de longitud variable. Si la longitud del campo *DATO* es cero, entonces *LARGO* es 0x04.

El campo *DESTINO* identifica al destinatario del paquete y consta de 1 byte. Los 4 bits más significativos indican el grupo y los 4 bits menos significativos, el número de *ID* de la placa de destino. Si el *ID* de placa es 0x0F, el paquete es distribuido a todos los *IDs* del grupo indicado y si el valor es 0xFF, el paquete es distribuido a todas las placas de todos los grupos. En el cuadro 12 establecemos el *ID* de cada grupo.

De igual forma, *ORIGEN* determina el emisor del paquete para la respuesta y es de 1 byte. Los 4 bits más significativos indican el grupo y los 4 bits menos significativos, el número de *ID* de la placa de origen. Los valores permitidos son del 0 al E, ya que F indica broadcast y no es válida una respuesta broadcast.

El campo *COMANDO* informa al destinatario la tarea a realizar o determina un pedido de información que puede o no tener parámetros. Ocupa 1 byte.

El campo *DATO* contiene los parámetros o datos extras que puedan ser necesarios para el comando enviado. En el caso que el comando no los requiera, el campo debe ser nulo y el campo *LARGO* será 0x04.

El control de errores lo realizamos mediante un checksum calculado, haciendo un *XOR* con cada byte del contenido del paquete y colocándolo en el campo *XOR*. Cuando un paquete se encuentre con errores o esté mal formado, el destinatario debería pedir la retransmisión. De igual forma, creímos necesaria la creación de un control adicional por medio de una lista de paquetes no confirmados mantenida por el controlador principal para evitar la pérdida de comandos.

Para más referencias recomendamos ver el apartado A.

Grupo	ID
MAIN CONTROLLER	0x00
DC MOTOR	0x10
SERVO MOTOR	0x20
DISTANCE SENSOR	0x30
BATTERY CONTROLLER	0x40
TRASH BIN	0x50
-BROADCAST-	0xF0

Cuadro 12: ID para cada grupo según su función.

2.6.2.1. Comandos comunes

Creamos comandos que son comunes a cualquier placa sea cual sea su grupo. La intención fue poder tener un espacio de comunicación común para poder intercambiar comandos de inicialización, identificación de controladores, identificación de versión o alguna otra necesidad genérica que se desarrolle en un futuro y deba ser incluida en el protocolo. El controlador principal debería ser quien envíe este tipo de comandos, pero no creamos restricciones al respecto.

En el cuadro 13 listamos y explicamos brevemente los comandos comunes del protocolo.

Valor	Comando	Detalle
0x01	INIT	Sincroniza el inicio de todas las placas en la cadena. Responde con los datos necesarios para identificar a la placa en la cadena.
0x02	RESET	Pide el reset de la tarjeta. Responde de igual forma que a <i>INIT</i> .
0x03	PING	Envía un ping a la placa.
0x04	ERROR	Informa que ha habido un error. Más información sobre el error viaja en el campo de <i>DATO</i> y depende del grupo de origen.

Cuadro 13: Comandos comunes a todos los controladores.

2.6.2.2. Comandos específicos

Cada grupo de placas tiene comandos propios y específicos dependiendo de la función que deban desempeñar en el sistema. Existen grupos con sus comandos predefinidos. Los comandos específicos para cada grupo deben estar dentro del rango de valores entre 0x40 y 0x7E.

El *MAIN CONTROLLER* no posee por ahora ningún comando específico pero le reservamos el espacio en el protocolo. En el cuadro 14 enumeramos los comandos para el controlador *DC MOTOR*. El controlador *SERVO MOTOR* responde a los comandos del cuadro 15. En el cuadro 16 listamos los comandos del controlador *DISTANCE SENSOR*. De igual forma, aunque no estén implementados, reservamos y especificamos los comandos para *BATTERY CONTROLLER* y *TRASH BIN* en los cuadros 17 y 18 respectivamente.

Valor	Comando	Detalle
0x40	SET DIRECTION	Fija el sentido de giro del motor. Envía el sentido de giro como parámetro.
0x41	SET DC SPEED	Fija la velocidad del motor. Envía el sentido de giro y velocidad en cuentas por segundos como parámetro.
0x42	SET EN-CODER	Seteo de las cuentas históricas del encoder. Envía como parámetro el valor que debe tener el histórico del encoder.
0x43	GET EN-CODER	Obtener las cuentas históricas del encoder. Responde con el valor histórico del encoder.
0x44	RESET EN-CODER	Pone las cuentas históricas a cero.
0x45	SET EN-CODER TO STOP	Establece en cuántas cuentas debe detenerse el motor. Envía como parámetro la cantidad de cuentas del encoder restantes para que el motor se detenga.
0x46	GET EN-CODER TO STOP	Obtener las cuentas restantes hasta detenerse. Responde con la cantidad de cuentas del encoder restantes para que el motor se detenga.
0x47	DONT STOP	Deshabilita el conteo de cuentas para frenar.
0x48	MOTOR CONSUMPTION	Consulta sobre el consumo actual del motor. Responde con el consumo promedio del último segundo.
0x49	MOTOR STRESS ALARM	Alarma sobre un consumo extremo en el motor. Responde con el consumo ante el cuál se disparó la alarma.
0x4A	MOTOR SHUT DOWN ALARM	Alarma informando que el motor se apagó. Envía el consumo ante el que se disparó la alarma.
0x4B	GET DC SPEED	Obtiene la velocidad en cuentas del encoder por segundo. Envía el sentido y velocidad de giro del motor.

Cuadro 14: Comandos específicos al *DC MOTOR*.

Valor	Comando	Detalle
0x40	SET POSITION	Determina la posición del servo motor indicado. Envía el <i>ID</i> del servo y la posición del eje como parámetro.
0x41	SET ALL POSITIONS	Establece la posición de cada uno de los servo motores. Envía las posiciones para cada uno de los 5 servos como parámetro.
0x42	GET POSITION	Obtiene la última posición del servo motor indicado. Envía como parámetro el <i>ID</i> del servo del que se quiere la posición y recibe el <i>ID</i> del servo y la posición del eje.
0x43	GET ALL POSITIONS	Obtiene todas las últimas posiciones de los servo motor. Responde con la posición para cada uno de los 5 servos.
0x44	SET SERVO SPEED	Fija la velocidad para el servo motor indicado. Envía como parámetro el <i>ID</i> del servo y la velocidad a setear.
0x45	SET ALL SPEEDS	Establece la velocidad para cada servo motor. Envía como parámetro la velocidad a setear para cada uno de los servos.
0x46	GET SERVO SPEED	Obtiene la velocidad para el servo motor indicado. Envía como parámetro el <i>ID</i> del servo y responde con <i>ID</i> y velocidad del servo.
0x47	GET ALL SPEEDS	Obtiene las velocidades de cada uno de los servo motores. Responde con la velocidad de cada uno de los servo motores.
0x48	FREE SERVO	Deja de aplicar fuerza sobre el servo indicado. Envía como parámetro el <i>ID</i> del servo a liberar.
0x49	FREE ALL SERVOS	Deja de aplicar fuerza sobre cada uno de los servo motores.
0x4A	GET STATUS	Obtiene el estado de cada uno de los switches. Responde con la 1 byte con el estado de cada switch.
0x4B	ALARM ON STATE	Establece si se desea recibir alarmas por cambio de estado. Envía como parámetro el <i>ID</i> y tipo de cambio en el switch.
0x4C	SWITCH ALARM	Alarma ante un cambio de estado programado. Envía como parámetro el <i>ID</i> y estado del switch que provocó el comando.

Cuadro 15: Comandos específicos al *SERVO MOTOR*.

Valor	Comando	Detalle
0x40	ON DISTANCE SENSOR	Enciende el sensor de distancia enviado como parámetro.
0x41	OFF DISTANCE SENSOR	Apaga el sensor de distancia enviado como parámetro.
0x42	SET DISTANCE SENSORS MASK	Habilita o no los sensores para lecturas. Envía 1 bit por cada <i>ID</i> del sensor a habilitar o deshabilitar.
0x43	GET DISTANCE SENSORS MASK	Obtiene la máscara de lectura. Responde con 1 bit por cada <i>ID</i> del sensor.
0x44	GET VALUE	Obtiene el valor promedio de la entrada de los sensores indicados. Envía 1 bit por cada <i>ID</i> del sensor y responde con los valores de lectura promedio de cada sensor.
0x45	GET ONE VALUE	Obtiene el valor de la entrada del sensor indicado. Envía 1 bit por cada <i>ID</i> del sensor y responde con los valores de lectura de cada sensor.
0x46	ALARM STATE	Setea el tipo de cambio de estado del switch para la alarma. Envía como parámetro el tipo de cambio de estado.
0x47	SWITCH ALARM	Alarma informando que fue satisfecha la condición. Envía como parámetro el tipo de cambio y estado actual del switch.

Cuadro 16: Comandos específicos al *DISTANCE SENSOR*.

2.7. Placas controladoras

Las funciones y requerimientos de nuestro proyecto tenían partes que eran comunes a otros trabajos en robótica o quizás de electrónica general. Pero tenían peculiaridades y aspectos que no pudimos satisfacer con placas prefabricadas. Por ejemplo, un protocolo de comunicación unificado entre todos los módulos era, de entrada, una barrera que acotaba las posibilidades. Además mantener distintos modos y protocolos dentro del proyecto generaba una carga de lógica que podíamos evitar.

Analizando las posibilidades y entendiendo que una de las principales razones de nuestro trabajo era generar las bases de conocimiento y el punto de partida a un proyecto más grande, decidimos producir nuestra propia versión de placas controladoras.

Aunque iban a tener un uso bien definido dentro de nuestro trabajo, también creímos necesario que tuvieran un nivel de generalidad suficiente como para poder utilizarlas en otros proyectos con mínimos cambios, adelantando así mucho trabajo a futuro.

La modularización fue un factor importante y muy presente durante todo el desarrollo. En este apartado explicamos el diseño, desarrollo, programación y especificaciones de las distintas placas que creamos durante nuestro proyecto.

Valor	Comando	Detalle
0x40	ENABLE	Habilita la alimentación del robot mediante la batería.
0x41	DISABLE	Deshabilita la alimentación del robot mediante la batería.
0x42	GET BATTERY VALUE	Obtiene el valor de la entrada de la batería. Responde con la lectura de la tensión en la batería.
0x43	BATTERY FULL ALARM	Mensaje informando que se completado la carga.
0x44	SET BATTERY EMPTY VALUE	Valor de la batería crítico. Envía la lectura de los volts de la batería.
0x45	BATTERY EMPTY ALARM	El voltaje llegó a un valor crítico. Envía la lectura de los volts de la batería.
0x46	SET FULL BATTERY VALUE	Establece el valor para la carga completa. Envía la lectura de la tensión en la batería.

Cuadro 17: Comandos específicos al *BATTERY CONTROLLER*.

Valor	Comando	Detalle
0x40	GET TRASH BIN VALUE	Consulta que tan lleno está el cesto de basura. Responde con el valor que representa el estado del cesto interno.
0x41	BIN FULL ALARM	El cesto de basura se ha completado y debe ser descargado.
0x42	SET FULL BIN VALUE	Establece el valor para determinar que el cesto está lleno. Valor que especifica que lectura se debe tomar como cesto lleno.

Cuadro 18: Comandos específicos al *TRASH BIN*.

2.7.1. Placa genérica

Durante el diseño nos surgió la necesidad de establecer un módulo común de comunicación y una base de prueba para los tests con los distintos sensores y periféricos que utilizaríamos en el robot. Es por ésto que luego de pruebas aisladas, creamos una placa para estas cuestiones. Nos ayudó y aceleró en gran medida el diseño de las placas definitivas y nos dió la posibilidad de diseñar futuras actualizaciones a nuestro proyecto.

2.7.1.1. Características principales

La necesidad principal era proveer de una interface común entre todas las placas para la comunicación y exportar los puertos y unidades del microcontrolador que elegimos a circuitos nuevos. Ésto nos sirvió para realizar pruebas de concepto para la conexión con los sensores, valores de componente pasivos

Pin	Voltaje
1	GND
2	5V
3	7V a 12V

Cuadro 19: Alimentación de la lógica

ideales, comportamiento e interacción entre placas, optimización del código del firmware o crear nuevas expansiones.

2.7.1.2. Módulo de comunicación

Como explicamos en la sección 2.6 para la comunicación nos basamos en el modelo de *Daisy-Chain* sobre una capa de transporte de *RS232*.

Los conectores y configuración son comunes y utilizamos dos tipos de conectores para realizar la interconexión entre las placas. Como mostramos en la figura 15 usamos conectores *RJ11* para generar el lazo entre nodos de la cadena y el conector *DB9* para la conexión con la PC. En los cuadros 9 y 10 explicamos, respectivamente, el conexionado para cada caso.

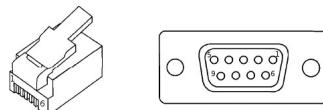


Figura 15: Conectores RJ11 (6P4C) y DB9.

Para determinar si la placa es un nodo más dentro de la cadena o es la terminación, colocamos una llave de dos posiciones que establece la configuración. En la posición *LINK* la placa actúa como nodo intermedio y en la posición *LAST* como terminación.

Es de vital importancia colocar en forma correcta esta llave porque podemos dejar sin conexión al resto de las placas o mismo perder todos los paquetes transmitidos al no cerrar la cadena en el final de la misma.

2.7.1.3. Alimentación de la placa

La alimentación principal de la placa es 7 a 20 volts, con la posibilidad de alimentarla directamente con 5 volts por uno de los conectores. En la figura 16 mostramos la bornera y en el cuadro 19 el pinout.



Figura 16: Bornera de alimentación.

La regulación interna de voltaje la realizamos por medio de un regulador 7805 y tiene una corriente máxima de 1A.

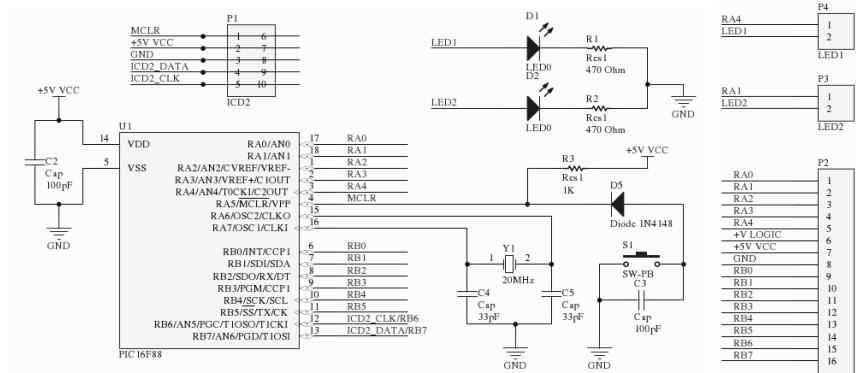


Figura 17: Microcontrolador y headers

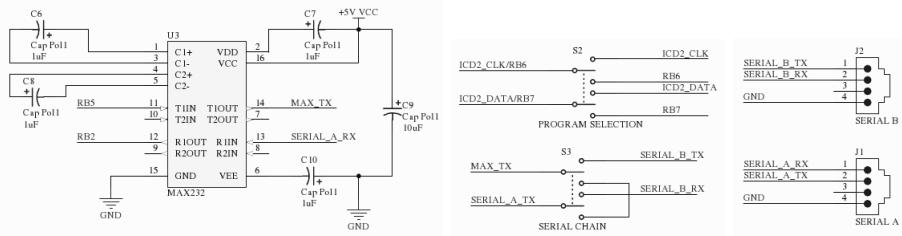


Figura 18: Comunicación, switch de modo y conectores de entrada y salida

2.7.1.4. Configuración

La fila de pines *P2* exporta todos los pines con funciones dentro del microcontrolador, para realizar conexiones con periféricos de prueba o nuevas extensiones. Los headers *P3* y *P4* son jumpers que vinculan los pines *RA1* y *RA4* del microcontrolador los leds 1 y 2 respectivamente.

El header de programación *P1* lo utilizamos para conectar la placa con el programador *ICD2* y depurar el código como explicamos en el apartado 2.5.2.3. El switch *S2* lo usamos para asociar los pines del microcontrolador con los canales de clock y data del conector de programación (modo *ICD2*) o con los pines *RB6* y *RB7* del header *P2*.

2.7.1.5. Esquemático

En la figura 17 mostramos el esquemático del microcontrolador y el conexionado con los conectores. El módulo de comunicación y conectores para conformar la cadena *Daisy-Chain* los detallamos en la figura 18. En la figura 19 mostramos el diagrama de la fuente de alimentación y bornera.

2.7.1.6. Circuito

En la figura 20 mostramos la máscara de componentes de la placa y en la figura 21 ambas capas de la placa.

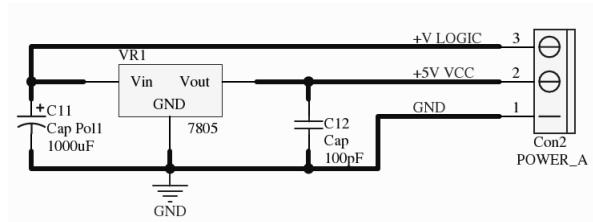


Figura 19: Fuente de alimentación

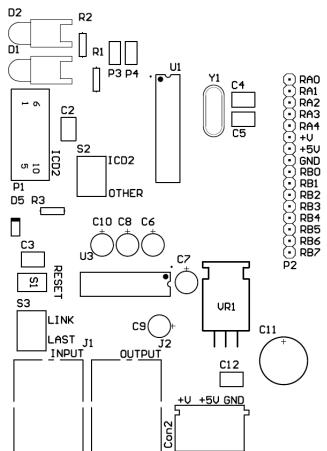


Figura 20: Máscara de componentes de la placa genérica.

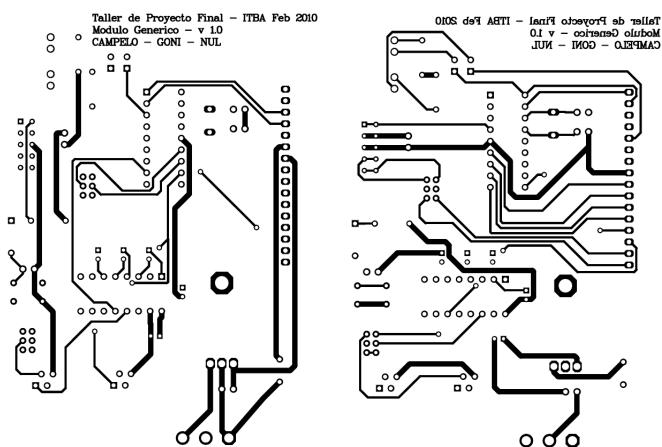


Figura 21: Capas superior e inferior de la placa genérica.

2.7.1.7. Código básico

Según el uso que le fuimos dando a las placas genéricas fuimos cambiando el código del firmware que le íbamos cargando, pero siempre manteníamos partes básicas que nos ayudaban.

Éstas principalmente incluían la configuración de flags de configuración base del microcontrolador, desde la velocidad del clock para los cálculos de tiempos hasta determinar qué pines eran mapeados a las funciones de RX y TX para la comunicación serial.

Para hacer pruebas sobre la comunicación RS232 o del protocolo, preparamos placas para las funciones que atienden a los comandos específicos y funciones de control del protocolo comunes para todas las placas. También generamos un modelo general del programa, con un loop principal y llamadas de inicialización. De esta sólo nos concentrábamos en la prueba que estábamos realizando.

2.7.1.8. Posibles extensiones

Son varias las extensiones que se podrían realizar a estas placas pero teniendo en cuenta que debían ser lo más genéricas posibles como para probar nuevas funcionalidades o componentes, creemos que aportan mucho valor en la fase de testeo. El montaje de las placas se realizó con componentes standard. En una fase posterior a este proyecto final, el uso de montaje superficial nos permitiría el diseño de placas más compactas.

2.7.2. Placa controladora de motores DC

Nuestro proyecto implicaba, entre otras cosas, el diseño y construcción de un robot móvil el cual debía poder trasladarse por el terreno como primer requerimiento. Y debido a que el desplazamiento iba a ser mediante ruedas, el control de los motores que las impulsaran era crítico.

Creamos una placa especializada para esta tarea sobredimensionando los requerimientos para no tener acotadas las opciones a futuro sobre qué modelo de motores usaríamos.

En este apartado explicamos los aspectos que tuvimos en cuenta para diseñar y crear la placa controladora de motores de corriente continua para el robot.

2.7.2.1. Características principales

La función principal, es entre otras, mantener una velocidad estable en el motor de continua, pero también poder sensar la cantidad y sentido del movimiento que realizó la rueda, informar el consumo y poder determinar el desplazamiento a realizar. Este último requerimiento era de vital importancia si queríamos tener precisión en el movimiento que realizaría la rueda ya que este control de rápida respuesta hubiera sido imposible hacerlo desde el controlador principal.

Todos los comandos desde y hacia el controlador principal viajan por la red de comunicación como explicamos en la sección 2.6 colocándose como un eslabón más en la cadena. Usamos como base la placa genérica que explicamos en la sección 2.7.1 y agregamos el circuito necesario para el control del motor como explicamos en el punto 2.7.2.6.

Pin	Voltaje
1	GND
2	12v (depende del motor)

Cuadro 20: Pines de alimentación del motor.

2.7.2.2. Comunicación

Al utilizar la placa genérica como base mantuvimos la sección de comunicación de la misma intacta, pues en principio, era un módulo probado y funcional, además manteníamos la misma estructura física que facilitaba el conexionado para futuros usuarios.

Como explicamos en la sección 2.6.2.2 esta placa responde a comandos específicos listados en el cuadro 14. La principal información disponible mediante estos comandos es la velocidad de la rueda en cuentas del encoder por segundo, el sentido de giro y el histórico de cuentas realizadas. Podemos enviar comandos para establecer estos valores y otros, como la cantidad de cuentas del encoder y a qué velocidad debe girar la rueda.

También podemos leer la cantidad de cuentas restantes antes de frenar y el consumo actual del motor para hacer cálculos sobre el tiempo restante de batería.

2.7.2.3. Alimentación de la placa

Como explicamos en la sección 2.7.1, la alimentación principal de la lógica es de 7 a 20 volts, con la posibilidad de alimentarla directamente con 5 volts a través de uno de los conectores. La alimentación del motor es por una bornera de 2 pines como listamos en el cuadro 20, aislada completamente para disminuir así el ruido generado por el funcionamiento del motor sobre la lógica.

Como la alimentación del motor se hace mediante el *puente H*, no puede superar los 46V que es el máximo que puede operar el L298.

Cabe aclarar también que al estar ambos circuitos completamente aislados, se necesita unificar ambas masas para establecer un punto de referencia común y así poder ejercer el control del *puente H* y la lectura del consumo, de forma correcta. Recomendamos hacer esto lo más cerca posible de la batería para disminuir al máximo el ruido que pueda generarse debido a los motores.

2.7.2.4. Configuración

La configuración de la placa es relativamente sencilla. Antes que nada debemos determinar el rol de la placa dentro de la cadena de comunicación, esto lo hacemos mediante el switch S3. Podemos colocarlo en modo *LINK* para que sea un eslabón más o en modo *LAST* para que sea la terminación de la cadena.

El conector del motor comunica la placa con el motor de continua (en el cuadro 21 detallamos la posición de los pines).

Las conexiones *MOTOR_B* y *MOTOR_A* son para la alimentación del motor. Dependiendo del sentido de circulación de la corriente entre estos pines será el sentido de giro del motor. Ésto se controla desde el puente H L298.

Los sensores dentro del motor reciben tensión mediante los pines *5V VCC* y *GND*. La señal *IDX* genera pulsos según las vueltas en el eje de salida de la

Pin	Conexión	Pin	Conexión
1	MOTOR_B	2	MOTOR_A
3	IDX	4	-
5	ENCODER_B	6	GND
7	ENCODER_A	8	GND
9	5V VCC	10	GND

Cuadro 21: Pines del header de comunicación con el motor.

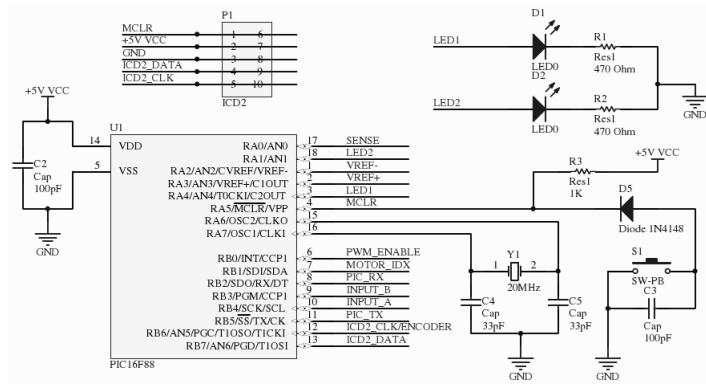


Figura 22: Microcontrolador y header de programación.

caja de reducción del motor. En cambio las señales *Encoder_A* o *Encoder_B* son pulsos en base a las vueltas del motor antes de entrar en la caja.

El switch *S2* lo utilizamos para asociar los pines del microcontrolador con el canal de datos del conector de programación, que explicamos en la sección 2.5.2.3, permitiendo el modo *ICD2* o el modo de lectura de la señal del encoder.

La señal del encoder proviene a su vez del switch *S4*, el cual nos da la posibilidad de elegir entre alguna de las dos señales *Encoder_A* o *Encoder_B*.

Podemos usar a los pines *P2* y *P3* para tomar como referencia del conversor analógico digital a *GND* y *VCC*, o al divisor de tensión formado por las resistencias *R5* y *R6*.

2.7.2.5. Esquemático

En la figura 22 mostramos el esquemático del microcontrolador y el conexionado el conector de programación. En la figura 23 detallamos el módulo de comunicación y el conexionado con los conectores entre placas. En la figura 24 mostramos el esquemático del driver y el header del motor. En la figura 25 mostramos los pines del divisor de tensión para el conversor analógico digital y en la figura 26 la fuente de alimentación y borneras.

2.7.2.6. Circuito

En la figura 27 mostramos la máscara de componentes de la placa. En la figura 28 mostramos ambas capas de la placa.

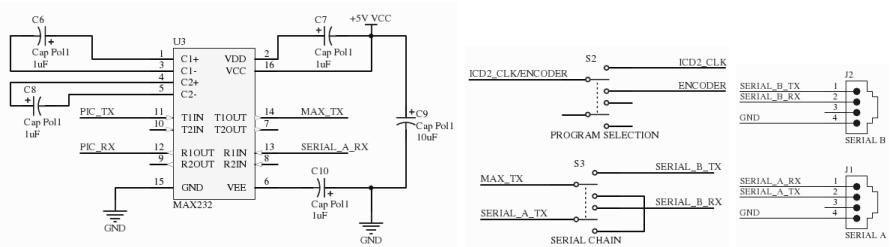


Figura 23: Comunicación, switch de modo y conectores de entrada y salida.

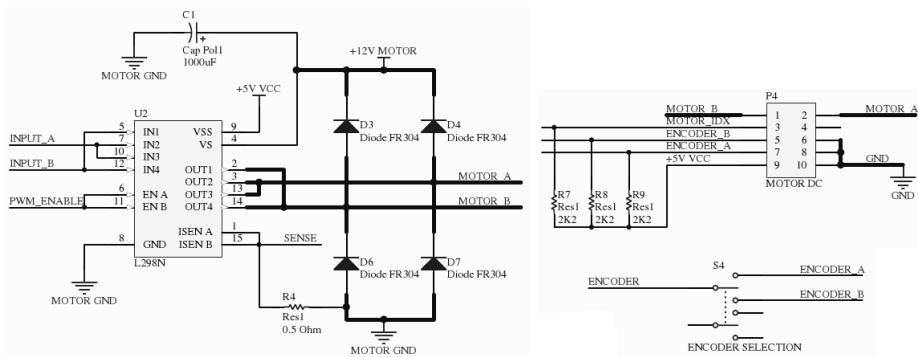


Figura 24: Driver y header de conexión con el motor.

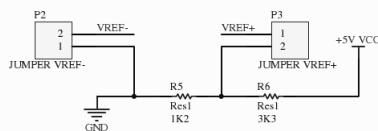


Figura 25: Divisor de tensión para el voltaje de referencia.

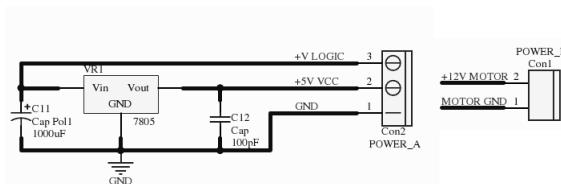


Figura 26: Fuente de alimentación de la lógica y motor.

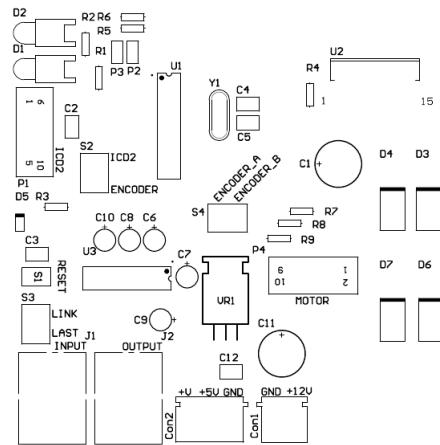


Figura 27: Máscara de componentes de la placa controladora de motores DC.

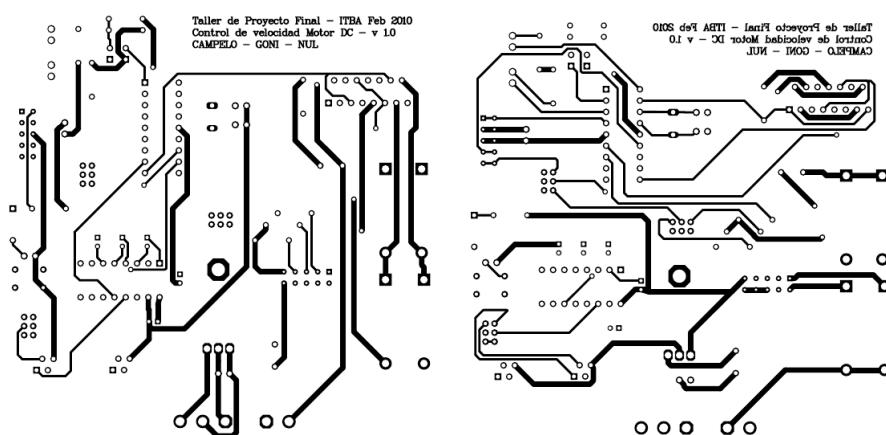


Figura 28: Capas superior e inferior de la placa controladora de motores DC.

2.7.2.7. Código básico

El código mínimo para ser parte de la cadena de comunicación es similar en todas las placas que armamos. La personalización del código está dentro de la función que interpreta el comando recibido y que realiza las tareas internas necesarias para llevar a cabo dicho comando y depende íntegramente del protocolo.

Para el sensado de la velocidad debemos determinar cuáles son los pines de lectura de la señal del encoder y debemos generar una base de tiempo que nos permita establecer cuántas cuentas del encoder hay por intervalo de tiempo. Con dicha información podemos aumentar o disminuir el ancho del pulso que controla al puente H *L298* y, en consecuencia, controlar la velocidad del motor de corriente continua.

Para mantener un histórico de cuentas de encoder ocurridas nosotros usamos el conocimiento del sentido de giro del motor para sumar o restar el valor calculado por cada intervalo de tiempo en una variable interna del microcontrolador.

2.7.2.8. Posibles extensiones

Un punto recurrente de discusión fue el uso de una placa por cada motor de corriente continua. Sabíamos que era posible realizar todas las funciones necesarias para el control de dos motores en una única placa, pero esto nos traía una mayor complejidad tanto a nivel físico como a nivel lógico de control, y un tiempo de implementación superior. Es por estas razones que decidimos esta configuración y dejarlo como futura extensión para la cual usaríamos los conocimientos y experiencias adquiridas en el desarrollo de la actual.

2.7.3. Placas de sensado

Para sensar el entorno utilizamos distintos tipos de sensores que debíamos leer y controlar según fuera requerido desde el controlador principal. Diseñamos una placa a la que podemos conectarle cualquiera de los sensores que usamos en nuestro proyecto, ya sea un sensor de distancia por ultrasonido, telémetros o sensores reflectivos de piso. También utilizamos esta placa para realizar las lecturas de la tensión de la batería. En este apartado detallamos cada una de las características que tuvimos en cuenta durante el desarrollo de las placas de sensado.

2.7.3.1. Características principales

El uso de diferentes tipos de sensores implicaba un método distinto de lectura según qué sensor estaba conectado. Los sensores de distancia por ultrasonido que usamos generan un pulso de ancho variable según la distancia al objetivo. De forma similar, los telémetros tienen un voltaje en salida dependiendo de la distancia con el objeto. Los sensores reflectivos de piso que usamos miden el nivel de luz infrarroja que refleja la superficie que tienen debajo de ellos.

Para los sensores que devuelven el valor sensado mediante un voltaje, utilizamos el conversor analógico digital para tomar las muestras necesarias. Salvo que se indique lo contrario, tomamos 5 muestras y luego las promediamos para minimizar el error de muestreo.

En el caso del sensor de ultrasonido usamos el módulo de timer para calcular

el ancho del pulso generado por el sensor. También usamos interrupciones para captar los flancos ascendentes y descendentes para determinar el comienzo y fin del pulso respectivamente.

Armamos la placa para poder conectar un sensor de distancia por ultrasonido o un switch binario sobre el mismo puerto de conexión. De manera conjunta podemos conectar a la misma placa, hasta cinco sensores por voltaje, telémetros, fototransistores (sensores de piso) o por ejemplo el divisor de tensión que mide la carga en la batería.

Generamos la posibilidad de habilitar y deshabilitar la alimentación de estos sensores mediante el uso de un transistor *BC327* y así aumentar la autonomía del robot. Este transistor tiene una corriente máxima de hasta $500mA$ para alimentar al sensor que sea necesario.

2.7.3.2. Módulo de comunicación

De igual forma que las otras placas en el proyecto, mantenemos la misma disposición física de la comunicación, desde el circuito base hasta la ubicación de los conectores para mayor facilidad a la hora de utilizarlas. También respetamos el protocolo de comunicación definiendo nuevos comandos específicos para la placa y retransmitiendo los paquetes con otro destinatario.

2.7.3.3. Alimentación de la placa

De igual forma que las otras placas la alimentación principal de la lógica es de 7 a 20 volts, con la posibilidad de alimentarla directamente con 5 volts como explicamos en la sección 2.7.1.

2.7.3.4. Configuración

Para la configuración física de la placa necesitamos definir primero el rol de la placa dentro de la cadena de comunicación con la llave *S3*. Como en todas las placas que diseñamos sólo tenemos que elegir el modo *LAST* o *LINK* para determinar si es la terminación o un eslabón más en la cadena.

Debemos saber que no podemos conectar más de 5 sensores de los cuales debamos medir una tensión para obtener el valor. Sólo un sensor de distancia por ultrasonido o switch en el puerto para medir estados lógicos y tiempo del pulso.

Con la llave *S2* cambiamos al modo programación vinculando el microcontrolador con el conector de *ICD2* o con la habilitación de los puertos de los sensores 3 y 4. Si esta llave está en la posición incorrecta nunca se excitará el transistor y por ende no habrá alimentación en el sensor.

En el caso que necesitemos una resistencia *pull-up* conectada con la salida del sensor podemos colocar un jumper en los pines de pull-up del puerto de sensor.

2.7.3.5. Esquemático

En la figura 29 mostramos el esquemático del microcontrolador y el conector de programación. En la figura 30 detallamos el módulo de comunicación y el conexionado con los conectores entre placas.

En la figura 31 mostramos los puertos de conexión para cada uno de los sensores y en la figura 32 la fuente de alimentación y bornera.

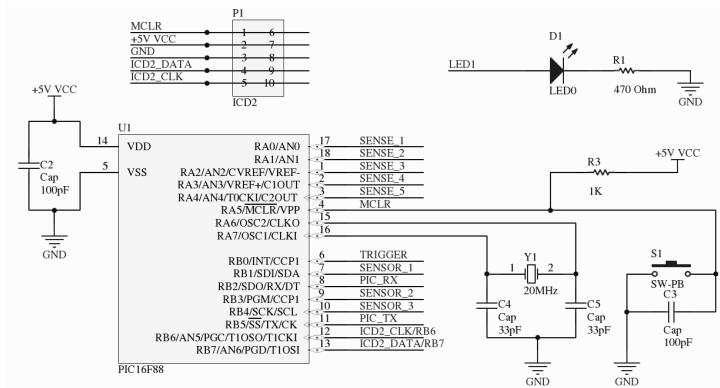


Figura 29: Microcontrolador y conector de programación.

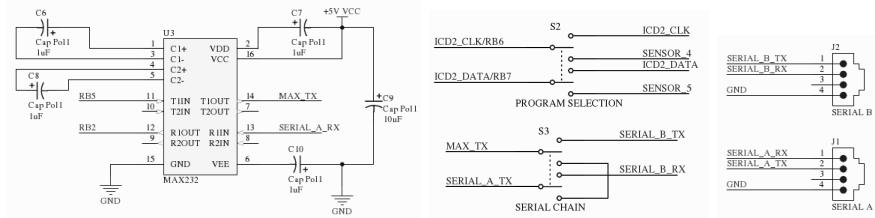


Figura 30: Comunicación, llaves de modo y conectores de entrada y salida.

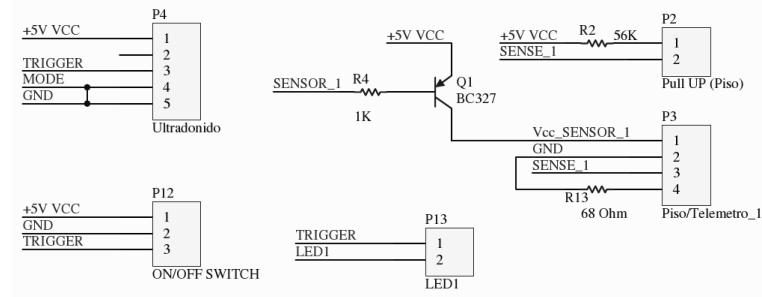


Figura 31: Puertos de conexión para los sensores y header de *pull-up*.

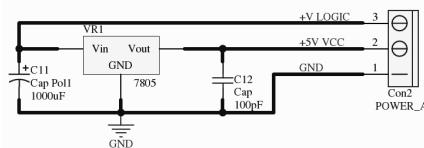


Figura 32: Fuente de alimentación de la lógica.

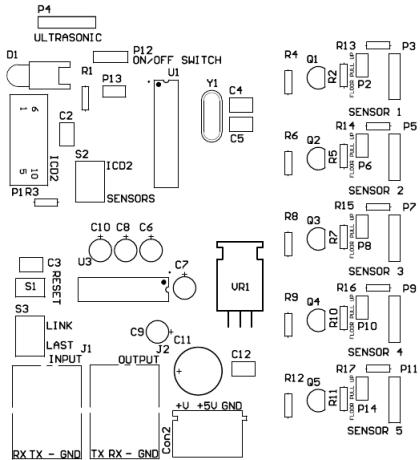


Figura 33: Máscara de componentes de la placa controladora de sensores.

2.7.3.6. Circuito

En la figura 33 mostramos la máscara de componentes de la placa. En la figura 34 mostramos ambas capas de la placa.

Una observación importante en el circuito de esta placa es que al momento de montarla observamos una gran inestabilidad en el microcontrolador cuando hacíamos lecturas simultáneas en más de 2 telémetros. Esto generaba picos de consumo que, a pesar del filtrado, generaban un reset en el microcontrolador. Para solucionar esto colocamos un capacitor de $100\mu F$ entre las pistas de $+5V$ y GND en medio de los puertos de los sensores 1 y 2.

Luego de este pequeño cambio pudimos seguir con nuestro desarrollo y las siguientes pruebas mostraron la estabilidad que necesitábamos en el microcontrolador.

2.7.3.7. Código básico

Al compartir el módulo de comunicación entre las placas y para que nuestra placa sea parte de la cadena y podamos interpretar los paquetes que viajan por ella, debemos incluir el código común de comunicación e implementar la función que analiza y ejecuta los paquetes que son dirigidos directamente a nuestra placa, grupo o a nivel broadcast.

Dependiendo de la selección de los sensores a conectar será la configuración interna del microcontrolador. Al principio del código colocamos una serie de scripts que nos ayudan a configurar los tiempos a los que se deben tomar las muestras de los sensores y los flags del microcontrolador.

Los pines que excitan a la base de los transistores usan una lógica inversa, es decir, un estado bajo habilita la alimentación del sensor y un estado alto, lo deshabilita.

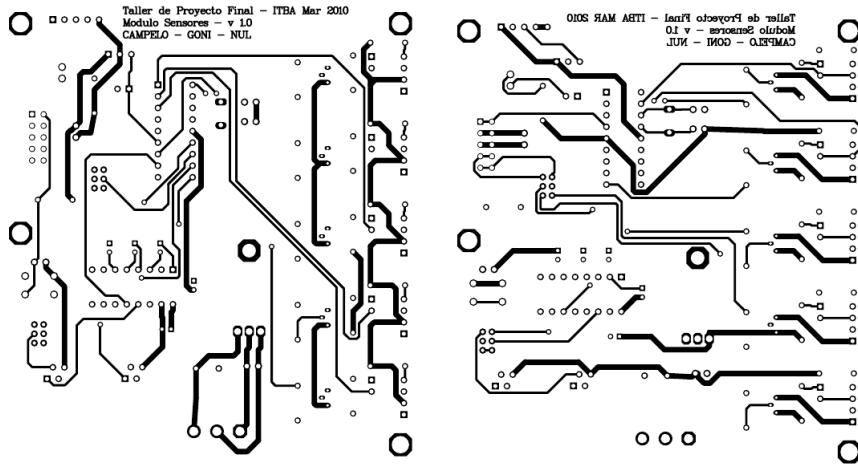


Figura 34: Capas superior e inferior de la placa controladora de sensores.

2.7.3.8. Posibles extensiones

Dentro de las posibles extensiones que creemos que serían útiles a futuro podemos decir que la más significativa sería usar potenciómetros en vez de resistencias fijas para usar como *pull-up* y para variar la alimentación de los sensores de piso, por ejemplo.

En una implementación futura también agregaríamos el uso de componentes de montaje superficial y aumentaríamos la cantidad de puertos para sensores disponibles. De igual forma, incorporar otro tipo de sensores nos parece una interesante mejora.

2.7.4. Placa controladora de servo motores

Para los mecanismos que implicaban algún tipo de movimiento y control de la posición elegimos usar servo motores, ya sea para el mecanismo de recolección de basura, inclinar el robot, reclinarse el cesto interno de basura o realizar algún movimiento de paneo o foco con la cámara.

Diseñamos esta placa teniendo en cuenta que usaríamos varios servos en el robot y que también necesitaríamos tener la posibilidad de recibir señales del estilo *fin de carrera* o algún estímulo generado por pulsadores que modificaran o no el accionar de los motores.

Esta placa no la construimos para este prototipo debido a que no contábamos con la necesidad. Al no implementar ningún método físico de recolección, no teníamos mecanismos que la usaran, aunque sí entendimos que era algo que debíamos hacer para tener el conocimiento y sentar una base para futuras implementaciones o proyectos similares.

2.7.4.1. Características principales

Diseñamos la placa pensando en que tendríamos que controlar cinco servo motores. Debido a que pensamos usar servo motores de tipo digital necesitábamos varios módulos de PWM para realizar esta tarea. Pensamos en el

uso de otros microcontroladores que pudieran proveernos de esta facilidad, pero finalmente decidimos implementar esta funcionalidad por software.

A los puertos que no están destinados al control de los servo motores les podemos dar varios usos. En un principio los pensamos como generadores de interrupciones ante un cambio de estado y en el puerto *P9* contamos con la posibilidad de determinar el flanco ante el cual se generará la interrupción. En el puerto *P8* podemos colocar algún tipo de sensado que haga uso del conversor analógico digital del microcontrolador.

De igual forma podríamos extender la rutina que genera el PWM por software, a todos los puertos y controlar hasta 11 servos con una única placa.

2.7.4.2. Módulo de comunicación

Como mantuvimos la sección de comunicación, para ser parte de la cadena sólo debemos incluir el código que implementa el protocolo e implementar las acciones necesarias dentro de la función que ejecuta los comandos.

Como explicamos en la sección 2.6.2.2 creamos ciertos comandos o paquetes específicos para el control de los servos. De igual forma contemplamos el uso de switches en la misma.

Un mayor cambio a nivel funcional implicaría que hicieramos otros cambios a nivel protocolo, el cual pensamos para ser expandido creando nuevos comandos o modificando los existentes.

2.7.4.3. Alimentación de la placa

Al igual que las otras placas utilizamos entre 7 y 20 volts para alimentarla, con la posibilidad de entregar 5 volts en forma directa usando el pin indicado.

Hicimos una modificación importante para la alimentación en esta placa que nos permitiría entregar una mayor cantidad de corriente que los $500mA$ que soporta el integrado *7805* usando transistores de potencia como los *TIP42*. Este cambio era importante porque los servo motores que íbamos a usar se alimentaban directamente a $5V$ y el consumo en conjunto superaba la cantidad de corriente que el regulador de tensión entregaba.

2.7.4.4. Configuración

La configuración física necesaria en la placa sería mínima. Mediante la llave inversora *S2* tendríamos la posibilidad de conectar el microcontrolador con el header de programación o puertos *P12* y *P13*.

Usando la llave *S3* establecemos la posición de la placa dentro de la cadena, ya sea como la terminación o un eslabón más. Todas las demás configuraciones serían a nivel software.

2.7.4.5. Esquemático

En la figura 35 mostramos el esquemático del microcontrolador y el conexionado del header de programación. En la figura 36 detallamos el módulo de comunicación y el conexionado con los conectores entre placas.

En la figura 37 mostramos los puertos de conexión para los servo motores y de uso general para otras conexiones. En la figura 38 la fuente de alimentación y bornera.

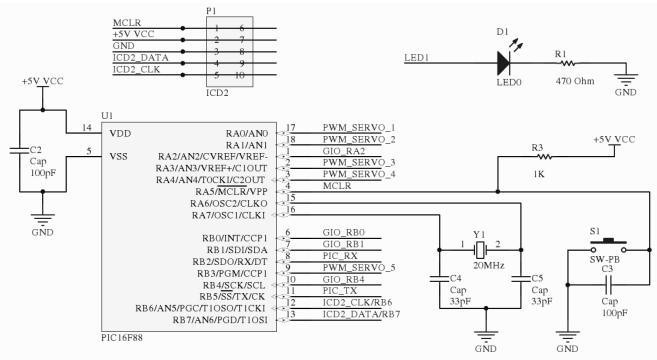


Figura 35: Microcontrolador y header de programación.

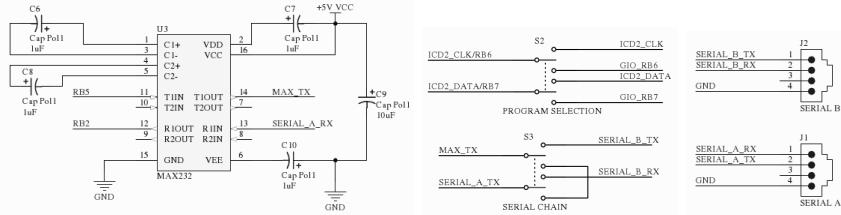


Figura 36: Comunicación, llaves de modo y conectores de entrada y salida.

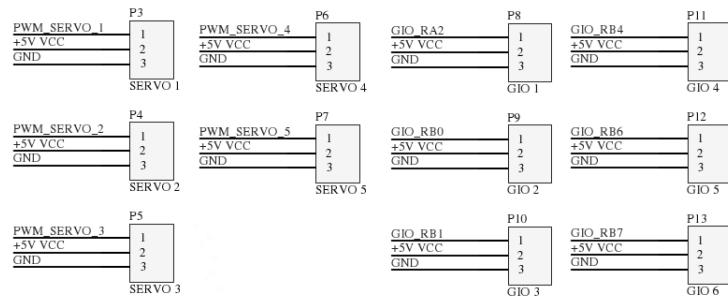


Figura 37: Puertos de conexión para los servo motores y de uso general.

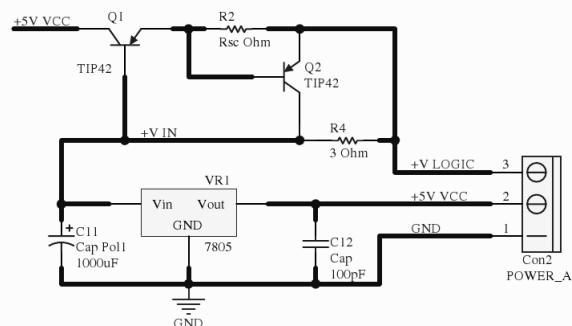


Figura 38: Fuente de alimentación extendida para la lógica y servo motores.

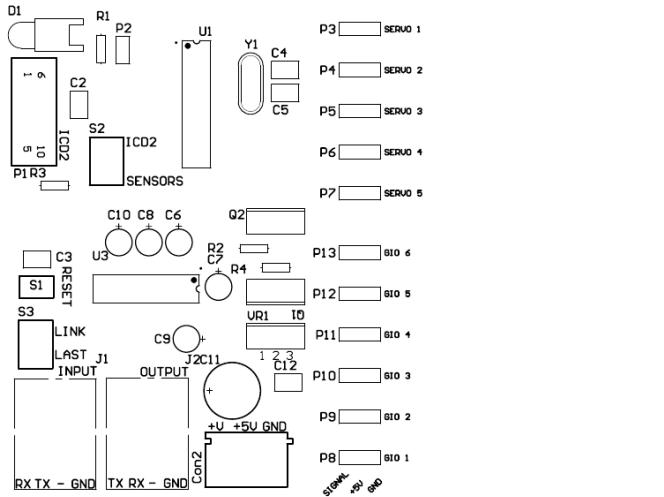


Figura 39: Máscara de componentes de la placa controladora de servo motores.

2.7.4.6. Circuito

En la figura 39 mostramos la máscara de componentes de la placa. En la figura 40 mostramos ambas capas de la placa.

2.7.4.7. Código básico

Como explicamos en la sección 2.3.2.2 creamos una rutina de control que usando el timer del microcontrolador como base de tiempo, genera pulsos del ancho necesario en cada puerto para establecer la posición necesaria de cada servo motor.

Para capturar cambios de estado en los puertos de uso general sólo deberíamos configurar las interrupciones del microcontrolador para que se generen ante cualquier cambio y verificar en la rutina de interrupción, que puerto la generó.

De querer utilizar los puertos para funciones no previstas en nuestro diseño deberíamos agregar las funciones de control en el ciclo principal de nuestro código.

2.7.4.8. Posibles extensiones

Possiblemente haya necesidades que no contemplamos en el diseño de esta placa aunque por no haberla construido no tenemos la experiencia sobre qué cosas hubiéramos hecho diferente.

2.8. Armado del prototipo

Una vez diseñadas, armadas y probadas cada una de las placas controladoras tanto en forma separada como en conjunto, procedimos al armado del robot prototipo. Como no teníamos las dimensiones o requerimientos de espacio del mecanismo de recolección armamos el prototipo sin estas consideraciones.

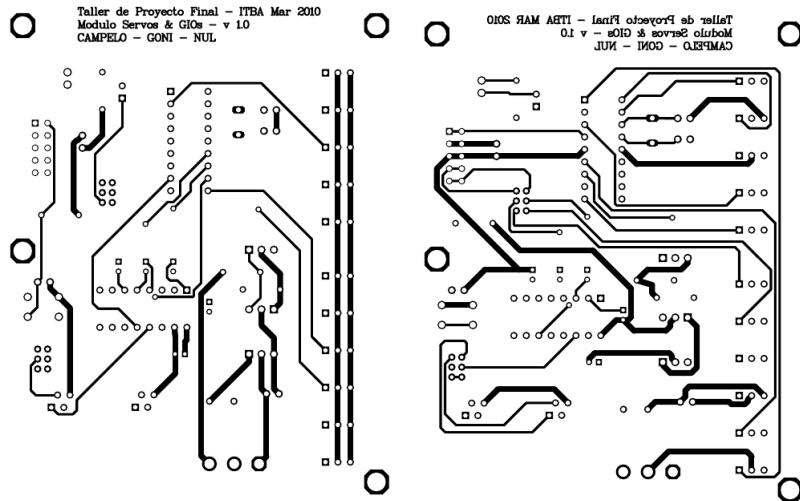


Figura 40: Capas superior e inferior de la placa controladora de servo motores.

2.8.1. Diseño

Para darle forma al prototipo tuvimos en consideración varias cuestiones como *a)* materiales de construcción; *b)* volumen interior para alojar los distintos componentes; *c)* posición de cada sensor respecto del borde exterior; *d)* posición de las ruedas y la maniobrabilidad que tendría; *e)* debía poder pasar por entre las sillas y mesas de la terraza.

Elegimos darle al chasis una forma de octágono. Ésto eliminaba los ángulos rectos de los bordes para minimizar la posibilidad de quedar atascados en las esquinas. A su vez, si colocábamos las ruedas alineadas con el centro geométrico del robot aumentaba la maniobrabilidad y podíamos realizar rápidas rotaciones sin chocar con el entorno.

Respetando nuestro diseño inicial y sabiendo que la cantidad de sensores era adecuada para cubrir nuestros requerimientos, definimos las posiciones de cada sensor. Para colocar los telémetros infrarrojos a la altura necesaria creamos un contorno con paredes, que a su vez, nos daban el volumen interior para alojar las placas controladoras y batería. El sensor de distancia por ultrasonido lo colocamos al frente, centrado y apuntando hacia adelante para tener el mayor campo de visión posible.

Para sostener las placas en el interior colocamos una varilla roscada de lado a lado en el interior y afirmamos entre dos tuercas a cada placa. Colocamos las placas de sensado en el centro y, contra las paredes, a las placas controladoras de motor. El cableado con los sensores lo hicimos interior a través de agujeros en las paredes. De igual forma, los cables de control para los motores y el cableado para los sensores de piso, los pasamos por un agujero en la base del robot.

Elegimos una batería de gel de 12V 7Ah que según nuestros cálculos nos daría una autonomía de entre 2 y 3 horas de uso continuo. Aprovechando su peso la colocamos lo más atrás posible dentro del volumen interior para mover el centro de gravedad y en conjunto con la rueda castor trasera, dar mayor estabilidad al robot.

Colocamos los motores en la parte inferior del robot alineados con el centro geométrico y las ruedas dentro del chasis para evitar atascos. Las ruedas que elegimos tienen una buje de teflón, con llanta de chapa y cubierta de goma compacta. Un problema que encontramos en este punto fue que el eje del motor era cilíndrico y no presentaba ninguna marca o corte para ajustar las ruedas, lo que dificultó el armado.

Con las ruedas actuales de 10cm de diámetro y teniendo en cuenta que el motor gira aproximadamente a 300 cuentas por segundo, logramos una velocidad cercana a 50cm/s.

En la parte superior armamos, con una madera más fina, un segundo piso para colgar la netbook y poder controlarla fácilmente. Pasamos el cable adaptador USB a serial RS232 por un agujero en el segundo piso y colocamos a la vista un interruptor de encendido desde el cual podemos recargar la batería del robot con los cables con punta de cocodrilo del cargador automático.

2.8.2. Desarme

Para lograr el desarme del prototipo recomendamos desconectar el cable del conversor USB - RS232 y retirar del piso superior la netbook. Luego seguir los siguientes pasos según las referencias en las figuras 41, 42 y 43.

- Retirar los 12 tornillos (1) que sostienen la tapa superior.
- Retirar el cable que conecta el sensor de ultrasonido (2) con las placas.
- Aflojar los tornillos (3) que sostienen al sensor de ultrasonido.
- Retirar la tapa (4).
- Desconectar los cables (5) de alimentación de las placas.
- Desconectar los cables (6) que conectan a los telémetros con las placas.
- Desconectar los cables (7) de comunicación entre las placas.
- Desconectar los cables (8) que conectan a las placas con los motores DC.
- Aflojar y retirar las tuercas (9) que mantienen a la varilla roscada (10).
- Despegar y retirar la batería (11) de la base del robot (12).
- Moviendo la varilla hacia los costados, retirar las placas (13).
- Retirar los telémetros (14).
- Retirar los sensores de piso (15).
- Aflojar y retirar las tuercas (16) que sostienen a los sensores de piso.
- Retirar las ruedas (17) de los motores.
- Retirar los tornillos (18) de las abrazaderas (19) de los motores (20).
- Retirar la rueda de tipo castor trasera (21).
- Retirar la rueda de tipo castor delantera (22).

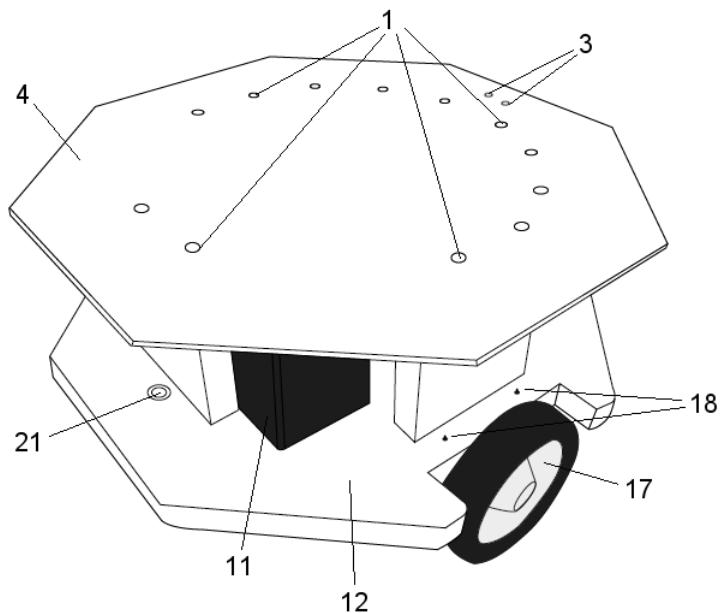


Figura 41: Diagrama de desarme exterior.

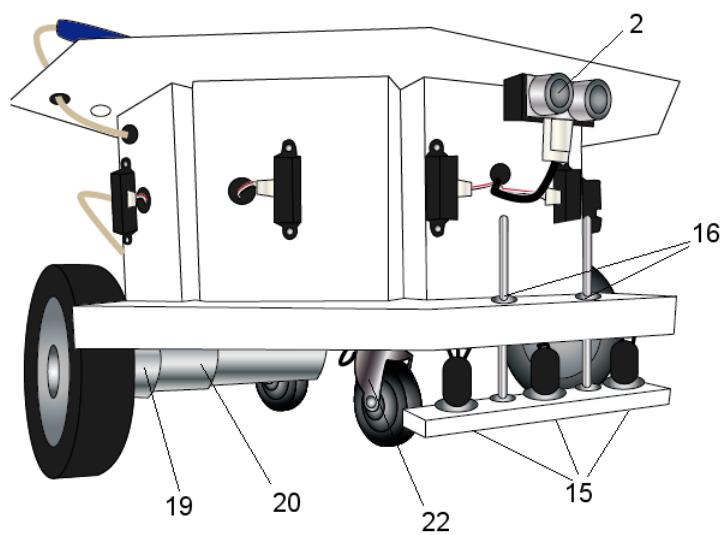


Figura 42: Diagrama de desarme frontal.

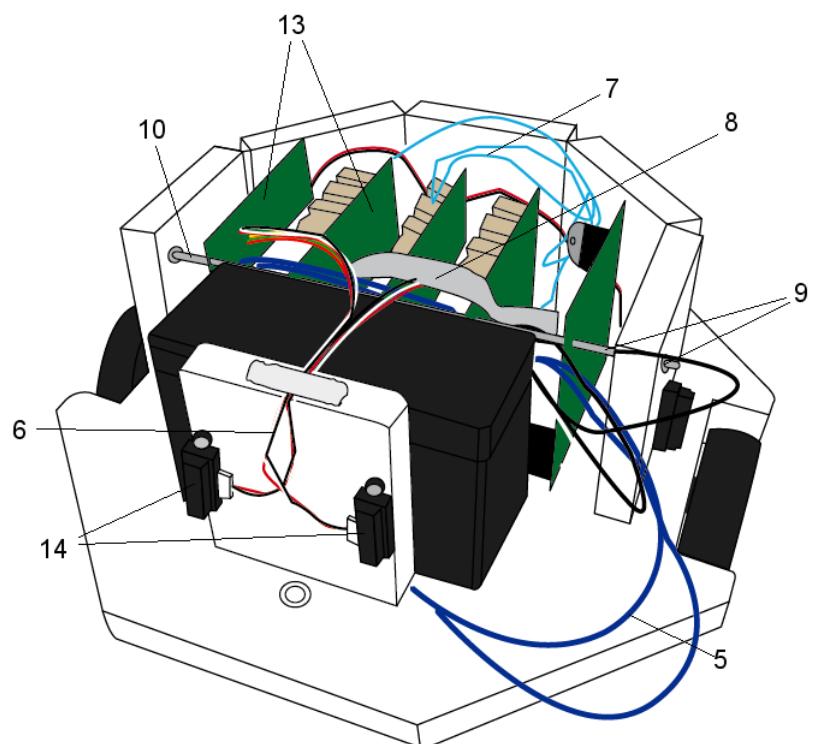


Figura 43: Diagrama de desarme interior.

3. Arquitectura de Comportamientos

3.1. Introducción

Como mencionamos en las secciones anteriores, el objetivo de éste trabajo es desarrollar un robot autónomo y reactivo que recolecte basura en un entorno estructurado pero dinámico, debido a que la arena (lugar donde se mueve el robot) es transitado por personas y está al aire libre.

En esta sección de *Arquitecturas de Comportamientos* detallamos las acciones que debería llevar a cabo el robot y cómo organizar las mismas para cumplir la meta de recolectar basura y ser autónomo, es decir, decidir por sí mismo las acciones a realizar en cada momento y ser capaz de mantenerse cargado para poder continuar recolectando.

En la sección 3.2 analizamos papers relacionados con este desarrollo, desde la forma de organizar los comportamientos, la definición y composición de los mismos, hasta su implementación. En la sección 3.3 detallamos la arquitectura elegida para llevar a cabo y organizar los comportamientos elegidos, y las ventajas y desventajas de usar la misma. En la sección 3.4 detallamos uno por uno los comportamientos que elegimos para que posea el robot, y sus correspondientes implementaciones en *pseudo-código*. En la sección 3.5 explicamos que es la odometría, para qué sirve y qué ventajas y desventajas tiene usarla. También detallamos el test utilizado para mejorar la eficiencia de la misma. En la sección 3.6 describimos cómo estructuramos el controlador del robot para que podamos ir desarrollando y probando el mismo con un simulador y minimizar el trabajo al pasarlo a el robot físico. Los resultados de performance y eficiencia del controlador desarrollado los obtuvimos de la simulación, y los mostramos en la sección 3.7. Finalmente, en la sección 3.8 sacamos conclusiones de los resultados obtenidos y las dificultades encontradas a lo largo del desarrollo de éste proyecto final.

3.2. Investigaciones previas

A continuación presentamos trabajos realizados por otros autores relacionados en alguna forma con el nuestro. Primero damos una breve descripción del trabajo del otro autor y luego lo comparamos con nuestro trabajo, analizando similitudes y diferencias entre ambos.

3.2.1. Desarrollo de comportamientos no triviales en robots reales : Robot recolector de basura - Stefano Nolfi [17]

En este paper se muestra el uso de un Khepera con el módulo Gripper en una arena delimitada por paredes para la recolección de "basura". Dicho robot se puede ver en la figura 44. La base tiene una fuente de luz asociada y el objetivo

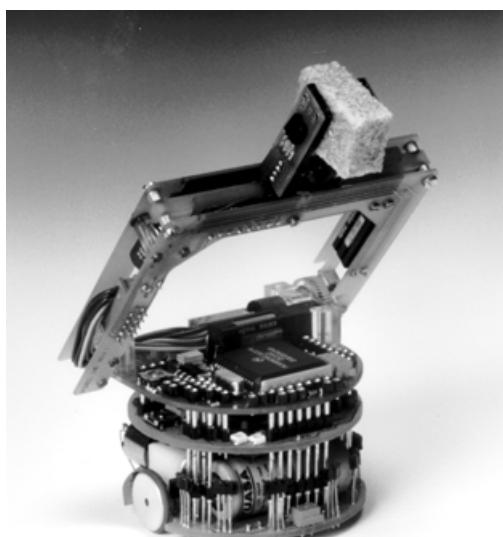


Figura 44: Robot Khepera con el módulo Gripper

del robot es llevar pequeños objetos a un lugar de depósito. Se utiliza *aprendizaje por refuerzo* para asociar las velocidades angulares de las ruedas a los diferentes tamaños de las "basuras". Para obtener el comportamiento deseado se hace uso de *algoritmos genéticos* y *redes neuronales*. Una vez obtenido el mismo mediante simulación en Webots, se lo probó en el Khepera físico. Comparando el trabajo de Stefano Nolfi con el nuestro podemos realizar las siguientes comparaciones:

- Existencia de un depósito: Ambos tienen un depósito en el cual dejar la basura recogida y una forma de identificarla: una fuente de luz usada por Nolfi y en nuestro trabajo, al llegar al final de una línea marcada sobre la arena.
- Autonomía: Ambos son autónomos en el sentido que no son manejados por un humano. En cuanto a la recarga de la batería, Nolfi no explicita

si es asistida o no; en nuestro caso, el robot se encarga de ir a la base de recarga una vez detectada la falta de energía.

- Método de Recolección: En el trabajo de Nolfi se utiliza el módulo “Gripper” agregado al Khepera. Este módulo simula un brazo con dos dedos. De ésta forma, la recolección se realiza juntándolos y la liberación se realiza separándolos. Ésta forma de recolectar está basada en el comportamiento humano. Nuestro trabajo se basa más en la actividad humana para realizar la recolección, ya que podría compararse con un recolector de basura que usa una pala para guardar el residuo en su tacho.
- Método de aprendizaje: En nuestro trabajo no utilizamos aprendizaje por refuerzo o algoritmos evolutivos. No es el caso de Nolfi, que utiliza ambas técnicas para evolucionar el comportamiento deseado.
- Robot Utilizado: Como mencionamos anteriormente, Nolfi utilizó un Khepera en la simulación. En nuestro trabajo utilizamos un E-puck para la simulación, una versión nueva del Khepera, pero sin el módulo Gripper.



Figura 45: Robot E-puck

- Reconocimiento del objeto a recolectar:

3.2.2. Arquitectura de control para un Robot Autónomo Móvil - Neves And Oliveira [16]

Neves y Oliveira describen en su trabajo una arquitectura de control basada en comportamientos para un robot móvil en un ambiente dinámico, utilizando muchos aspectos de la arquitectura *Subsumption* (Ver sección 3.3) propuesta por *Brooks* y que usamos al realizar éste proyecto.

La arquitectura propuesta en el paper, o *Control System Architecture*, se basa

también en la teoría de *The Society of Mind*, escrita por Minsky[15], donde el sistema es visto como una sociedad de agentes, cada uno con una competencia particular y que colaboran entre ellos para ayudar a la sociedad a alcanzar su meta. La arquitectura está compuesta por tres niveles: un nivel reflexivo, uno reactivo, y otro cognitivo, aumentando la complejidad al igual que el orden en que fueron presentados.

El nivel reflexivo incluye aquellos comportamientos innatos, es decir, actúan directamente como estímulo-respuesta. El segundo nivel, el reactivo, está compuesto por agentes que responden rápidamente a los estímulos ya que requieren poco nivel de procesamiento. Finalmente, en el nivel cognitivo, se encuentran los agentes encargados de guiar y administrar los comportamientos reactivos de forma tal que el robot muestre un comportamiento orientado.

Aunque la arquitectura explicada es similar a la utilizada en nuestro trabajo, no utilizamos ésta organización de los comportamientos en capas. Sin embargo, podemos divisar que algunos de los comportamientos que propusimos corresponden a la primera capa de reflexión, como por ejemplo el evitamiento de obstáculos y otros podrían incluirse en la segunda capa de comportamientos reactivos, como sería deambular. Finalmente en la última capa estaría el reconocimiento de objetos debido a su gran demanda de procesamiento.

3.2.3. Path Planning usando Algoritmos Genéticos - Salvatore Cандido [5]

En este paper se describe la utilización de algoritmos genéticos para resolver el problema de Path Planning. Éste consiste en armar un plan, una secuencia de acciones de forma tal que a partir de un punto de origen se llegue al punto de destino, siguiendo ese plan. Debido a la componente dinámica de nuestro trabajo, siempre tratamos de mantener los comportamientos del robot lo más reactivos posibles, por lo que armar un plan no sería la mejor opción a utilizar. Por el contrario, el uso de algoritmos genéticos puede ser una herramienta que se podría llegar a usar en una futura continuación de nuestro trabajo.

3.2.4. Navegación predictiva de un robot autónomo - Foka And Trahanias [9]

La navegación predictiva nace como una posible solución al problema de un robot navegando en un ambiente con muchas personas y obstáculos, tal como lo es el ambiente en el cual navegará nuestro robot.

La forma en que es implementada por los autores es mediante un *POMDP*, es decir, un proceso de decisión de markov parcialmente observable. Cabe destacar estas dos últimas palabras, ya que indican la naturaleza del ambiente: hay incertidumbre o falta de información acerca de ciertas variables del entorno. En este caso, se utiliza el *POMDP* para manejar tanto la navegación del robot como el evitamiento de obstáculos, un punto en que se diferencia de lo que propusimos nosotros, que es tratar el *wandering* de forma separada del comportamiento de evitamiento de obstáculos. Ésto, en parte, se debió a causa de la arquitectura utilizada ya que desde ese punto de vista, ambos comportamientos tienen niveles de jerarquía muy diferentes como se puede ver en la figura 49.

3.2.5. Algoritmo de navegación y evitamiento de obstáculos en un entorno desconocido - Clark Et. al [18]

En este paper se presentan dos algoritmos complementarios para la navegación en ese tipo de entornos.

El primero consiste en la navegación y un mapeo del entorno que garantiza una cobertura completa de una arena cuyas ubicaciones de la paredes no se conocen *a priori*. Consiste básicamente en un seguimiento de las paredes complementado por una variación de *flood filling* para asegurarse la cobertura completa de la arena. El algoritmo completo se puede apreciar en la figura 46.

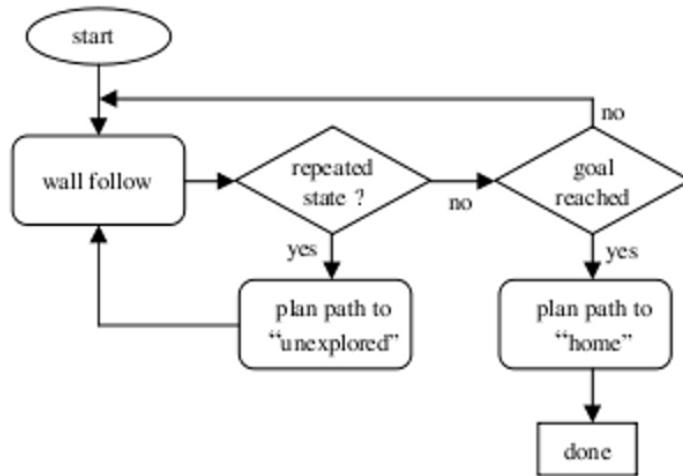


Figura 46: Algoritmo propuesto por Clark Et. al

Un autómata con aprendizaje estocástico es el algoritmo que complementa al primero, y su objetivo es el evitamiento de obstáculos. Para ésto, se utiliza un mecanismo de recompensa/castigo de forma tal que se adapten las probabilidades de las acciones a tomar.

En relación con nuestro trabajo, hay ciertas similitudes y diferencias, enumeradas a continuación:

- Ambos presentan un mecanismo de “seguimiento de”, en nuestro caso lo hacemos con líneas, en el paper se utiliza con paredes. Sin embargo, se utilizan para objetivos diferentes. Nosotros lo usamos para dirigirnos a la base ya que tratamos de mantener el conocimiento que el robot tiene sobre el mundo lo más acotado posible. En el paper se utiliza el mecanismo de seguir las líneas para obtener un modelo del mundo, algo que puede llegar a servir mucho en ambientes no tan dinámicos como el nuestro, razón por la cual no elegimos implementarlo.
- También coincidimos con la existencia de un método para evitar obstáculos. En el paper se implementa como un autómata que va aprendiendo según los premios o castigos que recibe. En nuestro caso el comportamiento es puramente reactivo y reacciona en base a los valores de los sensores de distancia y no tiene memoria.

3.3. Arquitectura propuesta

Una arquitectura basada en comportamientos define la forma en que los mismos son especificados, desde su granularidad (qué tan complejo o simple es un comportamiento), la base para su especificación, el tipo de respuesta y la forma en que se coordinan.

La arquitectura que elegimos para desarrollar nuestro trabajo es *Subsumption*, desarrollada por Rodney Brooks[4] en 1985. Esta arquitectura está basada en comportamientos puramente reactivos, rompiendo así con el esquema que estaba de moda en la época de *sensar-planear-actuar*. Algunos de los principios propuestos que tuvimos en cuenta durante el desarrollo de nuestro trabajo son:

- Un comportamiento complejo no es necesariamente el producto de un complejo sistema de control,
- El mundo es el mejor modelo de él mismo,
- La simplicidad es una virtud,
- Los sistemas deben ser construidos incrementalmente.

Cada comportamiento es un conjunto de pares estímulo-respuesta. Tal como mostramos en la figura 47, cada estímulo o respuesta puede ser inhibido o suprimida por otros comportamientos activos. Además cada comportamiento recibe una señal de reset, que lo devuelve a su estado original. El nombre

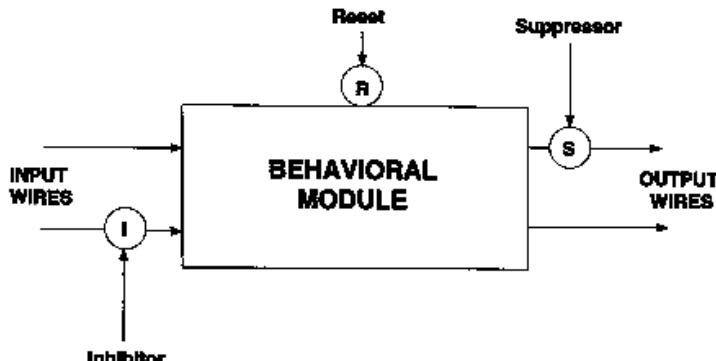


Figura 47: Esquema de comportamiento

Subsumption proviene de la forma en que los comportamientos son coordinados entre sí. Hay una jerarquía donde los comportamientos de la arquitectura tienen mayor o menor prioridad según su posición. Los comportamientos de los niveles inferiores no tienen conocimiento de los comportamientos de las capas superiores. Gracias a ésto, se puede plantear un diseño incremental, brindando flexibilidad, adaptación y paralelismo al desarrollo e implementación de los comportamientos.

La idea a seguir es que el mundo sea el principal medio de comunicación entre los comportamientos. Ésto se debe a que la respuesta de un comportamiento ante un estímulo resulta en un cambio en el mundo y, por lo tanto, en la relación del

robot con el mismo. De ésta manera, el robot en su próximo paso sensará otro estado del mundo.

El procedimiento básico para diseñar y desarrollar comportamientos para robots con esta arquitectura es sencillo:

1. Especificar cualitativamente la forma en que el robot responde al mundo, es decir, el comportamiento que realizará.
2. Descomponer la especificación como un conjunto de acciones disjuntas.
3. Determinar la granularidad del comportamiento, analizando en qué nivel de la jerarquía existente se encontrará y cuantas acciones disjuntas es necesario llevar a cabo para el cumplimiento de la tarea.

Un ejemplo de esta arquitectura se puede observar en la figura 48. En la misma hay 4 comportamientos: Homing, Pickup, Avoiding y Wandering. Las líneas que entran a cada comportamiento son los estímulos ante los cuales se activan y las salidas son las señales de respuesta correspondientes. La señal de respuesta de la arquitectura es la línea que sale por la derecha de la caja (Arquitectura) que contiene la relación entre los comportamientos. Puede verse como Homing inhibe la salida de Pickup ya que su salida entra al supresor (denotado con un círculo con una S dentro) de la salida de Pickup y por lo tanto, las salidas de los demás comportamientos, ya que su prioridad es mayor al resto. En el caso que no estén presentes los estímulos de Homing, Pickup y Avoiding, no hay inhibición en la salida de Wandering, y por lo tanto se lleva a cabo el comportamiento de Wandering.

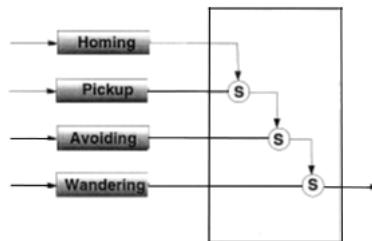


Figura 48: Ejemplo de la arquitectura Subsumption

3.4. Comportamientos e implementaciones

Una vez que decidimos utilizar la arquitectura explicada en la sección 3.3 para nuestro proyecto, tuvimos que analizar:

- Forma de implementación de la arquitectura en código,
- Comportamientos a realizar,
- Ordenes de inhibición y supresión entre los mismos,
- Forma de implementación de los mismos,
- Orden de implementación

Para implementar la arquitectura decidimos asignarle un *ID* numérico diferente a cada comportamiento. También tomamos la decisión de elegir como comportamiento activo en el instante t , aquel comportamiento que esté activo en ese instante y tenga mayor *ID*, suprimiendo así el resto de los comportamientos (con un *ID* menor) que podrían estar activos.

Como requerimiento de nuestro proyecto teníamos la realización de un robot autónomo que recolecte basura de su entorno dinámico¹⁴ pero estructuralmente estático¹⁵. De aquí se infieren algunos de los comportamientos que debe tener el robot:

- *Recolectar basura* (3.4.4)
- *Recargar batería* (3.4.8): Por ser autónomo, debe poder ser capaz de recargarse solo para poder continuar con su actividad.
- *Wandering* (3.4.1): El robot, al ser autónomo, no es radio-controlado y debe poder recorrer el entorno por sí mismo.
- *Evitamiento de obstáculos* (3.4.9): Debido a la naturaleza dinámica del entorno, el robot debe ser capaz de navegar sin chocarse contra las paredes ni con las personas que circulan por el mismo.

El comportamiento de recolectar basura y el requerimiento de la autonomía llevan a su vez a la aparición de más comportamientos: *Descargar basura* (3.4.6) e *Ir hacia basura* (3.4.3).

En la figura 49 mostramos los comportamientos implementados y su orden de jerarquía. Se puede ver que hay más comportamientos de los detallados anteriormente ya que la forma en que implementamos los comportamientos básicos del robot requirió el desarrollo de otros auxiliares.

Primero implementamos *Wandering* debido, en una primera aproximación, a su sencillez. Luego implementamos *Evitamiento de obstáculos* para lograr que el robot pueda navegar sin problemas por el entorno. Como el comportamiento de recolectar e ir hacia la basura dependía del módulo de reconocimiento de objetos y el mismo estaba siendo desarrollado en paralelo, decidimos implementar el comportamiento de *Recargar batería* y *Descargar basura*. Una vez

¹⁴Con entorno dinámico nos referimos a que es concurrido por personas y es al aire libre, por lo que el viento puede mover ciertos objetos

¹⁵Con estructuralmente estático nos referimos a que sus límites no varían ya que está rodeado por paredes

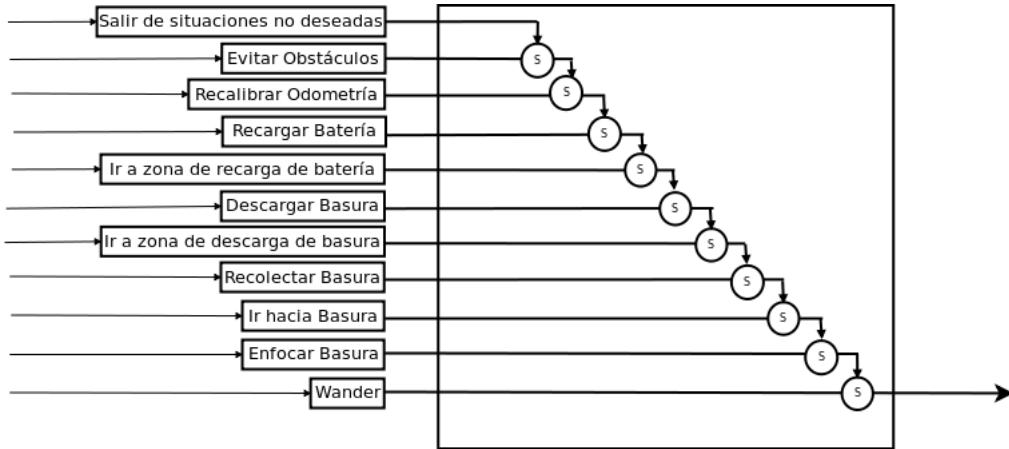


Figura 49: Arquitectura de comportamientos implementada

que tuvimos la primera implementación funcional del reconocimiento de objetos, procedimos a desarrollar *Ir hacia basura* y *Recolectar Basura*.

A continuación detallamos los comportamientos indicados en la figura 49, así como su implementación en *pseudo-código* y detalles tenidos en cuenta para su realización.

3.4.1. Wandering

3.4.1.1. Detalle del comportamiento

Por ser el comportamiento que menor jerarquía tiene (Ver figura 49), es el único comportamiento que está activo ante la ausencia de un estímulo, asegurándonos que siempre haya por lo menos un comportamiento activo.

En una primera aproximación de *Wandering*, sólo nos preocupamos por ir hacia adelante ya que eventualmente, el robot encuentra un obstáculo y realiza un giro cambiando su dirección.

Los resultados de la simulación nos indicaron que el robot no recorría ciertas zonas o las recorría después de un largo tiempo, lo que nos llevó a una segunda implementación. La misma lleva un seguimiento de los lugares que más recientemente recorrió el robot. Además, hicimos uso de la cámara para redefinir la idea de zona recorrida. Decimos que el robot recorrió cierta zona si:

- Estuvo físicamente en ella, o
- La zona fue alcanzada por la imagen que se ve en la cámara (Ver figura 51)

Ésto, en cierta forma, genera un modelo del mundo, un hecho que conflictúa con uno de los principios propuestos a seguir en la sección 3.3. Para minimizar el conflicto, decidimos mantener al mínimo la información almacenada para el funcionamiento del algoritmo, es decir, por cada zona de la arena sólo mantenemos el timestamp de la última vez que el robot la visitó.

3.4.1.2. Implementación del comportamiento

La implementación en *pseudo-código* es la siguiente:

```
por cada paso
    zona = pedir_zona_vista(camara)
    marcar_zona_como_vista(modelo_del_mundo,zona)
    ultima_zona_visitada = pedir_ultima_zona_visitada(modelo_del_mundo)
    velocidades = calcular_velocidades_de_ruedas(ultima_zona_visitada)
    poner_velocidades_en_ruedas(velocidades)
```

Para obtener la zona vista por la cámara, necesitamos de la altura C_h a la cual está ubicada la cámara en el robot, el campo de visión (de ahora en adelante *Field of View o FOV*) horizontal FOV_h o vertical FOV_v y el ángulo de inclinación de la cámara ac . Analizando la figura 50 podemos ver que:

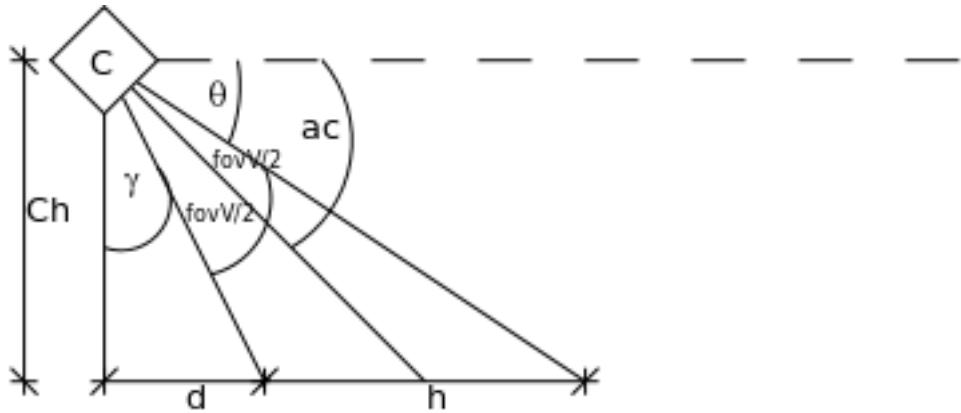


Figura 50: Diagrama de posición de la cámara

$$ac = \frac{FOV_v}{2} + \theta \quad (1)$$

$$\gamma + FOV_v + \theta = \frac{\pi}{2} \quad (2)$$

$$\tan(\gamma) = \frac{d}{C_h} \quad (3)$$

$$\tan(\gamma + FOV_v) = \frac{d + h}{C_h} \quad (4)$$

Organizando las ecuaciones, podemos deducir que:

$$\gamma = \frac{\pi}{2} - ac - \frac{FOV_v}{2} \quad (5)$$

$$d = \tan(\gamma) * C_h \quad (6)$$

$$d + h = \tan(\gamma + FOV_v) * C_h \quad (7)$$

Por lo que obtenemos el ángulo hasta el inicio de la imágen de la cámara γ y como consecuencia, la distancia d desde la posición de la cámara hasta el inicio de la imágen y $d + h$, la distancia desde la posición de la cámara hasta el final de la imágen. Usando estos datos, la posición del robot P y basándonos en la figura 51, podemos obtener los puntos A , B , C y D del trapezoide que determina la zona que ve la cámara.

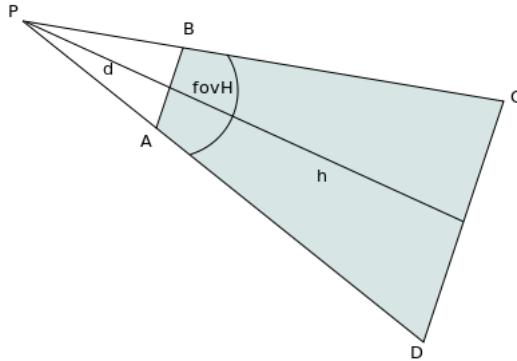


Figura 51: Zona vista por la cámara

3.4.2. Enfocar Basura

Una vez que el módulo de detección de objetos reconoce algo como basura (ver sección 4.3.6), hay que elegir la forma en que el robot se dirige hacia la basura. En la figura 52 mostramos una aproximación. Consiste en primero enfocar la basura de forma tal que la misma quede en el centro de la imagen de la cámara. De aquí surge el comportamiento *Enfocar Basura*.

Una segunda forma de lograr ésto (figura 53) no consiste en enfocar la basura, sino que se marca un arco hacia la misma. Ésto requiere que se establezcan las velocidades correspondientes a las ruedas de forma tal que la trayectoria del robot describa dicho arco. Con esta alternativa no haría falta un comportamiento para que el robot enfoque la basura.

Dado que la primera aproximación es más simple de implementar, y teniendo en cuenta el principio enunciado en la sección 3.3 “*La simplicidad es una virtud*”, elegimos implementarlo, a pesar de tener un posible inconveniente, como mostramos en la figura 54.

Cuando hay una basura en alguna esquina superior de la imagen de la cámara, y el robot gira sobre sí mismo para enfocarla, la basura puede llegar a perderse por el fondo de la imagen. Ésto se debe a que la distancia hacia dichas esquinas es mayor a la distancia hacia el centro del borde superior de la imagen (ver figura 51). Decimos que es un “possible” problema, ya que una vez que se pierde de vista la basura, el robot no enfocará más debido a que el estímulo desapareció, pero si luego se dirige hacia adelante (por la activación de algún otro comportamiento), la basura volverá a aparecer en la imagen, sin aprovechar la oportunidad de recogerla.

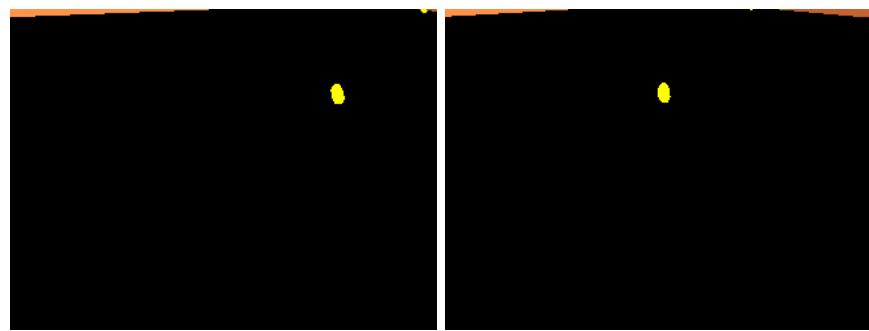


Figura 52: Primera aproximación de “Ir a la basura”. Inicialmente el objeto reconocido como basura se encuentra a un costado del eje vertical de la imagen. Luego el robot gira de forma tal que el objeto quede sobre el eje vertical.

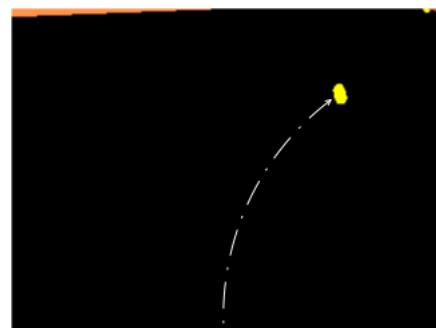


Figura 53: Segunda aproximación de “Ir a la basura”. Se traza un arco hacia el objetivo de forma tal que no sea necesario enfocar la basura.



Figura 54: Posible inconveniente con la primera aproximación de “Ir a la basura”. La basura se encuentra en una de las esquinas superiores de la imagen. El robot la enfoca, girando hacia la derecha y la basura desaparece de la imagen. En el caso que el robot se dirija hacia adelante, la basura volverá a aparecer en la imagen.

3.4.2.1. Detalle del comportamiento

La primer aproximación se puede describir como:

- Si la basura está a la izquierda de la imagen, se debe girar hacia la izquierda
- Si la basura está a la derecha de la imagen, se debe girar hacia la derecha
- Si la basura está en el centro, ya está enfocada

3.4.2.2. Implementación del comportamiento

Para la implementación se sigue el siguiente *pseudo-código*:

```

por cada paso
    lista_de_basuras = obtener_lista_de_basuras(modulo_de_reconocimiento)
    basura_mas_cercana = elijo_basura_mas_cercana(lista_de_basuras)
    angulo_a_basura = obtener_angulo(basura_mas_cercana)
    velocidad = VELOCIDAD_BASE * (abs(angulo_a_basura) / (PI/2))
                + VELOCIDAD_BASE_MINIMA

    si angulo_a_basura < 0 entonces
        veloc_izq = -velocidad
        veloc_der = velocidad
    sino
        veloc_izq = velocidad
        veloc_der = -velocidad
    fin_si
    poner_velocidades_en_ruedas(veloc_izq, veloc_der)

```

Se puede ver que la velocidad de giro del robot es proporcional al módulo del ángulo que hay hacia la basura, logrando enfocar más rápido cuando el ángulo es mayor y tener mayor precisión cuando el ángulo es más chico, además de tener mayor rapidez de enfoque y precisión que si la velocidad de giro fuera constante.

Como observamos, el algoritmo enfoca la basura que se encuentra más cerca del robot. Para ésto realiza una transformación en la cual obtiene la distancia al objeto según su coordenada (x, y) en la imagen. En la figura 55 podemos apreciar mejor el área vista por la cámara, delimitada por el trapecio gris. Las líneas azules marcan los segmentos hacia las esquinas izquierda, derecha y centro inferior de la imagen. Las verdes indican lo mismo pero para la parte superior de la imagen. En rojo se pueden ver los triángulos formados hacia una basura a distancia db . Ésta, puede calcularse mediante la siguiente ecuación:

$$db = \sqrt{dx^2 + dy^2} \quad (8)$$

Dado que los valores de dx y dy no son conocidos, nos ayudamos con la figura 56. Descomponiendo la misma en la figura 57, podemos afirmar que:

$$dx = 2b \sin\left(\frac{\alpha}{2}\right) \quad (9)$$

donde b , a su vez, lo calculamos como:

$$b = \sqrt{dy^2 + Ch^2} \quad (10)$$

donde Ch representa la altura de la cámara y es un valor conocido.

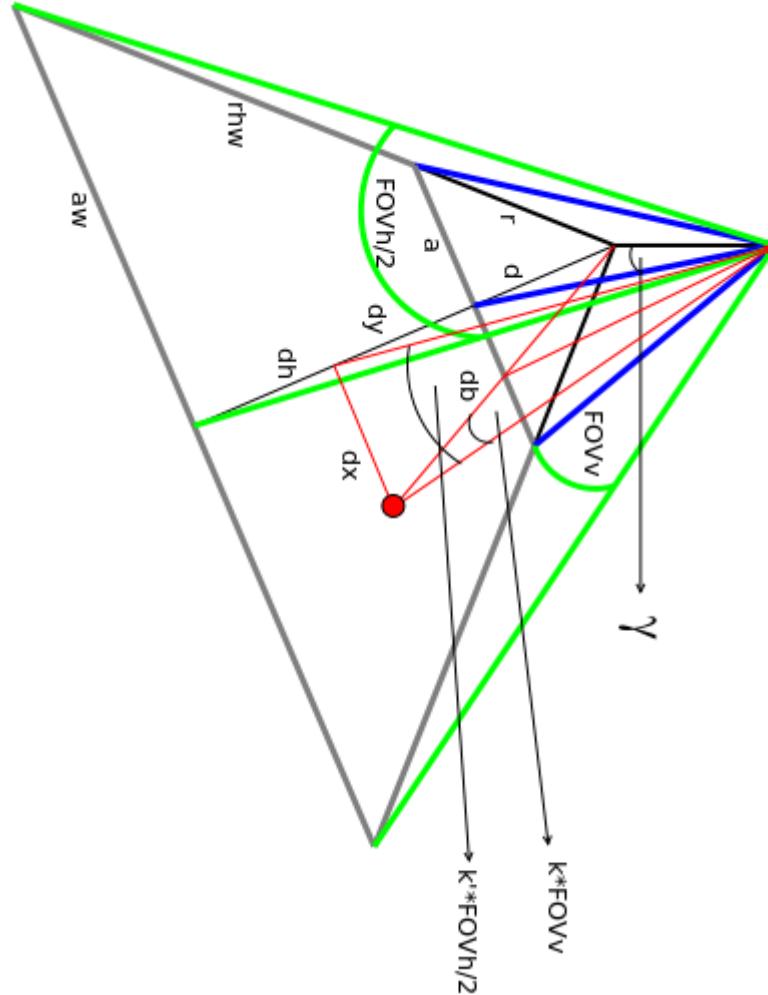


Figura 55: Área vista por la cámara y distancias y ángulos

Para el cálculo de dy tenemos que recurrir a la ecuación 7 donde reemplazamos FOV_v por β que es el ángulo vertical que forma la cámara con el objeto en cuestión. Éste se obtiene como $\beta = kFOV_v$ donde $k = \frac{dist_y}{C_{rh}}$ que se corresponden con la distancia en pixels sobre el eje y y la resolución vertical de la imagen respectivamente. El cálculo de k puede apreciarse mejor en la figura 60.

Para el cálculo de la ecuación 9 también es necesario el valor de α el cual obtenemos como $\alpha = k' \frac{FOV_h}{2}$ donde en este caso $k' = \frac{dist_x}{0.5C_{rh}}$ y FOV_h se corresponden con la distancia en pixeles sobre el eje x y el field of view horizontal de la cámara imagen. Habiendo obtenido estos valores, estamos en condiciones de calcular tanto la ecuación 10 como 9 y finalmente, utilizando estos resultados, reemplazamos en 8 y obtenemos la distancia hacia el objeto. En la figura 58

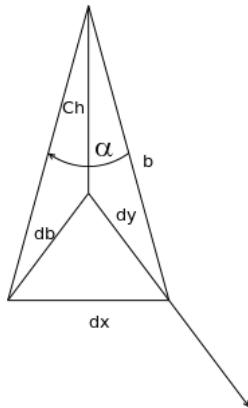


Figura 56: Caso particular de distancia hacia una basura a distancia db

podemos observar los arcos de distancia hacia cada punto de la imagen.

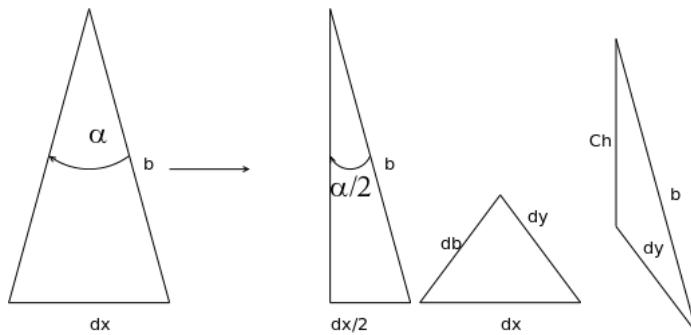


Figura 57: Descomposición de figura 56

3.4.3. Ir a Basura

Luego de la elección de la forma en que se resuelve el problema de encontrar una basura (ver figura 3.4.2), el comportamiento de ir a basura resulta trivial, ya que luego de ser enfocada, la basura está delante del robot y sólo basta ir hacia adelante.

3.4.3.1. Detalle del comportamiento

El estímulo necesario para que este comportamiento esté presente, está dado por dos condiciones:

1. El método de reconocimiento de objetos encontró una basura
2. La basura se encuentra en un entorno del centro del eje vertical de la imagen obtenida de la cámara.

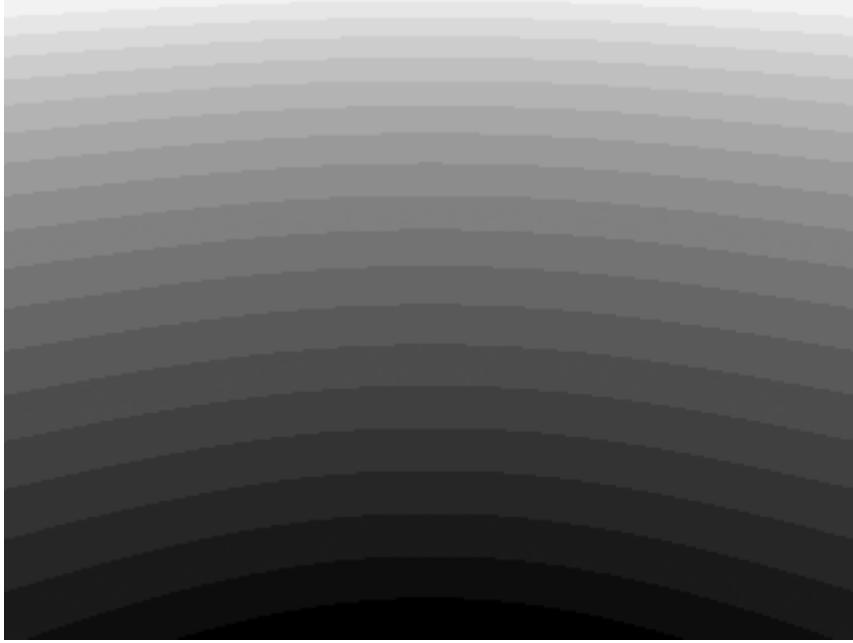


Figura 58: Distancias a pixeles en imagen según la variación de color. Los arcos marcan distancias dentro de un mismo rango de valores. Cuanto más clara la zona, más lejana está.

Para decidir si una basura se encuentra en un entorno del eje vertical, utilizamos un umbral δ_f que determina el ángulo de abertura. Si una basura se encuentra en la zona, entonces se la toma como enfocada. En la figura 59 se pueden ver 2 valores distintos para el umbral.

3.4.3.2. Implementación del comportamiento

```
por cada paso
    distancia = obtener_distancia_a_basura(modulo_de_reconocimiento)
    coeff = (distancia - DIST_MIN)/(DIST_MAX - DIST_MIN)
    veloc_der = VELOCIDAD_MIN*(1 - coeff) + coeff*VELOCIDAD_MAX
    veloc_izq = veloc_der
    poner_velocidades_en_ruedas(veloc_izq, veloc_der)
```

Al igual que en la implementación del comportamiento 3.4.2.2, las velocidades que se le otorgan a las ruedas dependen de la distancia hacia la basura, de forma tal que si una basura está muy lejos, la velocidad sea mayor y a medida que se va acercando, vaya disminuyendo linealmente.

Dado que la basura se encuentra en un entorno del eje Y en la imagen (Ver figura 60b) cometemos un error muy pequeño al estimar la distancia a la basura como si estuviera sobre el mismo (asumiendo que la coordenada X de la basura en la imagen es 0).

Como mostramos en las figuras 60 y 50, el ángulo vertical hacia el punto más alto de la imagen $(0, C_{rh})$ es $\gamma + FOV_v$ y hacia el más bajo, $(0, 0)$, el ángulo es γ .

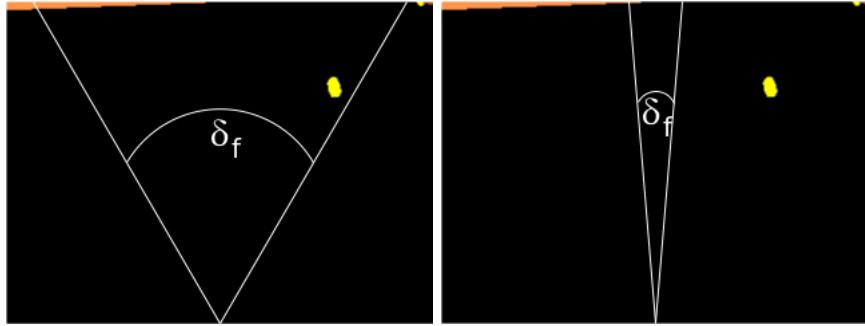


Figura 59: Distintos δ_f

De las ecuaciones (6) y (7) sabemos las distancias a los puntos $(0,0)$ y $(0,C_{rh})$ son (DIST_MIN) y (DIST_MAX) respectivamente. Entonces, para obtener la distancia dy hacia el punto $(0,y)$ basta con calcular:

$$y_{angle} = \text{FOV}_v * \frac{y}{h} \quad (11)$$

$$dy = \tan(\gamma + y_{angle}) * C_h \quad (12)$$

donde y_{angle} es la proporción de FOV_v hacia y .

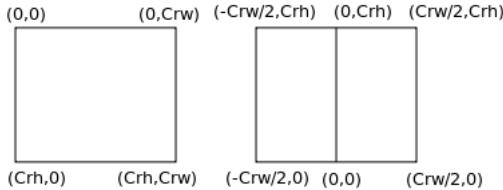


Figura 60: Conversión de coordenadas de imagen de cámara. Dado un pixel (y,x) , su nueva coordenada está dada por $(x - Crw/2, Crh - y)$

3.4.4. Recolectar Basura

Una vez que el robot llegó a estar posicionado para recolectar la basura deseada, el comportamiento de *recolectar basura* se activa. El mecanismo para lograr ésto fue cambiando a lo largo del desarrollo del proyecto.

En un principio pensamos en usar una rampa interna dentro del robot, de forma tal que la basura suba esa rampa para luego caer en un depósito interno. Por problemas con la simulación de este procedimiento, buscamos otro mecanismo.

El mecanismo que elegimos para usar en la simulación tiene estas componentes:

- El robot tiene un servo en su parte posterior (debajo de la cámara)
- Dos paredes delimitan el espacio a lo largo de la dirección que une el centro del robot con el servo anteriormente mencionado.

3.4.4.1. Detalle del comportamiento

La activación de *recolectar basura* depende de 3 condiciones:

- Las dos condiciones impuestas para *ir a basura*
- La distancia a la basura elegida para ser recolectada debe ser menor a un umbral.

Aquí se puede observar que tan importantes son los comportamientos anteriores para que el robot logre recolectar la basura.

Es importante la elección del umbral: un valor muy chico, cercano a DIST_MIN, puede llevar a que el comportamiento no se active porque la basura ya no está más en la imagen. Por otro lado, un valor muy grande, cercano a DIST_MAX, causaría que el robot se disponga a recolectar una basura que está muy lejos y podría llegar a moverse por cuestiones propias del ambiente, llevando así a una innecesaria activación del comportamiento.

3.4.4.2. Implementación del comportamiento

El *pseudo-código* de este comportamiento es el siguiente:

```
distanzia = obtener_distancia_a_basura(modulo_de_reconocimiento)
levantar(servo_delantero)
recorrer_distancia(distanzia)
cerrar(servo_delantero)
```

La distancia hacia la basura es obtenida de la misma forma que en la sección 3.4.3.2. Levantar y cerrar el servo consiste en establecer su posición en $\frac{\pi}{2}$ y 0 respectivamente. Para calcular la distancia recorrida utilizamos la distancia entre la posición del robot en el instante t , $P_r(t)$ y el instante $t + 1$, $P_r(t + 1)$, ambas dadas por la odometría (Ver sección 3.5). Las diferentes etapas de la recolección de una basura las detallamos en la figura 61.

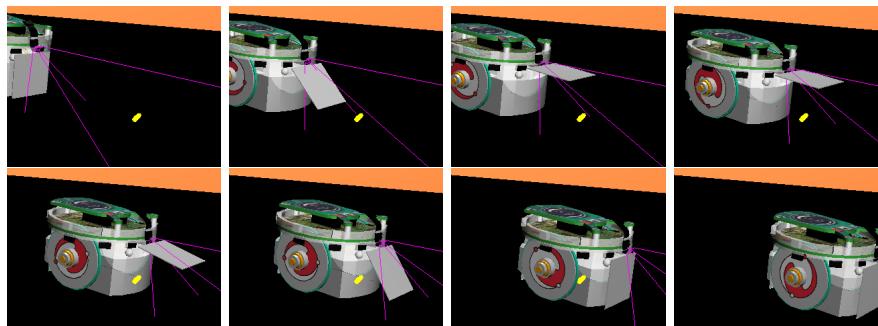


Figura 61: Etapas de recolección de basura

3.4.5. Ir a zona de descarga de basura

Como la capacidad del depósito interno de basura del robot tiene un límite, surge como necesidad que el robot sea capaz de ir hacia una zona donde descartará la basura que contiene. Entonces, al llenarse el depósito surge el estímulo

necesario para la activación de este comportamiento.

Para ir hacia dicha zona decidimos imponerle una condición al entorno donde el robot actuará. Esta condición consiste en poner líneas de forma tal que si el robot la sigue, lo lleve a la zona de descarga.

Por lo tanto para dirigirse a la zona de descarga de basura, es necesario:

- Buscar la línea e ir a la misma
- Entrar a la línea de forma tal que el robot y la línea queden alineados
- Seguir la línea

Siguiendo con la idea que es mejor descomponer un comportamiento complejo en otros más simples, decidimos separar el comportamiento de *Ir a zona de descarga de basura* en 3 comportamientos más simples: *Buscar línea*, *Entrar a la línea* y finalmente *Seguir la línea*.

3.4.5.1. Buscar línea

Inicialmente habíamos dispuesto las líneas de forma tal que sigan los límites de la arena. Entonces para buscar la línea tuvimos que calcular para cada una cuál es la distancia hacia la misma, luego elegir ir a la que menor distancia se encontraba. Ésta implementación, además de ser costosa, tenía un problema: a veces sucedía que al ir hacia una línea, la distancia hacia otra pasaba a ser más corta y ésta última podía estar más lejos de la zona de recarga que la primera.

Luego repensamos el problema y nos dimos cuenta que el objetivo era llegar a la zona de descarga, por lo que decidimos dejar sólo 2 las líneas que se encuentran a la misma distancia, como se puede apreciar en la figura 62. La zona de descarga se ubica cerca de donde se encuentra el cilindro de color verde.

Para distinguir si el robot está en una línea o no, utilizamos sensores de piso dispuestos como se muestra en la figura 63. Los sensores se encuentran a una distancia Fsd del centro del robot sobre el eje Y y aquellos de los costados a una distancia Fss del eje Y . La distancia desde el sensor del medio hacia el centro del robot es a , mientras que la distancia hacia un sensor del costado es $Fsl = \sqrt{Fsd^2 + Fss^2}$.

3.4.5.1.1. Detalle del comportamiento

La decisión de dejar sólo dos líneas, acompañada por la elección de la ubicación de la zona de descarga, nos facilitó la composición de éste comportamiento.

Como mostramos en la figura 62, para ir a la línea se debemos girar hasta tener un ángulo de $\frac{\pi}{2}$ y luego ir hacia adelante.

3.4.5.1.2. Implementación del comportamiento

El *pseudo-código* de este comportamiento es sencillo, ya que no requiere cálculos extras:

```
por cada paso
    angulo_actual = obtener_angulo_actual(odometria)
    si esta_en_un_entorno_de(angulo_actual, PI/2) entonces
        si angulo_actual > PI/2 && angulo_actual < 3PI/2 entonces
```

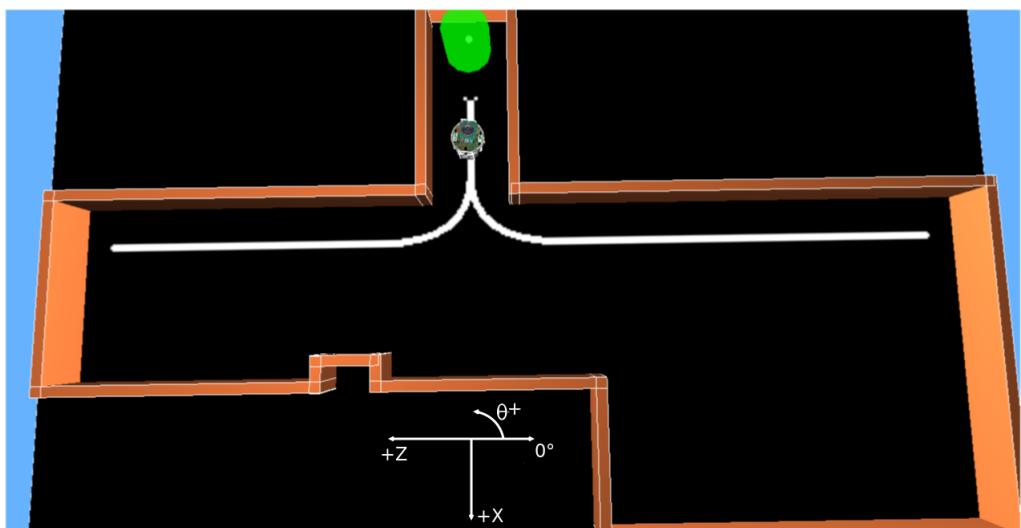


Figura 62: Arena de simulación y ejes de coordenadas

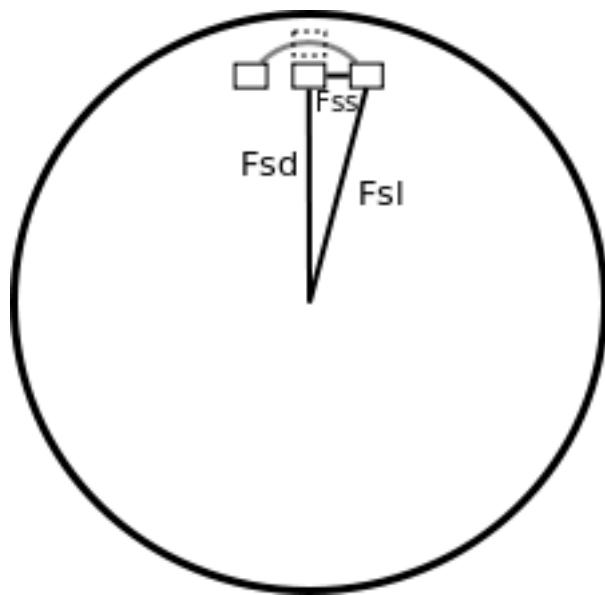


Figura 63: Disposición de sensores de piso

```

        giro_para_la_derecha
    sino
        giro_para_la_izquierda
    fin si
sino
    voy_hacia_linea
fin si

```

Hicimos la verificación de que el ángulo esté entre $\frac{\pi}{2}$ y $\frac{3\pi}{2}$ para evitar girar más de π , girando a la izquierda o a la derecha dependiendo cual sea el caso. Para ver esto más claramente, veamos que sucediese si no estuviera:

```

si esta_en_un_entorno_de(angulo_actual,PI/2) entonces
    giro_para_la_izquierda
sino

```

En el caso que el angulo actual sea π , el robot giraría un total de $\frac{3\pi}{2}$ hasta llegar al ángulo destino $\frac{\pi}{2}$, cuando en realidad girando para el sentido contrario sólo tendría que girar $\frac{\pi}{2}$.

Se puede observar en el código que usamos datos calculados por la odometría, en este caso, la orientación actual del robot. El lector se podría preguntar: “Si la odometría tiene la posición actual del robot, ¿porqué no se usaron los datos de la misma para ir hacia la base?”. En principio esto requiere que el robot tenga conocimiento acerca de la ubicación de la base. Por otro lado, se hubiera tenido que utilizar algún algoritmo de *Path Planning* para realizar el recorrido, algo que no concuerda con la arquitectura elegida. Es importante destacar el uso de datos calculados por la odometría porque si la misma llega a tener un error grande, puede llevar a una activación errónea de comportamientos. Decimos que la odometría tiene un error grande cuando, debido a esa magnitud, el robot no toma las decisiones correctas debido al error en la información en la cual se basó. En la sección 3.5.1 se puede ver la incidencia de un error grande de la odometría en el comportamiento emergente del robot.

3.4.5.2. Entrar a línea

3.4.5.2.1. Detalle del comportamiento

Para seguir la línea es necesario que primero el robot esté posicionado sobre ella y alineado con la misma. También se necesita que la dirección del robot sea la que lo lleve hacia la zona de descarga. Una vez en la línea, el robot deberá girar dependiendo de que lado se encuentre la misma (Ver figura 62).

3.4.5.2.2. Implementación del comportamiento

```

angulo_final = obtener_angulo_final(obtener_linea(odometria))
tita = atan(dist_sens_piso_X, dist_sens_piso_Y)
si (esta_en_la_linea(sensor_piso(DERECHA))) entonces
    tita = -tita
fin si
si (esta_en_la_linea(sensor_piso(MEDIO))) entonces
    tita = 0
fin si
si (tita != 0) entonces

```

```

        girar(tita)
fin si
distancia_a_recorrer = dist_sens_piso_Y;
si (tita != 0) entonces
    distancia_a_recorrer = sqrt(dist_sens_piso_X^2 + dist_sens_piso_Y^2)
fin si
recorrer(distancia_a_recorrer)
angulo_actual = obtener_angulo(odometria)
girar(normalizar(angulo_actual - angulo_final))

```

El primer giro del robot es para lograr que el segmento que une el sensor de piso del medio con el centro del robot quede ortogonal al segmento de línea. Como mostramos en la figura 64, en el caso (a) y (b) el robot girará de forma tal que llegue un caso parecido al que mostramos en la figura 65 ya posiblemente el sensor del medio no estará en la línea. Notar que si el sensor del medio está en la línea, como es el caso de la figura 64c, entonces el robot no gira, cualquiera sea el estado de los sensores de los costados. El ángulo que debe girar está dado por $\theta = \arctan\left(\frac{F_{ss}}{F_{sd}}\right)$ (Ver figura 63) si debe girar hacia la izquierda, o $-\theta$ en el otro caso.

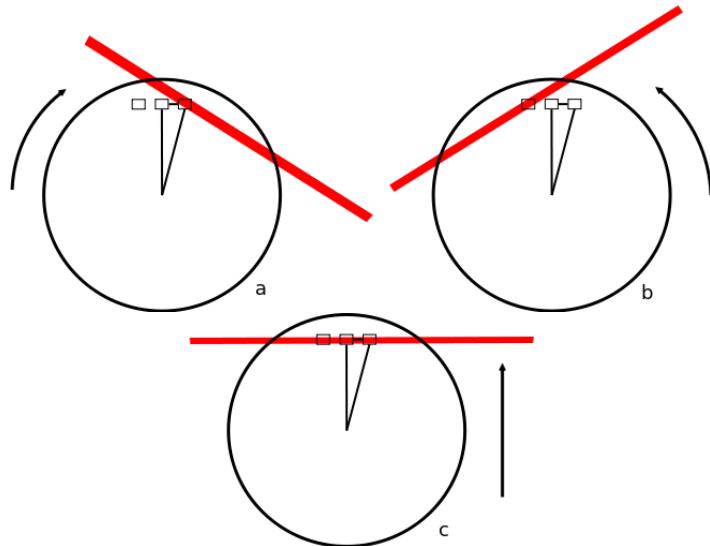


Figura 64: Posibles estados iniciales de “Entrar a la línea”. En el caso (a) el robot debe girar hacia la derecha. En el caso (b), el robot debe girar hacia la izquierda. En el caso (c) el robot no debe girar

Luego del posible giro para lograr que el sensor de piso del medio quede sobre la línea, se recorre una distancia de F_{sd} en el caso que el robot no haya girado anteriormente o F_{sl} en el caso que sí lo haya hecho. El motivo de realizar este trayecto es que el centro del robot quede sobre la línea, como muestra la figura 66. De ésta forma sólo queda girar nuevamente hacia el ángulo que se quiera. Si la línea es la izquierda entonces el robot deberá girar hasta que su orientación sea 0 o π en el caso que la línea sea la derecha.

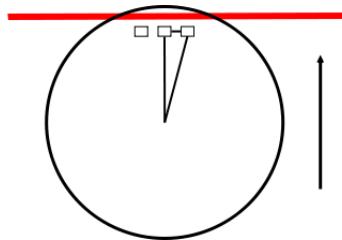


Figura 65: Posible estado final luego del primer giro del comportamiento “Entrar a la línea”

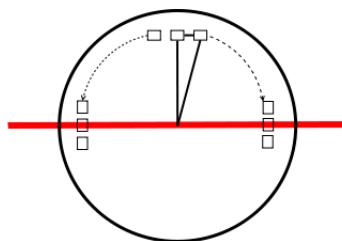


Figura 66: Centro del robot sobre la línea y posibles giros

3.4.5.3. Seguir línea

Una vez que el robot está posicionado y con el sensor del medio sobre la línea, sólo basta con seguirla para llegar hacia la zona deseada. Este comportamiento es sencillo de desarrollar.

3.4.5.3.1. Detalle del comportamiento

Para lograr que el robot siga la línea debemos tratar de mantener sólo el sensor del medio sobre la línea. Entonces, basta con analizar que se debe hacer en los siguientes casos:

1. El sensor de la derecha está sobre la línea
2. El sensor de la izquierda está sobre la línea

En el primer caso se debe lograr sacar el sensor de la derecha de la línea describiendo un pequeño arco hacia ese lado. El segundo caso es análogo: para sacar el sensor de la izquierda de la línea se debe seguir una trayectoria con un pequeño ángulo hacia la izquierda.

3.4.5.3.2. Implementación del comportamiento

El *pseudo-código* es simple:

```
por cada paso
    veloc_izq = veloc_der = VELOC_SEGUIR_LINEA
    si (esta_en_la_linea(sensor_piso(IZQUIERDA))) entonces
        veloc_izq *= (1 - FACTOR_DE_GIRO)
```

```

    veloc_der *= (1 + FACTOR_DE_GIRO)
fin si
si (esta_en_la_linea(sensor_piso(DERECHA))) entonces
    veloc_izq *= (1 + FACTOR_DE_GIRO)
    veloc_der *= (1 - FACTOR_DE_GIRO)
fin si
poner_velocidades_en_ruedas(veloc_izq, veloc_der)

```

3.4.6. Descargar Basura

Una vez que el robot logró llegar a la zona de descarga de basura, debe descargarla. Para hacer ésto decidimos ubicar un servo en la parte trasera del robot, con la misma idea del servo en el frente utilizado para recolectar.

3.4.6.1. Detalle del comportamiento

Para descargar la basura el robot debe posicionarse de forma tal que la compuerta de descarga quede adyacente a la zona. Dado que para llegar a la misma el robot siguió la línea, al finalizar va a estar orientado con un ángulo en un entorno de $\frac{\pi}{2}$ mirando la zona de descarga. Como el servo de descarga se encuentra en la parte trasera, deberá realizar un giro de aproximadamente π para luego poder descargar la basura.

3.4.6.2. Implementación del comportamiento

```

posicionarse()
levantar(servo_trasero)
recorrerdistancia(ANCHO_ROBOT)
cerrar(servo_trasero)

```

3.4.7. Ir a base de recarga de batería

Ayudados por la elección que tomamos de poner la zona de descarga de basura muy cercana a la zona donde se recarga la batería, decidimos utilizar la misma estrategia de seguir la línea para llegar hacia la misma.

La diferencia entre ambos casos es el estímulo ante el cual se activan. En el caso de ir a la zona de descarga, el estímulo proviene del sensor del depósito interno de basura que indica que el mismo está lleno. En el caso de ir a la base de recarga de batería, el estímulo para la activación depende de los valores de dos sensores de batería que indican batería baja:

- El sensor de la batería del robot, de donde se alimentan los sensores, actuadores y motores.
- El sensor de la computadora que corre el controlador.

3.4.8. Cargar Batería

3.4.8.1. Detalle del comportamiento

Una vez que el robot logró llegar a la zona de recarga de batería, debe recargarse. Para ésto, debe posicionarse para lograr ubicarse el cargador y esperar

hasta que la batería alcance un valor de carga tal que le permita seguir buscando basura sin problemas de batería.

3.4.8.2. Implementación del comportamiento

```

posicionarse()
mientras(bateria_no_llena(BATERIA_ROBOT)
          o bateria_no_llena(BATERIA_PC)) hacer

    esperar(time_step)
fin mientras

```

3.4.9. Evitar Obstáculos

Evitar obstáculos es uno de los comportamientos con mayor jerarquía en la arquitectura que elegimos. Su nivel se debe a la importancia que tiene en un ambiente estructurado pero dinámico como es el nuestro. El objetivo del robot es facilitar una tarea sin entorpecer el tránsito de personas.

3.4.9.1. Detalle del comportamiento

Para lograr tener conocimiento sobre la proximidad de un obstáculo, utilizamos sensores de proximidad explicados en 2.4. Dependiendo de la proximidad indicada por un determinado sensor, el robot deberá alejarse de ese lado para evitar un posible choque. La activación del comportamiento dependerá entonces del valor sensado tal que la distancia hacia un obstáculo lleve al robot a una colisión o le imposibilite moverse.

3.4.9.2. Implementación del comportamiento

La implementación de este comportamiento la hicimos utilizando una red neuronal sin capas ocultas (ver figura 67) usando los sensores de distancia como entradas y 2 neuronas de salida indicando los valores a ser seteados en los motores de las ruedas.

Debemos notar que hay conexiones tanto inhibitorias como excitatorias. Como es de esperarse, ambos sensores traseros (4 y 5) exitan a ambos motores. Distinto es el caso de los sensores del lado izquierdo (5, 6 y 7) que exitan el motor ubicado de su lado e inhiben el motor del lado opuesto, de forma tal que el robot gire para el lado opuesto de la ubicación de los sensores. La misma idea se sigue con los sensores (0, 1 y 2) ubicados en el costado derecho del robot.

Luego de entrenar la red, obtuvimos los siguientes valores para los pesos de la misma:

Rueda	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
Izquierda	-0.9	-0.85	-0.2	0.6	0.5	0.35	0.8	0.6
Derecha	0.9	0.85	0.2	0.6	0.5	-0.35	-0.8	-0.6

Cuadro 22: Asignación de pesos para evitar obstáculos

Se puede ver que los pesos que influyen en el motor de una rueda influyen

con igual fuerza pero distinto signo en el motor de la rueda contraria.

Este comportamiento, en *pseudo-código* puede verse como:

```
por cada paso
    veloc_izq = suma(coeficiente(RUEDA_IQZ, SENSOR_I) * VALOR(SENSOR_I))
    veloc_izq *= FACTOR_DE_INCIDENCIA
    veloc_der = suma(coeficiente(RUEDA_DER, SENSOR_I) * VALOR(SENSOR_I))
    veloc_der *= FACTOR_DE_INCIDENCIA
    poner_velocidades_en_ruedas(veloc_izq, veloc_der)
```

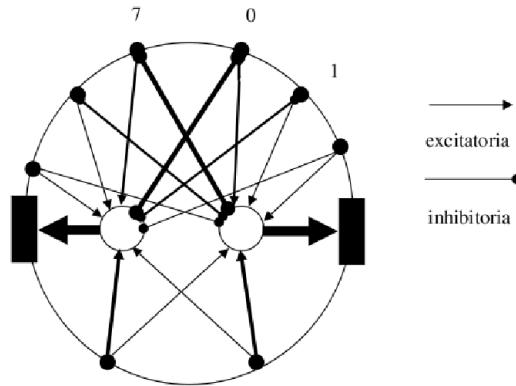


Figura 67: Red neuronal entre los sensores de distancia y los motores

3.4.10. Salir de situaciones no deseadas

Salir de situaciones no deseadas surgió como un comportamiento para ayudar al objetivo de crear un robot autónomo. Con las sucesivas simulaciones observamos que había situaciones donde corría peligro la autonomía del robot. Un ejemplo de estas situaciones es el caso donde dos comportamientos se “activan mutuamente”.

Para ver esto más claramente, supongamos dos comportamientos *A* y *B* con nivel en la jerarquía $N(A)$ y $N(B)$, siendo $N(A) > N(B)$. Si la respuesta a un estímulo de *A* lleva a la activación de *B* y a la desactivación de *A* y luego la respuesta de *B* lleva a la activación de *A*, podría llegar a entrarse en un ciclo si es que esta situación se da por un tiempo prolongado o indeterminado. El mayor peligro para la autonomía ocurre cuando *A* es el comportamiento de *evitar obstáculos* y *B* es el comportamiento de *ir a zona de recarga de batería* ya que el robot podría terminar quedándose sin batería en ese ciclo.

3.4.10.1. Detalle del comportamiento

Las situaciones no deseadas se dan la mayoría de los casos cuando un comportamiento hace girar al robot hacia un lado y el otro comportamiento hacia el lado contrario, aproximadamente en la misma magnitud. Ésto quiere decir que la posición del robot se mantiene alrededor de un punto por un período prolongado de tiempo, por lo que decidimos tomar este hecho como estímulo para la activación de este comportamiento.

La respuesta del comportamiento es, girar un ángulo que cambie la dirección del robot y además que la suma de ese ángulo repetidas veces no sea periódica o lo sea luego de una gran cantidad de giros como por ejemplo, un valor de $\frac{\pi}{4} + 0.1$. Esta última condición se pide por el siguiente escenario:

- Los comportamientos *A* y *B* se “activan mutuamente” cuando la orientación del robot es 0.
- Los comportamientos *C* y *D* se “activan mutuamente” cuando la orientación del robot es π .
- La respuesta de *salir de situaciones no deseadas* es girar un ángulo π .

3.4.10.2. Implementación del comportamiento

```
angulo_actual = obtener_angulo_actual(odometria)
nuevo_angulo = angulo_actual + ANGULO_A_SUMAR
girar(nuevo_angulo)
```

3.5. Odometría

Para saber donde se encuentra el robot en cierto momento utilizamos odometría. Esta técnica se basa en la medición del encoder en cuentas realizadas por cada motor para obtener el desplazamiento realizado por la rueda asociada. A diferencia de los métodos de posicionamiento absoluto, la odometría da una estimación del desplazamiento *local* a la ubicación anterior del robot, por lo cual *un error en una estimación se propaga* hacia las siguientes estimaciones.

Para calcular la posición P_n en el instante de tiempo n y la orientación O_n en base a la posición P_{n-1} en el instante (n-1) y la correspondiente orientación O_{n-1} , usamos las siguientes fórmulas:

$$d_l = \frac{(e_l(n) - e_l(n-1)) * R_l}{EncRes} \quad (13)$$

$$d_r = \frac{(e_r(n) - e_r(n-1)) * R_r}{EncRes} \quad (14)$$

$$lc = \frac{d_r + d_l}{2} \quad (15)$$

$$P_n = P_{n-1} + (lc * \cos O_{n-1}, lc * \sin O_{n-1}) \quad (16)$$

$$O_n = O_{n-1} + \frac{d_r - d_l}{dbw} \quad (17)$$

donde d_l y d_r son las distancias recorridas por las ruedas izquierda y derecha respectivamente. Ambas son calculadas teniendo en cuenta los valores anteriores y actuales de los encoders $e_i(n)$, el radio de la rueda R_i , la resolución del encoder $EncRes$ y es la distancia entre ruedas dbw .

Los errores que influyen en el cálculo de la odometría pueden ser de dos tipos:

- Sistemáticos: Son aquellos que pueden ser corregidos o tenidos en cuenta para disminuir el error.
- No sistemáticos: Son aquellos que pueden intentarse corregir pero *no* eliminar.

Decidimos tener en cuenta dos errores sistemáticos para disminuir el error en la odometría :

- Incertezza sobre la distancia entre las ruedas (dbw)
- Ruedas con radios diferentes (R_l y R_r)

Tuvimos en cuenta estos errores dado que son los que más contribuyen al error acumulado a lo largo del trayecto del robot. Para corregir estas fuentes de error, utilizamos el *test del camino bidireccional describiendo un cuadrado* (UMB-mark)¹⁶. Dado que en el momento de realizar el test no teníamos un robot físico, decidimos realizar el test sobre un e-puck simulado en Webots. Así fuimos obteniendo los valores de corrección para los diámetros de las ruedas c_i y la corrección sobre la distancia entre las ruedas c_{dbw} hasta que el error sistemático máximo dejara de disminuir. En la figura 68 mostramos cómo disminuye el error a lo largo de las iteraciones.

¹⁶<http://www-personal.umich.edu/~johannb/Papers/umbmark.pdf>

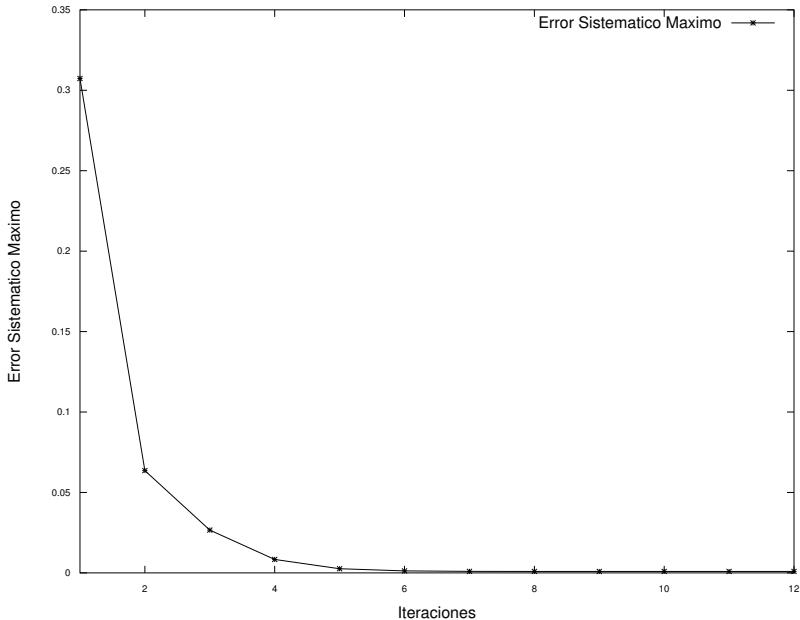


Figura 68: Error Sistémico Máximo a lo largo de las iteraciones

En la última iteración obtuvimos los coeficientes de corrección $c_l = 0.99998703$, $c_r = 1.00001297$ y $c_{dbw} = 1.092171094$ para ruedas con radio $R_l = R_r = 0.205$, una distancia entre ruedas de 0.052 y una resolución de encoder $EncRes = 159.23$.

3.5.1. Problemas con la odometría

En las simulaciones previas a la presentada en la sección 3.7 observamos que el comportamiento emergente del robot no era el esperado. Viendo más en detalle, el robot tomaba decisiones erróneas en los comportamientos que utilizaban la odometría. Después de una extensiva revisión de código, llegamos a la conclusión que el mismo no tenía problemas.

Fue entonces que decidimos agregar un GPS (con error mínimo y que luego eliminamos) al e-puck en la simulación con Webots, con el objetivo de obtener un gráfico que indique el error en la odometría a lo largo de la simulación. El resultado esperado era que la odometría inicie con un error cercano a 0 y vaya creciendo constantemente a medida que avanzaba la simulación. El resultado no fue el esperado pero si iluminativo: el crecimiento era en partes constantes, pero había momentos en los cuales crecía abruptamente.

Decidimos entonces simular nuevamente y ver que comportamiento estaba activo en dichos momentos. Nos sorprendió ver que en esos momentos el comportamiento activo era *Evitar obstáculos* ya que era uno de los más simples y de los que más teníamos confianza que no haya un error de lógica de código. Nuevamente revisamos el código, sin encontrar problemas.

Ésto nos llevó a pensar qué relación había entre *Evitar obstáculos* y la odometría. El comportamiento setea velocidades en las ruedas y la odometría utiliza los valores de los encoders de las ruedas. Revisando las ecuaciones 13 y 14 nos dimos cuenta de lo siguiente: dado que las correcciones c_l y c_r de los radios R_l y R_r de las ruedas minimizan el error pero no lo eliminan, el error de las ecuaciones aumenta proporcionalmente a la diferencia entre $e_i(n)$ y $e_i(n - 1)$. Y ésto sucedía cuando se activaba el evitamiento de obstáculos ya que daba velocidades a las ruedas de un orden de magnitud mayor a los que tenían anteriormente.

Finalmente, teníamos que buscar una solución a éste problema, ya sabiendo la verdadera causa del mismo. Pensamos en dos opciones:

1. Disminuir las velocidades que *Evitar obstáculos* asignada a las ruedas, dividiéndolas por una cte.
2. Hacer que cada vez que se setee una velocidad, se haga un cálculo con la velocidad seteada anteriormente, de forma tal que no haya saltos de órdenes de magnitud.

Por el tiempo que disponíamos en ese momento y por simplicidad y rapidez de implementación, elegimos la primer opción. Cabe aclarar que la segunda opción es más abarcativa y robusta, ya que sirve para cualquier comportamiento. Sin embargo, es invasiva en parte ya que puede que el desarrollador quiera, por algún motivo, que haya ese tipo de variaciones y no va a poder lograrlo.

3.6. Interfaces con hardware y módulo de reconocimiento de objetos

Decidimos hacer que el controlador encargado de realizar los comportamientos sea independiente de la forma con la que se implemente el hardware y el reconocimiento de los objetos. Para ésto definimos interfaces para la comunicación entre el controlador y el hardware y el módulo de reconocimiento de objetos de forma tal que los dos últimos puedan cambiar su implementación pero brindando siempre la información que necesita el controlador para poder realizar los comportamientos. Esta decisión nos posibilitó realizar un controlador que sea capaz de ser ejecutado tanto en Webots, donde se hizo el desarrollo, como en la base real del robot. Cabe aclarar que el traspaso de la simulación a la realidad no es instantánea, ya que hay que calibrar los dispositivos y realizar nuevamente la odometría, entre otras cosas, pero el trabajo demandado es mucho menor ya que una corrección en la lógica de los comportamientos se puede observar tanto en un simulador como en la realidad.

3.6.1. Interfaz con hardware

La interfaz con el hardware se basa en tener clases encargadas de obtener los valores de los sensores o del accionar de los actuadores.

En la implementación de la interfaz que corrimos en Webots, hicimos llamadas al controlador del simulador, que utiliza sensores y actuadores simulados.

En la implementación que se comunica con el robot físico, las llamadas las hicimos a un servidor encargado de enviar y recibir paquetes del protocolo descripto en la sección 2.6 a través del puerto serial.

De esta forma es cuestión de decidir que implementación se usa en base a si se quiere correr en un simulador como Webots o en el robot físico. Si quisieramos usar otro simulador u otro tipo de hardware, sólo haría falta que implementemos la interfaz que cumpla con lo establecido e indicarle a la capa de comportamientos que la utilice para realizar su ejecución.

3.6.2. Interfaz con módulo de reconocimiento de objetos usando Visión

Como el controlador de comportamientos es **cliente** del módulo de reconocimiento, necesita conocer en el instante de tiempo t , que objetos están siendo reconocidos. Para esto el módulo debe tener una fuente de información, en nuestro caso, una cámara usada para Visión.

Desde el punto de vista anatómico, ésto se puede ver como los ojos (cámara), la parte del cerebro encargada de analizar el estímulo recibido (imágenes) por los mismos (módulo de reconocimiento) y la parte del cerebro encargada de analizar los estímulos recibidos y realizar las acciones (controlador de comportamientos). Si hubiésemos usado algún tipo de conjunto de sensores táctiles para reconocer objetos (mano), en vez de visión, el módulo de reconocimiento de objetos bien podría analizar la información que ellos proveen e informar al controlador sobre su análisis.

Esta analogía puede llevar a pensar que los sensores de distancia u otros dispositivos que usamos en este desarrollo bien podrían formar parte de otro módulo, y no se estaría equivocado. La razón por la cual el módulo de visión está sepa-

rado y el resto de los sensores no, es que el procesamiento de una secuencia de imágenes es muy complejo y demandante, tal como detallamos en la sección 4.

3.7. Resultados obtenidos

En esta sección describimos los parámetros que utilizamos en la simulación para obtener los resultados sobre los diferentes comportamientos. Luego presentamos dichos resultados y extraemos conclusiones sobre los mismos.

A lo largo del proceso de desarrollo fuimos realizando diferentes simulaciones, de las cuales observamos defectos en las implementaciones de los comportamientos así como retardos y detalles que podíamos mejorar en algunos y que fuimos corrigiendo.

La arena utilizada en dichas simulaciones la mostramos en la figura 69. A lo largo de la misma hay ubicados 15 objetos que simulan ser basuras, un área de recarga de batería y un área de descarga de basura, marcadas por el cilindro naranja. Los valores de los parámetros que utilizamos los detallamos en el cuadro 23. Los ángulos los expresamos en radianes y las distancias las expresamos en metros, así como también los radios, correcciones y separaciones. La resolución de la cámara la expresamos en pixeles, la de la grilla en unidades y los tiempos están en milisegundos.

Parámetro	Descripción del parámetro	Valor
$P_r(0)$	Posición Inicial del robot	(-0.295,-0.4)
$O_r(0)$	Orientación Inicial del robot	0
R_r	Radio del robot	0.026
dbw	Distancia entre ruedas	0.052
c_{dbw}	Corrección de la distancia entre ruedas	1.09217109
R_l	Radio de la rueda izquierda	0.0205
R_r	Radio de la rueda derecha	0.0205
c_l	Corrección del radio de la rueda izquierda	0.99998702
c_r	Corrección del radio de la rueda derecha	1.00001297
$EncRes$	Resolución del encoder	159.23
Cd	Distancia de la cámara al centro del robot	0.0355
Ch ó C_h	Altura de la cámara	0.038
FOV_h	Field of view horizontal	1.1
ac	Ángulo de la cámara	-0.5
C_{rw}	Ancho de la imagen obtenida de la cámara	640
C_{rh}	Alto de la imagen obtenida de la cámara	480
A_w	Ancho del arena	2
A_h	Alto del arena	1.2
A_{rw}	Ancho de la grilla	100
A_{rh}	Alto de la grilla	38
Fsd	Dist. s.de piso del medio al centro del robot	0.03
Fss	Separación entre sensores de piso	0.01
δ_f	Ángulo umbral de basura enfocada	0.1
T_s	Time Step	32

Cuadro 23: Parámetros utilizados para las simulaciones

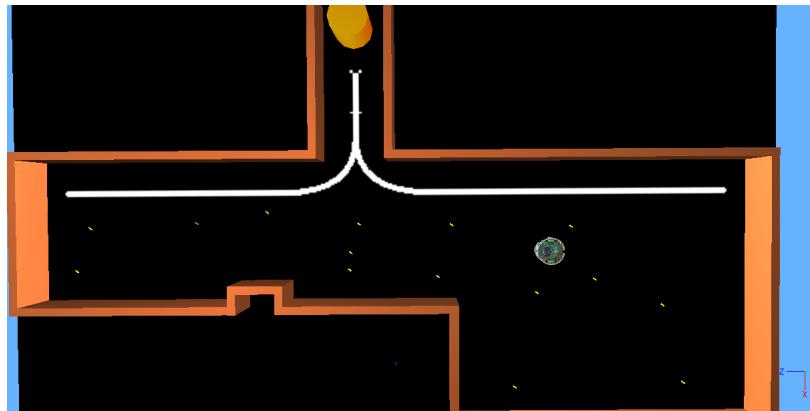


Figura 69: Arena que utilizamos para las simulaciones

3.7.1. Distribución de tiempos y progreso de comportamientos en la simulación

En la figura 70 mostramos la distribución de time steps en la simulación de cada comportamiento. Podemos ver que *Recargar batería* y *Descargar basura* en conjunto están activos el 54 % de la simulación. Los comportamientos involucrados en la recolección de basura (*Enfocar basura*, *Ir a basura* y *Recolectar basura*) abarcan el 10.5 % del tiempo total de simulación. El 35.5 % restante está repartido en los comportamientos encargados de una navegación libre de obstáculos y situaciones peligrosas (*Wandering*, *Evitar obstáculos* y *Salir de situaciones no deseadas*). Todos los comportamientos relacionados con la basura(desde que se detecta hasta que se descarga) abarcan un 43 % del tiempo total de simulación. También se puede observar que el comportamiento que mayor tiempo está activo a lo largo de la simulación es *Descargar basura* y el que menor tiempo está activo es *Salir de situaciones no deseadas*.



Figura 70: Distribución de los tiempos de los comportamientos en la simulación

En la figura 71 mostramos el progreso a lo largo de la simulación de todos los comportamientos. Observamos que en el comienzo ($0 < t < 10000$)¹⁷ la mayoría

¹⁷t es el número de time step de la simulación

de los comportamientos está activo aproximadamente la misma cantidad de time steps, salvando el caso de *Descargar Basura*. En el período ($40000 < t < 50000$), el comportamiento *Wandering* alcanza a estar activo la misma cantidad de steps que *Ir a basura*. Podemos ver también que la forma en que evolucionan *Enfocar basura*, *Ir a basura* y *Recolectar basura* a lo largo de la simulación es muy similar, salvando la cantidad de steps que están activos. La evolución de *Recargar batería* es periódica: inicia con una meseta de time steps en los cuales no está activo y luego tiene una pendiente pronunciada. Diferente es el progreso de *Evitamiento de obstáculos* ya que sus mesetas son las menos prolongadas y sus pendientes son poco abruptas.

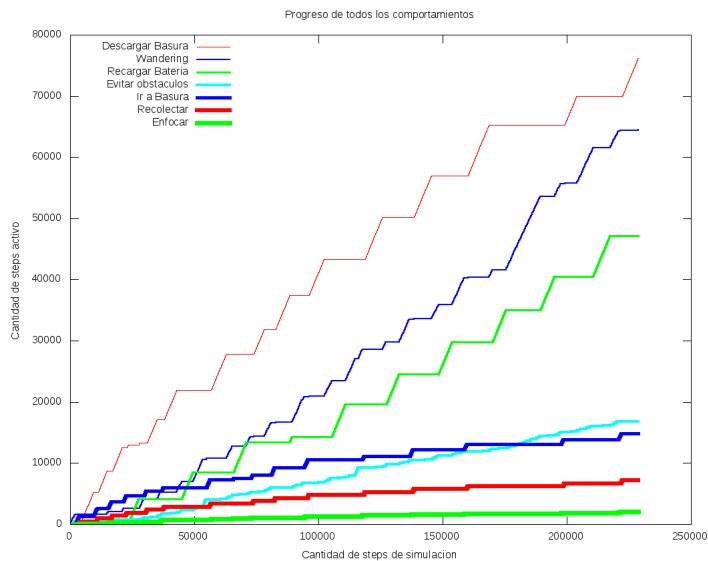


Figura 71: Evolución de los comportamientos a lo largo de la simulación

En la figura 72 mostramos más en detalle el comportamiento vital para nuestro proyecto, *Recolectar basura*. Podemos ver que en el período ($0 < t < 50000$) acumula aproximadamente la misma cantidad de steps hechos que en el período ($50000 < t < 150000$). El progreso describe mesetas cuyas duraciones se van prolongando a medida que se avanza la simulación y las pendientes tienen aproximadamente la misma magnitud a lo largo de la misma.

Para que pudieramos ver más en detalle lo que sucedió con los comportamientos, obtuvimos la cantidad de veces que cada uno se activó (Ver cuadro 24). En el mismo observamos que el comportamiento que más veces se activó fue *Evitar obstáculos*, seguido por *Enfocar basura* y *Wandering*. En un orden de magnitud menor están *Ir a descargar basura* e *Ir a recargar batería*. Finalmente, *Recolectar basura* se activó 15 veces y *Salir de situaciones indeseadas* nunca.

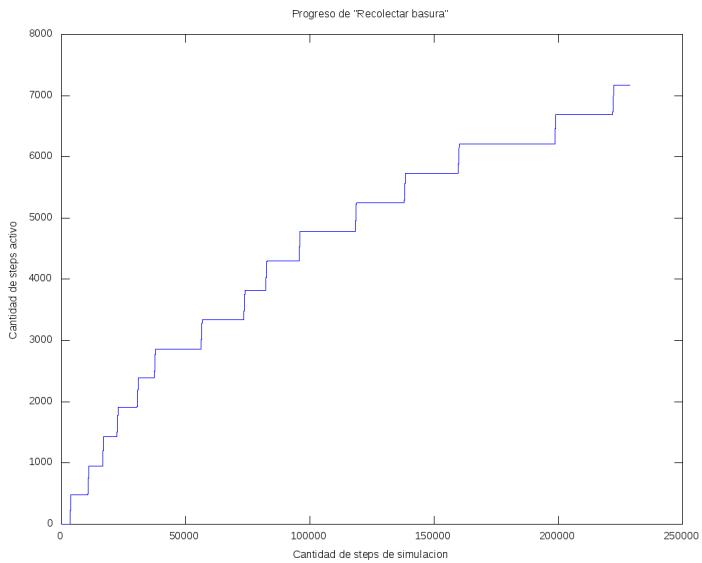


Figura 72: Progreso del comportamiento *Recolectar Basura*

Comportamiento	# total de steps activo	# veces que se activó
Wandering	64476	1330
Enfocar basura	2057	1384
Ir a basura	14832	1218
Recolectar basura	7166	15
Ir a descargar basura	76160	321
Ir a recargar batería	47188	265
Evitar obstáculos	16843	1507
Salir situaciones indeseadas	0	0

Cuadro 24: Resultados sobre steps en los cuales los comportamientos están activos

3.7.2. Tiempos en que los comportamientos están continuamente activos e inactivos

En los cuadros 25 y 26 mostramos cálculos sobre los steps que cada comportamiento estuvo *caa*¹⁸ y *cai*¹⁹, respectivamente.

Podemos ver que *Recolectar basura* es en promedio el mayor en ambos cuadros y que tiene el menor desvío estándar en *caa*. *Wandering* es en promedio el que menor tiempo está *cai*, con el segundo menor desvío. Sigue lo mismo con *Evitar obstáculos*, segundo en promedio en estar *cai* y con el menor desvío. Ambos, además, tienen los mínimos máximos en estar *cai*, con una diferencia de un orden de magnitud con respecto al resto de los comportamientos. *Ir a descargar basura* e *Ir a recargar batería* son de los que mayor promedio y desvío tienen tanto en estar *caa* como en estar *cai*. Los comportamientos de *Enfocar basura* y *Ir a basura* tienen un promedio y desvío estándar de los más bajos en estar *caa*.

Comportamiento	Promedio	Desvío Estándard	# máximo
Wandering	48.44	225.07	2212
Enfocar basura	1.48	3.27	67
Ir a basura	12.17	58.44	1222
Recolectar basura	477.73	0.70	478
Ir a descargar basura	237.25	840.49	6012
Ir a recargar batería	178.06	647.39	3950
Evitar obstáculos	11.17	54.72	1386
Salir situaciones indeseadas	-	-	-

Cuadro 25: Resultados sobre steps que los comportamientos están continuamente activos

Comportamiento	Promedio	Desvío Estándard	# máximo
Wandering	123.4	754.4	8351
Enfocar basura	163.7	1766	38000
Ir a basura	175.5	1811	38040
Recolectar basura	13850	9479	38400
Ir a descargar basura	473.8	2689	30520
Ir a recargar batería	682.5	3322	23270
Evitar obstáculos	140.5	724.3	9485
Salir situaciones indeseadas	-	-	-

Cuadro 26: Resultados sobre steps que los comportamientos están continuamente inactivos

¹⁸caa: continuamente activo. Por continuamente activo se entiende que estuvo activo en el time step $t - 1$ y en el t

¹⁹cai: continuamente inactivo. Por continuamente inactivo se entiende que no estuvo activo en el timestep $t - 1$ ni en el t

3.7.3. Transiciones hacia y desde un comportamiento a otro

En el cuadro 27 mostramos las transiciones entre los comportamientos. Éstos están abreviados y presentados en el mismo orden que en la sección 3.4. El valor en la posición $(i,j) = t_{ij}$, siendo i y j el número de fila y columna respectivamente, se lee como: “desde el comportamiento i se pasó t_{ij} veces al comportamiento j ”. También se puede leer como: “el comportamiento j se activó t_{ij} veces siendo el comportamiento i el que anteriormente estaba activo”. Consideraremos que hay una transición cuando $i \neq j$, por lo que $t_{ii} = 0$. Una transición de i a j con $i > j$ puede ser porque el estímulo necesario para la activación del comportamiento i ya no está presente. Una transición de i a j con $i < j$ puede ser porque el estímulo necesario para la activación del comportamiento j pasó a ser sensado.

El comportamiento que más veces “activó” a otro comportamiento es *Enfocar basura*, activando a *Ir a basura*. Observamos también que el caso contrario es la 2da transición más hecha y ambos valores están relativamente cercanos. El caso que los valores de t_{ij} y t_{ji} estén cercanos también sucede con *Evitar obstáculos* y el resto de los comportamientos, aunque en estos casos la diferencia es mucho menor.

Podemos ver que hay transiciones que sólo se dan 1 sola vez. Éstas son:

- *Wandering* a *Recolectar basura*,
- *Enfocar basura* a *Descargar basura* e
- *Ir a basura* a *Recargar batería*

	W	EB	IAB	RB	DB	R	EO
W	0	303	176	1	12	5	833
EB	238	0	1036	2	1	0	107
IAB	229	967	0	12	2	1	7
RB	12	3	0	0	0	0	0
DB	14	0	0	0	0	2	305
R	9	0	0	0	1	0	255
EO	828	111	6	0	305	257	0

Cuadro 27: Transiciones entre comportamientos

Mostramos también las transiciones en forma de porcentajes en los cuadros 28 y 29. En el primero detallamos las transiciones desde i hacia j y en el segundo, transiciones de j hacia i . Se puede ver que las transiciones que más sucedieron fueron desde *Descargar basura* y *Recargar batería* hacia *Evitar obstáculos*, con un porcentaje mayor al 95 %, y que el segundo también es la causa principal de la activación de los primeros, con porcentajes de 95 % y 97 % respectivamente.

En segundo lugar se ubican las transiciones desde *Enfocar basura* hacia *Ir a basura* y el recíproco, con valores cercanos al 75 %. Algo parecido sucede en el segundo cuadro.

El 80 % de las veces que estuvo activo *Recolectar basura* luego se activó *Wandering* y el 80 % de las veces que se activó el primero fue por parte de *Ir a basura*.

	W	EB	IAB	RB	DB	R	EO
W	0 %	0.227 %	0.132 %	0.001 %	0.009 %	0.003 %	0.626 %
EB	0.172 %	0 %	0.748 %	0.002 %	0.001 %	0 %	0.077 %
IAB	0.188 %	0.794 %	0 %	0.01 %	0.002 %	0.001 %	0.006 %
RB	0.8 %	0.2 %	0 %	0 %	0 %	0 %	0 %
DB	0.043 %	0 %	0 %	0 %	0 %	0.006 %	0.951 %
R	0.034 %	0 %	0 %	0 %	0.004 %	0 %	0.962 %
EO	0.55 %	0.074 %	0.004 %	0 %	0.202 %	0.170 %	0 %

Cuadro 28: Transiciones desde i hacia j en porcentajes

	W	EB	IAB	RB	DB	R	EO
W	0 %	0.179 %	0.172 %	0.009 %	0.011 %	0.007 %	0.623 %
EB	0.219 %	0 %	0.699 %	0.002 %	0 %	0 %	0.08 %
IAB	0.145 %	0.851 %	0 %	0 %	0 %	0 %	0.004 %
RB	0.066 %	0.134 %	0.8 %	0 %	0 %	0 %	0 %
DB	0.037 %	0.003 %	0.006 %	0 %	0 %	0.003 %	0.95 %
R	0.019 %	0 %	0.003 %	0 %	0.008 %	0 %	0.97 %
EO	0.553 %	0.071 %	0.004 %	0 %	0.202 %	0.170 %	0 %

Cuadro 29: Transiciones desde j hacia i en porcentajes

3.7.4. Progreso de recorrido de la arena a lo largo de la simulación

En la figura 73 mostramos el progreso del área recorrida por el robot a lo largo de la simulación. El arena estaba dividida en A_{rw} . $A_{rh} = 3800$ slots. Como es de esperarse, la suma de cantidad de slots vistos y slots restantes en un instante de tiempo t es $cte = 3800$. El tiempo en que se llegó a recorrer todo el arena fue $t_{fv} = 95241$. Dado que la duración de la simulación fue de $t_{fs} = 228721$, el arena se recorrió en el 0.42 % del tiempo total de simulación.

Se puede ver que al principio de la simulación ($0 < t < 17000$) hay pendientes abruptas, con mesetas de pocos steps. Luego las pendientes tienen menor valor y las mesetas son más prolongadas, salvando el caso del intervalo ($35000 < t < 40000$).

En el cuadro 30 vemos más en detalle el progreso del porcentaje de arena cubierta. Tomamos 10 muestras uniformes en el período ($0 < t < t_{fv}$) y obtuvimos el porcentaje cubierto en esa muestra, así como también el acumulado la misma. Podemos ver que ya en la 2da muestra, $0 < t < 0.042t_{fs}$, se alcanza a recorrer el 50 % del total del arena. Después los incrementos son más pequeños en comparación a los primeros, salvando el caso de la cuarta y última muestra. En la primer muestra el robot recorrió aproximadamente el 25 % de la arena, en la segunda el 50 % y en la cuarta el 75 %.

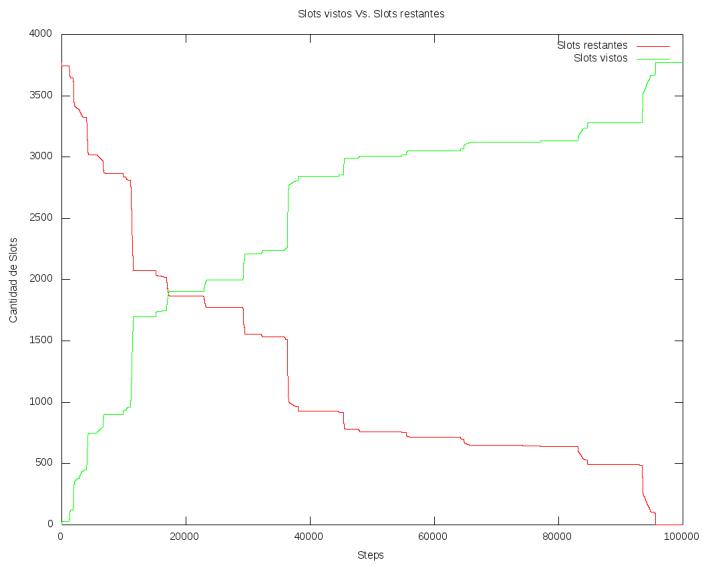


Figura 73: Área visualizada y área no visualizada a lo largo de la simulación

<i>Nº muestra</i>	(%)	(%) Acumulado
1	0.246	0.246
2	0.258	0.505
3	0.081	0.586
4	0.167	0.754
5	0.044	0.798
6	0.011	0.809
7	0.017	0.827
8	0.003	0.830
9	0.038	0.869
10	0.130	1

Cuadro 30: Porcentaje de arena cubierta

3.8. Conclusión

En ésta sección sacamos conclusiones sobre los resultados obtenidos en la sección 3.7, así como dar posibles explicaciones a los mismos.

Dado que en la simulación había 15 basuras, era esperado que *Recolectar basura* se activase la misma cantidad de veces. En promedio, el robot recolectó

$$\frac{t_{fs}}{15\text{basuras}} = \frac{1.52 \times 10^5 \text{Ts}}{\text{basura}}$$

es decir, 1 basura cada 8 minutos. El mayor tiempo que pasó entre entre una recolección y otra fue de $38400 \text{Ts} = 20m28s$.

Los comportamientos íntimamente relacionados, *Enfocar basura* e *Ir a basura* se activaron más de 15 veces. Ambos comportamientos tuvieron un nivel de activación parecido a la cantidad de steps que estuvieron activos, provocando un bajo promedio de steps *caa*. Ésto pudo haber sido causado debido al umbral δ_f en el cual se decide si una basura está enfocada o no (ver figura 59). Un valor muy grande del mismo, figura (a), cubre más porcentaje de la imagen, por lo que va a llevar a indicar que la basura se enfoca más rápido, disminuyendo el tiempo en que el comportamiento está activo. Un valor chico (figura (b)), hará que la mayor cantidad de veces que se detecte una basura, la misma no esté enfocada y al cubrir menos porción de la imagen, aumentará el promedio de time steps en los cuales tiene que enfocar. Éste caso es más sensible a la situación en la que la basura se mueve, ya que va a ocasionar que se tenga que enfocar nuevamente. El primer caso es menos sensible al movimiento de la basura, pero tiene el siguiente defecto: cuando la basura se sienta como enfocada, la misma *NO* está frente al robot, y el comportamiento de *Ir a basura* sólo va hacia adelante, suponiendo que la basura se encuentra frente al robot, ocasionando que la basura pase a no estar enfocada y una nueva activación de *Enfocar basura*. En los cuadros de transiciones 28 y 29 podemos ver que ambos comportamientos son los mayores proveedores de transiciones de dichos estados al otro.

Como ya mencionamos, *Ir a descargar basura* e *Ir a recargar batería* en conjunto abarcan el 54 % del tiempo total de simulación. Además son de los que mayor promedio están *caa*, lo que nos lleva a pensar que una activación incorrecta de los mismos es indeseable, ya que son de los que más jerarquía tienen y por lo tanto inhibirían innecesariamente y por un tiempo relativamente prolongado a otros. Una activación incorrecta puede estar dada por:

- Mal funcionamiento de los sensores
- Haber juntado un objeto que no era basura
- Una falla en la lógica de activación de comportamientos

El gran desvío estándar de ambos en *caa* se debe a que su acción es llevar al robot hacia las líneas, y la distancia y el recorrido por las mismas varía según la posición del robot cuando se hizo presente el estímulo correspondiente.

El promedio en estar *cai* de *Ir a descargar basura* depende directamente de la cantidad de basuras que haya en la arena y la efectividad del módulo de reconocimiento. En el caso que no haya basuras en la arena, lo ideal es que el comportamiento nunca se active, aunque ésto depende de las posibles activaciones incorrectas. El gran desvió estándar en estar *cai* se debe a que al

principio de la simulación había más basuras para recolectar, disminuyendo el tiempo entre activaciones, y llegando al final de la simulación había menos basuras, disminuyendo la probabilidad que el robot las encuentre y aumentando el tiempo entre activaciones.

Los valores de *Wandering* dependen totalmente de las activaciones de los otros comportamientos, ya que está activo si y sólo si ningún otro está activo. En un arena con 15 basuras, estuvo activo un 28 % del tiempo de simulación. Teniendo en cuenta que los comportamientos relacionados con las basuras abarcaron un 43 % de t_{fs} , es de esperarse que en una simulación sin basuras el protagonismo de *Wandering* sea el mayor de todos superando el 70 %. Su bajo promedio y desvío estándar en estar *caí* puede estar dado por situaciones en que está activo, un comportamiento pasa a estar activo por un período corto de tiempo(por ejemplo, *Evitar obstáculos*, que causa el 62 % de activaciones) hasta que desaparece el estímulo que lo activó y el robot vuelve a hacer *Wandering*. En la mayor parte de los casos entre una activación y otra de *Wandering* no hay muchos comportamientos que se activan.

Evitar obstáculos es un comportamiento que tiene ciertas cualidades. Una de ellas se puede ver en los cuadros de transiciones. Las transiciones hacia él y desde él se dan en proporciones muy parecidas, esto quiere decir que $t_{7j}^1 \approx t_{7j}^2$, siendo t^1 y t^2 transiciones de los cuadros 28 y 29 respectivamente.

Ésto da la siguiente idea: hay un comportamiento activo *A* y sus respuestas están llevando al robot a una posible colisión. En ese momento se activa *Evitar obstáculos*, detiene *A*, saca al robot de la posible colisión y vuelve a activar a *A*. Cabe aclarar que en realidad no lo activa a *A*, sino que desaparece el estímulo de *Evitar obstáculos* y *A* toma el control nuevamente. Su promedio y desvío estandar de *caa* son bajos y se ven reflejados en la figura 71. En la misma figura se puede observar el efecto de tener un promedio y desvío estandar bajos, reflejados en la duración muy corta de las mesetas. El poder de detener a otros comportamientos está dado por su nivel en la arquitectura. Es importante resaltar que este beneficio viene solo por el hecho de usar *Subsumption*, y en el caso de haber utilizado otra arquitectura, nos hubiese demandado trabajo.

El comportamiento de *Salir de situaciones no deseadas* no se activó nunca. Éste hecho es muy importante: indica que el resto de los comportamientos nunca se “activaron mutuamente” y por lo tanto no hubo situaciones que puedan llevar al robot a quedarse sin carga en la batería, arriesgando su autonomía. El hecho que no se active nunca el comportamiento puede llevar a pensar que no es necesario tenerlo. Éste pensamiento es totalmente erróneo ya que uno de nuestros objetivos era que la autonomía del robot no corra riesgo bajo ningún concepto. Cabe aclarar que hay situaciones que escapan al alcance de este proyecto, por ejemplo, que se rompa una rueda.

El funcionamiento de *Salir de situaciones no deseadas* lo probamos en diferentes simulaciones anteriores a la que presentamos y el resultado fue el esperado.

En la sección 3.7 vimos que algunas transiciones sucedían sólo una vez. A continuación detallamos posibles causas de las mismas:

1. *Wandering a Recolectar basura,*
2. *Enfocar basura a Descargar basura e*
3. *Ir a basura a Recargar batería*

El primer caso puede estar dado por el siguiente escenario:

- El robot esquivó un obstáculo.
- En el time step que desapareció ese estímulo, había una basura en la imagen detectada por la cámara.
- En dicho time step, el sistema de reconocimiento no la detectó, activando *Wandering*.
- En el siguiente time step, el sistema de reconocimiento detecta la basura, y además se encuentra enfocada y a una distancia en la cual puede activarse el comportamiento de *Recolectar basura*.

El segundo caso puede estar dado por el siguiente escenario:

- El robot recolectó una basura.
- En el time step que se desactivó la recolección, había otra basura en la imagen detectada por la cámara y por el sistema de reconocimiento que era necesario enfocar para que sea recolectada, pero la basura recolectada no había llegado a activar el sensor del depósito de basura. Estó llevó a la activación de *Enfocar basura*.
- En el siguiente time step, la basura recolectada activa el sensor del depósito y por ende, activa el comportamiento *Descargar basura*.

El tercer caso es más sencillo de explicar:

- En el time step t el robot se dirigía hacia una basura.
- En el time step $t + 1$, el valor de la batería hizo que se active el comportamiento de *Recargar batería*. Ésto es posible debido a que el último tiene mayor nivel en la arquitectura que el primero.

Otro dato curioso es que el 80 % de las veces que estuvo activo *Recolectar basura* luego se activó *Wandering* y el restante a *Enfocar basura*. Teniendo un almacenamiento máximo de 1 basura, es de esperarse que luego de recolectar, se active *Descargar basura*. Ésto no sucede ya que, como explicamos en el escenario del segundo caso, hay un período de time steps en los cuales la basura recolectada está yendo hacia el sensor del depósito, permitiendo la activación de otros comportamientos.

Finalmente, el robot recorrió toda la arena en el 42 % del tiempo total de simulación. Éste porcentaje disminuye o aumenta dependiendo de si aumenta el tiempo de simulación o disminuye. Es decir, lo importante no es el porcentaje del tiempo total, sino el tiempo en que recorrió el área en su totalidad. Dicho tiempo fue de $95241Ts = 50m47s$. Durante éste tiempo el robot no sólo recorrió un poco menos de $200m^2$ (la escala es 10:1), sino que recolectó 9 basuras de las 15 que había en la arena y fue a descargarlas (uno de los comportamientos que más tiempo lleva). Además, se recargó 3 veces en ese período, como dijimos anteriormente, es otro de los comportamientos que más tiempo demanda. Otro factor importante para destacar es que el 75 % del área se recorrió en un cuarto del tiempo en cuestión, es decir, en aproximadamente $13mins$. Por todo ésto, concluimos que el recorrido de la arena es muy eficiente.

3.8.1. Posibles extensiones

Una de las posibles extensiones del trabajo puede ser minimizar el tiempo consumido por los comportamientos de *Ir a zona de descarga de basura* y *Ir a zona de recarga de batería*. Dado que ambos van hacia la línea, podemos pensar en algunas formas para minimizar este tiempo. Entre las mismas están: cambiar las ubicaciones de las líneas, agregar más líneas o cambiar las formas de las mismas. Otra alternativa podría ser utilizar otro mecanismo para ir hacia dichas zonas que no use el seguimiento de línea, o lo use en menor tiempo. Cabe aclarar que no es el seguimiento de la línea lo que más tiempo demora sino ir hacia la misma.

En cuanto a los comportamientos que posee el robot podemos pensar en agregar algunos para ampliar las prestaciones que puede brindar el robot. Entre ellos se encuentran:

- Seguir la pared, con un objetivo a pensar.
- Interactuar con las personas, mediante sonidos o reconocimiento visual.
- Agregar un mecanismo que le permita recolectar basuras que están pegadas o en lugares no alcanzables por el robot por la presencia de obstáculos.
- Mover una basura hacia un área, en caso que no pueda recolectarla.
- Notificar a una central sobre sucesos inesperados o difíciles de que sucedan.
- Proveer una interfaz web que permita ver el estado de los sensores y actuadores del robot en forma on-line, así como también gráficos mostrando los últimos comportamientos activos. Además podría ver el estado actual del sistema de predicción y la imagen vista por la cámara.
- Subir estadísticas a un servidor central de forma tal que se pueda analizar qué cambios se podrían hacer en el controlador para mejorar la efectividad y eficiencia de la recolección o de otros comportamientos.

4. Visión

4.1. Introducción

Como es sabido, el análisis de imágenes es una tarea demandante en cuanto a complejidad y tiempo de procesamiento requerido ya que involucra, entre otras cosas, operar sobre grandes matrices. Este motivo, nos llevó a implementar un módulo de visión separado del resto del software del robot. En esta sección exhibimos el sistema de análisis de imágenes por el cual el robot en cuestión da cuenta de la existencia o ausencia de elementos residuales en el entorno descripto. El sistema se basa en la utilización de una cámara de tipo webcam con una única lente para obtener información del ambiente. Esta información es luego procesada por una computadora a bordo del robot utilizando un algoritmo de visión cuyo único objetivo es la detección de residuos en las imágenes obtenidas. Probamos este algoritmo diseñándolo para la detección de colillas de cigarrillo, vasos descartables y platos de plástico. Para el desarrollo del algoritmo utilizamos la librería de visión computacional OpenCV [3] acompañado de un sistema desarrollado en C++. Como plataforma utilizamos un sistema linux ubuntu corriendo en una netbook con procesador intel atom n450 y 1 gb de memoria RAM como se detalla en 2.5.1. Esta sección se divide en las siguientes partes: en la sub-sección 2 mostramos los trabajos previos estudiados, en la sub-sección 3 describimos el algoritmo de detección, en las secciones 4 y 5 explicamos los módulos de predicción y focalización respectivamente mientras que en la sub-sección 6 se muestran los resultados. En la sub-sección 7 concluimos la sección.

4.2. Trabajos previos

En esta sección mostramos la información obtenida a partir de trabajos anteriores relacionados con visión computacional, procesamiento de imágenes y robots autónomos.

4.2.1. Mobile Field Robot with Vision-Based Detection of Volunteer Potato Plants in a Corn Crop [22]

El objetivo de este trabajo es la construcción de un robot de tamaño reducido y de bajo costo capaz de detectar un tipo de planta de papas (volunteer potato) en un campo de trigo y proporcionar control automático de esta planta. El interés en esta especie se debe a que esta planta resulta difícil de controlar en las cosechas y favorece la proliferación de otras sustancias que pueden perjudicar otros cultivos. Con este propósito, se utilizó un camión de juguete como soporte para un robot autónomo móvil. Las partes de radio-control del mismo fueron removidas y remplazadas por micro-controladores para manejar la dirección y la velocidad del robot. Además se agregaron, como instrumentos de sensado, una webcam estándar montada sobre un mástil a una altura de 1.5m, un contador de revoluciones y un giroscopio de estado sólido. Las imágenes tomadas por la webcam se procesaban por una PC a bordo con un procesador Pentium M de 1.6 MHz. El control robótico de *volunteer potatoes* involucró al menos 3 grandes

etapas: navegación a través del cultivo, detección de plantas de papa y el control de las mismas.

En los campos de trigo la cosecha se dispone en filas dejando espacio entre hilera e hilera para el paso del robot. Para que este pueda navegar correctamente debía poder reconocer dichas hileras de modo de orientarse correctamente en el cultivo. Ésto se logró mediante un algoritmo de visión. Este comenzaba tomando una imagen RGB de 320x240 de la cámara. Para separar las plantas del fondo se tomó el doble del valor del canal verde (G) y se sustraen los canales rojos y azules (R y B respectivamente) (2G -R -B). Pixeles con un valor mayor a cierto umbral son considerados planta y todos los demás, fondo. En un segundo paso de procesamiento, se detectaron las filas de plantas utilizando un algoritmo inspirado en la transformada de Hough [2]. Este dibujaba líneas imaginarias sobre la imagen segmentada y establecía un puntaje para cada una según cuantos pixeles blancos (plantas) cubrían. Estos puntajes se convertían en valores de pixeles de una nueva imagen en donde la coordenada vertical de cada pixel correspondía con la pendiente de la línea imaginaria y donde la coordenada horizontal con la intersección con el eje de coordenadas. Las hileras de plantas contribuían pixeles a varias líneas imaginarias y aparecían en la nueva imagen como áreas con pixeles brillantes. Se utilizaron las operaciones de threshold y dilatación para combinar estas áreas brillantes en un única región contigua de la cual se extraía el centro de gravedad para representar una hilera de plantas. Ya que el ruido podía ocasionar el reconocimiento de hileras de plantas inexistentes se utilizaron algunas reglas sencillas para validar estas detecciones. Dos ejemplos de estas reglas son “las hileras de plantas deben estar a uno de los dos costados del robot” y “las hileras deben ser paralelas entre sí”. Luego, la orientación del robot se establecía según la pendiente de estas líneas.

Para la detección de las plantas *volunteer potato* se utilizó una segunda cámara ubicada a una menor altura para obtener mayor resolución sobre las plantas en sí. El algoritmo consistía en extraer las siguientes características de las imágenes:

1. promedio de rojo para pixeles de planta.
2. promedio de verde para pixeles de planta.
3. promedio de azul para pixeles de planta.
4. número total de pixeles de planta en la imagen binaria.
5. número total de pixeles de contorno en la imagen binaria.
6. número total de pixeles de borde en la imagen Canny binaria.

Estas características se relacionan al color (1,2 y 3), tamaño (4), figura (combinación de 4 y 5) y textura (6). Utilizando la misma fórmula (2G -B -R) y la operación de threshold se segmentó nuevamente la imagen en pixeles de planta o fondo. Sobre esta imagen se calculó el promedio de rojo, azul y verde para las características 1,2 y 3. Contando el número de pixeles planta (pixeles considerados como correspondientes a los de una planta) en la misma imagen se obtuvo la característica 4. Para la característica 5 se extraen los contornos de esta imagen y se cuenta el número de pixeles plantas contenidos en ellos. Finalmente, para la característica 6 se repite este último procedimiento extrayendo los bordes de la imagen pero, utilizando una implementación del algoritmo de Canny

[6]. Se esperaba que el número de pixeles de borde sea mayor si se trata de una planta de papa ya que esta contiene un número mayor de hojas pequeñas que se solapan entre sí mientras que las de trigo sólo contienen algunas pocas hojas. Para clasificar las imágenes se usó un conjunto de imágenes de entrenamiento que fueron clasificadas manualmente según si contenían plantas de papa o no. Luego se entrenó un clasificador de discriminante lineal de Fisher [10] para distinguir entre imágenes que contienen plantas de papa de las que no.

Para poner a prueba el algoritmo se tomaron dos conjuntos de imágenes capturando el recorrido del mismo robot autónomo a través de un campo de trigo. La tasa global de error para el clasificador fue de 1.5 % en el primer conjunto y 10.6 % en el segundo. En cuanto al rendimiento en la navegación, el robot fue capaz de recorrer con éxito 120m de hileras de cultivo con una velocidad promedio de 0.67 m/s. El autor remarcó el alto tiempo de procesamiento requerido por el detector de plantas de papa ya que este bajó la tasa de cuadros del algoritmo de navegación de 30 a 8 cuadros/segundo. Por dicho motivo se redujo la velocidad de desplazamiento del robot a 0.2 m/s para que fuera capaz de capturar correctamente todas las plantas. El mecanismo de control no fue implementado pero se incluyó un speaker que reprodujo un sonido cuando éste encontraba una planta del tipo buscado. Los algoritmos de visión utilizados fueron desarrollados usando Matlab con la librería Delft [11].

4.2.2. Review of shape representation and description techniques [24]

El objetivo de este trabajo es realizar un resumen de las técnicas más importantes en cuanto a la representación y descripción de figuras o formas. Al clasificar dichas técnicas el autor sugiere una primera sub-división en 2 grupos: métodos basados en contornos y métodos basados en regiones. Los primeros se caracterizan por extraer la información solo del contorno mientras que los segundos, de la región completa abarcada por la figura. Dentro de cada clase se dividen en técnicas estructurales o técnicas globales de acuerdo a si la figura se representa como un todo o por segmentos y secciones respectivamente. Exhibimos la jerarquía completa en la figura 74.

Dentro de la categoría de los basados en contornos, en la sub-categoría de globales, los métodos suelen calcular vectores de propiedades numéricas extraídas de la información del borde de una figura. La comparación entre formas se realiza generalmente utilizando métricas de distancia, como la distancia euclídea. Ejemplos de métodos sencillos que caen dentro de esta categoría son *area*, *circularidad*(*perímetro*² / área), *excentricidad* (*longitud del eje mayor* / *longitud del eje menor*). Otros métodos de la misma familia pero más complejos son los métodos basados en la correspondencia de puntos. En estos métodos se trata a cada punto como una característica de la figura que compone. La distancia de *Hausdorff* es una técnica clásica de esta categoría. Dadas dos formas representadas por dos conjuntos de puntos $A = x_1, x_2, \dots, x_n$ y $B = y_1, y_2, \dots, y_n$ la distancia Haussdorff se define como

$$d_H(A, B) = \max\left\{ \sup_{x \in A} \inf_{y \in B} d(x, y), \sup_{y \in B} \inf_{x \in A} d(x, y) \right\},$$

La comparación de figuras usando distancia Hausdorff resulta sensible al ruido y a pequeñas alteraciones en dos formas iguales. Existen otros métodos

basados en *firma de contorno* que representan una figura con una función unidimensional derivada de la información del borde de la misma, algunos son *perfil centroidal, coordenadas complejas y distancia al centroide*. Estos métodos deben ser normalizados para compensar traslaciones o escalas lo que los hace relativamente costosos. Ésto, sumado a la sensibilidad al ruido y a pequeños cambios en los bordes, hace que el método no sea lo suficientemente robusto y que necesite de procesamiento adicional. Otra familia de descriptores se basa en los *Momentos de borde*. Dada una parametrización como una función unidimensional $z(i)$, se define el momento r -ésimo m_r y el momento central μ_r como:

$$m_r = \frac{1}{N} \sum_{i=1}^N [z(i)]^r \quad \mu_r = \frac{1}{N} \sum_{i=1}^N [z(i) - m_1]^r$$

donde N es el número de puntos de borde. El momento normalizado $\bar{m}_r = m_r / (\mu_2)^{r/2}$ es invariante frente a traslaciones, rotaciones y escalamientos. Sin embargo, los momentos de alto orden son difíciles de asociar con interpretaciones físicas del contorno.

La otra sub-categoría dentro de los métodos basados en contornos es la de los métodos estructurales. En estos métodos se suele descomponer un contorno en segmentos de bordes llamado primitivas. Los métodos estructurales entonces se diferencian entre sí de acuerdo a la definición y organización de estas primitivas. Algunos métodos de descomposición son: aproximación por polígonos, descomposición por curvatura, y ajuste de curva. El resultado de la descomposición se codifica como una cadena $S = \{s_1, s_2, \dots, s_n\}$ donde s_i puede ser un lado de un polígono, una arista cuadrática, una función spline, etc. Además, s_i puede contener varios atributos como longitud, orientación o curvatura promedio, entre otros. Un método en esta línea es por ejemplo el de código en cadena, en el cual se describe un objeto por una serie de segmentos de longitud unitaria y una orientación determinada. En su implementación se superpone una imagen con una grilla y se aproximan los puntos del contorno al punto de la grilla más cercano. Luego se selecciona un punto de partida y se codifica una secuencia, estableciendo la dirección del segmento que conecta un punto con el siguiente según la dirección de éste en la grilla (respecto del punto anterior). Para la comparación de dos contornos es necesario realizar normalizaciones ya que el punto de partida puede variar y frente a un cambio de escala pueden parecer contornos distintos. Otros métodos en esta sub-categoría pueden ser el de la descomposición polinómica o la descomposición en curvas suaves.

Dentro de la categoría de los métodos basados en la región de una figura y la sub-categoría de globales podemos encontrar los *momentos Hu*. Un momento de Hu de un contorno 2D se define como:

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y)$$

Usando combinaciones lineales de momentos de Hu de órdenes bajos (valores de p y q menores a 2) se obtiene un descriptor invariante frente a rotaciones, traslaciones o escalamientos. Sin embargo esta técnica probó servir sólo para figuras simples ya que en formas más complejas no tiene un buena rendimiento. Otros métodos en esta sub-categoría pueden ser *momentos algebraicos* o *descriptor general de Fourier*. En la otra sub-categoría de métodos estructurales se destaca la *envoltura convexa*. La *envoltura convexa* de un set de puntos es

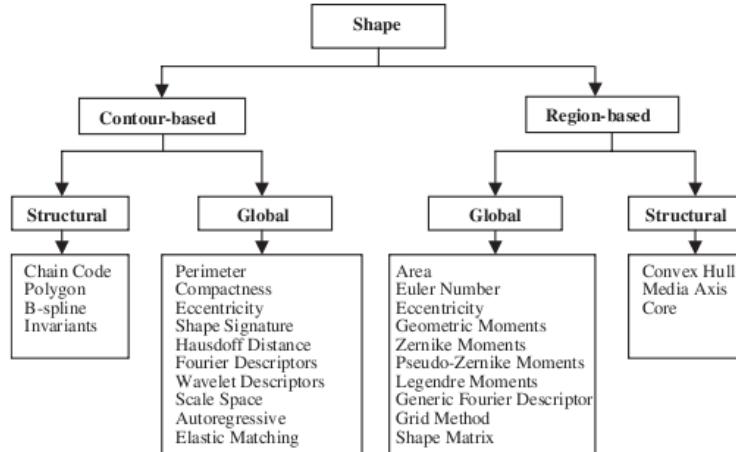


Figura 74: Clasificación de las técnicas de descripción y representación de figuras.[24]

la mínima región que contiene dichos puntos y es convexa. Este método obtiene dicha región y la compara con el contorno original obteniendo una serie de pequeñas componentes que marcan las deficiencias de concavidad del contorno. La figura en cuestión puede entonces representarse utilizando estas componentes. Finalmente la comparación entre dos contornos representados con esta técnica resulta en la comparación de grafos (caracterizando las deficiencias mencionadas) lo cual es otro problema a resolver por sí mismo. Además, algunos métodos en esta familia son difíciles y complejos de implementar.

Como conclusión, el autor establece que los métodos basados en contorno son más populares ya que están más relacionados a la percepción humana y que para muchas aplicaciones es sólo necesario la información sobre el contorno y no sobre el interior del mismo. Sin embargo, también establece que como solo usan parte de una figura, son más sensibles al ruido y pequeñas variaciones.

4.2.3. Sidewalk Following Using Color Histograms [19]

Este trabajo describe un algoritmo para detectar y seguir veredas o áreas transitables en un video filmado, con el objetivo de lograr la navegación autónoma de un robot dentro del campus de una universidad. La plataforma en la cual se tomaron los videos estaba compuesta de una silla de ruedas motorizada controlada por una CPU a bordo corriendo un sistema operativo Linux. El robot tenía una única cámara NTSC que proveía imágenes color con una tasa de 15 cuadros por segundo. Todos los videos de prueba fueron tomados utilizando esta plataforma conduciéndola remotamente por un operador a través de un sistema de radio.

El algoritmo en cuestión utilizaba histogramas 2D de los canales de tono y saturación de los pixeles para modelar las áreas transitables y las no transitables. Se utilizaba un área de entrenamiento ubicada directamente en frente del robot en la cual se asumía que no habían obstáculos y contenía pixeles representativos

del área que estaba transitando. Esta área de entrenamiento se utilizaba para actualizar uno de los 4 histogramas que modelaban el área transitable. Además, se mantenía un histograma que modelaba el área no transitable o fondo de la imagen, tomando una muestra de pixeles del último cuadro que fueron clasificados como tales. La clasificación de pixeles entonces se realizaba utilizando la siguiente métrica:

$$P_{transitable} = T(h.s)/(T(h.s) + NT(h.s))$$

donde T y NT son los histogramas que modelan las áreas transitables y no transitables respectivamente y donde $T(h.s)$ y $NT(h.s)$ representan la frecuencia relativa de aparición de un pixel con valor h de tono y s de saturación en dichos histogramas. $P_{transitable}$ se calculaba para los 4 histogramas tomando el máximo de éstos. Si este último (el máximo) superaba cierto umbral entonces se consideraba a dicho pixel como correspondiente a un área transitable. La actualización de los histogramas se realizaba reemplazando el histograma de menor cantidad de aciertos con el histograma del área de entrenamiento. Entre los resultados se destacaba el hecho de que 4 histogramas resultaron suficientes para modelar la variación de color de las áreas transitables pero, sin embargo, no se lograron clasificar a las zonas cubiertas de sombra como transitables. Bajo condiciones de luz favorables, 2 histogramas fueron suficientes para clasificar, en promedio, el 50% del área transitable. Las posibles mejoras incluyen la incorporación de bordes o texturas para el reconocimiento de áreas transitables como así la posibilidad de tener áreas de entrenamiento con posiciones dinámicas de acuerdo a las condiciones de iluminación.

4.2.4. Skin Detection using HSV color space [21]

En este artículo se trabajó sobre la detección de piel en imágenes utilizando el espacio de color HSV. El algoritmo propuesto por el autor consiste en el filtrado de los pixeles en base al valor del canal de tonalidad (canal H). Los rangos de valores utilizados para representar el color de la piel fueron obtenidos de otros artículos relacionados. Con el objetivo de eliminar ruido, se aplican filtros morfológicos y de suavizado, estos son (en este orden) dilatación, erosión y un filtro de mediana. Para los filtros morfológicos se utilizaron núcleos de 5x5 pixeles mientras que para el filtro de mediana se usaron núcleos de 3x3 pixeles. Finalmente, el autor midió la performance de su algoritmo utilizando imágenes de prueba en donde se aprecian distintas zonas con piel aisladas. A partir de estas imágenes, se obtuvieron las coordenadas de los pixeles que se corresponden con piel y se las contrastó con aquellas que identificó el algoritmo. Los resultados presentados para 4 imágenes son satisfactorios obteniendo en promedio un porcentaje de falsos negativos cercano al 1% y de falsos positivos menor al 6%.

4.2.5. Line Detection and Lane Following for an Autonomous Mobile Robot [1]

En este artículo se describió el software utilizado para uno de los robots ganadores de la competencia “Intelligent Ground Vehicle Competition” (IGVC) en donde se requiere que los robots naveguen a través de un camino delimitado por líneas pintadas sobre césped en un ambiente al aire libre en donde se pueden

presentar obstáculos. El robot utilizaba una única cámara para el sistema de visión y su objetivo era detectar las líneas pintadas para determinar la dirección de movimiento del robot. El algoritmo descripto por el autor cuenta con una etapa de pre-procesamiento y otra de detección de líneas. En la etapa de pre-procesamiento se comienza por convertir la imagen a escala de grises utilizando una transformación en la que la intensidad de cada pixel es determinada usando $2 * B(i, j) - G(i, j)$ ²⁰ con el argumento que el canal verde contiene ruido provocado por la exposición de luz en el pasto y de esta manera se lo elimina. Luego el autor propone utilizar un ajuste de brillo argumentando que los píxeles de la zona más alta se encuentran más expuestos a la luz por la perspectiva de la cámara, para esto sustrae una máscara que compensa este efecto. Además, el autor elimina la proyección del robot mismo sobre la imagen capturada por la cámara. En la etapa de detección se realizó un threshold para obtener los píxeles más brillantes. Luego, la salida del threshold se utilizó como entrada para un detector de líneas basado en el algoritmo de Hough [2]. Finalmente, según la orientación de las líneas detectadas se determinaba la dirección del robot. El algoritmo se consideró exitoso ya que el robot propuesto fue el ganador de la competencia.

4.3. Algoritmo de detección

En esta sección describimos las distintas etapas del algoritmo de análisis de imágenes utilizado. Éstas se pueden separar en una etapa de pre-procesamiento donde realizamos un tratamiento de la imagen obtenida para resaltar las características de interés y una segunda etapa donde extraemos estas características para interpretarlas y buscar los objetos residuales en las imágenes.

4.3.1. Estructura general

La figura 75 ilustra las etapas del algoritmo. Programamos a éste utilizando el lenguaje C++ y la librería de visión computacional OpenCV. El algoritmo comienza por tomar una imagen nueva de la cámara, las imágenes capturadas poseen el formato 24-bit RGB. Con el objetivo de minimizar el impacto producido por cambios en la iluminación convertimos la imagen al formato HSV (tono, saturación y brillo). Luego utilizamos el canal del tono de esta nueva imagen para establecer qué píxeles se corresponden con el color buscado, filtrando aquellos píxeles que no coinciden con el rango deseado (ver inciso 4.3.2). A su vez, usamos la información brindada por los canales de saturación y/o brillo para obtener más precisión sobre el color de los píxeles.

Descartamos los píxeles filtrados en el canal de saturación realizando una operación lógica AND bit a bit con la imagen filtrada. En la siguiente etapa, utilizamos filtros morfológicos cuyo objetivo es la eliminación de ruido. El resultado de aplicar estos filtros resulta en la expansión de las áreas donde hay mucha presencia de píxeles de interés, produciendo un área de intensidad homogénea mientras que en las áreas con poca presencia de estos píxeles elimina la aparición aislada de los mismos, generados por el ruido (ver inciso 4.3.4). El paso siguiente consiste en la aplicación de un umbral. El umbral filtra aquellos píxeles que no se encuentren por encima de un valor mínimo de intensidad marcándolos con valor cero y conserva únicamente los píxeles que sí lo hacen

²⁰B(i,j) se refiere al pixel de la coordenada (i,j) del canal Azul de la imagen

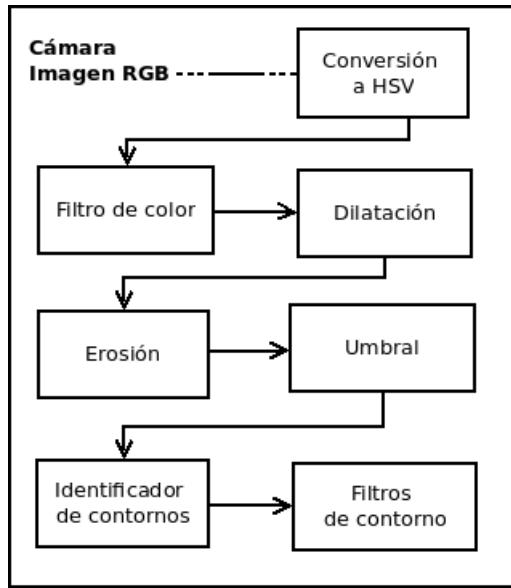


Figura 75: Etapas del algoritmo de visión

estableciéndoles un valor predeterminado (mayor a cero) (ver inciso 4.3.3).

Las operaciones que mencionamos hasta aquí componen la etapa de preprocesamiento de la imagen. Finalizada dicha etapa del algoritmo deseamos tener sólo los pixeles de los potenciales objetos de interés. Vale notar que, como consecuencia de aplicar un filtro umbral, la imagen se encuentra binarizada, es decir, los pixeles sólo pueden tener dos posibles valores: cero (negro) si no es un pixel de interés o mayor a cero en el caso contrario. Es entonces que automáticamente quedan delimitadas las áreas que contienen pixeles de interés. El próximo paso del algoritmo se encarga de obtener los contornos de estas áreas. Para esto utilizamos el algoritmo de Suzuki [20] que recorre los bordes de estas zonas y crea secuencias de puntos (x, y) que definen el contorno de las mismas. Las secuencias de puntos son luego aproximadas para generar polígonos cerrados mediante el algoritmo de Douglas-Pecker [7]. Tomando estos polígonos definimos distintos parámetros tales como el área, perímetro o figura que nos permiten discernir si se trata de un objeto a reconocer o no. Luego, informamos la posición de un objeto a partir de su rectángulo contenedor mínimo.

4.3.2. Detección de color

El problema de decidir si un pixel determinado se corresponde con un color particular resulta mucho más sencillo de resolver en el espacio de color HSV que en el RGB. Este espacio de color se compone de un canal H que describe el color de un pixel, un canal S que pondera la saturación de dicho color y un canal V que mide la luminosidad de dicho pixel. En el espacio RGB, de haber un cambio en el brillo o iluminación de la imagen, los tres canales se ven afectados de igual manera mientras que en el espacio HSV el canal H, que codifica la información sobre el tono del color, se ve mucho menos influenciado

```

V=max(R, G, B)
S=(V-min(R,G,B))*255/V    if V!=0, 0 otherwise

(G - B)*60/S,   if V=R
H= 180+(B - R)*60/S,   if V=G
240+(R - G)*60/S,   if V=B

if H<0 then H=H+360

```

Figura 76: pseudo-código de la conversión RGB-HSV

en comparación con los otros dos canales (saturación y valor). De esta manera, podemos caracterizar el color de un pixel determinado observando su valor en el canal H, sin molestarlos demasiado por las condiciones de brillo o iluminación al cual se encuentra expuesto.

Realizamos la detección de color verificando los valores correspondientes al canal H (tono) de la representación HSV de la imagen. Para esto convertimos de la imagen obtenida por la cámara de formato RGB a HSV usando el algoritmo expuesto en la figura 76.

Obteniendo el canal H, nos fijamos que los valores de los pixeles estén dentro de cierto rango correspondiente al color que buscamos. Como podemos observar en la figura 77, en el espacio de color HSV los colores se encuentran dispuestos a lo largo de una circunferencia, donde cada tonalidad representa un ángulo en la misma. Por ejemplo, si queremos abarcar las distintas tonalidades del azul podemos elegir el rango que va de 200° a 260° . Cuando buscamos un color en particular es preciso elegir este rango cuidadosamente ya que de ser un rango muy restrictivo podemos despreciar pixeles de interés arruinando la figura del contorno del objeto a buscar y si elegimos un rango más abarcadero podemos tomar pixeles de colores distintos al buscado, introduciendo ruido. Experimentalmente comprobamos los resultados de un filtro de color con distintos rangos, como se muestra en la figura 78. Una vez obtenidos los pixeles que se corresponden con el color buscado se combinan estos con los del canal de saturación utilizando un and bit a bit entre ambos pixeles. A mayor saturación consideraremos mayor presencia del color buscado en dicho pixel y por lo tanto mayor probabilidad de que éste corresponda al objeto buscado.

En el caso de las colillas se utilizó el rango para el canal de tono $20 < h < 40$ con valor en el canal de saturación mayor a 60. En el caso de los platos y los vasos se buscó detectar el color blanco. La particularidad de este color es que este se encuentra presente en todos los colores del rango H y se lo caracteriza por su valor alto en el canal V. Por este motivo, para platos y vasos sólo se verifica que se cumpla que el valor del pixel tenga $v > 170$.

4.3.3. Threshold

Threshold o umbral es una operación que nos permite pasar de una imagen en escala de grises a una imagen binaria. El proceso de thresholding consiste en distinguir los pixeles que se encuentran por encima de un cierto valor de los que están por debajo del mismo. Con este objetivo, se crea una imagen binaria asignándoles valores 1 o 0 según su condición respecto del valor. El algoritmo

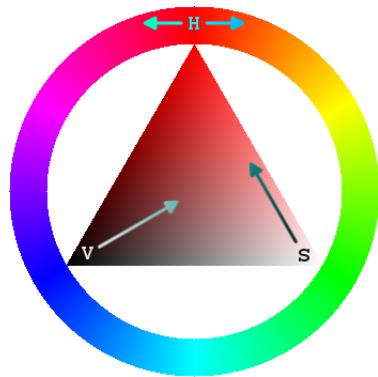


Figura 77: Espacio de color HSV. Los distintos tonos de colores se encuentran dispuestos a lo largo de la circunferencia.

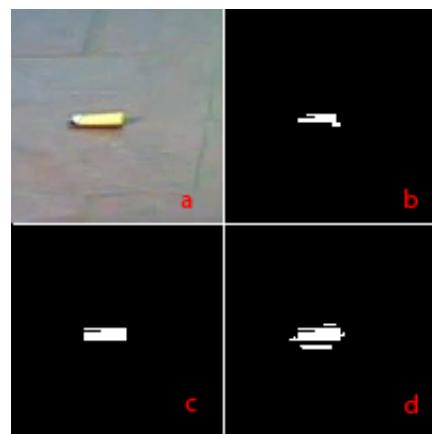


Figura 78: Resultados de aplicar un filtro de color utilizando distintos rangos. (a) Captura original. (b) Resultado de aplicar el filtro $35 \leq h \leq 45$, se pierden algunos pixeles. (c) Resultado de aplicar el filtro $20 \leq h \leq 40$. (d) Resultado de aplicar el filtro $10 \leq h \leq 50$, se introducen pixeles extra.

de thresholding nos puede servir para distinguir un objeto determinado de un contexto o fondo siempre y cuando el objeto posea un mayor brillo que el fondo en el que se encuentra. Esto significa que los pixeles correspondientes a un objeto se encontrarán por encima de cierto valor característico, determinado por el fondo. En nuestro caso ésto no sucede de ésta forma ya que no siempre la intensidad o brillo de los objetos supera al fondo en el que se encuentra, por lo cual resulta difícil utilizar esta técnica. Podemos apreciar un ejemplo de este efecto en la figura 79.

Existen diversas variantes de threshold, siendo los parámetros utilizados Val para indicar el valor umbral y M para indicar el valor a tomar por los pixeles.

- Threshold binario: Si un pixel se encuentra por encima de Val , se le asigna un valor M , de otro modo se le asigna 0.
- Threshold binario invertido: Si un pixel se encuentra por encima de Val , se le asigna 0, de otro modo se le asigna M .
- Threshold truncado: Si un pixel se encuentra por encima de Val , se le asigna V , de otro modo conserva su valor.
- Threshold a cero invertido : Si un pixel se encuentra por encima de Val se le asigna 0, de otro modo, conserva su valor.
- Threshold a cero invertido : Si un pixel se encuentra por encima de Val conserva su valor, de otro modo se le asigna 0.

El problema con esta familia de algoritmos es que en todos los casos el valor Val permanece constante para toda la imagen haciéndolo propenso a errores cuando la imagen presenta diferentes niveles de iluminación. Una manera de solucionar esto es pre-calculando el valor Val para distintas zonas de la imagen como lo hace la técnica de thresholding adaptativo, que calcula el valor de Val a medida que recorre la imagen, usando para esto, una ventana cuyo tamaño es definido por el usuario. Siguiendo esta lógica, se utilizan los valores de los pixeles abarcado por esta ventana para definir un valor de Val (por ejemplo calculando el promedio de intensidad) y luego se aplica alguna de las técnicas de thresholding simples mencionadas, utilizando este mismo valor. Sin embargo, este algoritmo es más costoso y puede provocar efectos indeseados como se observa en la figura 80 en donde el valor de Val se calcula utilizando un promedio ponderado por una función gausseana de acuerdo a la distancia del centro de la ventana.

En nuestro caso, utilizamos thresholding no para distinguir los objetos del fondo (esto es trabajo del filtro de color), sino como método de eliminación de ruido. Cuando se combina el canal de saturación con el resultado del filtro de color, se obtiene una imagen de escala de grises que está segmentada en un conjunto de áreas con distintos valores de intensidad. Luego de aplicar dilatación y erosión, los niveles de intensidad de cada área se vuelven homogéneos (ya que se toma el máximo o mínimo local) y es entonces donde podemos diferenciar las zonas de intensidad alta de las zonas de intensidad baja, motivo por el cual utilizamos la operación de threshold. En las figuras 81 y 82 exhibimos dos casos de ejemplo de threshold. Para la detección de colillas, vasos y platos establecemos el umbral de la operación de threshold en 100, asignándoles un valor de 255 (valor máximo) a los pixeles que superen dicho umbral o 0 en caso contrario. Podemos apreciar un ejemplo de threshold en la figura 82.

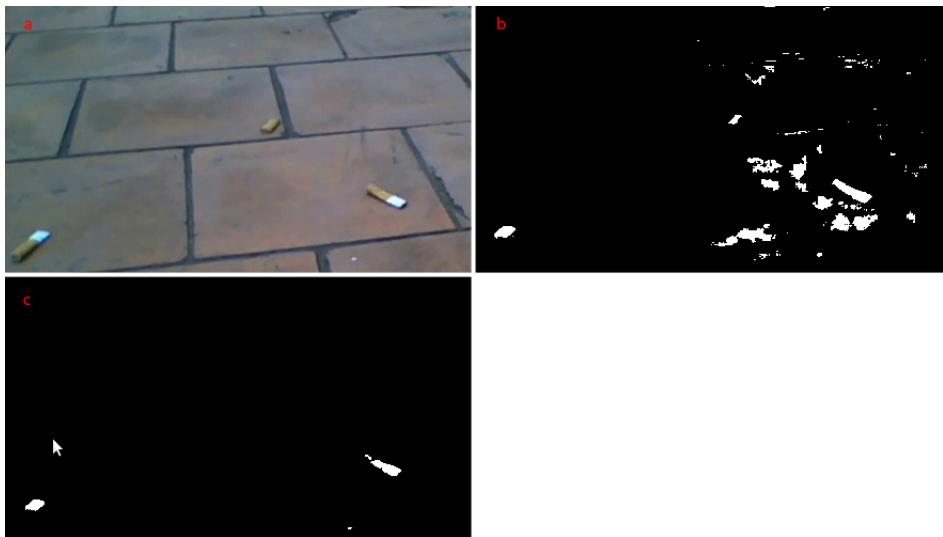


Figura 79: Distintos valores de umbral. (a) Captura original. (b) Threshold binario con umbral 120. Se observan pixeles extra en la detección. (c) Threshold binario con umbral 160. Se observa la ausencia de pixeles.

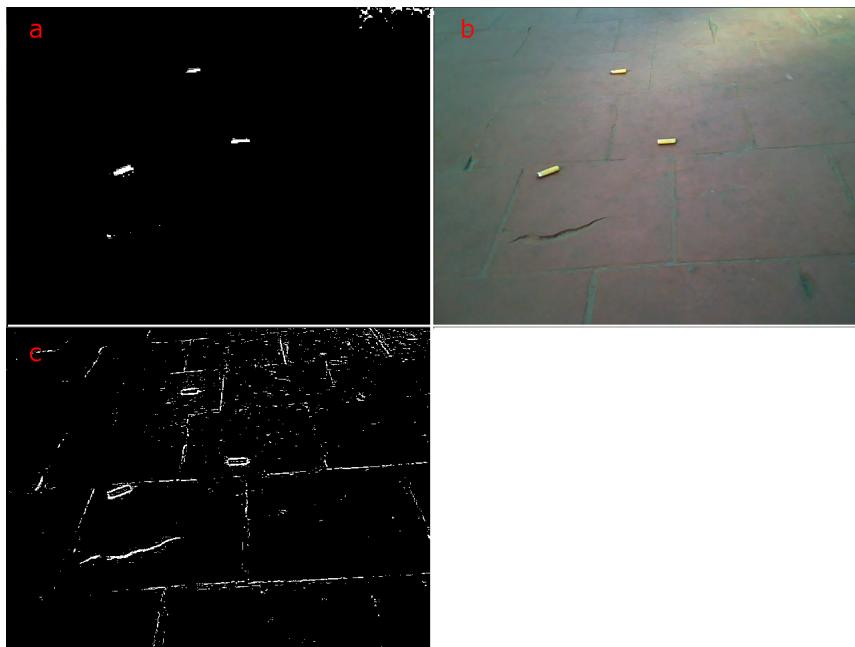


Figura 80: Comparación de threshold adaptativo con el filtro de color. (b) Imagen obtenida de la cámara. (a) Salida del filtro de color. (c) Threshold adaptativo sobre la imagen b previa conversión a blanco y negro utilizando una ventana de 9x9 pixeles. Se observa que el filtro adaptativo identifica correctamente los contornos de las colillas pero agrega mucha información redundante que resultara en procesamiento extra.

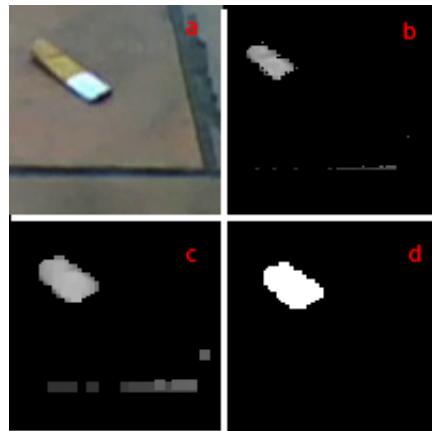


Figura 81: Resultados de aplicar thresholding binario. (a) Captura original. (b) Salida del detector de color, se observan algunos pixeles extra cerca de la línea de brea. (c) Salida de la operación de dilatación-erosión. (d) Salida de la operación de thresholding binario con un umbral de 100. La baja intensidad de los pixeles extra en la línea de brea no supera el umbral propuesto y por lo tanto son descartados. El contorno del objeto de interés se conserva aislado.

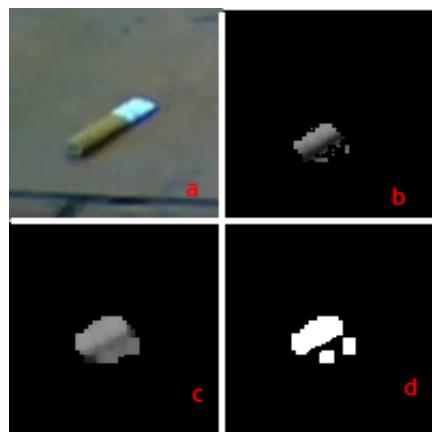


Figura 82: Resultados de aplicar thresholding. (a) Captura original. (b) Salida del detector de color, se observan algunos pixeles extra debajo del objeto de interés. (c) Salida de la operación de dilatación-erosión, los pixeles extra se combinaron con los pixeles del objeto.(d) Salida de la operación de thresholding binario con un umbral de 100. La baja intensidad de los pixeles extra en la zona debajo del objeto no supera el umbral propuesto y por lo tanto son descartados. El contorno del objeto de interés se conserva aislado.

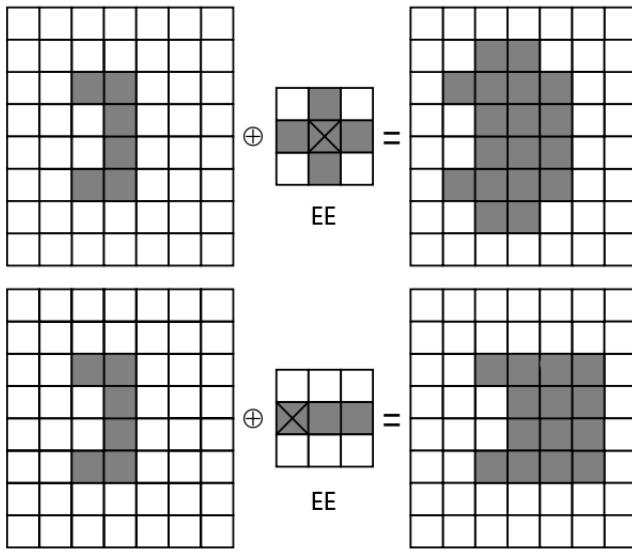


Figura 83: Operación de dilatación en imágenes binarias. Se reemplazan los pixeles no vacíos por una copia del elemento estructural con punto principal en esa posición. La cruz indica la posición del punto principal dentro del elemento estructural (EE). Imagen obtenida del curso de visión artificial de la Universidad Politécnica de Madrid.

4.3.4. Operaciones morfológicas

Sombras, luces y otros efectos pueden alterar el resultado del filtro de color introduciendo ruido en los objetos a detectar. Este ruido puede surgir tanto de la omisión de pixeles de interés como inclusión de pixeles extra. Para subsanar esto, utilizamos las operaciones morfológicas de dilatación y erosión. Ambas se basan en la utilización de un elemento estructural, esto es, una figura de cualquier tamaño y forma que tiene definido un punto principal y que recorre la imagen solapándose pixel a pixel. De acuerdo a operaciones locales a este elemento, el pixel que coincide con el punto principal se ve modificado. Usualmente se utilizan como elementos estructurales, pequeños discos o cuadrados, donde el punto principal se encuentra en el centro del mismo. El efecto generado se entiende mejor observándolo en imágenes binarias. En el caso de la dilatación, la idea intuitiva es remplazar cada pixel no vacío con una copia del elemento estructural cuyo punto principal se encuentra en esa posición. Para la erosión, la idea intuitiva es quedarse con aquellos pixeles tal que podamos hacer caber un elemento estructural cuyo punto principal se encuentra en esa posición y dicho elemento solo cubra pixeles no vacíos. Ilustramos esto en las figuras 83 y 84.

4.3.4.1. Dilatación

En imágenes no binarias, la operación de dilatación se define tomando el máximo local bajo el elemento estructural y asignándole ese valor al punto principal. El efecto producido es un aumento general en el brillo de la imagen y un probable aumento el tamaño de las figuras [23]. Cuando un área grande

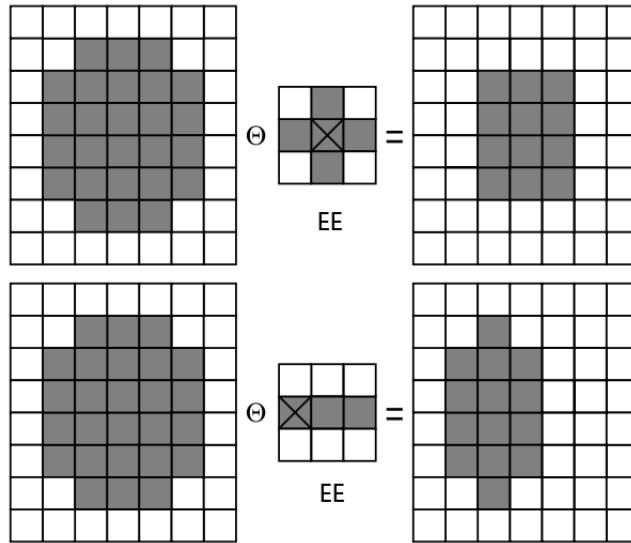


Figura 84: Operación de erosión en imágenes binarias. Se persisten los pixeles no vacíos en donde cabe una copia del elemento estructural cuyo punto principal se encuentra en esa posición. La cruz indica la posición del punto principal dentro del elemento estructural (ee). Imagen obtenida del curso de visión artificial de la Universidad Politécnica de Madrid.

aparece, a causa del ruido, partida en varias componentes, el uso de la operación de dilatación provoca que se combinen nuevamente en una sola. Un ejemplo de esto puede ser apreciado en la figura 85. Otro beneficio de utilizar dilatación es la eliminación del ruido espurio como se aprecia en la figura 86.

4.3.4.2. Erosión

En imágenes no binarias, la operación de erosión se define tomando el mínimo local bajo el elemento estructural y poniéndole ese valor al punto principal. Contrario a la dilatación, este operador generalmente reduce el brillo de la imagen y disminuye el tamaño de las figuras [23]. Usamos la operación de erosión para eliminar las protuberancias que pueden surgir de aplicar la operación de dilatación. En la figura 87 podemos apreciar el efecto de aplicar dilatación y luego erosión.

4.3.5. Detección de contornos

La detección de contornos se realiza sobre las imágenes binarias producidas en la etapa de pre-procesamiento. Dichos contornos quedan definidos implícitamente como la separación de regiones positivas de las negativas²¹. Por lo tanto para obtener estas regiones hacemos uso de los algoritmos explicados en esta sub-sección.

²¹ Como se trata de imágenes binarias, llamamos a una región positiva a las que contienen pixeles con valores mayores a 0 y a las negativas a las que tienen un valor de 0.

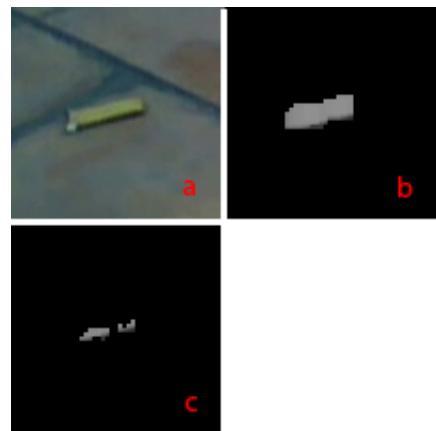


Figura 85: Efecto de la operación de dilatación. (a) Imagen original tomada de la cámara. (b) Imagen luego de aplicar dilatación a la imagen c con un elemento estructural cuadrado de 3x3 pixeles, con punto principal en el centro. (c) Salida del filtro de color. La colilla de cigarrillo esta 'partida' en dos componentes conexas.

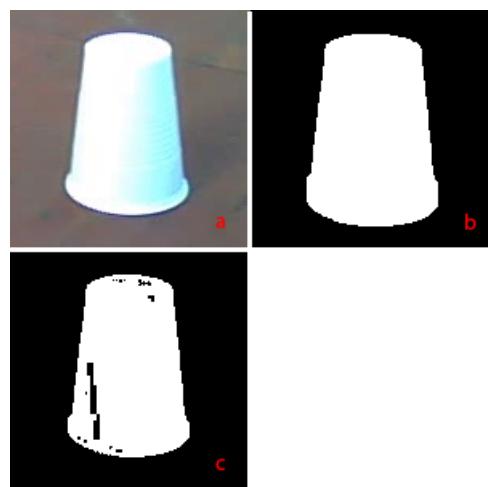


Figura 86: Eliminación de ruido mediante la operación de dilatación. (a) Imagen original tomada de la cámara. (b) Imagen luego de aplicar dilatación a la imagen c con un elemento estructural cuadrado de 3x3 pixeles, con punto principal en el centro. (c) Salida del filtro de color. Observamos áreas oscuras en el interior del vaso.

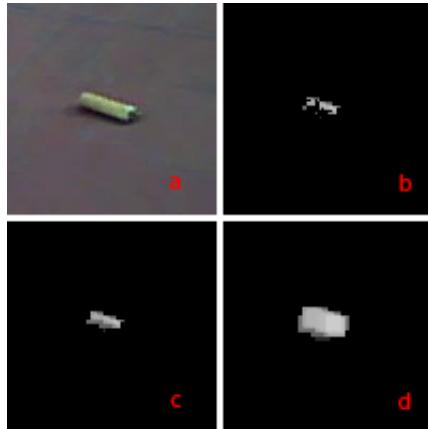


Figura 87: Ejemplo de erosión. (a) Captura original. (b) Salida del filtro de color. Se observan fallas en la detección del objeto. (d) Se aplica la operación de dilatación a la imagen b. Se observa la producción de protuberancias en la figura del objeto. (c) Salida luego de aplicar erosión a la imagen d. La figura del objeto se corrige eliminando parte de las protuberancias. Sin embargo, no se observan fallas en la detección del objeto.

4.3.5.1. Algoritmo

Recordemos que las imágenes binarias se caracterizan por tener sólo dos tipos de píxeles, los que tienen un valor de 0 y los que tienen un valor mayor a 0. Denominamos a estos píxeles como 0-píxeles o 1-píxeles respectivamente. Decimos que un conjunto de 0-píxeles o 1-píxeles *conectados* forman una componente. Entendemos dos tipos de conectividad: la 4-conectividad y la 8-conectividad. Dos píxeles con coordenadas (x', y') y (x'', y'') son 4-conexos si y sólo si $|x' - x''| + |y' - y''| = 1$ y 8-conexos si y sólo si $\max(|x' - x''|, |y' - y''|) = 1$, la figura 88 ilustra estas relaciones. Usando este concepto de conectividad se puede segmentar una imagen binaria en varias componentes 4-conexas u 8-conexas formadas por 0-píxeles o 1-píxeles. Cada una de estas componentes se compone de píxeles de valores iguales y, para cualquier par de píxeles pertenecientes a la componente, existe un 4-camino u 8-camino²² que los conecta. Ya que las 1-componentes (componentes que contienen 1-píxeles) son complementarias con las 0-componentes podemos sólo considerar las primeras y asumir que todo lo demás corresponde a 0-pixels conformando el fondo de la imagen. Cada 1-componente posee un único borde exterior que lo separa de la 0-componente que lo rodea y cero o más bordes interiores que lo separan de las 0-componentes que rodea (agujeros). Un ejemplo de esto puede apreciarse en la figura 89. El algoritmo que utilizamos no tiene en cuenta agujeros interiores de un contorno. Definimos entonces un punto de borde a un elemento de una 1-componente que es 4-conexo con una 0-componente y al *borde* o *contorno* de una figura como un conjunto de puntos de borde conectados.

El objetivo de este algoritmo es entonces recolectar todos los puntos de la imagen caracterizados como puntos de borde. Para hacer esto, recorre la ima-

²² Un 4-camino u 8-camino, es una secuencia de píxeles tal que cada pixel es 4-conexo u 8-conexo con el pixel siguiente.

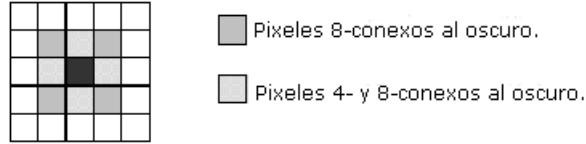


Figura 88: Tipos de conectividad entre píxeles de una imagen binaria

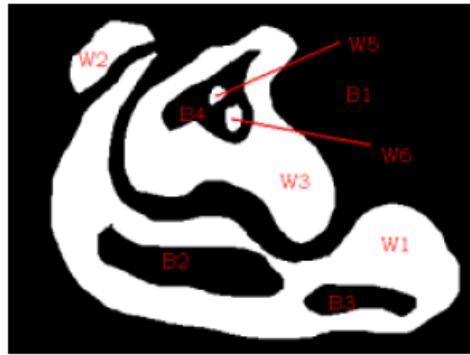


Figura 89: Ejemplo de segmentación en imágenes binarias. Las 1-componentes w1,w2 y w3 se encuentran rodeadas por la componente de fondo b1 y a su vez rodean a 0-componentes b2, b3 y b4

gen línea por línea buscando nuevos puntos de borde, cuando encuentra uno nuevo, activa un proceso de seguimiento de borde almacenando las coordenadas de cada punto que lo compone. Durante este proceso va marcando los píxeles ya visitados asignándoles a estos un valor especial para no volver a incidir en ellos. Finalmente se disponen todos los contornos encontrados en una lista. Este algoritmo de reconocimiento de contornos está basado en la técnica expuesta por Suzuki y Abe (para mayor detalle referimos al autor a [20]).

4.3.5.2. Representación

Como se expresa en el trabajo de Zhang y Lu [24], existen diversas formas de representar los contornos, cada una con distintas aplicaciones. En nuestro caso utilizamos la codificación encadenada de Freeman, que consiste en describir una sucesión de puntos de acuerdo a la posición de un punto con respecto a su predecesor en la secuencia. Para lograr esto se numeran los vecinos de un pixel con los números del 0 al 7 como indica la figura 91. Así un contorno puede almacenarse como la coordenada de un punto inicial seguido de una cadena de códigos (del 0 al 7) que indican la posición del próximo punto respecto al actual. Esta es una representación más compacta en comparación con una secuencia de coordenadas (x, y) y posibilita la extracción de características tales como el

CCH o NCCH ²³. La figura 90 exhibe un ejemplo de codificación utilizando este método.

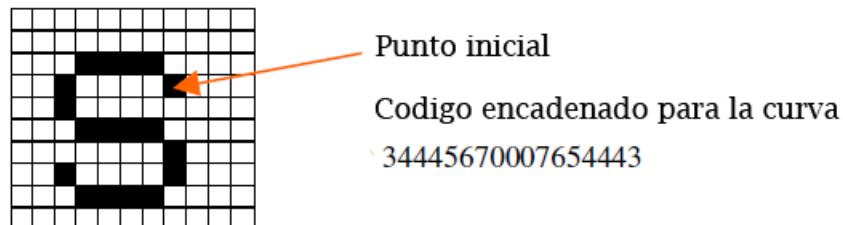


Figura 90: Ejemplo de codificación de un contorno utilizando los códigos de Freeman.

3	2	1
4		0
5	6	7

Figura 91: Matriz para la codificación de Freeman. El número indica la dirección de desplazamiento del próximo punto respecto del actual (punto negro)

4.3.5.3. Polígonos

Como mencionamos, los contornos son representados utilizando la codificación de Freeman. Sin embargo, la precisión con la que se muestran estos bordes suele ser excesiva o redundante para nuestras necesidades lo cual hace el análisis de figuras más difícil. Con el fin de reducir esta complejidad utilizamos un algoritmo para representar el mismo contorno con una menor cantidad de puntos. Esto se realiza mediante un algoritmo de aproximación por polígonos basado en el trabajo de Douglas y Pecker [7]. Previo a la aplicación de este algoritmo, convertimos el contorno a una representación tradicional de secuencias de puntos (x, y) correspondientes a las coordenadas del borde en la imagen. Esto se puede lograr fácilmente partiendo del punto inicial y calculando la variación en la coordenada según el código de vecino correspondiente (0-7).

La técnica de aproximación comienza tomando los dos puntos pertenecientes al contorno más alejados entre sí. Estos puntos se agregan a la aproximación y se traza una línea entre ellos. Luego se recorre el resto de los puntos del contorno buscando a aquel que se encuentre a mayor distancia de esta línea. A dicho punto se lo agrega a la aproximación, generando dos nuevas líneas con los dos

²³ CCH o chain code histogram es un histograma que se extrae a partir de la frecuencia de aparición de los números 0 a 7 en un contorno codificado con cadenas de Freeman. Este descriptor es invariante frente a traslaciones y escalamientos pero no frente a rotaciones para lo que existe su versión normalizada NCCH. Para más detalle ver [13]

puntos iniciales. Este proceso se itera agregando siempre el punto que se encuentra más distante de la aproximación hasta que todos los puntos del contorno se encuentren a una distancia menor que cierto parámetro de precisión establecido. En la figura 93 podemos apreciar una esquematización de este proceso.

Otra ventaja que otorga la utilización de esta aproximación es que si un contorno presenta una serie de puntos alineados, estos serán remplazados por un único segmento o lado del polígono, lo que nos permite identificar líneas en los contornos. Esta característica es aprovechada para realizar la detección de vasos como se ve en la sección 4.3.6. En la figura 92 podemos observar un contorno y su aproximación por polígonos.

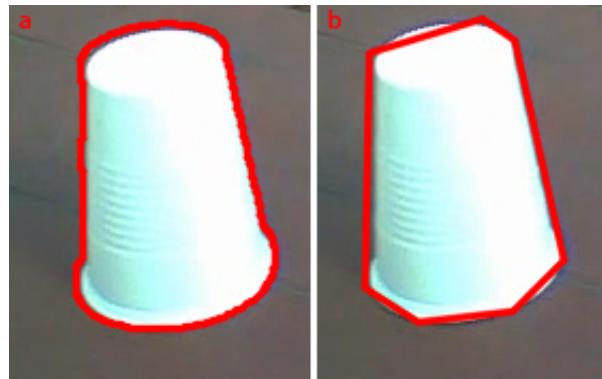


Figura 92: Aproximación de contornos mediante polígonos. (a) Se observa que el contorno se encuentra acorde el borde del vaso, las partes laterales del mismo se encuentran segmentadas (efecto serrucho) por la cantidad de puntos. (b) El contorno es representado mediante 8 puntos, los lados laterales solo involucran un segmento posibilitando la detección de líneas.

4.3.6. Filtros

Realizada la etapa de pre-procesamiento y la recolección de contornos de la imagen, el paso que sigue es verificar que estos contornos tengan características que correspondan con los objetos buscados. Para realizar esto, se implementan una serie de filtros que verifican propiedades puntuales sobre los mismos de forma tal que, si un contorno supera todos los filtros entonces consideramos a tal como un objeto reconocido.

Los filtros se disponen en cascada, es decir, se prueba un filtro en un contorno si y sólo si este superó todos los filtros anteriores. De esta forma podemos ordenar los filtros de más general a más específico, optimizando (generalmente) los tiempos de detección. Por ejemplo, conocer el área que abarca un contorno determina fácil y rápidamente si un contorno corresponde a una colilla o a un plato evitando la ejecución de filtros más específicos. En la figura 94 se exhiben los valores obtenidos por algunos de los filtros que detallamos a continuación:

4.3.6.1. Filtro de área

El filtro de área calcula el área encerrada por un contorno y verifica que la misma se encuentre por encima y/o por debajo de valores determinados. El

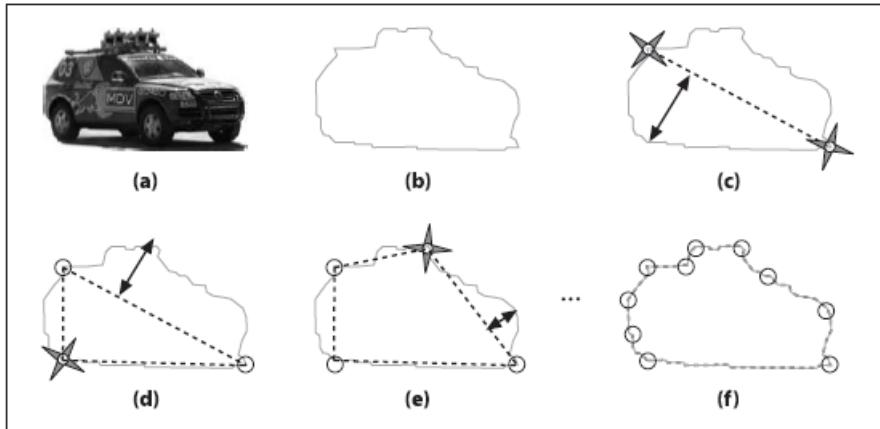


Figura 93: Iteraciones del algoritmo de aproximación de polígonos de Douglas-Peucker. (a) Imagen original. (b) Extracción de contorno (c) Se seleccionan dos puntos iniciales y se traza una línea entre ellos (d) Se agrega a la aproximación el punto más lejano del contorno a esta línea. (e) Se repite el proceso agregando siempre el punto más lejano (f) hasta que todos los puntos estén a cierta distancia requerida.

área encerrada por un contorno se computa utilizando la fórmula de Green [14]. Si bien esto es más veloz que contabilizar todos los pixeles encerrados en el contorno, posee la desventaja que no se tienen en cuenta agujeros en el interior del mismo.

4.3.6.2. Filtro de área por zona

Un objeto que posee un área determinada puede ocupar más o menos pixeles dependiendo de su ubicación en la imagen. Esto se debe a la perspectiva de la cámara y puede afectar al reconocimiento de un contorno. Para contemplar esta variación se pueden establecer distintas zonas en la imagen y definir distintos valores de área mínimos y máximo para cada una de ellas. En la figura 58 podemos apreciar las diferencia entre las distintas zonas de la imagen. Al ejecutarse este filtro se calcula la zona a la cual pertenece dependiendo de su ubicación en la imagen y se verifican los valores para dicha zona, si este los satisface entonces el contorno supera al filtro.

4.3.6.3. Filtro de perímetro

El filtro de perímetro calcula, efectivamente, el perímetro cubierto por un contorno. Esto se hace contando la cantidad de pixeles que lo definen. Vale remarcar, que el valor del perímetro depende ampliamente de la precisión con la que se obtengan los bordes ya que se ve drásticamente modificado si se aumenta o disminuye dicha precisión.

4.3.6.4. Filtro de excentricidad

La excentricidad es un parámetro que relaciona la longitud del eje más largo con la del más corto. Esta se define como $\text{longitud del eje más largo} / \text{longitud}$

del eje más corto. Algunas figuras pueden caracterizarse por tener valores de excentricidad que se mantienen en cierto rango y permite identificarlas usando este filtro.

4.3.6.5. Filtro de circularidad

La circularidad es un parámetro que relaciona área con perímetro y define la redondez de una figura. Se define como $\text{perímetro}^2/\text{area}$. Algunas figuras pueden caracterizarse por tener valores de circularidad que se mantienen en cierto rango y permite identificarlas usando este filtro.

4.3.6.6. Filtro de área rectangular

El filtro de área rectangular relaciona el área del objeto con el área del rectángulo contenedor mínimo. Este verifica que el área del contorno cubra en gran parte el área del rectángulo. Esto se logra mediante la división ($\text{areacontorno}/\text{arearectangulo}) > p$, donde p es el porcentaje de área que queremos que sea cubierto.

4.3.6.7. Filtro de elipse

El filtro de elipse verifica que un contorno determinado se ajuste o no a una elipse. Esto se logra usando un algoritmo basado en cuadrados mínimos. El mismo retorna como resultado las longitudes de los semi-ejes de la elipse a y b . Con estas medidas calculamos el área según la fórmula $A = \pi ab$ y se la compara con la obtenida a través del filtro de área. Se computa como $|A - AReal|/AReal < p$ donde p determina el valor de verosimilitud. Un valor de p mas chico significa mayor semejanza con una elipse. Si estas dos áreas son lo suficientemente similares consideramos que el contorno supera el filtro. Para el ajuste de elipse se utiliza el algoritmo Fitzgibbon, Pilu y Fisher [8].

4.3.6.8. Filtro de vaso

El filtro para vasos consiste en varias etapas. Recordando que el contorno se codifica como un polígono, se obtienen los dos segmentos más largos y se verifica que estos no compartan vértices. Por otro lado se pide que la suma de la longitud de estos dos segmentos supere al 50 % del perímetro del contorno, indicado por dicho filtro. Finalmente, se verifica que la distancia mínima entre estos dos ejes sea mayor a cierto valor prefijado, proporcional a la longitud del segmento más largo. El filtro se basa en la idea de que los dos segmentos más largos se correspondan con los lados laterales del vaso. Se puede apreciar una foto de un contorno que ha verificado este filtro en la figura 92.

4.3.6.9. Filtro de histograma

El filtro de histograma verifica la similitud de un histograma obtenido a partir del contorno con otro histograma obtenido a partir de una imagen modelo del objeto a detectar. El histograma modelo se calcula en el inicio de la ejecución mientras que el del contorno se computa en tiempo real. Esto se hace obteniendo el rectángulo contenedor mínimo de dicho contorno y computando un histograma 2D de la imagen utilizando los canales de saturación y tono. De esta manera, se puede caracterizar a un objeto por una composición de colores y no por su forma. El problema de este filtro es que requiere mayor procesamiento.

4.3.7. Reconocimiento de vasos

Para el reconocimiento de vasos se utilizan(en este orden) los filtros de área, perímetro, circularidad y el de vasos. El filtro de área verifica que $500 \leq area \leq 10000$, el de perímetro que $100 \leq per \leq 800$ y el de circularidad que $14 \leq circ \leq 18$. El filtro de vasos no es parametrizado.

4.3.8. Reconocimiento de colillas

Para el reconocimiento de vasos se utilizan(en este orden) los filtros de área, perímetro y área rectangular. El filtro de área verifica que $50 \leq area \leq 800$, el de perímetro que $10 \leq per \leq 150$ y el de área rectangular que sea cubierto el 70 % del área del rectángulo contenedor mínimo.

4.3.9. Reconocimiento de platos

Para el reconocimiento de platos se utilizan(en este orden) los filtros de área, perímetro, circularidad, excentricidad y de elipse. El filtro de área verifica que $500 \leq area \leq 10000$, el de perímetro que $500 \leq per \leq 1500$, el de circularidad que $10 \leq circ \leq 16$, el de excentricidad que $0.65 \leq ecc \leq 0.95$ y el de elipse que $|a - aReal|/aReal < 0.2$.

4.3.10. Centroide

El centroide es el centro geométrico o centro de gravedad de una figura y sirve como punto de referencia para la misma. Para calcular dicho punto hacemos uso de las formulas propuestas por Hu [12] para la descripción de figuras. El momento de Hu de un contorno 2D se define como:

$$m_{pq} = \sum_x \sum_y x^p y^q I(x, y) \quad (18)$$

donde $I(x, y)$ es la intensidad del pixel de coordenadas (x, y) . Notamos que $m_{0,0}$ representa la suma de la intensidad de todos los pixeles del contorno. Entonces, podemos definir al centroide (x_c, y_c) como $x_c = m_{1,0}/m_{0,0}$ e $y_c = m_{0,1}/m_{0,0}$. Estas dos fórmulas se corresponden con el promedio de las coordenadas x e y de los puntos de la figura.

4.4. Sistema de predicción

El algoritmo de visión que hemos descripto hasta aquí funciona procesando la imagen correspondiente a un único cuadro. Sin embargo, el sistema de visión del robot debe permitir el reconocimiento de objetos a medida que éste se desplaza por su entorno y por lo tanto dicho sistema debe procesar imágenes secuenciales del mismo. Con esto afirmamos que existe cierta relación (tanto temporal como espacial) entre las imágenes que el sistema de visión debe procesar y que podemos aprovechar esta característica para reforzar la detección de objetos. El mecanismo por el cual buscamos lograr este objetivo es el del sistema de predicción.

El sistema de predicción consume la información que el sistema de visión le entrega en cada cuadro y, con ella, realiza el seguimiento de objetos a lo largo del tiempo. Es a partir de este seguimiento que el sistema de predicción toma



Figura 94: Ejemplo de descriptores de contorno. Se visualiza el área, perímetro, zona y circularidad del contorno detectado de una colilla y un vaso

decisiones sobre la posición o existencia de los objetos que el sistema de visión reconoce. Vale destacar que estos dos sistemas pueden encontrarse, a menudo, en desacuerdo acerca de presencia u ausencia de un objeto. El sistema de predicción se basa en las siguientes asunciones:

1. Los objetos no realizaran grandes desplazamientos entre cuadros consecutivos - esto se debe a la proximidad temporal entre cuadros y que el robot no realiza movimientos bruscos.
2. Los objetos no sufrirán grandes alteraciones en su forma, área o perímetro entre cuadros consecutivos - con un razonamiento similar, la perspectiva no cambia lo suficiente como para alterar la figura, área o perímetro de los objetos.

Con estas premisas, dado un objeto A reconocido en el cuadro c_0 , cuyo centroide se encuentra en las coordenadas (en la imagen) (x_0, y_0) , área a y perímetro p , el sistema de predicción asume que si se encuentra un objeto A' en el cuadro siguiente c_1 con coordenadas (x_1, y_1) tal que $\|(x_1, y_1) - (x_0, y_0)\| \leq \alpha$, área a' tal que $|a - a'| \leq \gamma * a$ y perímetro p' tal que $|p - p'| \leq \theta * p$ entonces $A \simeq A'$, donde α , γ y θ son parámetros del sistema. Esta afirmación no solo vale para el cuadro subsiguiente c_1 sino también para los siguientes $c_2, c_3, \dots, c_{0+\eta}$ donde η y α se computan, para este caso, según el historial de detección de cada objeto. Basándose en esta regla, el sistema de predicción puede estimar la posición de un objeto que, para un cuadro en particular, no fue detectado por el sistema de visión. Para concretar este mecanismo, cuando un objeto es detectado a lo largo de varios cuadros se persiste la siguiente información:

- Última posición, área y perímetro - Se toman los datos de la detección más reciente del sistema de visión.

- Antigüedad del objeto - Cantidad de cuadros transcurridos desde que se comenzó a persistir el objeto en el sistema de predicción.
- Cantidad de detecciones - Se mantiene la cantidad de cuadros en la cual el sistema de visión detectó al objeto.
- Desplazamiento en la posición - Si el objeto es detectado más de una vez, se almacena el último desplazamiento en la posición $(x_i - x_{i-1}, y_i - y_{i-1})$.
- Antigüedad de la última detección - Cantidad de cuadros que transcurrieron desde la última detección por parte del sistema de visión.

Esta información no sólo permite corregir al sistema de visión cuando este no detecta un objeto sino que también podemos corregirlo cuando este no rechaza falsas detecciones de objetos inexistentes. Para implementar estos mecanismos se establecen las siguientes reglas sobre la persistencia de objetos:

1. Un objeto debe ser detectado por el sistema de visión en δ cuadros consecutivos para ser considerado como tal por el sistema de predicción
2. El sistema de predicción descarta un objeto considerado si el sistema de visión no lo detecta en un número η de cuadros consecutivos.
3. η se calcula cada λ cuadros como $\frac{\text{detecciones}}{\text{antiguedad}} \times \rho$ donde ρ es un parámetro del sistema.
4. Si un objeto es considerado por el sistema de predicción y este no es detectado por el sistema de visión para algún cuadro entonces el sistema de visión realiza una estimación de su posición de acuerdo a la información persistida a partir de sus apariciones en cuadros anteriores.

Estas reglas, aunque parezcan complejas por el número de variables que manejan, son realmente intuitivas. El sistema de predicción consume la información que el sistema de visión le brinda y la manipula con cierta desconfianza de acuerdo a su conveniencia. En una primera instancia, los objetos deben ganarse la confianza del sistema de predicción apareciendo ²⁴ regularmente en cuadros consecutivos, en concordancia con la asunción número 1. De no cumplir con esta premisa, consideramos que no se trata de un objeto de interés y utilizamos la regla 1 para eliminar dicha detección.

Una vez ganada la confianza del sistema de predicción, este último comienza a tener a estos objetos en consideración ²⁵ y veta por su existencia aunque el sistema de visión no los detecte. El grado en que el sistema cree en la existencia de estos objetos se regula de acuerdo al historial de apariciones de los mismos. Si estos abusan de la confianza otorgada entonces el sistema de predicción los descarta como indica la regla 2. Este grado de confianza o de libertad se traduce en cantidad de cuadros que puede pasar un objeto sin ser detectado por el sistema de visión y siendo considerado por el sistema de predicción. Dicho número se determina periódicamente utilizando la fórmula de la regla 3. Finalmente,

²⁴Con aparición nos referimos a que el objeto sea reconocido por el sistema de visión.

²⁵Decimos que el sistema de predicción considera a un objeto cuando este empieza a informar la posición de este dentro de la imagen. Nótese que, para un cuadro determinado, el sistema de visión puede detectar un objeto y el sistema de predicción, como todavía no considera a dicho objeto, ignora la detección y no informa de su presencia.

durante este lapso de confianza, el sistema de predicción ‘aproxima’ la posición de los objetos si estos no son detectados por el sistema de visión, como dice la regla 4.

La variable α determina la distancia máxima que puede desplazarse un objeto entre cuadros para que el sistema de predicción lo considere como el mismo. Las variables γ y θ determinan el máximo porcentaje de variación en el área y perímetro respectivamente, que puede soportar un objeto entre cuadros para ser considerado como el mismo. Valores típicos para la detección de colillas y vasos son de $0.1 \leq \gamma \leq 0.3$ y $0.1 \leq \theta \leq 0.3$. Para el caso de α definimos un valor aproximado de 30 pixeles²⁶. Sin embargo, la tolerancia en el desplazamiento varía según los cuadros transcurridos desde la última detección. Si en el cuadro anterior fue detectado entonces se utiliza α , si fue detectado hace dos cuadros entonces 2α , tres cuadros 3α y así. La variable δ establece el umbral de detección de un objeto por parte del sistema de visión para que el sistema de predicción lo empiece a considerar como basura. ρ establece, independientemente de la regularidad de detección de un objeto, la cantidad máxima de cuadros que puede permanecer este sin ser detectado por el sistema de visión y considerado por el sistema de predicción.

Es importante destacar que tanto ρ como δ repercuten directamente en la performance de detección. En el caso del parámetro δ , el sistema de visión debe reconocer un objeto $\delta - 1$ veces antes de que sea tenido en consideración por el sistema de predicción y, en este lapso, las detecciones serán ignoradas²⁷. Sin embargo, las apariciones no sostenidas (de hasta $\delta - 1$ cuadros) de elementos que aparentan ser objetos para el sistema de visión pero no lo son, son también ignoradas, disminuyendo el porcentaje de falsos negativos.

La variable ρ , en cambio, establece otro tipo de balance. Supongamos que un objeto que tiene un historial de detección positivo repentinamente deja de ser detectado, debido a una oclusión o un cambio en la iluminación, por el sistema de visión pero permanece en el campo de visión del robot. En esta circunstancia, el sistema de predicción arriesgará la posición (hasta ρ veces) pudiéndole acertar o no hasta que el objeto sea nuevamente detectado por el sistema de visión o sea descartado. En esta situación es donde se obtienen nuevas detecciones que el sistema de visión nunca podría obtener por sí solo. Sin embargo, esto acarrea el siguiente problema: si un objeto (de aparición regular) deja de aparecer en la imagen (por desplazamiento del objeto o del robot), el sistema de visión, lógicamente, deja de detectarlo. Cuando esto sucede, el sistema de predicción arriesga la posición del objeto un máximo de ρ veces (según el grado de regularidad de detección) y, en este caso, no acierta a ninguna ya que el objeto dejó de estar en el rango de visión del robot.

A partir de estas reflexiones surge la importancia de configurar estos parámetros de acuerdo al sistema de visión que estemos usando. Si tenemos un sistema de visión con alto porcentaje de falsos positivos y bajo porcentaje de falsos negativos entonces es probable que arriesgar más posiciones de objetos cuando

²⁶ El valor de 30 pixeles nos dió resultados para imágenes de 640x480, para otras resoluciones habría que buscar el equivalente.

²⁷ El sistema de predicción ignora una detección del sistema de visión en el sentido de que este primero no informa a dicha detección como un objeto residual encontrado para ese cuadro, sino que sólo registra dicha detección para cuadros futuros.

estos no son detectados mejore el rendimiento²⁸. En este caso debemos elegir valores de ρ , no muy pequeños, para afrontar el déficit de detección del sistema de visión y valores de δ , más bien pequeños, para no entorpecer la detección. Es importante remarcar que el bajo porcentaje de falsos positivos, en este caso, nos permite confiar en el sistema de visión en el sentido de que las cosas que este detecte tienen altas probabilidades de ser objetos.

En el caso recíproco, un sistema de visión con alto porcentaje de falsos negativos y bajo porcentaje de falsos positivos quizás convenga tener un valor de δ alto, para filtrar la mayor cantidad de detecciones falsas. Al hacerlo, puede suceder que también filtremos detecciones positivas pero en ese caso habría que estudiar cuánto se gana y cuánto se pierde y buscar un balance.

Otra posibilidad es que el sistema de visión sea muy confiable (bajo porcentaje de falsos positivos/negativos). En esta situación conviene configurar al sistema de predicción para que haga más caso de lo que el sistema de visión le informa. Esto significa que cuando el sistema de visión no encuentra un objeto el sistema de predicción no debe arriesgar mucho ya que el objeto probablemente no se encuentre. Asimismo, unas pocas detecciones de un objeto bastan para que el sistema de predicción lo tenga en consideración ya que existe una alta probabilidad de que realmente se trate de un objeto. Esto se traduce en valores de δ y ρ bajos.

4.5. Focalización

Focalización es el mecanismo que busca que el sistema de visión se centre en la detección de un único objeto a lo largo del tiempo. Esta situación es deseable cuando el sistema de visión ya detectó una basura y el robot se dirige a recolectarla. En esta circunstancia, no nos interesan otros objetos que puedan aparecer durante el trayecto y por lo tanto tratamos de evitar cualquier tipo de procesamiento innecesario sobre ellos. Con este objetivo, el mecanismo selecciona una sub-imagen a procesar en la cual supone que va a encontrar al objeto en cuestión. El mecanismo de focalización se encuentra muy ligado al sistema de predicción ya que debe persistir cierta información sobre la posición del objeto para poder tomar una sub-imagen que lo contenga.

El mecanismo de focalización puede separarse en dos etapas. En una primera etapa, utilizando la información brindada por el sistema de predicción, se selecciona un objeto a focalizar. Existen varios criterios de selección, algunos que consideramos más relevantes son:

- Objeto más cercano - Se estiman las distancias a los objetos y se selecciona aquel que se encuentre más cercano al robot.
- Objeto más detectado en los últimos 20 cuadros - Se selecciona al objeto que fue reconocido más veces por el sistema de visión en las últimas 20 imágenes.
- Objeto más antiguo - Se elige al objeto que comenzó a ser persistido hace más cuadros según el sistema de predicción.

²⁸ Decimos que el rendimiento del algoritmo mejora cuando éste disminuye el porcentaje de falsos positivos o falsos negativos.

- Combinación ponderada - Se utilizan los criterios mencionados dándoles mayor o menor peso a cada uno y eligiendo al objeto que obtenga mayor puntuación.

Pensamos todos los criterios con el objetivo de maximizar las chances de que el robot tenga éxito en la recolección del objeto. El criterio de menor distancia implica un menor recorrido hacia el objeto y por lo tanto menos posibilidades de cometer error durante el recorrido. Si utilizamos el criterio de mayor detección en los últimos 20 cuadros estamos maximizando las chances de que el sistema de visión detecte al objeto durante el trayecto del robot hacia el mismo. Con un razonamiento similar, el criterio de mayor antiguedad maximiza las chances de que el sistema de visión detecte al objeto pero utilizando información más general y no tan local como el resultado de los últimos cuadros. Previo a la focalización es necesario alimentar al sistema de predicción ya que sin este, no es posible utilizar ninguno de los criterios mencionados.

Una vez que seleccionamos el objeto, utilizamos el sistema de predicción para dar cuenta de la posición del mismo. Con esta información, tomamos sub-imágenes (ventanas) de la captura original que sólo abarquen un entorno del objeto como se aprecia en la figura 95 y 96. Como beneficio, obtenemos una disminución en el tiempo de procesamiento requerido ya que sólo se procesa una fracción de la imagen. El tamaño de la ventana queda determinado por las dimensiones del contorno del objeto a focalizar. Si consideramos al sistema de visión como un sensor podríamos decir que este mecanismo incrementa la frecuencia de muestreo del mismo ya que permite tomar una mayor cantidad de muestras en menor tiempo. La focalización sobre un objeto finaliza únicamente cuando este objeto deja de ser considerado por el sistema de predicción y entonces retomamos al procesamiento de la imagen completa donde se seleccionara un nuevo objeto a focalizar.

4.6. Resultados

En esta sección se describen los parámetros utilizados para la medición de la performance y se muestran los resultados obtenidos para el sistema de visión y un sistema de visión con predicción. De este último exhibimos dos configuraciones. Tomamos 8 videos de prueba en los cuales 3 pertenecen a colillas, 3 a vasos y 2 a platos. Los videos fueron tomados utilizando una silla de oficina giratoria para poder desplazarse suavemente por el entorno simulando el movimiento del robot. La cámara se ubicó a 30 cm del piso formando un ángulo de 30° con el asiento en dirección hacia el suelo. La cámara utilizada es una microsoft lifecam vx-700 del tipo webcam y posee una resolución de 640x480. Esta misma resolución es utilizada para el análisis de las imágenes.

4.6.1. Parámetros de medición

Con el objetivo de medir la efectividad del algoritmo de detección, desarrollamos un sistema de benchmarking. Este sistema permite a un usuario indicar la presencia de objetos a detectar en un video, marcándolos con rectángulos en las sucesiones de cuadros. De esta manera, corremos el algoritmo de detección por el mismo video y verificamos que los centroides de los objetos detectados por el algoritmo se encuentren contenidos dentro de los rectángulos indicados por el



Figura 95: Se observa un vaso delimitado por un rectángulo contenedor mínimo rojo. El rectángulo se utiliza para calcular el tamaño de la sub-imagen previo al ventaneo.



Figura 96: Ventana o sub-imagen obtenida de un vaso en focalización. El tamaño de la ventana es de tamaño proporcional al contorno del objeto en cuestión.

usuario. Para obtener estadísticas sobre la performance medimos los siguientes parámetros:

- Cantidad de objetos a detectar ($\#O$): Se corresponde con la cantidad de rectángulos marcados por el usuario
- Hits(**H**) : Cantidad de detecciones por parte del algoritmo que se corresponden con rectángulos marcados.
- Misses o detecciones falsas (**M**) : Cantidad de detecciones por parte del algoritmo que no se corresponden con rectángulos marcados.

Se desprende directamente, de estos valores, que la cantidad de detecciones: $\#D = H + M$. Con estos datos obtenemos los porcentajes de error, falsos positivos :

$$fp = 1 - \frac{H}{\#O}$$

y falsos negativos:

$$fn = \frac{M}{\#D}$$

En el caso óptimo, ambas medidas de error tendrán un valor de 0 %.

A estas estadísticas se agregan las particulares para el sistema de predicción. Recolectamos información sobre cuántos casos el sistema de predicción estimó la posición y acertó. Denominamos a la cantidad de estimaciones realizada por el sistema de predicción con **G** mientras que a las estimaciones que además fueron acertadas **G&H**.

4.6.2. Análisis

Exhibimos los resultados obtenidos tras utilizar sólo el sistema de visión en el cuadro 31 e utilizando el sistema de visión con el sistema de predicción, utilizando dos configuraciones distintas, en los cuadros 32 y 33. Además exhibimos los resultados promedios de falsos positivos y falsos negativos agrupados por objeto para todas las configuraciones en los cuadros 34, 35 y 36. Las configuraciones utilizadas fueron seleccionadas para destacar distintos comportamientos del sistema de predicción. Éstas son:

- **Configuración 1:** $\lambda = 10, \rho = 10, \delta = 3$
- **Configuración 2:** $\lambda = 10, \rho = 4, \delta = 2$

En los resultados arrojados en el cuadro 31 (sólo por el sistema de visión) observamos problemas en la detección de colillas ya que posee altos porcentajes de falsos negativos. Esto indica que el detector, en el caso de las colillas, tiene problemas en distinguir a estas de su entorno ya que comete demasiados errores al reconocer colillas que no existen. Esto concuerda con el hecho de que, en el caso de las colillas, el número de detecciones $\#D$ es siempre mayor que $\#O$ también observado en el cuadro 31. No obstante, al utilizar el sistema de predicción observamos una gran reducción de este porcentaje producto del filtrado realizado por dicho sistema el cual no considera una basura hasta que esta esté presente en δ cuadros consecutivos. Esta mejora viene acompañada de efecto negativos ya que la cantidad de hits se ve afectada pero con menor intensidad. Esto prueba

que el filtrado realizado es efectivo ya que elimina, sobre el total de las detecciones, en mayor proporción a aquellas que son falsas. El cuadro 34 exhibe las ventajas de utilizar un sistema de predicción para las colillas.

En el caso de los vasos, observamos que el sistema de visión brinda una muy buena performance por sí solo, como se ve en el cuadro 31 donde se muestran valores menores al 10% en falsos positivos y falsos negativos para todos los videos. La diferencia de performance con la detección de colillas es comprensible ya que los vasos poseen características como tamaño y color que hacen que sea más fácil reconocerlos y mas difícil confundirlos en relación con estas últimas, que poseen un color similar al del fondo y son de tamaño pequeño. En este caso, el sistema de predicción no altera demasiado los resultados como en el caso de las colillas, sin embargo, destacamos un aumento en la cantidad de detecciones falsas en la configuración 1, atribuido a el valor elevado de ρ , acompañado de un incremento similar en el porcentaje de aciertos. Distinto es el caso de la configuración 2 que disminuye levemente el porcentaje de falsas detecciones y el porcentaje de aciertos respecto de los resultados obtenidos por el sistema de visión. El comportamiento de la configuración 2 es razonable ya que esta es menos invasiva²⁹ y más respetuosa del sistema de visión subyacente en comparación con la configuración 1. Esto se ve reflejado, en parte, en la cantidad de estimaciones (**G**) y estimaciones correctas (**G&H**) cometidas por cada configuración. Podemos apreciar las diferencias entre las distintas configuraciones para los vasos en el cuadro 35.

El caso de los platos es similar al de los vasos. El sistema de visión obtiene buenos resultados por sí sólo pero se ve perjudicado por la configuración 1 del sistema de predicción ya que éste no acierta en gran proporción de las estimaciones que realiza. Sin embargo, en este caso, el aumento en el porcentaje de detecciones falsas es relativamente mayor que el aumento de aciertos. La configuración 2, por otro lado, aumenta el porcentaje de aciertos y disminuye la cantidad de detecciones falsas aunque en ambos casos, de manera leve, mejorando apenas los resultados obtenidos por el sistema de visión. En este caso, los porcentajes de aciertos de las estimaciones son bajos pero observamos que utilizando la configuración 2 el número de estimaciones es mucho menor con lo que se cometen menos errores. Un cuadro comparativo de las distintas configuraciones para vasos pueden observarse en el cuadro 36.

Las dos configuraciones del sistema de predicción modifican de distintas maneras al algoritmo de visión subyacente. Para cada objeto hemos obtenido resultados y comportamiento particulares lo que justifica la necesidad de ajustar los parámetros del mismo para obtener la mejor performance posible. Para poder realizar este ajuste es necesario establecer que grado de incidencia tienen los dos tipos de errores en la performance del algoritmo. Por ejemplo, bajo distintas circunstancias podemos preferir que el sistema de visión cometa más errores de tipo I (falsos positivos) con el objetivo de minimizar errores de tipo II (falsos negativos) o viceversa. En ese sentido el sistema de predicción proporciona los mecanismos que permiten lograr estos ajustes sin necesidad de modificar el algoritmo de detección subyacente. Para nuestro caso, por lo explicado en 3.8, priorizamos la disminución en los errores de tipo I frente a los de tipo II y, por tanto, configuraríamos el sistema de predicción acorde.

²⁹Nos referimos a una configuración menos invasiva en el sentido de que la presencia del sistema de predicción no modifica drásticamente los resultados arrojados por el sistema de visión.

En cuanto al tiempo de procesamiento demandado por el algoritmo obtuvimos, en la notebook cuyas características explicamos en la sección 2.5.1, una velocidad de procesamiento de aproximadamente 8.3 *cuadros/seg* sólo utilizando el sistema de visión en conjunto con el sistema de predicción. No obstante, como hemos explicado en 4.5, los mecanismos de focalización y ventaneo reducen notablemente la complejidad del algoritmo una vez que se enfoca un residuo.

Video	(#O)	(#D)	(H)	(M)	fp	fn
vasos1	5050	4880	4849	31	0.6 %	3.9 %
vasos2	1777	1708	1696	12	0.7 %	4.5 %
vasos3	6196	5354	5251	103	1.9 %	15.2 %
colillas1	1645	1648	1282	367	22.2 %	22.0 %
colillas2	891	1045	636	409	39.1 %	28.6 %
colillas3	2708	3899	2373	1526	39.2 %	12.7 %
platos1	2236	1966	1832	134	6.8 %	18.0 %
platos2	1128	1228	1117	111	9.0 %	0.9 %

Cuadro 31: Performance del sistema sin el algoritmo de predicción

Video	(#O)	(#D)	(H)	(M)	fp	fn	G	G&H
vasos1	5050	5017	4888	118	2.3 %	3.2 %	245	142
vasos2	1777	1773	1707	66	3.7 %	3.9 %	121	60
vasos3	6196	5703	5459	240	4.2 %	11.8 %	536	373
colillas1	1645	1444	1330	98	6.8 %	19.1 %	296	229
colillas2	891	687	586	95	13.9 %	34.2 %	127	57
colillas3	2708	2555	2242	254	10.1 %	17.2	516	320
platos1	2236	1966	1854	218	10.5 %	17.0 %	179	34
platos2	1128	1228	1112	167	13.0 %	1.4 %	115	16

Cuadro 32: Performance del sistema con el sistema de predicción para $\lambda = 10$, $\rho = 10$, $\delta = 3$

4.7. Conclusión

Presentamos un algoritmo para el reconocimiento de distintos objetos en un ambiente dinámico en el marco de un robot autónomo capaz de recolectar basura. El mismo funciona sobre la librería de visión computacional OpenCV y el lenguaje C++. El algoritmo cuenta con una etapa de pre-procesado donde se extraen los pixeles de colores de interés y se disminuye el ruido utilizando operaciones morfológicas y de threshold. La etapa de procesamiento extrae los contornos y utiliza una aproximación por polígonos para simplificar el análisis posterior. Este consiste en una serie de filtros pre-definidos dispuestos en cascada que interpretan la información de contorno y determinan si se trata de un objeto de interés o no. Implementamos un mecanismo de predicción parametrizable que funciona sobre el algoritmo en cuestión por el cual se puede optimizar la performance de detección según las características propias de cada detector. Se midieron los resultados obtenidos realizando el reconocimiento de vasos, colillas y platos obteniendo un promedio de 14 % de falsos positivos y 5 % de

Video	(#O)	(#D)	(H)	(M)	fp	fn	G	G&H
vasos1	5050	4866	4839	27	0.5 %	4.1 %	83	72
vasos2	1777	1700	1689	11	0.6 %	4.9 %	32	27
vasos3	6196	5318	5222	96	1.8 %	15.7 %	241	226
colillas1	1645	1220	1168	53	4.3 %	28.9 %	142	126
colillas2	891	652	566	86	13.1 %	36.4 %	78	38
colillas3	2708	2414	2142	272	11.2 %	20.0 %	303	158
platos1	2236	1958	1843	115	5.8 %	17.5 %	50	16
platos2	1128	1211	1115	96	7.9 %	1.1 %	26	8

Cuadro 33: Performance del sistema con el sistema de predicción con $\lambda = 10$, $\rho = 4$, $\delta = 2$

Configuración	<i>fp</i>	<i>fn</i>	G	G&H	G&H/G
Sin predicción	18.1 %	34.9 %	-	-	-
Configuración 1	20.7 %	9.5 %	939	606	64.5 %
Configuración 2	26.0 %	9.5 %	523	322	61.5 %

Cuadro 34: Performance promedio de las colillas por configuración y efectividad de las estimaciones del sistema de predicción

falsos negativos. Finalmente agregamos un mecanismo de focalización por el cual el sistema completo puede concentrarse sobre un único objeto reduciendo los tiempos de procesamiento.

4.7.1. Posibles extensiones

Existen varios puntos donde consideramos se pudo mejorar este sistema de reconocimiento de objetos. Una de las áreas donde se pueden introducir mejoras es en la sección de filtros. Como hemos visto, existen novedosas técnicas de descripción y representación de contornos que pueden ser introducidas como un filtro adicional sin necesidad de modificar la estructura del algoritmo. La utilización de dichas técnicas pueden servir para la caracterización de nuevos objetos a reconocer o bien para reforzar los objetos ya tratados. Por otro lado, hallamos al sistema de filtros en cascada un tanto limitado para la tarea propuesta ya que frente a pequeñas perturbaciones que provoquen que un único filtro rechaze un determinado contorno, este último se verá descartado por el sistema. Como alternativas a este sistema se puede implementar un sistema de votación donde se reúnan los resultados de cada filtro y se tome una decisión en base a ellos. Adicionalmente, sería interesante que en este sistema de votación se pueda indicar un valor que establezca la importancia de cada filtro en relación a un objeto determinado. Otra posibilidad, basándonos en el trabajo [22] expuesto en la sección 4.2.1, es utilizar las características medidas por los filtros para entrenar un clasificador. Luego este será capaz de caracterizar a los residuos utilizando los valores medidos de manera óptima.

En lo que respecta a la etapa de pre-procesamiento se pueden tomar las ideas del trabajo [19] expuesto en la sección 4.2.3 para caracterizar el entorno del robot. Esto es, mantener histogramas que modelen el entorno del robot utilizando aquellas regiones de las imágenes donde no se hallaron objetos de interés.

Configuración	<i>fp</i>	<i>fn</i>	G	G&H	G&H/G
Sin predicción	9.4 %	1.2 %	-	-	-
Configuración 1	7.4 %	3.3 %	902	575	63.7 %
Configuración 2	9.7 %	1.1 %	356	325	91.2 %

Cuadro 35: Performance promedio de los vasos según configuración del sistema y efectividad de las estimaciones del sistema de predicción

Configuración	<i>fp</i>	<i>fn</i>	G	G&H	G&H/G
Sin predicción	12.3 %	7.6 %	-	-	-
Configuración 1	11.8 %	12.0 %	294	50	17.0 %
Configuración 2	12.0 %	6.6 %	76	24	31.5 %

Cuadro 36: Performance promedio de los platos según configuración del sistema y efectividad de las estimaciones del sistema de predicción

De este modo sería más fácil distinguir un objeto del fondo o entorno que lo rodea. Esto también ayudaría al algoritmo de visión a adaptarse a los cambios de iluminación que pueden ocurrir ya que este histograma se podría actualizar periódicamente.

Introducir algún sistema de sensado por el cual el robot pueda detectar si efectivamente se ha recolectado un residuo reconocido o no, resulta de gran interés ya que permitiría implementar algún mecanismo de aprendizaje por reforzamiento para el sistema de visión. De esta manera, el sistema de visión podría adaptar parámetros como el valor del umbral en las operaciones de filtrado, el rango en el canal de tono para el filtro de color o los límites superiores e inferiores en el filtro de área de acuerdo a los refuerzos recibidos.

El sistema de predicción puede cometer errores cuando, por efectos de iluminación o oclusión, un objeto cambia su área o perímetro en gran proporción. Por lo tanto, la inclusión de otros *descriptores de contorno* en este sistema podría resultar en un seguimiento más efectivo de los mismos a lo largo de los cuadros. También es posible mejorar la estimación de las posiciones utilizando fórmulas que tengan en cuenta no solo el último desplazamiento sino el desplazamiento promedio del objeto en los últimos cuadros.

5. Conclusiones

En este trabajo hemos desarrollado la construcción e implementación de un robot autónomo, móvil, capaz de recolectar residuos en un entorno dinámico. Como mecanismo de locomoción utilizamos dos ruedas de tracción y una tercer rueda de tipo castor para proveer equilibrio, movilizadas por un par de motores de corriente continua provistos de una caja reductora y encoders. Para el sensado del entorno equipamos al robot con sensores infrarrojos, sensores de distancia por ultrasonido, sensores reflectivos de piso y un medidor de tensión en la batería. Para cada uno de estos dispositivos desarollamos circuitos de control y rutinas que nos permitieron la obtención de datos como así la interacción con el entorno. Los actuadores y sensores son controlados desde una computadora a bordo de tipo netbook que utilizamos como unidad principal de procesamiento. Realizamos el interconexiónado de las placas de control con el computador utilizando una red del tipo *daisy-chain* en donde la información se transmite a través de un protocolo escalable capaz de conectar distintos tipos de placas entre sí y con el computador principal.

Implementamos la interacción del robot con su entorno mediante la definición de distintos comportamientos dispuestos bajo un arquitectura de tipo *Subsumption*, los más importantes son: wandering, evitamiento de obstáculos, recargar batería, descargar y recolectar basura. Para la navegación, desarollamos un sistema de odometría que permite estimar la posición del robot en el entorno a través del uso de los encoders de los motores. Para el reconocimiento de residuos, instalamos una cámara de tipo webcam conectada al computador principal donde corre un sistema de visión que diseñamos para el reconocimiento de colillas de cigarrillo, vasos de plástico y platos descartables. Éste sistema está compuesto por una etapa de pre-procesamiento que consiste en un filtro de color, operaciones morfológicas y operaciones de tipo umbral para la eliminación de ruido y una segunda etapa de análisis de contornos en donde el sistema verifica que las figuras encontradas correspondan con los residuos buscados. Agregamos a este sistema de visión un módulo de predicción y un módulo de focalización por el cual aumentamos la confianza y eficiencia del algoritmo.

Realizamos la simulación de los comportamientos del robot en el ambiente Webots en donde se utilizaron versiones virtuales de los sensores, actuadores y entorno para verificar su buen funcionamiento.

6. Agradecimientos

Queremos agradecer a nuestro tutor, Juan Miguel Santos por la guía y horas de dedicación, a las autoridades del departamento de informática por la confianza y apoyo durante todos estos meses y a los amigos y familiares que nos acompañaron con consejos, críticas y tasas de café... Muchas gracias.

Referencias

- [1] Andrew Reed Bacha, Dr. A. Lynn Abbott, and Andrew Reed Bacha. Processing line detection and lane following for an autonomous mobile robot, 2005.
- [2] Nick Bennett, Robert Burridge, and Naoki Saito. A method to detect and characterize ellipses using the hough transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21:652–657, 1999.
- [3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [4] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA, 1985.
- [5] Salvatore Candido. Autonomous robot path planning using a genetic algorithm.
- [6] F. John Canny. A Computational Approach to Edge Detection. 8(6):679–698, 1986.
- [7] D.H. Douglas and T.K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10:112–122, 1973.
- [8] M. Fitzgibbon, A. W. and Pilu and R. B. Fisher. Direct least-squares fitting of ellipses. 21(5):476–480, May 1999.
- [9] Amalia F. Foka and Panos E. Trahanias. Predictive autonomous robot navigation. In *In Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 490–495, 2002.
- [10] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2008.
- [11] van Vliet L.J. Rieger B. van Ginkel M. Ligteringen R. Hendriks, C.L.L. DIPimage: A Scientific Image Processing Toolbox for MATLAB. 2005.
- [12] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *Information Theory, IRE Transactions on*, 8(2):179–187, February 1962.
- [13] Jukka Iivarinen and Ari Visa. Shape recognition of irregular objects. In *Intelligent Robots and Computer Vision XV: Algorithms, Techniques, Active Vision, and Materials Handling, Proc. SPIE 2904*, pages 25–32, 1996.
- [14] The IDL Astronomy User's Library. Eroding and dilating image objects. 2007.
- [15] Marvin Minsky. *The society of mind*. Simon & Schuster, Inc., New York, NY, USA, 1986.
- [16] Maria C. Neves and R. S. Tome. A control architecture for an autonomous mobile robot. In *Proceedings of Autonomous Agents*. ACM, 1997.

- [17] Stefano Nolfi. Evolving non-trivial behaviors on real robots: A garbage collecting robot. *Robotics and Autonomous Systems*.
- [18] K. Wedeward R. Clark, A. El-Osery and S. Bruder. A navigation and obstacle avoidance algorithm for mobile robots operating in unknown, maze-type environments. December 2004.
- [19] John S. Seng and Thomas J. Norrie. Sidewalk following using color histograms. *J. Comput. Small Coll.*, 23(6):172–180, 2008.
- [20] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. 30(1):32–46, April 1985.
- [21] A. CONCI V. A. OLIVEIRA. Skin detection using hsv color space, 2009.
- [22] FRITS K. VAN EVERD, GERIE WAM VAN DER HELJDEN, LAMBERTUS A.P. LOTZ, GERRIT POLDER, ARJAN LAMAKER, ARJAN DE JONG, MARJOLIJN C. KUYPER, ELTJE J.K. GROENDIJK, JACQUES J. NEETESON, and TON VAN DER ZALM. A mobile field robot with vision-based detection of volunteer potato plants in a corn crop1. *Weed Technology*, 20(4):853–861, 2006.
- [23] Eric W. Weisstein. Green’s theorem. from mathworld—a wolfram web resource.
- [24] Dengsheng Zhang and Guojun Lu. Review of shape representation and description techniques, 2002.

A. Protocolo de comunicación

La comunicación entre los distintos componentes del robot es imprescindible para su correcto funcionamiento y por lo tanto, un protocolo acorde es la base fundamental sobre la cual se construye dicha comunicación. Toda interacción con el entorno es precedida por los mensajes enviados desde el controlador principal hacia las placas controladora y debemos asegurar que su correctitud. Las placas controladoras se dividen en grupos según su función y periféricos que tenga conectados.

A.1. Formato del paquete

El paquete consta de un encabezado común con datos que identifican el emisor y receptor del paquete, el comando a enviar y posibles datos extras que sean requeridos. En el cuadro 37 mostramos la estructura interna de un paquete típico.

LARGO	DESTINO	ORIGEN	COMANDO	DATO	XOR
-------	---------	--------	---------	------	-----

Cuadro 37: Formato y encabezado del paquete de datos

Tanto los paquetes de envío de datos como los de respuesta tienen el mismo formato y comparten el valor en el campo de comando.

LARGO

Indica el largo en bytes del paquete que viene detrás. Consta de 1 byte. Necesario ante la existencia del campo *DATO* de longitud variable. En caso que la longitud del campo *DATO* sea cero, es 0x04.

DESTINO

Identifica al destinatario del paquete, consta de 1 byte. Los 4 bits más significativos indican el grupo y los 4 bits menos significativos, el número de *ID* de la placa de destino. Los grupos predefinidos se tratan en el apartado A.3. Si el *ID* de placa es F, entonces el paquete es transmitido a todos los *IDs* del grupo indicado. El valor 0xFF indica que el paquete es transmitido a todas las placas de todos los grupos.

ORIGEN

Similar al *DESTINO*, consta de 1 byte. Determina el emisor del paquete para la respuesta. Los 4 bits más significativos indican el grupo y los 4 bits menos significativos, el número de *ID* de la placa de origen. Los grupos predefinidos se tratan en el apartado A.3. Los valores permitidos son del 0 al E, ya que F indica transmitido y no es válida una respuesta transmitido.

COMANDO

El comando consta de 1 byte. Comando enviado al destino, que puede o no tener datos en el campo *DATO*. Definidos en la sección A.2.

DATO

Contiene los parámetros o datos extras que puedan ser necesarios para el comando enviado. En el caso que el comando no los requiera, el campo debe ser nulo y el campo *LARGO* será 0x04.

En el apartado A.10 se explican las reglas para la codificación de números utilizadas en el protocolo.

XOR

El comando consta de 1 byte. Se realiza un XOR con cada uno de los bytes que conforman el paquete.

A.2. Posibles comandos

El campo *COMANDO* determina la acción que debe realizarse en el destinatario o la respuesta al comando recibido. El rango para los comandos comunes a todos los grupos de placas son desde 0x00 hasta 0x3F. Los comandos específicos para cada grupo deben ser desde 0x40 hasta 0x7F. El bit más significativo indica que el paquete es una respuesta, los bits menos significativos, indicarán a qué comando se está respondiendo.

Todos los comandos generan una respuesta o al menos un ACK con el campo *DATOS* vacío.

A.2.1. INIT

Sincroniza el inicio de todas las placas en la cadena. Debe ser recibido por la placa para inicializarse y poder informar al controlador principal de su existencia.

También utilizado para obtener la confirmación del listado de dispositivos conectados.

Comando enviado

- COMANDO: 0x01
- DATO: vacío

Respuesta al comando

- COMANDO: 0x81
- DATO: Descripción de la placa en texto plano

A.2.2. RESET

Pide el reset de la placa.

Comando enviado

- COMANDO: 0x02
- DATO: vacío

Respuesta al comando

- COMANDO: 0x82
- DATO: Descripción de la placa en texto plano

A.2.3. PING

Envia un ping a la placa

Comando enviado

- COMANDO: 0x03
- DATO: vacío

Respuesta al comando

- COMANDO: 0x83
- DATO: vacío

A.2.4. ERROR

Informa que ha habido un error.

Comando enviado

- COMANDO: 0x04
- DATO: 1 byte con el código de error.

El valor 0x00 indica un error de XOR en el paquete. En este caso, también se agrega el paquete con el XOR erróneo y luego el XOR esperado.

El valor 0x01 indica que el comando es desconocido para la placa destinataria.

Cualquier otro valor, indica un error que depende de la placa y el comando enviado.

Respuesta al comando

Único comando sin respuesta directa. En caso de error de XOR debería retransmitir el paquete o tomar alguna otra acción ante otro tipo de error.

A.3. Comandos específicos

Cada grupo de placas tiene comandos propios y específicos dependiendo de la función que deban desempeñar en el sistema. Existen grupos con comandos predefinidos y cada uno se trata en las secciones como se detalla en el listado. Los comandos específicos para cada grupo deben ser desde 0x40 hasta el valor 0x7E.

- *MAIN CONTROLLER* - sección A.4
- *DC MOTOR* - sección A.5
- *SERVO MOTOR* - sección A.6
- *DISTANCE SENSOR* - sección A.7
- *BATTERY CONTROLLER* - sección A.8
- *TRASH BIN* - sección A.9

A.4. MAIN CONTROLLER

El controlador principal no posee comandos específicos. El identificador de grupo es 0x0X.

A.5. DC MOTOR

Comandos específicos del controlador de velocidad de motor de corriente continua. El identificador de grupo es 0x1X.

A.5.1. SET DIRECTION

Establece el sentido de giro del motor.

Comando enviado

- COMANDO: 0x40
- DATO: 0x00 para sentido horario ó 0x01 para sentido anti-horario.

Respuesta al comando

- COMANDO: 0xC0
- DATO: vacío

A.5.2. SET DC SPEED

Establece la velocidad del motor en cuentas del encoder por segundo.

Comando enviado

- COMANDO: 0x41
- DATO: 0x00 para sentido horario ó 0x01 para sentido anti-horario. Número entero de 16 bits con signo, que representa la velocidad en cuentas por segundos. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC1
- DATO: vacío

A.5.3. SET ENCODER

Establece la cantidad de cuentas históricas del encoder.

Comando enviado

- COMANDO: 0x42
- DATO: Número entero de 32 bits con signo, con el valor para establecer en el histórico del encoder. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC2
- DATO: vacío

A.5.4. GET ENCODER

Obtener la cantidad de cuentas históricas del encoder.

Comando enviado

- COMANDO: 0x43
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC3
- DATO: Número entero de 32 bits con signo, que representa el valor histórico del encoder. Ver apartado A.10 para mayor información.

A.5.5. RESET ENCODER

Reestablecer las cuentas históricas a cero.

Comando enviado

- COMANDO: 0x44
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC4
- DATO: vacío

A.5.6. SET ENCODER TO STOP

Establece cuántas cuentas debe girar hasta detenerse.

Comando enviado

- COMANDO: 0x45
- DATO: Número entero de 16 bits con signo, que representa la cantidad de cuentas del encoder restantes para que el motor se detenga. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC5
- DATO: vacío

A.5.7. GET ENCODER TO STOP

Obtener la cantidad de las cuentas restantes que quedan por realizar hasta detenerse.

Comando enviado

- COMANDO: 0x46
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC6
- DATO: Número entero de 16 bits con signo, que representa la cantidad de cuentas del encoder restantes para detener el motor. Ver apartado A.10 para mayor información.

A.5.8. DONT STOP

Deshace los comandos A.5.6 y A.5.7, deshabilita el conteo de cuentas para frenar y sigue en el estado actual.

Comando enviado

- COMANDO: 0x47
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC7
- DATO: vacío

A.5.9. MOTOR CONSUMPTION

Consulta sobre el consumo de corriente actual del motor.

Comando enviado

- COMANDO: 0x48
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC8
- DATO: Número entero positivo de 16 bits en el rango desde 0x0000 hasta 0x03FF, que representa el consumo de corriente promedio del último segundo. Ver apartado A.10 para mayor información.

A.5.10. MOTOR STRESS ALARM

Indica al controlador principal que hay un consumo de corriente extremo en el motor, posiblemente un atasco del motor o de la rueda.

Comando enviado

- COMANDO: 0x49
- DATO: Número entero positivo de 16 bits en el rango desde 0x0000 hasta 0x03FF, que representa el consumo de corriente antes que sonó la alarma. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC9
- DATO: vacío

A.5.11. MOTOR SHUT DOWN ALARM

Indica al controlador principal que el motor ha sido apagado debido al alto consumo. Enviado luego de sucesivos avisos del comando A.5.10.

Comando enviado

- COMANDO: 0x4A
- DATO: Número entero positivo de 16 bits en el rango desde 0x0000 hasta 0x03FF, que representa el consumo antes que sonó la alarma. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xCA
- DATO: vacío

A.5.12. GET DC SPEED

Obtiene la velocidad del motor en cuentas del encoder por segundo.

Comando enviado

- COMANDO: 0x4B
- DATO: vacío

Respuesta al comando

- COMANDO: 0xCB
- DATO: 0x00 para sentido horario ó 0x01 para sentido anti-horario. Número entero de 16 bits con signo, que representa la velocidad en cuentas por segundos. Ver apartado A.10 para mayor información.

A.6. SERVO MOTOR

Comandos específicos del controlador de servomotores. El identificador de grupo es 0x2X.

A.6.1. SET POSITION

Determina la posición en la que debe colocarse el servo motor indicado.

Comando enviado

- COMANDO: 0x40
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo al que se le aplicará la posición. Valor entre 0x00 y 0xB4 que representa el rango de 0° a 180° con 1° de presición.

Respuesta al comando

- COMANDO: 0xC0
- DATO: vacío

A.6.2. SET ALL POSITIONS

Determina las posiciones en la que deben colocarse cada uno de los servomotores.

Comando enviado

- COMANDO: 0x41
- DATO: Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para cada uno de los servos conectados al controlador. Cada valor representa el rango de 0° a 180° con 1° de presión.

Respuesta al comando

- COMANDO: 0xC1
- DATO: vacío

A.6.3. GET POSITION

Obtiene la última posición del servomotor indicado.

Comando enviado

- COMANDO: 0x42
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo del que se requiere la posición.

Respuesta al comando

- COMANDO: 0xC2
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo del que se requirió la posición. Valor entre 0x00 y 0xB4 que representa el rango de 0° a 180° con 1° de presión.

A.6.4. GET ALL POSITIONS

Obtiene las últimas posiciones de todos los servomotor conectados al controlador.

Comando enviado

- COMANDO: 0x43
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC3
- DATO: Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para cada uno de los servos conectados al controlador. Cada valor representa el rango de 0° a 180° con 1° de presión.

A.6.5. SET SERVO SPEED

Determina la velocidad a la que el servomotor indicado llegará a la posición.

Comando enviado

- COMANDO: 0x44
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo al que se le aplicará la velocidad. Valor entre 0x00 y 0xB4, velocidad en grados por segundo.

Respuesta al comando

- COMANDO: 0xC4
- DATO: vacío

A.6.6. SET ALL SPEEDS

Determina las velocidades a la que cada uno de los servomotores llegará a la posición indicada.

Comando enviado

- COMANDO: 0x45
- DATO: Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para cada uno de los servos conectados al controlador. Cada valor representa a la velocidad en grados por segundo.

Respuesta al comando

- COMANDO: 0xC5
- DATO: vacío

A.6.7. GET SERVO SPEED

Obtiene la velocidad asignada al servomotor indicado.

Comando enviado

- COMANDO: 0x46
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo del que se requiere la velocidad.

Respuesta al comando

- COMANDO: 0xC6
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo del que se requirió la velocidad. Valor entre 0x00 y 0xB4, velocidad en grados por segundo.

A.6.8. GET ALL SPEEDS

Obtiene las velocidades de cada uno de los servomotores conectados al controlador.

Comando enviado

- COMANDO: 0x47
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC7
- DATO: Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para cada uno de los servos conectados al controlador. Cada valor representa a la velocidad en grados por segundo.

A.6.9. FREE SERVO

Deja de aplicar fuerza sobre el servo indicado.

Comando enviado

- COMANDO: 0x48
- DATO: Valor de 0x00 a 0x04 que determina el *ID* del servo a liberar.

Respuesta al comando

- COMANDO: 0xC8
- DATO: vacío

A.6.10. FREE ALL SERVOS

Deja de aplicar fuerza sobre cada uno de los servomotores conectados al controlador.

Comando enviado

- COMANDO: 0x49
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC9
- DATO: vacío

A.6.11. GET STATUS

Obtiene el estado de cada uno de los switches conectados al controlador.

Comando enviado

- COMANDO: 0x4A
- DATO: vacío

Respuesta al comando

- COMANDO: 0xCA
- DATO: Valor de 0x00 a 0x7F donde cada bit representa el estado del switch con ese *ID*. Si $2^{ID} = 1$ entonces el switch con *ID* está en un estado lógico alto. Si $2^{ID} = 0$ entonces el switch con *ID* está en un estado lógico bajo.

A.6.12. ALARM ON STATE

Establece si se desea o no recibir una alarma ante cierto cambio de estado en el switch indicado. Puede ser ante cualquier cambio o sobre un flanco ascendente o descendente.

Comando enviado

- COMANDO: 0x4B
- DATO: Valor de 0x00 a 0x05 con el *ID* del switch que se está configurando. Valor entre 0x00 y 0x03 con el tipo de cambio ante el cual generar la alarma. Con un 0x00 ignora cualquier cambio en el switch. Se utiliza 0x01 para que sea ante un flanco ascendente y 0x02 para que sea sólo ante un flanco descendente y 0x03 para que cualquier cambio en el switch genere el mensaje (opcional).

Respuesta al comando

- COMANDO: 0xCB
- DATO: vacío

A.6.13. SWITCH ALARM

Comando enviado desde la placa a la controladora principal informando que fue satisfecha la condición de la alarma.

Comando enviado

- COMANDO: 0x4C
- DATO: Valor de 0x00 a 0x05 con el *ID* del switch que provocó el comando. Valor entre 0x00 y 0x03 con el tipo de cambio configurado y el estado actual, 0x00 para un estado bajo y 0x01 para un estado alto.

Respuesta al comando

- COMANDO: 0xCC
- DATO: vacío

A.7. DISTANCE SENSOR

Comandos específicos del controlador de sensores de distancia, telémetros o sensores de piso. También puede estar presente un sensor de ultrasonido o un switch.

El identificador de grupo es 0x3X.

A.7.1. ON DISTANCE SENSOR

Enciende el sensor de distancia indicado.

Comando enviado

- COMANDO: 0x40
- DATO: Valor de 0x00 a 0x05 que representa el *ID* del sensor a encender. El *ID* 0x05 hace referencia al led de la placa si está presente, en caso contrario se ignora.

Respuesta al comando

- COMANDO: 0xC0
- DATO: vacío

A.7.2. OFF DISTANCE SENSOR

Apaga el sensor de distancia indicado.

Comando enviado

- COMANDO: 0x41
- DATO: Valor de 0x00 a 0x05 que representa el *ID* del sensor a apagar. El *ID* 0x05 hace referencia al sensor de ultrasonido o switch conectado a la placa.

Respuesta al comando

- COMANDO: 0xC1
- DATO: vacío

A.7.3. SET DISTANCE SENSORS MASK

Habilita o deshabilita cada uno de los sensores de distancia conectados al controlador para las próximas lecturas. Permite identificar los sensores a los que se deberá tener en cuenta para futuras lecturas.

Comando enviado

- COMANDO: 0x42
- DATO: Valor de 0x00 a 0x3F donde cada bit representa el *ID* del sensor a habilitar o deshabilitar. Si $2^{ID} = 1$ entonces el sensor *ID* está habilitado. Si $2^{ID} = 0$ entonces el sensor *ID* está deshabilitado.

Respuesta al comando

- COMANDO: 0xC2
- DATO: vacío

A.7.4. GET DISTANCE SENSORS MASK

Obtiene el estado de habilitación de cada uno de los sensores de distancia conectados al controlador para las próximas lecturas.

Comando enviado

- COMANDO: 0x43
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC3
- DATO: Valor de 0x00 a 0x3F donde cada bit representa el *ID* del sensor a habilitar o deshabilitar. Si $2^{ID} = 1$ entonces el sensor *ID* está habilitado. Si $2^{ID} = 0$ entonces el sensor *ID* está deshabilitado.

A.7.5. GET VALUE

Obtiene el valor promedio de la entrada de los sensores indicados.

Comando enviado

- COMANDO: 0x44
- DATO: Valor de 0x00 a 0x3F donde cada bit representa el *ID* del sensor del cual obtener la lectura.

Respuesta al comando

- COMANDO: 0xC4
- DATO: Valor de 0x00 a 0x3F donde cada bit representa el *ID* del sensor del cual proviene el la lectura de distancia. Secuencia de números enteros positivos de 16 bits en el rango desde 0x0000 hasta 0x03FF, con el valor de la lectura que representa la distancia al objeto. En la secuencia de números el orden está dado de izquierda a derecha comenzando por el bit menos significativo.

En el caso del sensor de ultrasonido el rango es desde 0x0000 hasta 0x7594 que representa la mínima y máxima lectura del sensor.

En el caso del switch, un estado lógico bajo se lee como 0x0000 y un estado lógico alto se lee como 0xFFFF.

Ver apartado A.10 para mayor información sobre la codificación de números.

A.7.6. GET ONE VALUE

Obtiene el valor de la entrada del sensor indicado. Igual al comando A.7.5 pero sin realizar un promedio de lecturas.

Comando enviado

- COMANDO: 0x45
- DATO: Valor de 0x00 a 0x3F donde cada bit representa el *ID* del sensor del cual obtener la lectura.

Respuesta al comando

- COMANDO: 0xC5
- DATO: Valor de 0x00 a 0x3F donde cada bit representa el *ID* del sensor del cual proviene el la lectura de distancia. Secuencia de números enteros positivos de 16 bits en el rango desde 0x0000 hasta 0x03FF, con el valor de la lectura que representa la distancia al objeto. En la secuencia de números el orden está dado de izquierda a derecha comenzando por el bit menos significativo.

En el caso del sensor de ultrasonido el rango es desde 0x0000 hasta 0x7594 que representa la mínima y máxima lectura del sensor.

En el caso del switch, un estado lógico bajo se lee como 0x0000 y un estado lógico alto se lee como 0xFFFF.

Ver apartado A.10 para mayor información sobre la codificación de números.

A.7.7. ALARM ON STATE

Cuando un switch está presente en el ID: 0x05, establece si se desea o no recibir una alarma ante cierto cambio de estado en el mismo. Puede ser ante cualquier cambio o sobre un flanco ascendente o descendente.

Comando enviado

- COMANDO: 0x46
- DATO: Valor entre 0x00 y 0x03 con el tipo de cambio ante el cual generar la alarma. Con un 0x00 ignora cualquier cambio en el switch. Se utiliza 0x01 para que sea ante un flanco ascendente y 0x02 para que sea sólo ante un flanco descendente y 0x03 para que cualquier cambio en el switch genere el mensaje (opcional).

Respuesta al comando

- COMANDO: 0xC6
- DATO: vacío

A.7.8. SWITCH ALARM

Comando enviado desde la placa a la controladora principal informando que fue satisfecha la condición de la alarma.

Comando enviado

- COMANDO: 0x47
- DATO: Valor entre 0x00 y 0x03 con el tipo de cambio configurado y el estado actual, 0x00 para un estado bajo y 0x01 para un estado alto.

Respuesta al comando

- COMANDO: 0xC7
- DATO: vacío

A.8. BATTERY CONTROLLER

Comandos específicos del controlador de carga y consumo de la batería. El identificador de grupo es 0x4X.

A.8.1. ENABLE

Habilita la alimentación del robot mediante la batería.

Comando enviado

- COMANDO: 0x40
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC0
- DATO: vacío.

A.8.2. DISABLE

Deshabilita la alimentación del robot mediante la batería.

Comando enviado

- COMANDO: 0x41
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC1
- DATO: vacío.

A.8.3. GET BATTERY VALUE

Obtiene el valor de la entrada de la batería.

Comando enviado

- COMANDO: 0x42
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC2
- DATO: Número entero positivo de 16 bits, en el rango desde 0x0000 hasta 0x03FF, que representa la lectura de la tensión en la batería. Ver apartado A.10 para mayor información.

A.8.4. BATTERY FULL ALARM

Mensaje enviado desde el controlador de la batería informando que se ha completado la carga de la misma.

Comando enviado

- COMANDO: 0x43
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC3
- DATO: vacío

A.8.5. SET BATTERY EMPTY VALUE

Establece el valor de la batería para ser tomado como crítico.

Comando enviado

- COMANDO: 0x44
- DATO: Número entero positivo de 16 bits, en el rango desde 0x0000 hasta 0x03FF, que representa la lectura de la tensión en la batería. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC4
- DATO: vacío

A.8.6. BATTERY EMPTY ALARM

Mensaje enviado desde el controlador de batería informando que el voltaje llegó a un valor crítico.

Comando enviado

- COMANDO: 0x45
- DATO: Número entero positivo de 16 bits, en el rango desde 0x0000 hasta 0x03FF, que representa la lectura de la tensión en la batería. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC5
- DATO: vacío

A.8.7. SET FULL BATTERY VALUE

Establece el valor de batería para ser tomado como carga completa.

Comando enviado

- COMANDO: 0x46
- DATO: Número entero positivo de 16 bits, en el rango desde 0x0000 hasta 0x03FF, que representa la lectura de la tensión a ser tomada como carga completa de la batería. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC6
- DATO: vacío

A.9. TRASH BIN

Comandos específicos del controlador de carga en el cesto de basura. El identificador de grupo es 0x5X.

A.9.1. GET TRASH BIN VALUE

Obtiene el valor que representa que tan lleno está el cesto interno de basura.

Comando enviado

- COMANDO: 0x40
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC0
- DATO: Número entero positivo de 16 bits, en el rango desde 0x0000 hasta 0x03FF, que representa que tan lleno está el cesto interno de basura. Ver apartado A.10 para mayor información.

A.9.2. BIN FULL ALARM

Mensaje enviado desde el controlador del cesto de basura informando que se ha completado y debe ser descargado.

Comando enviado

- COMANDO: 0x41
- DATO: vacío

Respuesta al comando

- COMANDO: 0xC1
- DATO: vacío

A.9.3. SET FULL BIN VALUE

Establece el valor para el cual se envía la alarma A.9.2 indicando un nivel alto en el cesto de basura.

Comando enviado

- COMANDO: 0x42
- DATO: Número entero positivo de 16 bits, en el rango desde 0x0000 hasta 0x03FF, que representa la lectura del nivel del cesto de basura. Ver apartado A.10 para mayor información.

Respuesta al comando

- COMANDO: 0xC2
- DATO: vacío

A.10. Codificación de los valores enviados

Para mantener la compatibilidad entre los distintos agentes dentro de la cadena de transmisión de datos, los números enteros a través del protocolo son codificados bajo el formato *little-endian* con elementos atómicos de 8 bits.

Por ejemplo, si el valor 1023, codificado en un entero de 16 bits en hexadecimal es 0x03FF, entonces se envía primero el byte 0xFF y luego el 0x03. Si fuera el caso del valor 1193046, que codificado en un entero de 32 bits, cuyo valor en hexadecimal es 0x00123456, se envía primero el byte 0x56, luego el 0x34, 0x12 y por último, 0x00.

El envío de números con punto flotante no está previsto en el protocolo.

A.11. Ejemplos

Se detallan paquetes de ejemplo para una mejor comprensión del protocolo.

A.11.1. Ejemplo 1

En el cuadro 38 mostramos un paquete enviado desde el controlador principal con *ID* igual a 0, a una placa controladora de motor de corriente continua con *ID* igual a 1, estableciendo el sentido de giro anti-horario. El campo *XOR* es 0x55, valor que corresponde con el resultado de realizar la operación *XOR* entre todos los bytes del paquete.

05	11	00	40	01	55
----	----	----	----	----	----

Cuadro 38: Paquete de datos del ejemplo 1

A.11.2. Ejemplo 2

En el cuadro 39 mostramos un paquete enviado desde el controlador de baterías con *ID* igual a 2, al controlador principal con *ID* igual a 0, informando que la batería se encuentra con un valor crítico de 0x036B. El campo *XOR* es 0x49, valor que corresponde con el resultado de realizar la operación *XOR* entre todos los bytes del paquete.

06	00	62	45	6B	03	49
----	----	----	----	----	----	----

Cuadro 39: Paquete de datos del ejemplo 2

B. Código fuente

En este apartado mostramos el código fuente usado para cada una de las placas controladoras del proyecto. Pequeños cambios en el *ID* de grupo y placa fueron necesarios para poder lograr que sean únicas dentro de la cadena de comunicación.

B.1. Placa genérica

A la placa genérica la pensamos para testeo y como punto de partida de nuevas implementaciones, agregamos el código que usamos como base para la programación de las otras placas.

```
#define CARD_GROUP      MOTOR_DC      // Ver protocol.h
#define CARD_ID          0             // Valor entre 0 y E

// Descripcion de la placa
#define DESC              "PLACA GENERICA - 1.0" // Maximo DATA_SIZE bytes

/* Modulo Generico - main.c
 * PIC16F88 - MAX232 - GENERICO
 *
 *          PIC16F88
 *
 *          -----
 *          -|RA2/AN2/CVREF/VREF      RA1/AN1|- LED
 *          -|RA3/AN3/VREF+/C1OUT    RAO/ANO|-_
 *          LED -|RA4/AN4/TOCKI/C2OUT RA7/OSC1/CLKII|- XT CLOCK pin1, 27pF to GND
 *          RST/ICD2:MCLR -|RA5/MCLR/VPP   RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *          GND -|VSS                  VDD|- +5v
 *          -|RB0/INT/CCP1           RB7/AN6/PGD/T1OSI|- ICD2:PGD/
 *          -|RB1/SDI/SDA             RB6/AN5/PGC/T1SO/T1CKII|- ICD2:PGC/
 *          MAX232:R1OUT -|RB2/SD0/RX/DT   RB5/SS/TX/CK|- MAX232:T1IN
 *          -|RB3/PGM/CCP1           RB4/SCK/SCL|-_
 *          '-----',
 */
#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// Led
#bit led1=porta.1
#bit led2=porta.4

// MAX232
#bit tx=portb.5
#bit rx=portb.2

#include <.../protocolo/src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica
```

```

char buffer[MAX_BUFFER_SIZE]; // Buffer de recepcion de comandos
int buffer_write; // Indice de escritura
int buffer_read; // Indice de lectura
int data_length; // Largo de los datos en el buffer

struct command_t command; // Comando parseado
struct command_t response; // Respuesta

/** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecula el comando

**/


void init()
{
    // Inicializa puertos
    set_tris_a(0b1100101);
    set_tris_b(0b1100110);

    // Variable para hacer el reset
    reset = false;

    return;
}

void main()
{
    // Placa Generica - Implementacion del protocolo
    init();

    // Init del protocol
    initProtocol();

    // FOREVER
    while(true)
    {
        // Hace sus funciones...

        // Protocolo
        runProtocol(&command);
    }

    return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
    int crc, i, len;

    // Calculo del CRC
    crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

    // CRC ok?
    if (cmd->crc != crc)
    {
        // Creo respuesta de error
        response.len = MIN_LENGTH + cmd->len + 2 + 1;
        response.to = cmd->from;
        response.from = THIS_CARD;
        response.cmd = COMMON_ERROR;
        response.data[0] = 0x00;
        // Agrego el paquete que contiene el error de CRC
        response.data[1] = cmd->len;
        response.data[2] = cmd->to;
        response.data[3] = cmd->from;
        response.data[4] = cmd->cmd;
        // Campo data
        len = cmd->len - MIN_LENGTH;
        for (i = 0; i < len; i++)
            response.data[5 + i] = (cmd->data)[i];
        // CRC erroneo
    }
}

```

```

        response.data[5 + len] = cmd->crc;
        // CRC esperado
        response.data[5 + len + 1] = crc;
        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

        crcOK = false;
        return;
    }

    crcOK = true;

    // Minimo todos setean esto
    response.len = MIN_LENGTH;
    response.to = cmd->from & 0x77;
    response.from = THIS_CARD;
    response.cmd = cmd->cmd | 0x80;

    switch (cmd->cmd)
    {
        // Comandos comunes
        case COMMON_INIT:
            init();
            // Enviar la descripcion de la placa en texto plano
            strcpy(response.data, DESC);
            response.len += strlen(response.data);
            break;
        case COMMON_RESET:
            // Enviar la descripcion de la placa en texto plano
            strcpy(response.data, DESC);
            response.len += strlen(response.data);
            // Reset!
            reset = true;
            break;
        case COMMON_PING:
            // No hace falta hacer mas nada
            break;
        case COMMON_ERROR:
            // Por ahora se ignora el comando
            break;

        /* Comandos especificos */

        case 0x40:
            /*
             :DATO:
             -
             :RESP:
             -
             */
            break;
        default:
            response.len++;
            response.cmd = COMMON_ERROR;
            response.data[0] = 0x01; // Comando desconocido
            break;
    }

    // CRC de la respuesta
    response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

    return;
}

```

B.2. Archivo *protocolo.c*

Código fuente común que genera un buffer cíclico con los paquetes entrantes, parsea el buffer y llama a la función que analiza al paquete según la implementación propia para cada placa.

```
#include <../src/protocol.h>
```

```

short reset;
short crcOK;
short sendResponse;

char buffer[MAX_BUFFER_SIZE];
int buffer_write;
int buffer_read;
int data_length;

// Comando parseado
struct command_t command;
// Respuesta
struct command_t response;

// Interrupcion del RS232
#define INT_RDA
void RS232()
{
    disable_interrupts(INT_RDA);
    // Un nuevo dato...
    buffer[buffer_write++] = getc();
    data_length++;
    if (buffer_write == MAX_BUFFER_SIZE)
        buffer_write -= MAX_BUFFER_SIZE;
    enable_interrupts(INT_RDA);
    return;
}

/* Envia los datos por el pto serial */
void initProtocol()
{
    // Variables de comunicacion
    buffer_write = 0;
    buffer_read = 0;
    data_length = 0;
    crcOK = false;

    // Interrupcion Rcv
    enable_interrupts(INT_RDA);

    // Habilito las interrupciones
    enable_interrupts(GLOBAL);
}

/* Envia los datos por el pto serial */
void send(struct command_t * cmd)
{
    int i, len;

    len = cmd->len - 4;
    putc(cmd->len);
    putc(cmd->to);
    putc(cmd->from);
    putc(cmd->cmd);

    for (i = 0; i < len; i++)
    {
        putc((cmd->data)[i]);
    }

    // Enviar el CRC
    putc(cmd->crc);

    return;
}

void runProtocol(struct command_t * cmd)
{
    // Analiza el buffer
    if (buffer[buffer_read] < data_length)
    {
        data_length -= buffer[buffer_read] + 1;
}

```

```

cmd->len = buffer[buffer_read++];

if (buffer_read == MAX_BUFFER_SIZE)
    buffer_read = 0;

cmd->to = buffer[buffer_read++];

if (buffer_read == MAX_BUFFER_SIZE)
    buffer_read = 0;

cmd->from = buffer[buffer_read++];

if (buffer_read == MAX_BUFFER_SIZE)
    buffer_read = 0;

cmd->cmd = buffer[buffer_read++];

if (buffer_read == MAX_BUFFER_SIZE)
    buffer_read = 0;

// Obtiene el campo DATA
if ((buffer_read + cmd->len - MIN_LENGTH) > MAX_BUFFER_SIZE)
{
    // DATA esta partido en el buffer ciclico
    memcpy(cmd->data, buffer + buffer_read, MAX_BUFFER_SIZE - buffer_read);
    memcpy(cmd->data + MAX_BUFFER_SIZE - buffer_read, buffer,
           cmd->len - MIN_LENGTH - MAX_BUFFER_SIZE + buffer_read);
} else {
    // DATA esta continuo
    memcpy(cmd->data, buffer + buffer_read, cmd->len - MIN_LENGTH);
}

buffer_read += cmd->len - MIN_LENGTH;
if (buffer_read >= MAX_BUFFER_SIZE)
    buffer_read -= MAX_BUFFER_SIZE;

cmd->crc = buffer[buffer_read++];

if (buffer_read == MAX_BUFFER_SIZE)
    buffer_read = 0;

sendResponse = true;

// Soy el destinatario?
if (cmd->to == THIS_CARD)
{
    // Ejecuta el comando
    doCommand(cmd);
} else // Es broadcast?
    if (((cmd->to & 0xF0) == 0xFO)
{
    // Ejecuta el comando
    doCommand(cmd);

    if (crcOK == true)
    {
        // Envia la respuesta
        send(&response);
        // Envia nuevamente el comando recibido
        response = *cmd;
    }
} else // Es broadcast para mi grupo?
    if (((cmd->to & 0x0F) == 0x0F) &&
        ((cmd->to & 0xFO) == THIS_GROUP))
{
    // Ejecuta el comando
    doCommand(cmd);
#endif RESEND_GROUP_BROADCAST
    if (crcOK == true)
    {
        // Envia la respuesta
        send(&response);
        // Envia nuevamente el comando recibido
        response = *cmd;
    }
}

```

```

        }
#endif
    } else {
        response = *cmd;
    }

    // Envia la respuesta?
    if (sendResponse == true)
    {
        send(&response);
    }

}

// Reset del micro
if (reset == true)
{
    reset_cpu();
}

int generate_8bit_crc(char* data, int length, int pattern)
{
    int crc_byte, i;

    crc_byte = data[0];

    for (i = 1; i < length; i++)
        crc_byte ^= data[i];

    return crc_byte;
}

```

B.3. Placa controladora de motor de dc

Código fuente de la placa controladora de motor de corriente continua.

```

//CCS PCM V4.023 COMPILER

#define CARD_GROUP      MOTOR_DC      // Ver protocol.h
#define CARD_ID         0            // Valor entre 0 y E

// Descripcion de la placa
#define DESC           "CONTROL MOTOR DC 1.0" // Maximo DATA_SIZE bytes

/* Modulo Motor - main.c
 * PIC16F88 - MAX232 - L298 - MR-2-60-FA
 *
 *          PIC16F88
 *
 *          .-----.
 *          VREF -|RA2/AN2/CVREF/VREF      RA1/AN1|- MOTOR:CHA_B
 *          LED -|RA3/AN3/VREF+/C1OUT      RAO/ANO|- L298:SEN
 *          LED -|RA4/AN4/TOCKI/C2OUT      RA7/OSC1/CLKII|- XT CLOCK pin1, 27pF to GND
 *          RST/ICD2:MCLR -|RA5/MCLR/VPP   RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *          GND -|VSS                  VDD|- +5v
 *          L298:ENABLE -|RB0/INT/CCP1     RB7/AN6/PGD/T1OSI|- ICD2:PGD
 *          MOTOR:IDX -|RB1/SDI/SDA      RB6/AN5/PGC/T1SO/T1CKI|- ICD2:PGC/MOTOR:CHA_A
 *          MAX232:R1OUT -|RB2/SDO/RX/DT   RB5/SS/TX/CK|- MAX232:T1IN
 *          L298:INPUT_B -|RB3/PGM/CCP1   RB4/SCK/SCL|- L298:INPUT_A
 *          ICD2:PGM   '-----'
 *
 */

#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

```

```

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// MAX232
#bit tx=portb.5
#bit rx=portb.2

// Led
#bit led1=porta.3
#bit led2=porta.4

// L298
#bit inputA=portb.4
#bit inputB=portb.3
#bit enable=portb.0
#bit sensor=porta.0

// Motor inputs
#bit motorIDX=portb.1
#bit channel1A=portb.6
#bit channel1B=porta.1

#include <../../protocolo/src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE]; // Buffer de recepcion de comandos
int buffer_write; // Indice de escritura
int buffer_read; // Indice de lectura
int data_length; // Largo de los datos en el buffer

struct command_t command; // Comando parseado
struct command_t response; // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecuta el comando

/**/

#define MAX_CONSUMPTION 150
#define MAX_CONSUMPTION_COUNT 5

// Correcion de la cantidad de cuentas por segundo en base al periodo del TMRO
#define INTERVAL_CORRECTION 5

// Girar -> clockwise or unclockwise
// Intercambiar entre el motor derecho y el izquierdo
#define CLOCKWISE 1
#define UNCLOCKWISE -1

// Sentido de giro del motor
signed int turn;
// Cantidad de overflows del TMRO
long tmr0_ticks;
// Valor acumulado del ADC - Consumo aprox
long adc_value;
// Ultimo valor del consumo
long last_consumption;
// Valor de duty del PWM
signed long duty;
// Cantidad de cuentas del encoder medidas por intervalo
signed long counts_real;
// Cantidad de cuentas del encoder esperadas por intervalo

```

```

signed long counts_expected;
// Cantidad de cuentas del encoder historicas (32 bits)
signed int32 counts_total;
// Cantidad de cuentas del encoder restantes para deterner el motor
signed long counts_to_stop;
// Cantidad de cuentas del encoder desde el ultimo intevalo
signed long last_counts;
signed long last_counts2;
// Tengo una cantidad de cuentas para hacer?
short counts_check;
// Corrijo el PWM segun lo esperado?
short correct_duty;
// Indica que hay que enviar una alarma de consumo
short consumption_alarm;
// Cuenta cuantas alarmas se enviaron
int alarm_count;
// Indica que hay que avisar que el motor se detuvo
short shutdown_alarm;
// Indica si se apagaron los motores por el alto consumo
short motor_shutdown;

// Variables temporales
signed int32 * tmp32;
signed long * tmp16;

/* Setea el PWM */
void SetPWM(signed long pwm);

// Interrupcion del Timer0
#INT_RTCC
void Timer0_INT()
{
    // Seteo el valor para que interrumpa cada 6.25ms
    set_timer0(12);

    // Comienza la lectura del ADC
    read_adc(ADC_START_ONLY);

    // Agrego al historico de cuentas el ultimo acumulado
    counts_real = get_timer1();
    counts_total += (counts_real - last_counts2) * turn;
    last_counts2 = counts_real;

    // Tengo una cantidad de cuentas para hacer?
    if (counts_check == 1)
    {
        // Verifico si pasaron las cuentas que se habian pedido
        if (counts_to_stop < 1)
        {
            // Detengo el motor
            counts_check = 0;
            counts_to_stop = 0;
            counts_expected = 0;
            duty = 0;
            SetPWM(duty);
            correct_duty = 0;
            last_counts = 0;
        } else {
            counts_real = get_timer1();
            counts_to_stop -= (counts_real - last_counts); // * turn;
            correct_duty = 1;
            last_counts = counts_real;
        }
    } else {
        correct_duty = 1;
    }

    // Tomo la muestra
    adc_value += read_adc(ADC_READ_ONLY);

    if (++tmr0_ticks == 32)
    {
        // Entra cada 200ms

```

```

// Obtengo la cantidad de cuentas desde la ultima entrada
counts_real = get_timer1();
set_timer1(0);
last_counts = 0;
last_counts2 = 0;
// Promedio el consumo segun la cantidad de tmr0_ticks
last_consumption = adc_value / tmr0_ticks;

if (last_consumption >= MAX_CONSUMPTION)
{
    consumption_alarm = 1;
    if (alarm_count++ == MAX_CONSUMPTION_COUNT)
    {
        motor_shutdown = 1;
        alarm_count = 0;
        consumption_alarm = 0;
        shutdown_alarm = 1;
    }
}

tmr0_ticks = 0;

// Mantengo el consumo promedio desde que arranque y borro el temporal
adc_value = 0;

// Corrijo el PWM segun lo esperado
if ((correct_duty == 1) && (counts_real != counts_expected))
{
    duty += (counts_expected - counts_real) * 5;
    if (duty > 1023L)
        duty = 1023;
    else if (duty < 0)
        duty = 0;
    SetPWM(duty * turn);
} else if ((counts_expected == 0) && (duty != 0)) {
    SetPWM(duty = 0);
}
}

return;
}

void init()
{
    // Inicializa puertos
    set_tris_a(0b11100111);
    set_tris_b(0b11100110);

    // ***ADC***
    setup_port_a(sANO); //|VSS_VREF);
    setup_adc(ADC_CLOCK_INTERNAL);
    set_adc_channel(0);
    setup_adc_ports(sANO);
    // Deberia usar VREF_A2... probar
    setup_vref(VREF_HIGH | 8); // VREF a 2.5V -> no hay cambios...

    // ***PWM***
    setup_ccp1(CCP_PWM);
    // Seteo al PWM con f: 4.88 kHz, duty = 0
    set_pwm1_duty(0);
    setup_timer_2(T2_DIV_BY_4, 255, 1);

    // ***TIMER1 - ENCODER COUNTER***
    // Seteo el Timer1 como fuente externa y sin divisor
    setup_timer_1(T1_EXTERNAL | T1_DIV_BY_1);
    set_timer1(0);

    // ***TMR0 - TIME BASE***
    // Seteo el Timer0 como clock -> dt = 6.25ms
    setup_counters(RTCC_INTERNAL, RTCC_DIV_128);
    set_timer0(12);
    // Interrupcion sobre el Timer0
    enable_interrupts(INT_RTCC);

    // Habilito las interrupciones
}

```

```

enable_interrupts(GLOBAL);

// Variable para hacer el reset
reset = false;

// Sentido de giro del motor
turn = CLOCKWISE;
// Cantidad de overflows del TMRO
tmr0_ticks = 0;
// Valor acumulado del ADC - Consumo aprox
adc_value = 0;
last_consumption = 0;
// Valor de duty del PWM
duty = 0;
// Cantidad de cuentas del encoder esperadas por intervalo
counts_expected = 0;
// Cantidad de cuentas del encoder totales
counts_total = 0;
// Cantidad de cuentas del encoder restantes para deterner el motor
counts_to_stop = 0;
// Cantidad de cuentas del encoder desde el ultimo intevalo
last_counts = 0;
last_counts2 = 0;
// Tengo una cantidad de cuentas para hacer?
counts_check = 0;
// Corrijo el PWM segun lo esperado?
correct_duty = 1;
// Indica que hay que enviar una alarma de consumo
consumption_alarm = 0;
// Cuenta cuantas alarmas se enviaron
alarm_count = 0;
// Indica que hay que enviar una alarma de consumo
shutdown_alarm = 0;
// Indica si se apagaron los motores por el alto consumo
motor_shutdown = 0;

return;
}

/* Setea el duty del PWM segun el valor. Positivo o negativo determina el sentido */
void setPWM(signed long pwm)
{
    if (pwm < 0)
    {
        inputA = 0;
        inputB = 1;
    } else {
        inputA = 1;
        inputB = 0;
    }

    if (motor_shutdown == 1)
    {
        pwm = 0;
    } else {
        pwm = (abs(pwm));

        if (pwm > 1023L)
            pwm = 1023;
    }

    set_pwm1_duty(pwm);
}

return;
}

void main()
{
    // Placa Generica - Implementacion del protocolo
    init();

    // Init del protocol
    initProtocol();
}

```

```

counts_expected = 30;

    // FOREVER
    while(true)
    {
        // Hace sus funciones -> interrupcion

        // Enviar alarma de alto consumo
        if (consumption_alarm == 1)
        {
            /* Indica al controlador principal que hay un consumo extremo en el motor,
            posiblemente un atasco del motor o de la rueda.
            :DATO:
            Numero entero positivo de 16 bits en el rango desde 0x0000 hasta
            0x03FF, que representa el consumo ante el que son la alarma.
            :RESP:
            -
            */
            command.len = MIN_LENGTH + 2;
            command.to = MAIN_CONTROLLER;
            command.from = THIS_CARD;
            command.cmd = DC_MOTOR_MOTOR_STRESS_ALARM;
            // A la posicion 0 dentro de command.data la tomo como signed long *
            tmp16 = (command.data);
            // Le asigno el valor del ultimo consumo del motor
            (*tmp16) = last_consumption;
            command.crc = generate_8bit_crc((char *)(&command), command.len, CRC_PATTERN);
            consumption_alarm = 0;
            // Envio del comando
            send(&command);
        }

        // Enviar aviso de motor apagado
        if (shutdown_alarm == 1)
        {
            /* Indica al controlador principal que el motor ha sido apagado debido al alto
            consumo. Enviado luego de sucesivos avisos del comando DC_MOTOR_MOTOR_STRESS_ALARM.
            :DATO:
            Numero entero positivo de 16 bits en el rango desde 0x0000 hasta
            0x03FF, que representa el consumo ante el que son la alarma.
            :RESP:
            -
            */
            command.len = MIN_LENGTH + 2;
            command.to = MAIN_CONTROLLER;
            command.from = THIS_CARD;
            command.cmd = DC_MOTOR_MOTOR_SHUT_DOWN_ALARM;
            // A la posicion 0 dentro de response->data la tomo como signed long *
            tmp16 = (command.data);
            // Le asigno el valor del ultimo consumo del motor
            (*tmp16) = last_consumption;
            command.crc = generate_8bit_crc((char *)(&command), command.len, CRC_PATTERN);
            shutdown_alarm = 0;
            // Envio del comando
            send(&command);
        }

        // Protocolo
        runProtocol(&command);
    }

    return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
    int crc, i, len;

    // Calculo del CRC
    crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

    // CRC ok?
    if (cmd->crc != crc)

```

```

{
    // Creo respuesta de error
    response.len = MIN_LENGTH + cmd->len + 2 + 1;
    response.to = cmd->from;
    response.from = THIS_CARD;
    response.cmd = COMMON_ERROR;
    response.data[0] = 0x00;
    // Agrego el paquete que contiene el error de CRC
    response.data[1] = cmd->len;
    response.data[2] = cmd->to;
    response.data[3] = cmd->from;
    response.data[4] = cmd->cmd;
    // Campo data
    len = cmd->len - MIN_LENGTH;
    for (i = 0; i < len; i++)
        response.data[5 + i] = (cmd->data)[i];
    // CRC erroneo
    response.data[5 + len] = cmd->crc;
    // CRC esperado
    response.data[5 + len + 1] = crc;
    // CRC de la respuesta
    response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

    crcOK = false;
    return;
}

crcOK = true;

// Minimo todos setean esto
response.len = MIN_LENGTH;
response.to = cmd->from & 0x77;
response.from = THIS_CARD;
response.cmd = cmd->cmd | 0x80;

switch (cmd->cmd)
{
    // Comandos comunes
    case COMMON_INIT:
        init();
        // Enviar la descripcion de la placa en texto plano
        strcpy(response.data, DESC);
        response.len += strlen(response.data);
        break;
    case COMMON_RESET:
        // Enviar la descripcion de la placa en texto plano
        strcpy(response.data, DESC);
        response.len += strlen(response.data);
        // Reset!
        reset = true;
        break;
    case COMMON_PING:
        // No hace falta hacer mas nada
        break;
    case COMMON_ERROR:
        // Por ahora se ignora el comando
        break;

    /* Comandos especificos */

    case DC_MOTOR_SET_DIRECTION:
        /* Seteo del sentido de giro del motor
         * :DATO:
         * 0x00 para sentido horario o 0x01 para sentido anti-horario.
         * :RESP:
         * -
         */
        if (((cmd->data)[0] & 0x01) == 0)
        {
            turn = CLOCKWISE;
        } else {
            turn = UNCLOCKWISE;
        }
        break;
}

```

```

case DC_MOTOR_SET_DC_SPEED:
    /* Seteo de la velocidad del motor en cuentas del encoder por segundo
     :DATO:
     0x00 para sentido horario o 0x01 para sentido anti-horario. Numero
     entero de 16 bits con signo, que representa la velocidad en cuentas
     por segundos.
     :RESP:
     -
     */
    if (((cmd->data)[0] & 0x01) == 0)
    {
        turn = CLOCKWISE;
    } else {
        turn = UNCLOCKWISE;
    }
    // A la posicion 1 dentro de cmd->data la tomo como signed long *
    tmp16 = (cmd->data) + 1;
    // Le asigno el valor de la velocidad ajustada a 1 segundo
    counts_expected = (*tmp16) / INTERVAL_CORRECTION;
    // Habilita el motor
    motor_shutdown = 0;
break;
case DC_MOTOR_SET_ENCODER:
    /* Seteo de la cantidad de cuentas historicas del encoder
     :DATO:
     Numero entero de 32 bits con signo, con el valor para setear en el
     historico del encoder.
     :RESP:
     -
     */
    // A la posicion 0 dentro de cmd->data la tomo como signed int32 *
    tmp32 = (cmd->data);
    // Le asigno el valor de las cuentas historicas
    counts_total = (*tmp32);
break;
case DC_MOTOR_GET_ENCODER:
    /* Obtener la cantidad de cuentas historicas del encoder
     :DATO:
     -
     :RESP:
     Numero entero de 32 bits con signo, que representa el valor historico
     del encoder.
     */
    // A la posicion 0 dentro de response->data la tomo como signed int32 *
    tmp32 = (response.data);
    // Le asigno el valor de las cuentas historicas
    (*tmp32) = counts_total;
    // Corrijo el largo del paquete
    response.len += 4;
break;
case DC_MOTOR_RESET_ENCODER:
    /* Resetear las cuentas historicas a cero
     :DATO:
     -
     :RESP:
     -
     */
    counts_total = 0;
break;
case DC_MOTOR_SET_ENCODER_TO_STOP:
    /* Seteo de cuantas cuentas debe girar hasta detenerse
     :DATO:
     Numero entero de 16 bits con signo, que representa la cantidad de
     cuentas del encoder restantes para que el motor se detenga.
     :RESP:
     -
     */
    // A la posicion 0 dentro de cmd->data la tomo como signed long *
    tmp16 = (cmd->data);
    // Le asigno el valor de la velocidad ajustada a 1 segundo
    counts_to_stop = (*tmp16);
    // Habilito el chequeo de cuentas para detener el motor
    counts_check = 1;
break;

```

```

case DC_MOTOR_GET_ENCODER_TO_STOP:
    /* Obtener la cantidad de las cuentas restantes que quedan por
       realizar hasta detenerse.
    :DATO:
    -
    :RESP:
    Numero entero de 16 bits con signo, que representa la cantidad
    de cuentas del encoder restantes para detener el motor.
    */
    // A la posicion 0 dentro de response->data la tomo como signed long *
    tmp16 = (response.data);
    // Le asigno el valor de la velocidad ajustada a 1 segundo
    (*tmp16) = counts_to_stop;
    // Corrijo el largo del paquete
    response.len += 2;
break;
case DC_MOTOR_DONT_STOP:
    /* Deshace los comandos DC_MOTOR_DONT_STOP y DC_MOTOR_GET_ENCODER_TO_STOP,
       deshabilita el conteo de cuentas para frenar y sigue en el estado actual.
    :DATO:
    -
    :RESP:
    -
    */
    counts_check = 0;
break;
case DC_MOTOR_MOTOR_CONSUMPTION:
    /* Numero entero positivo de 16 bits en el rango desde 0x0000 hasta
       0x03FF, que representa el consumo promedio del ultimo segundo.
    :DATO:
    -
    :RESP:
    -
    */
    // A la posicion 0 dentro de response->data la tomo como signed long *
    tmp16 = (response.data);
    // Le asigno el valor del ultimo consumo del motor
    (*tmp16) = last_consumption;
    // Corrijo el largo del paquete
    response.len += 2;
break;
/*case DC_MOTOR_MOTOR_STRESS_ALARM:
break;
case DC_MOTOR_MOTOR_SHUT_DOWN_ALARM:
break;*/
case DC_MOTOR_GET_DC_SPEED:
    /* Obtiene la velocidad del motor en cuentas del encoder por segundo
    :DATO:
    0x00 para sentido horario o 0x01 para sentido anti-horario. Numero
    entero de 16 bits con signo, que representa la velocidad en cuentas
    por segundos.
    :RESP:
    -
    */
    // Sentido de giro del motor
    if (turn == CLOCKWISE)
    {
        (response.data)[0] = 0x00;
    } else {
        (response.data)[0] = 0x01;
    }
    // A la posicion 1 dentro de response->data la tomo como signed long *
    tmp16 = (response.data) + 1;
    // Le asigno el valor de la velocidad ajustada a 1 segundo
    (*tmp16) = counts_real * INTERVAL_CORRECTION;
    // Corrijo el largo del paquete
    response.len += 2;
break;
default:
    response.len++;
    response.cmd = COMMON_ERROR;
    response.data[0] = 0x01; // Comando desconocido
break;

```

```

    }

    // CRC de la respuesta
    response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

    return;
}

```

B.4. Placa controladora de sensores

Código fuente de la placa controladora de sensores.

```

//CCS PCM V4.023 COMPILER

#define CARD_GROUP      DISTANCE_SENSOR // Ver protocol.h
#define CARD_ID          1                // Valor entre 0 y E

// ID:0 -> Placa de telemetros
// ID:1 -> Placa de sensores de piso y sensor de ultrasonido
// ID:2 -> Placa de telemetros

// Descripcion de la placa
#define DESC           "PLACA SENSORES 1.0" // Maximo DATA_SIZE bytes

// Determina el tipo de sensores principales en la placa - CAMBIARLO SEGUN CORRESPONDA
// Posibles valores: FLOOR_SENSORS o TELEMETERS_SENSORS o NONE
// #define SENSORS_TYPE TELEMETERS_SENSORS

// Determina contra que esta conectado el pin TRIGGER - CAMBIARLO SEGUN CORRESPONDA
// Posibles valores: NONE o ULTRASONIC_SENSOR o SWITCH_SENSOR o LED
// #define TRIGGER_TYPE ULTRASONIC_SENSOR
// #define TRIGGER_TYPE SWITCH_SENSOR

#if CARD_ID == 0
    #define SENSORS_TYPE    TELEMETERS_SENSORS
    #define TRIGGER_TYPE   SWITCH_SENSOR
    #define DEFAULT_MASK   0x2F
#elif CARD_ID == 1
    #define SENSORS_TYPE    FLOOR_SENSORS
    #define TRIGGER_TYPE   ULTRASONIC_SENSOR
    #define DEFAULT_MASK   0x37
#elif CARD_ID == 2
    #define SENSORS_TYPE    TELEMETERS_SENSORS
    #define TRIGGER_TYPE   SWITCH_SENSOR
    #define DEFAULT_MASK   0x2F
#endif

// Define el estado logico para prender o apagar los sensores - CAMBIARLO SEGUN CORRESPONDA
#define SENSOR_ON        0
#define SENSOR_OFF       1

#define SAMPLES_DEFAULT 5

// Ancho del pulso que se debe enviar al sensor de ultrasonido como INIT - CAMBIARLO SEGUN CORRESPONDA
#define ULTRASONIC_INIT_PULSE_WIDTH_US 15

// Frame de muestreo. Cada este tiempo se realiza una nueva lectura ver CalculoTMR.xls
#define COMPLEMENT_TIME      12411 // 85ms
// Tiempo necesario para realizar la lectura en los telemetros - CAMBIARLO SEGUN CORRESPONDA
#define TELEMETERS_WAITING_TIME_CYCLES 26792 //62ms
#define TELEMETERS_COMPLEMENT_TIME 51161 //23ms complemento a COMPLEMENT_TIME
// Tiempo necesario para realizar la lectura en los sensores de piso - CAMBIARLO SEGUN CORRESPONDA
#define FLOOR_WAITING_TIME_CYCLES 62411 //5ms
#define FLOOR_COMPLEMENT_TIME 15536 //80ms complemento a COMPLEMENT_TIME
// Tiempo necesario para realizar la lectura del sensor de ultrasonido - CAMBIARLO SEGUN CORRESPONDA
#define ULTRASONIC_WAITING_TIME_CYCLES 45536 //32ms
#define ULTRASONIC_COMPLEMENT_TIME 32411 //53ms complemento a COMPLEMENT_TIME

// Tiempo entre el cambio de canal del ADC y una muestra estable (minimo 10us) - CAMBIARLO SEGUN CORRESPONDA
#define ADC_DELAY         20

/* Modulo Generico - main.c

```

```

* PIC16F88 - MAX232 - SENSORES
*
*          PIC16F88
*
*      SENSE_3 -|RA2/AN2/CVREF/VREF      RA1/AN1|- SENSE_2
*      SENSE_4 -|RA3/AN3/VREF+/C1OUT    RAO/AN0|- SENSE_1
*      SENSE_5 -|RA4/AN4/TOCKI/C2OUT    RA7/OSC1/CLKI|- XT CLOCK pin1, 27pF to GND
*      RST/ICD2:MCLR -|RA5/MCLR/VPP    RA6/OSC2/CLK0|- XT CLOCK pin2, 27pF to GND
*      GND -|VSS                      VDD|- +5v
*      TRIGGER -|RB0/INT/CCP1         RB7/AN6/PGD/T10SI|- ICD2:PGD/SENSOR_5
*      SENSOR_1 -|RB1/SDI/SDA         RB6/AN5/PGC/T10SO/T1CKI|- ICD2:PGC/SENSOR_4
*      MAX232:R1OUT -|RB2/SDO/RX/DT    RB5/SS/TX/CK|- MAX232:T1IN
*      SENSOR_2 -|RB3/PGM/CCP1        RB4/SCK/SCL|- SENSOR_3
*      ,
*
*/
#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// MAX232
#bit tx=portb.5
#bit rx=portb.2

// Pin para el control del sensores de ultrasonido, switch on/off o led
#bit trigger=portb.0
// Led
#bit led1=portb.0
// Pin para switch On/Off
#bit inOnOff=portb.0
// Pin de control del sensor de ultrasonido
#bit ultrasonido=portb.0

// Sensores - Base de los transistores
#bit sensor1=portb.1
#bit sensor2=portb.3
#bit sensor3=portb.4
#bit sensor4=portb.6
#bit sensor5=portb.7

// Sensores - Salida de los sensores (SENSE)
#bit sense1=porta.0
#bit sense2=porta.1
#bit sense3=porta.2
#bit sense4=porta.3
#bit sense5=porta.4

// Estados para la maquina de estados que controla las funciones de la placa
#define STATE_FREE           0
#define STATE_START_READING   1
#define STATE_WAIT_TO_READ    2
#define STATE_READ_VALUES     3
#define STATE_WAITING         4

// Estados para la lectura del sensor de ultrasonido
#define USONIC_STATE_START    0
#define USONIC_STATE_STOP      1

// Posibles conexiones del pin TRIGGER
#define NONE                  (0x00)
#define ULTRASONIC_SENSOR (NONE + 1)

```

```

#define SWITCH_SENSOR      (ULTRASONIC_SENSOR + 1)
#define LED                (SWITCH_SENSOR + 1)

// Posibles sensores principales
#define TELEMETERS_SENSORS (NONE + 1)
#define FLOOR_SENSORS     (TELEMETERS_SENSORS + 1)

// Tiempo maximo para NONE
#define NONE_TIME          65530 //0ms
#define SWITCH_TIME         65530 //0ms
#define LED_TIME            65530 //0ms

#if SENSORS_TYPE == FLOOR_SENSORS
    #if TRIGGER_TYPE == ULTRASONIC_SENSOR
        #define SENSORS_WAITING_TIME ULTRASONIC_WAITING_TIME_CYCLES
        #define WAITING_TIME ULTRASONIC_COMPLEMENT_TIME //Complemento 85ms -> 53ms
    #else
        #define SENSORS_WAITING_TIME FLOOR_WAITING_TIME_CYCLES
        #define WAITING_TIME FLOOR_COMPLEMENT_TIME //Complemento 85ms -> 80ms
    #endif
#elif SENSORS_TYPE == TELEMETERS_SENSORS
    #define SENSORS_WAITING_TIME TELEMETERS_WAITING_TIME_CYCLES
    #define WAITING_TIME TELEMETERS_COMPLEMENT_TIME //Complemento 85ms -> 23ms
#elif SENSORS_TYPE == NONE
    #if TRIGGER_TYPE == ULTRASONIC_SENSOR
        #define SENSORS_WAITING_TIME ULTRASONIC_WAITING_TIME_CYCLES
        #define WAITING_TIME ULTRASONIC_COMPLEMENT_TIME //Complemento 85ms -> 53ms
    #else
        #define SENSORS_WAITING_TIME NONE_TIME
        #define WAITING_TIME COMPLEMENT_TIME //85ms
    #endif
#endif

#include <../src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE]; // Buffer de recepcion de comandos
int buffer_write; // Indice de escritura
int buffer_read; // Indice de lectura
int data_length; // Largo de los datos en el buffer

struct command_t command; // Comando parseado
struct command_t response; // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecuta el comando

*/
// IO como entrada o salida
int trisB_value = 0b00100101;
// Determina el estado actual
int state;
// Vector donde se almacenan los valores de los sensores
unsigned long values[6];
// Cantidad de muestras a tomar por sensor
int samples;
// mascara que ignora ciertos sensores
int sensorMask;
// Determina cuales sensores estan siendo leidos en este instante
int actualReadSensor;
// Determina cuales son los sensores de los que se pide la lectura
int readSensor;
// Bufferea el pedido de sensores de los que se pide la lectura
int bufferedReadSensor;
// Determina el destinatario actual
int actalTO;

```

```

// Informa quien hizo el pedido
int requestFrom;
// Informa de quien vino el ultimo pedido (mientras habia una lectura en curso)
int bufferedFrom;
// Determina el comando actual
int actualCmd;
// Informa cual fue el comando del pedido
int requestCmd;
// Informa cual fue el comando del ultimo pedido (mientras habia una lectura en curso)
int bufferedCmd;
// Estado para el sensor de ultrasonido
short usonic_state;
// Tiempo para el comienzo del pulso del sensor de ultrasonido
unsigned long pulseStart;
// Flag de aviso sobre la interrupcion en TIMER1
short intTMR;
// Tipo de alarma en trigger
int alarmType;
// Flag de alarma en trigger
short triggerAlarm;

/* Habilita los sensores segun corresponda para comenzar la lectura */
void startReading(int sensors);
/* Realiza la lectura sobre los sensores segun corresponda */
void readSensors(int sensors);
/* Interrupcion del TIMER1 */
void int_timer(void);
/* Interrupcion de RBO */
void int_trigger(void);
/* Crea la respuesta sobre la lectura de los sensores */
void sendValues(int to, int cmd, long * values, int sensors);

/* Interrupcion del TIMER1 */
#define INT_TIMER1
void int_timer(void)
{
    // Habilita el flag de aviso que sucedio la interrupcion
    intTMR = 1;
    disable_interrupts(INT_TIMER1);
}

/* Interrupcion de RBO */
#define INT_EXT
void int_trigger(void)
{
    #if TRIGGER_TYPE == ULTRASONIC_SENSOR
        if (usonic_state == USONIC_STATE_START)
        {
            // Tomo el tiempo en que comienza el pulso
            pulseStart = get_timer1();
            // Cambio el tipo de flanco
            ext_int_edge(H_TO_L);
            // Cambio el estado
            usonic_state = USONIC_STATE_STOP;
        } else {
            // Tomo el tiempo y guardo el valor
            values[5] = (get_timer1() - pulseStart)/2;
            // Deshabilita la interrupcion
            disable_interrupts(INT_EXT);
        }
    #elif TRIGGER_TYPE == SWITCH_SENSOR
        triggerAlarm = 1;
    #endif
}

void init()
{
    // Inicializa puertos
    set_tris_a(0b11111111);
    set_tris_b(trisB_value);

    // ***ADC***
    setup_port_a(sANO);
    setup_adc(ADC_CLOCK_INTERNAL);
}

```

```

set_adc_channel(0);
setup_adc_ports(sANO);
setup_vref(VREF_HIGH | 8);

// Seteo el Timer1 como fuente interna
setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);
set_timer1(26786);

// Interrupcion sobre el Timer1
enable_interrupts(INT_TIMER1);

// Seteo el pin RBO - Sensor de ultrasonido :)
ext_int_edge(L_TO_H);
enable_interrupts(INT_EXT);

// Habilito las interrupciones
enable_interrupts(GLOBAL);

// Variable para hacer el reset
reset = false;

// Apaga todos los sensores
sensor1 = SENSOR_OFF;
sensor2 = SENSOR_OFF;
sensor3 = SENSOR_OFF;
sensor4 = SENSOR_OFF;
sensor5 = SENSOR_OFF;

// Inicializa los valores
values[0] = 0x0000;
values[1] = 0x0000;
values[2] = 0x0000;
values[3] = 0x0000;
values[4] = 0x0000;
values[5] = 0x0000;

// Muestras iniciales
samples = SAMPLES_DEFAULT;

// Inicializa la mascara -> todos habilitados (0x3F)
sensorMask = DEFAULT_MASK;

//Determina el estado actual
state = STATE_FREE;

// Sin lectura temprana
readSensor = 0x00;
bufferedReadSensor = 0x00;
actalTO = 0x00;
requestFrom = 0x00;
bufferedFrom = 0x00;
actalCmd = 0x00;
requestCmd = 0x00;
bufferedCmd = 0x00;

alarmType = 0x00;
triggerAlarm = 0;
#if TRIGGER_TYPE == SWITCH_SENSOR
    disable_interrupts(INT_EXT);
#endif

#if TRIGGER_TYPE == LED
    // TRIGGER como escritura
    bit_clear(trisB_value, 0);
    set_tris_b(trisB_value);
#endif

return;
}

void sendAlarm()
{
    triggerAlarm = 0;
    return;
}

```

```

}

void main()
{
    // Placa Generica - Implementacion del protocolo
    init();

    // Init del protocol
    initProtocol();

    // FOREVER
    while(true)
    {
        // Ejecucion de la maquina de estados
        switch (state)
        {
            case STATE_FREE:
                /*
                 * Analiza los paquetes y retransmite o lanza el pedido de lectura
                 */
            #if TRIGGER_TYPE == SWITCH_SENSOR
                // Enviar alarma de trigger
                if (triggerAlarm == 1)
                    sendAlarm();
            #endif
                // Protocolo
                runProtocol(&command);

                if (readSensor != 0x00)
                {
                    // Almacena el pedido sobre los sensores
                    actualReadSensor = readSensor;
                    readSensor = 0x00;
                    actalTO = requestFrom;
                    actalCmd = requestCmd;
                    // Cambio de estado
                    state = STATE_START_READING;
                }
                break;
            case STATE_START_READING:
                /*
                 * Genera un pedido de lectura a los sensores que lo necesiten
                 * y setea los TIMERS para contabilizar los tiempos
                 */
                // Limpio la variable
                bufferedReadSensor = 0x00;

                // Inhabilita los sensores enmascarados
                actualReadSensor &= sensorMask;

                // Pedido de -START-
                startReading(actualReadSensor);

                // Flag de interrupcion
                intTMR = 0;

                // Seteo de TIMERS
                set_timer1(SENSORS_WAITING_TIME);
                enable_interrupts(INT_TIMER1);

                // Cambio de estado
                state = STATE_WAIT_TO_READ;

                break;
            case STATE_WAIT_TO_READ:
                /*
                 * Espera el tiempo necesario para tomar las muestras en los sensores
                 * Recibe nuevos pedidos y los agraga para una proxima lectura
                 */
            #if TRIGGER_TYPE == SWITCH_SENSOR
                // Enviar alarma de trigger
                if (triggerAlarm == 1)

```

```

        sendAlarm();

#endif
// Protocolo
runProtocol(&command);

// Almacena el pedido sobre los sensores si no hay otro ya
if ((readSensor != 0x00) && (bufferedReadSensor == 0x00))
{
    bufferedReadSensor = readSensor;
    bufferedFrom = requestFrom;
    bufferedCmd = requestCmd;
    readSensor = 0x00;
}

// Cambio de estado?
if (intTMR == 1)
{
    // El TIMER1 hizo timeout -> leer sensores
    state = STATE_READ_VALUES;
}

break;
case STATE_READ_VALUES:
/*
 * Toma las muestras en los sensores y envia el paquete de respuesta
 */

// Toma las muestras de los sensores
readSensors(actualReadSensor);

// Mandar paquete de respuesta
sendValues(actalT0, actalCMD, values, actualReadSensor);

// Flag de interrupcion
intTMR = 0;

// Setea el tiempo de espera y pasa al estado de espera
set_timer1(WAITING_TIME);
enable_interrupts(INT_TIMER1);

state = STATE_WAITING;

break;
case STATE_WAITING:
/*
 * Espera a que pase el tiempo de espera entre lecturas para evitar rebotes
 * Recibe nuevos pedidos y los agraga para una proxima lectura
 */

#endif
#if TRIGGER_TYPE == SWITCH_SENSOR
    // Enviar alarma de trigger
    if (triggerAlarm == 1)
        sendAlarm();
#endif
// Protocolo
runProtocol(&command);

// Almacena el pedido sobre los sensores si no hay otro ya
if ((readSensor != 0x00) && (bufferedReadSensor == 0x00))
{
    bufferedReadSensor = readSensor;
    bufferedFrom = requestFrom;
    bufferedCmd = requestCmd;
    readSensor = 0x00;
}

// Cambio de estado?
if (intTMR == 1)
{
    // El TIMER1 hizo timeout
    if (bufferedReadSensor != 0x00)
    {
        // Hay un pedido pendiente y lo carga
        actualReadSensor = bufferedReadSensor;
    }
}

```

```

        bufferedReadSensor = 0x00;
        actalTO = bufferedFrom;
        actalCmd = bufferedCmd;
        // Cambio de estado
        state = STATE_START_READING;
    } else {
        state = STATE_FREE;
    }
}
break;
default:
    init();
    break;
}
}
return;
}

/* Habilita los sensores segun corresponda para comenzar la lectura*/
void startReading(int sensors)
{
    // Sensor1
    if (bit_test(sensors, 0) == 1)
        sensor1 = SENSOR_ON;
    // Sensor2
    if (bit_test(sensors, 1) == 1)
        sensor2 = SENSOR_ON;
    // Sensor3
    if (bit_test(sensors, 2) == 1)
        sensor3 = SENSOR_ON;
    // Sensor4
    if (bit_test(sensors, 3) == 1)
        sensor4 = SENSOR_ON;
    // Sensor5
    if (bit_test(sensors, 4) == 1)
        sensor5 = SENSOR_ON;
    // Sensor6
#ifndef TRIGGER_TYPE == ULTRASONIC_SENSOR
    // Sensor6 -> ULTRASONIC_SENSOR
    if (bit_test(sensors, 5) == 1)
    {
        // Comienza el pulso de habilitacion -> TRIGGER como escritura
        bit_clear(trisB_value, 0);
        set_tris_b(trisB_value);
        // Pin en estado habilitado -> envio del pulso INIT
        trigger = 1;
        delay_us(ULTRASONIC_INIT_PULSE_WIDTH_US);
        trigger = 0;
        // Termina el pulso de habilitacion -> TRIGGER como lectura
        bit_set(trisB_value, 0);
        set_tris_b(trisB_value);
        // Seteo la interrupcion sobre RBO en flanco ascendente
        ext_int_edge(L_TO_H);
        // Seteo el estado actual del pulso del sensor de ultrasonido
        usonic_state = USONIC_STATE_START;
        // Habilita la interrupcion
        enable_interrupts(INT_EXT);
    }
#endif
#ifndef TRIGGER_TYPE == SWITCH_SENSOR
    // Sensor6 -> SWITCH_SENSOR
    if (bit_test(sensors, 5) == 1)
    {
        if (trigger == 1)
            values[5] = 0xFFFF;
        else
            values[5] = 0;
    }
#endif
    return;
}

/* Realiza la lectura sobre los sensores segun corresponda */
void readSensors(int sensors)
{

```

```

int i;
values[0] = 0;
values[1] = 0;
values[2] = 0;
values[3] = 0;
values[4] = 0;

for (i = 0; i < samples; i++)
{
    // Sensor1
    if (bit_test(sensors, 0) == 1)
    {
        // ADC en el pin correcto
        set_adc_channel(0);
        // Espera el tiempo necesario
        delay_us(ADC_DELAY);
        // Toma la muestra
        values[0] += read_adc();
    }
    // Sensor2
    if (bit_test(sensors, 1) == 1)
    {
        // ADC en el pin correcto
        set_adc_channel(1);
        // Espera el tiempo necesario
        delay_us(ADC_DELAY);
        // Toma la muestra
        values[1] += read_adc();
    }
    // Sensor3
    if (bit_test(sensors, 2) == 1)
    {
        // ADC en el pin correcto
        set_adc_channel(2);
        // Espera el tiempo necesario
        delay_us(ADC_DELAY);
        // Toma la muestra
        values[2] += read_adc();
    }
    // Sensor4
    if (bit_test(sensors, 3) == 1)
    {
        // ADC en el pin correcto
        set_adc_channel(3);
        // Espera el tiempo necesario
        delay_us(ADC_DELAY);
        // Toma la muestra
        values[3] += read_adc();
    }
    // Sensor5
    if (bit_test(sensors, 4) == 1)
    {
        // ADC en el pin correcto
        set_adc_channel(4);
        // Espera el tiempo necesario
        delay_us(ADC_DELAY);
        // Toma la muestra
        values[4] += read_adc();
    }
}

values[0] /= samples;
values[1] /= samples;
values[2] /= samples;
values[3] /= samples;
values[4] /= samples;

return;
}

/* Crea la respuesta sobre la lectura de los sensores */
void sendValues(int to, int cmd, long * values, int sensors)
{
    int idx = 1, i;

```

```

signed long * tmp16;
command.len = MIN_LENGTH + 1;
command.to = to;
command.from = THIS_CARD;
command.cmd = cmd;
command.data[0] = sensors;

// Valores de los sensores en command.data segun corresponda
for (i = 0; i < 6; i++)
{
    if (bit_test(sensors, i) == 1)
    {
        // A la posicion 0 dentro de command.data la tomo como signed long *
        tmp16 = (command.data + idx);
        // Le asigno el valor del sensor
        (*tmp16) = values[i];
        idx+=2;
        command.len += 2;
    }
}

command.crc = generate_8bit_crc((char *)(&command), command.len, CRC_PATTERN);
// Envio del comando
send(&command);

return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
    int crc, i, len;
    // Calculo del CRC
    crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

    // CRC ok?
    if (cmd->crc != crc)
    {
        // Creo respuesta de error
        response.len = MIN_LENGTH + cmd->len + 2 + 1;
        response.to = cmd->from;
        response.from = THIS_CARD;
        response.cmd = COMMON_ERROR;
        response.data[0] = 0x00;
        // Agrego el paquete que contiene el error de CRC
        response.data[1] = cmd->len;
        response.data[2] = cmd->to;
        response.data[3] = cmd->from;
        response.data[4] = cmd->cmd;
        // Campo data
        len = cmd->len - MIN_LENGTH;
        for (i = 0; i < len; i++)
            response.data[5 + i] = (cmd->data)[i];
        // CRC erroneo
        response.data[5 + len] = cmd->crc;
        // CRC esperado
        response.data[5 + len + 1] = crc;
        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);
        crcOK = false;
        return;
    }

    crcOK = true;

    // Minimo todos setean esto
    response.len = MIN_LENGTH;
    response.to = cmd->from & 0x77;
    response.from = THIS_CARD;
    response.cmd = cmd->cmd | 0x80;

    switch (cmd->cmd)
    {
        // Comandos comunes

```

```

    case COMMON_INIT:
        init();
        // Enviar la descripcion de la placa en texto plano
        strcpy(response.data, DESC);
        response.len += strlen(response.data);
    break;
    case COMMON_RESET:
        // Enviar la descripcion de la placa en texto plano
        strcpy(response.data, DESC);
        response.len += strlen(response.data);
        // Reset!
        reset = true;
    break;
    case COMMON_PING:
        // No hace falta hacer mas nada
    break;
    case COMMON_ERROR:
        // Por ahora se ignora el comando
    break;

    /* Comandos especificos */

    case DISTANCE_SENSOR_ON_DISTANCE_SENSOR:
        /* Enciende el sensor de distancia indicado.
        :DATO:
        Valor de 0x00 a 0x05 que representa el ID del sensor a encender.
        El ID 0x05 hace referencia al led de la placa si esta presente,
        en caso contrario se ignora.
        :RESP:
        -
        */
        // Enciende los sensores segun corresponda
        if ((cmd->data)[0] == 0)
            // Sensor1
            sensor1 = SENSOR_ON;
        else if ((cmd->data)[0] == 1)
            // Sensor2
            sensor2 = SENSOR_ON;
        else if ((cmd->data)[0] == 2)
            // Sensor3
            sensor3 = SENSOR_ON;
        else if ((cmd->data)[0] == 3)
            // Sensor4
            sensor4 = SENSOR_ON;
        else if ((cmd->data)[0] == 4)
            // Sensor5
            sensor5 = SENSOR_ON;
        #if TRIGGER_TYPE == LED
        else if ((cmd->data)[0] == 5)
            // Led
            led1 = 1;
        #endif
        break;
    case DISTANCE_SENSOR_OFF_DISTANCE_SENSOR:
        /* Apaga el sensor de distancia indicado.
        :DATO:
        Valor de 0x00 a 0x05 que representa el ID del sensor a apagar. El
        ID 0x05 hace referencia al sensor de ultrasonido o switch de la placa.
        :RESP:
        -
        */
        // Apaga los sensores segun corresponda
        if ((cmd->data)[0] == 0)
            // Sensor1
            sensor1 = SENSOR_OFF;
        else if ((cmd->data)[0] == 1)
            // Sensor2
            sensor2 = SENSOR_OFF;
        else if ((cmd->data)[0] == 2)
            // Sensor3
            sensor3 = SENSOR_OFF;
        else if ((cmd->data)[0] == 3)
            // Sensor4
            sensor4 = SENSOR_OFF;

```

```

        else if ((cmd->data)[0] == 4)
            // Sensor5
            sensor5 = SENSOR_OFF;
    #if TRIGGER_TYPE == LED
        else if ((cmd->data)[0] == 5)
            // Led
            led1 = 0;
    #endif
    break;
    case DISTANCE_SENSOR_SET_MASK:
        /* Habilita o deshabilita cada uno de los sensores de distancia conectados al
        controlador. Permite identificar los sensores a los que se deberá tener en
        cuenta para futuras lecturas.
        :DATO:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor a
        habilitar o deshabilitar. Si  $2^{\text{ID}}$  = 1 entonces el sensor ID está habilitado.
        Si  $2^{\text{ID}}$  = 0 entonces el sensor ID está deshabilitado.
        :RESP:
        -
        */
        sensorMask = (cmd->data)[0];
    break;
    case DISTANCE_SENSOR_GET_MASK:
        /* Obtiene el estado de habilitación de cada uno de los sensores de distancia
        conectados al controlador.
        :DATO:
        -
        :RESP:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor a
        habilitar o deshabilitar. Si  $2^{\text{ID}}$  = 1 entonces el sensor ID está habilitado.
        Si  $2^{\text{ID}}$  = 0 entonces el sensor ID está deshabilitado.
        */
        // Envío la máscara de sensores
        response.data[0] = sensorMask;
        // Corrige el largo del paquete
        response.len++;
    break;
    case DISTANCE_SENSOR_GET_VALUE:
        /* Obtiene el valor promedio de la entrada de los sensores indicados.
        :DATO:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual obtener la lectura.
        :RESP:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual proviene la lectura de distancia. Secuencia de números enteros
        positivos de 16 bits en el rango desde 0x0000 hasta 0x03FF, con el valor de
        la lectura que representa la distancia al objeto. En la secuencia de números
        el orden está dado de izquierda a derecha comenzando por el bit menos
        significativo.
        En el caso del sensor de ultrasonido el rango es desde 0x0000 hasta 0x7594
        que representa la mínima y máxima lectura del sensor.
        En el caso del switch, un estado lógico bajo se lee como 0x0000 y un estado
        lógico alto se lee como 0xFFFF.
        */
        readSensor = (cmd->data)[0];
        requestFrom = response.to;
        requestCmd = response.cmd;
        sendResponse = false;
        samples = SAMPLES_DEFAULT;
    break;
    case DISTANCE_SENSOR_GET_ONE_VALUE:
        /* Obtiene el valor de la entrada del sensor indicado. Igual al comando 8.5 pero
        sin realizar un promedio de lecturas.
        :DATO:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual obtener la lectura.
        :RESP:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual proviene la lectura de distancia. Secuencia de números enteros
        positivos de 16 bits en el rango desde 0x0000 hasta 0x03FF, con el valor de
        la lectura que representa la distancia al objeto. En la secuencia de números
        el orden está dado de izquierda a derecha comenzando por el bit menos
        significativo.
        En el caso del sensor de ultrasonido el rango es desde 0x0000 hasta 0x7594

```

```

que representa la minima y maxima lectura del sensor.
En el caso del switch, un estado logico bajo se lee como 0x0000 y un estado
logico alto se lee como 0xFFFF.
*/
readSensor = (cmd->data)[0];
requestFrom = response.to;
requestCmd = response.cmd;
sendResponse = false;
samples = 1;
break;
case DISTANCE_SENSOR_ALARM_ON_STATE:
/* Cuando un switch esta presente en el ID: 0x05, establece si se desea o no
recibir una alarma ante cierto cambio de estado en el mismo. Puede ser ante
cualquier cambio o sobre un flanco ascendente o descendente.
:DATO:
Valor entre 0x00 y 0x03 con el tipo de cambio ante el cual generar
la alarma. Con un 0x00 ignora cualquier cambio en el switch. Se utiliza
0x01 para que cualquier cambio en el switch genere el mensaje, 0x02 para
que sea solo ante un flanco ascendente y 0x03 para que sea solo ante un
flanco descendente.
:RESP:
-
*/
#endif
#if TRIGGER_TYPE == SWITCH_SENSOR
switch ((cmd->data)[0])
{
    case 0x00:
        // Ignorar
        disable_interrupts(INT_EXT);
        break;
    case 0x01:
        // Cualquier cambio
        enable_interrupts(INT_EXT);
        break;
    case 0x02:
        // flanco ascendente
        ext_int_edge(H_TO_L);
        enable_interrupts(INT_EXT);
        break;
    case 0x03:
        // flanco descendente
        ext_int_edge(L_TO_H);
        enable_interrupts(INT_EXT);
        break;
    default:
        (cmd->data)[0] = 0x00;
        break;
}
alarmType = (cmd->data)[0];
#endif
break;
default:
    response.len++;
    response.cmd = COMMON_ERROR;
    response.data[0] = 0x01; // Comando desconocido
    break;
}

// CRC de la respuesta
response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

return;
}

```

B.5. Placa controladora de servo motores

Código fuente de la placa controladora de servo motores.

```

//CCS PCM V4.023 COMPILER

#define CARD_GROUP      SERVO_MOTOR      // Ver protocol.h
#define CARD_ID          0                // Valor entre 0 y E

```

```

// Descripcion de la placa
#define DESC           "SERVOR CONTROL - 1.0" // Maximo DATA_SIZE bytes

/* Modulo Servo - main.c
 * PIC16F88 - MAX232 - SERVO
 *
 *          .
 *          PIC16F88
 *          .
 *          -----
 *      MOTOR_4 -|RA2/AN2/CVREF/VREF      RA1/AN1|- MOTOR_5
 *      LED -|RA3/AN3/VREF+/C1OUT      RAO/ANO|-_
 *      LED -|RA4/AN4/TOCKI/C2OUT      RA7/OSC1/CLKII|- XT CLOCK pin1, 27pF to GND
 *      RST/ICD2:MCCLR -|RA5/MCLR/VPP    RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *      GND -|VSS                  VDD|- +5v
 *      MOTOR_1 -|RB0/INT/CCP1      RB7/AN6/PGD/T1OSI|- ICD2:PGD
 *      MOTOR_2 -|RB1/SDI/SDA      RB6/AN5/PGC/T1OSO/T1CKII|- ICD2:PGC
 *      MAX232:R1OUT -|RB2/SDO/RX/DT    RB5/SS/TX/CK|- MAX232:T1IN
 *      ICD2:PGM/ -|RB3/PGM/CCP1      RB4/SCK/SCL|-_
 *      MOTOR_3      '-----'
 *
 */
#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// Led
#bit led1=porta.1
#bit led2=porta.4

// MAX232
#bit tx=portb.5
#bit rx=portb.2

// PWMs
#bit pwm1=portb.0
#bit pwm2=portb.1
#bit pwm3=portb.3
#bit pwm4=porta.2
#bit pwm5=porta.1

#include <../../protocalo/src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE]; // Buffer de recepcion de comandos
int buffer_write; // Indice de escritura
int buffer_read; // Indice de lectura
int data_length; // Largo de los datos en el buffer

struct command_t command; // Comando parseado
struct command_t response; // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecuta el comando

```

```

****/

// Software PWM - Minimo 1750 (~0.71ms)
#define PULSE_MIN 1750
// Tiempo maximo que puede durar un pulso - Maximo 6755(~2.71ms)
#define PULSE_MAX 6755
// Tiempo entre pulsos (~25ms -> ~22.29ms fijos de espera)
#define PWM_MAX 62500
// 1 ~ 27.8 cuentas ~ 69.4us
#define DEGREE 27.8f

// Valor que representa el ancho del pulso para cada servo
long pwm_t[5];
// Angulo de cada servo
long pos[5];
// On/Off de cada servo
short servo[5];

void init()
{
    // Inicializa puertos
    set_tris_a(0b11100001);
    set_tris_b(0b11110100);

    // Seteo el Timer1 como fuente interna
    setup_timer_1(T1_INTERNAL | T1_DIV_BY_2);
    set_timer1(0);

    // Variable para hacer el reset
    reset = false;

    // Valor que representa el ancho del pulso para cada servo
    pwm_t[0] = PULSE_MIN;
    pwm_t[1] = PULSE_MIN;
    pwm_t[2] = PULSE_MIN;
    pwm_t[3] = PULSE_MIN;
    pwm_t[4] = PULSE_MIN;

    // Angulo de cada servo
    pos[0] = 0;
    pos[1] = 0;
    pos[2] = 0;
    pos[3] = 0;
    pos[4] = 0;

    // On/Off de cada servo
    servo[0] = 0;
    servo[1] = 0;
    servo[2] = 0;
    servo[3] = 0;
    servo[4] = 0;

    // Activo los servos segun este o no habilitado
    pwm1 = servo[0];
    pwm2 = servo[1];
    pwm3 = servo[2];
    pwm4 = servo[3];
    pwm5 = servo[4];

    return;
}

void main()
{
    short check_com = 0;
    long tmr1;

    // Control de servomotores
    init();

    // Init del protocol
    initProtocol();

    // FOREVER
}

```

```

while(true)
{
    // Software PWM

    // Tomo el tiempo
    tmr1 = get_timer1();

    // Es hora de reiniciar el pulso?
    if (tmr1 >= PWM_MAX)
    {
        // Detiene el analisis de comandos
        check_comm = 0;
        // Inicio del periodo
        set_timer1(0);
        // Activo los servos segun este o no habilitado
        pwm1 = servo[0];
        pwm2 = servo[1];
        pwm3 = servo[2];
        pwm4 = servo[3];
        pwm5 = servo[4];
    } else
    // Llego al final del pulso?
    if (tmr1 >= PULSE_MAX)
    {
        // Pone las salidas a 0
        pwm1 = 0;
        pwm2 = 0;
        pwm3 = 0;
        pwm4 = 0;
        pwm5 = 0;
        // Analiza si hay comandos para ser atendidos
        check_comm = 1;
    } else {
        // Tomo el tiempo
        tmr1 = get_timer1();

        // Es tiempo de desactivar el PWM?
        if (tmr1 >= pwm_t[0])
            pwm1 = 0;
        if (tmr1 >= pwm_t[1])
            pwm2 = 0;
        if (tmr1 >= pwm_t[2])
            pwm3 = 0;
        if (tmr1 >= pwm_t[3])
            pwm4 = 0;
        if (tmr1 >= pwm_t[4])
            pwm5 = 0;
    }

    // Protocolo
    if (check_comm == 1)
        runProtocol(&command);
}

return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
    int crc, i, len;

    // Calculo del CRC
    crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

    // CRC ok?
    if (cmd->crc != crc)
    {
        // Creo respuesta de error
        response.len = MIN_LENGTH + cmd->len + 2 + 1;
        response.to = cmd->from;
        response.from = THIS_CARD;
        response.cmd = COMMON_ERROR;
    }
}

```

```

response.data[0] = 0x00;
// Agrego el paquete que contiene el error de CRC
response.data[1] = cmd->len;
response.data[2] = cmd->to;
response.data[3] = cmd->from;
response.data[4] = cmd->cmd;
// Campo data
len = cmd->len - MIN_LENGTH;
for (i = 0; i < len; i++)
    response.data[5 + i] = (cmd->data)[i];
// CRC erroneo
response.data[5 + len] = cmd->crc;
// CRC esperado
response.data[5 + len + 1] = crc;
// CRC de la respuesta
response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

crcOK = false;
return;
}

crcOK = true;

// Minimo todos setean esto
response.len = MIN_LENGTH;
response.to = cmd->from & 0x77;
response.from = THIS_CARD;
response.cmd = cmd->cmd | 0x80;

switch (cmd->cmd)
{
    // Comandos comunes
    case COMMON_INIT:
        init();
        // Enviar la descripcion de la placa en texto plano
        strcpy(response.data, DESC);
        response.len += strlen(response.data);
        break;
    case COMMON_RESET:
        // Enviar la descripcion de la placa en texto plano
        strcpy(response.data, DESC);
        response.len += strlen(response.data);
        // Reset!
        reset = true;
        break;
    case COMMON_PING:
        // No hace falta hacer mas nada
        break;
    case COMMON_ERROR:
        // Por ahora se ignora el comando
        break;

    /* Comandos especificos */

    case SERVO_MOTOR_SET_POSITION:
        /* Determina la posicion en la que debe colocarse el
        servo motor indicado.
        :DATO:
        Valor de 0x00 a 0x04 que determina el id del servo al
        que se le aplicara la posicion. Valor entre 0x00 y 0xB4
        que representa el rango de 0 a 180 con 1 de presicion.
        :RESP:
        -
        */
        i = ((cmd->data)[0] & 0x07); // Servo destinatario
        if (i < 5)
        {
            servo[i] = 1;
            pos[i] = (unsigned char)((cmd->data)[1]);
            pwm_t[i] = PULSE_MIN + pos[i] * DEGREE;
        }
        break;
    case SERVO_MOTOR_SET_ALL_POSITIONS:
        /* Determina las posiciones en la que deben colocarse

```

```

        cada uno de los servomotores
:DATO:
Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno
para cada uno de los servos conectados al controlador.
Cada valor representa el rango de 0 a 180 con 1 de presicion.
:RESP:
-
*/
for (i = 0; i < 5; i++)
{
    servo[i] = 1;
    pos[i] = (unsigned char)((cmd->data)[i]);
    pwm_t[i] = PULSE_MIN + pos[i] * DEGREE;
}
break;
case SERVO_MOTOR_GET_POSITION:
/* Obtiene la ultima posicion del servomotor indicado.
:DATO:
Valor de 0x00 a 0x04 que determina el id del servo del que
se requiere la posicion.
:RESP:
Valor de 0x00 a 0x04 que determina el id del servo del que
se requiro la posicion. Valor entre 0x00 y 0xB4 que representa
el rango de 0 a 180 con 1 de presicion.
*/
i = ((cmd->data)[0] & 0x07); // Servo destinatario
if (i < 5)
{
    response.data[0] = pos[i];
    response.len++;
}
break;
case SERVO_MOTOR_GET_ALL_POSITIONS:
/* Obtiene las ultimas posiciones de todos los servomotor
conectados al controlador.
:DATO:
-
:RESP:
Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para
cada uno de los servos conectados al controlador. Cada valor
representa el rango de 0 a 180 con 1 de presicion.
*/
response.data[0] = pos[0];
response.data[1] = pos[1];
response.data[2] = pos[2];
response.data[3] = pos[3];
response.data[4] = pos[4];
response.len += 5;
break;
case SERVO_MOTOR_SET_SERVO_SPEED:
/* Determina la velocidad a la que el servomotor indicado
llegara a la posicion.
:DATO:
Valor de 0x00 a 0x04 que determina el id del servo al que
se le aplicara la velocidad. Valor entre 0x00 y 0xB4,
velocidad en grados por segundo.
:RESP:
-
*/
break;
case SERVO_MOTOR_SET_ALL_SPEEDS:
/* Determina las velocidades a la que cada uno de los
servomotores llegara a la posicion indicada.
:DATO:
Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno
para cada uno de los servos conectados al controlador.
Cada valor representa a la velocidad en grados por segundo.
:RESP:
-
*/
break;
case SERVO_MOTOR_GET_SERVO_SPEED:
/* Obtiene la velocidad asignada al servomotor indicado.
:DATO:

```

```

        Valor de 0x00 a 0x04 que determina el id del servo del que
        se requiere la velocidad.
        :RESP:
        Valor de 0x00 a 0x04 que determina el id del servo del que
        se requirió la velocidad. Valor entre 0x00 y 0xB4, velocidad
        en grados por segundo.
        */
    break;
    case SERVO_MOTOR_GET_ALL_SPEEDS:
        /* Obtiene las velocidades de cada uno de los servomotor
        conectados al controlador.
        :DATO:
        -
        :RESP:
        Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para
        cada uno de los servos conectados al controlador. Cada valor
        representa a la velocidad en grados por segundo.
        */
    break;
    case SERVO_MOTOR_FREE_SERVO:
        /* Deja de aplicar fuerza sobre el servo indicado.
        :DATO:
        Valor de 0x00 a 0x04 que determina el id del servo a liberar.
        :RESP:
        -
        */
        i = ((cmd->data)[0] & 0x07); // Servo destinatario
        if ((i < 5) && (i >= 0))
        {
            servo[i] = 0;
        }
    break;
    case SERVO_MOTOR_FREE_ALL_SERVOS:
        /* Deja de aplicar fuerza sobre cada uno de los servomotor
        conectados al controlador.
        :DATO:
        -
        :RESP:
        -
        */
        servo[0] = 0;
        servo[1] = 0;
        servo[2] = 0;
        servo[3] = 0;
        servo[4] = 0;
    break;
    default:
        response.len++;
        response.cmd = COMMON_ERROR;
        response.data[0] = 0x01; // Comando desconocido
    break;
}

// CRC de la respuesta
response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

return;
}

```

C. Costo del prototipo

Principales costos de armado del prototipo.

Cantidad	Detalle	Unitario	Total
1	Netbook	2451	2451
1	Adaptador USB - Serial RS232	90	90
10	Microcontrolador PIC16F88	23	230
4	Placa genérica	97	388
2	Placa controladora motor DC	151	302
2	Motorreductor IGNIS MR2-FA	50	100
2	Driver L298	21	42
4	Placa controladora de sensores	106	424
8	Telémetro infrarrojo GP2D120	35	280
1	Sensor de ultrasonido SRF05	106	106
5	Sensor de piso CNY70	10	50
1	Batería 12v 7Ah	85	85
1	Cargador de batería automático	68	68
1	Rueda castor 30C	11	11
1	Rueda castor 40C	14	14
2	Rueda 100x26	30	60

Cuadro 40: Lista de materiales y costos (en pesos a marzo de 2010).

D. Implementación de arquitectura de software del controlador

La implementación de la arquitectura de software del controlador la hicimos en C++, tal como explicamos en la sección 3.6. La relación explicada en dicha sección entre el robot (con sus dispositivos), los comportamientos y el sistema de reconocimiento se puede ver más claramente en la figura 97, unidos por la clase *GarbageCleaner*, que implementa la arquitectura *Subsumption*. En la misma también está el main-loop. Mostramos en la figura 98 que *GarbageCleaner* posee comportamientos, así como también un robot y el mismo posee dispositivos que pueden ser sensores o actuadores, tal como se ve en la figura 99. También posee el módulo de reconocimiento de objetos, mostrado en la figura 100. La implementación de *GarbageCleaner* es simple, ya que debe:

- Obtener los sensores y actuadores del robot.
- Inicializar el módulo de reconocimiento de basuras.
- Instanciar los comportamientos, brindándoles los sensores, actuadores y eventualmente el módulo de reconocimiento, necesarios para saber si su estímulo está activo y para poder interactuar con el entorno.
- Correr el main-loop, utilizando la arquitectura de comportamientos elegida (*Subsumption*), haciendo actuar al comportamiento cuyo estímulo esté activo en ese instante de tiempo y tenga mayor nivel en la arquitectura.

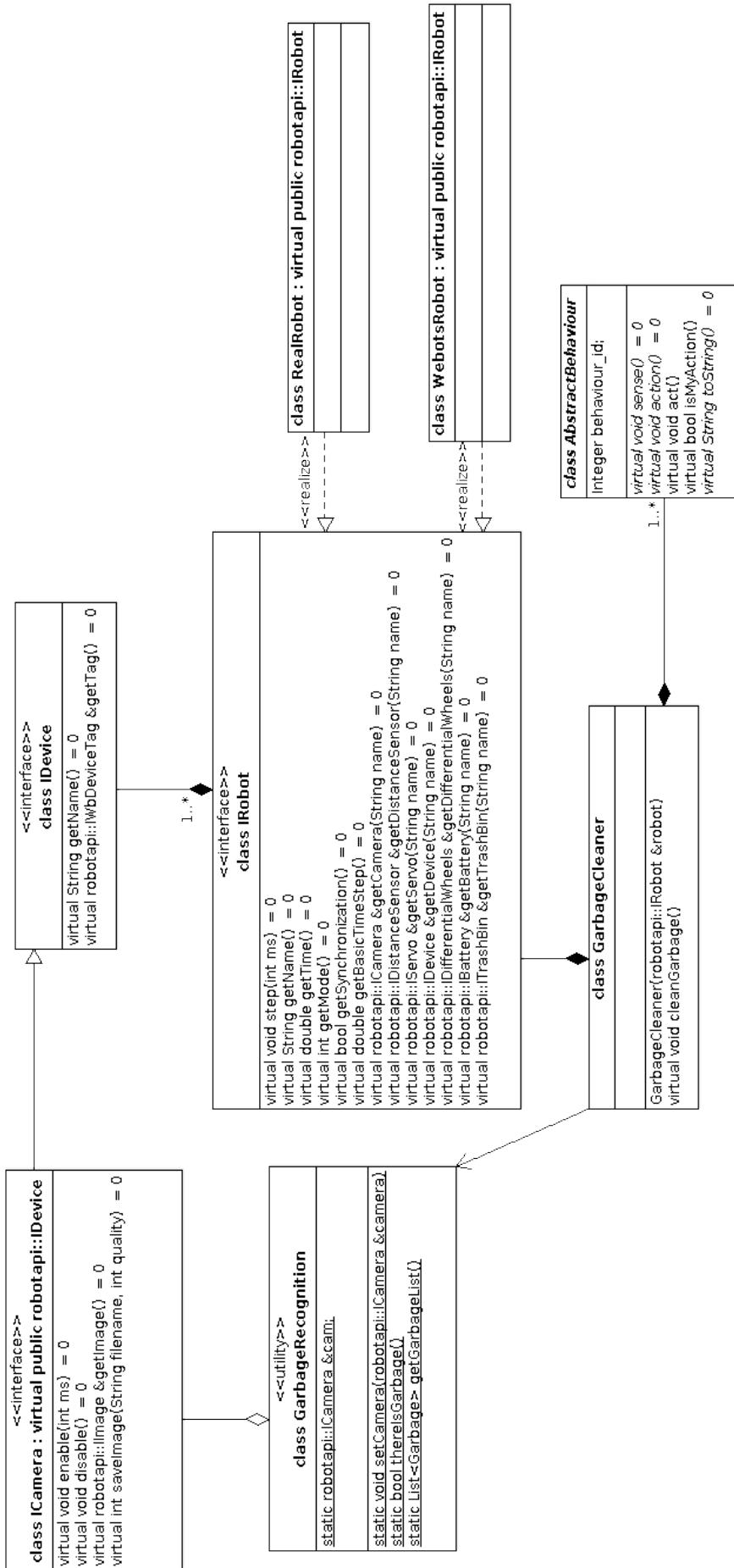


Figura 97: Diagrama de clases de la arquitectura de software. Relación entre las 3 grandes componentes del proyecto

D.1. Comportamientos

Como ya mencionamos anteriormente, en la figura 98 mostramos la relación de *GarbageCleaner* con los comportamientos. Podemos ver que más precisamente posee *AbstractBehaviours*. Ésto se debe a que la única información que necesita para coordinarlos es saber si están activos (método *sense*) y luego poder indicarles que den su respuesta (método *act*).

Para agregar un comportamiento, es necesario que extienda de *AbstractBehaviour* e implemente los métodos abstractos de la clase: *sense*, *action* y *toString*. En el primero el comportamiento debe indicar si está activo o no en base a la información de los sensores que utilice. En el segundo se debe implementar la respuesta del comportamiento en el caso que esté activo. El último método provee una descripción del comportamiento implementado. Finalmente, hay que instanciarlo y agregarlo a la lista de comportamientos que posee *GarbageCleaner*.

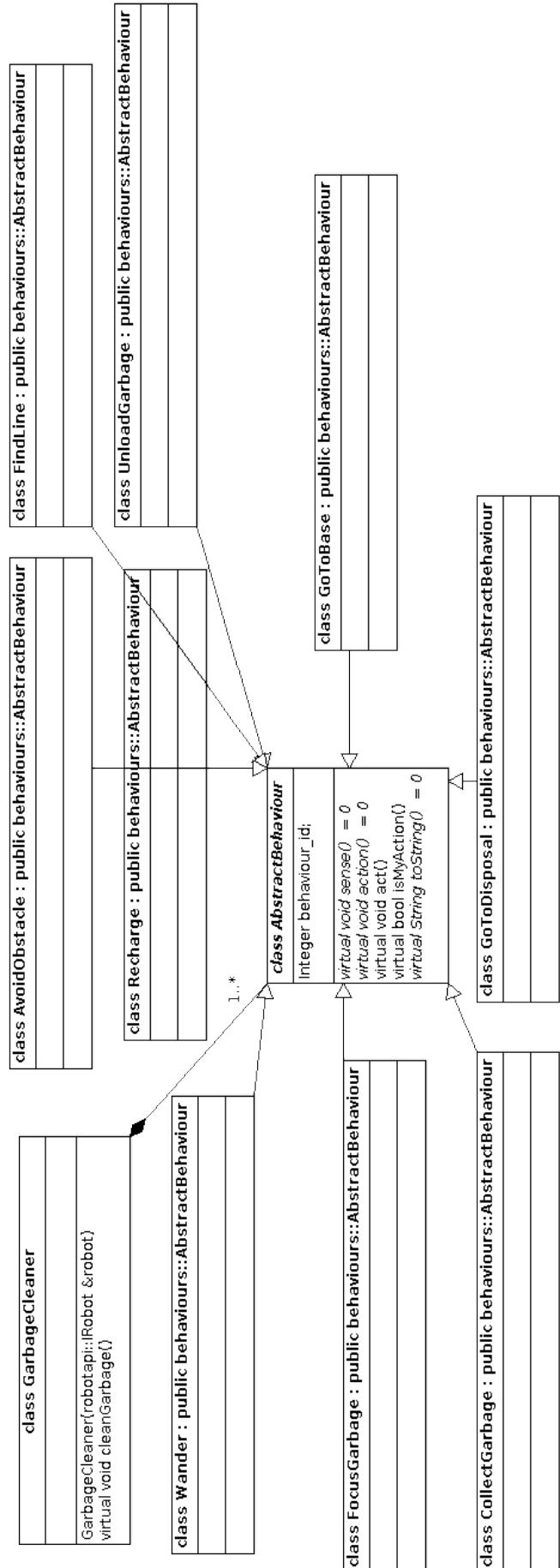


Figura 98: Diagrama de clases de la arquitectura de software. Paquete de comportamientos

D.2. Dispositivos del robot

Para que el robot sense y actúe en el entorno a través de sus comportamientos, debe disponer de dispositivos que le permita hacerlo. Éstos últimos forman parte del *IRobot* que le es entregado a *GarbageCleaner* para que instancie los comportamientos. Como se puede observar, la interfaz que representa al robot provee métodos de obtención de sensores como *getDistanceSensor* y de actuadores, como *getDifferentialWheels*. Para el desarrollo de la arquitectura nos basamos fuertemente en la api que provee Webots para interactuar con su robot simulado.

Para minimizar el acoplamiento entre *IRobot* y *GarbageCleaner*, hicimos que los dispositivos sean pedidos por un nombre que los representa. En la implementación de la interfaz para la simulación con Webots (*WebotsRobot*), la mayor cantidad de llamadas a éstos métodos son simples wrappers de una invocación al dispositivo virtual del programa de simulación. La implementación para el robot real, *RealRobot*, posee un mapa que relaciona los nombres de los dispositivos con instancias de implementaciones de los mismos. Éstas implementaciones, como por ejemplo *RealDistanceSensor*, tienen los handlers necesarios para poder enviar y recibir información de los sensores y actuadores de los cuales son responsables. En el caso de *RealDifferentialWheels*, el mismo posee dos handlers, dado que cada handler es capaz de interactuar con una placa con determinado groupid y boardid, y cada motor posee una placa que lo controla.

En el caso que se quiera agregar un sensor nuevo a la api, como por ejemplo un GPS, es necesario crear la interfaz que represente el mismo, con métodos de obtención de valores de los sensores o establecimiento de valores para los actuadores. Luego de crear la interfaz, es necesario agregar un método a la interfaz de *IRobot* para poder obtener una instancia del dispositivo en cuestión. Después es necesario implementar éste método en las clases *WebotsRobot* y *RealRobot*, dependiendo si se va a querer simular con Webots, correr en la realidad o ambos. Finalmente, se deben realizar las implementaciones correspondientes de la interfaz creada para el dispositivo.

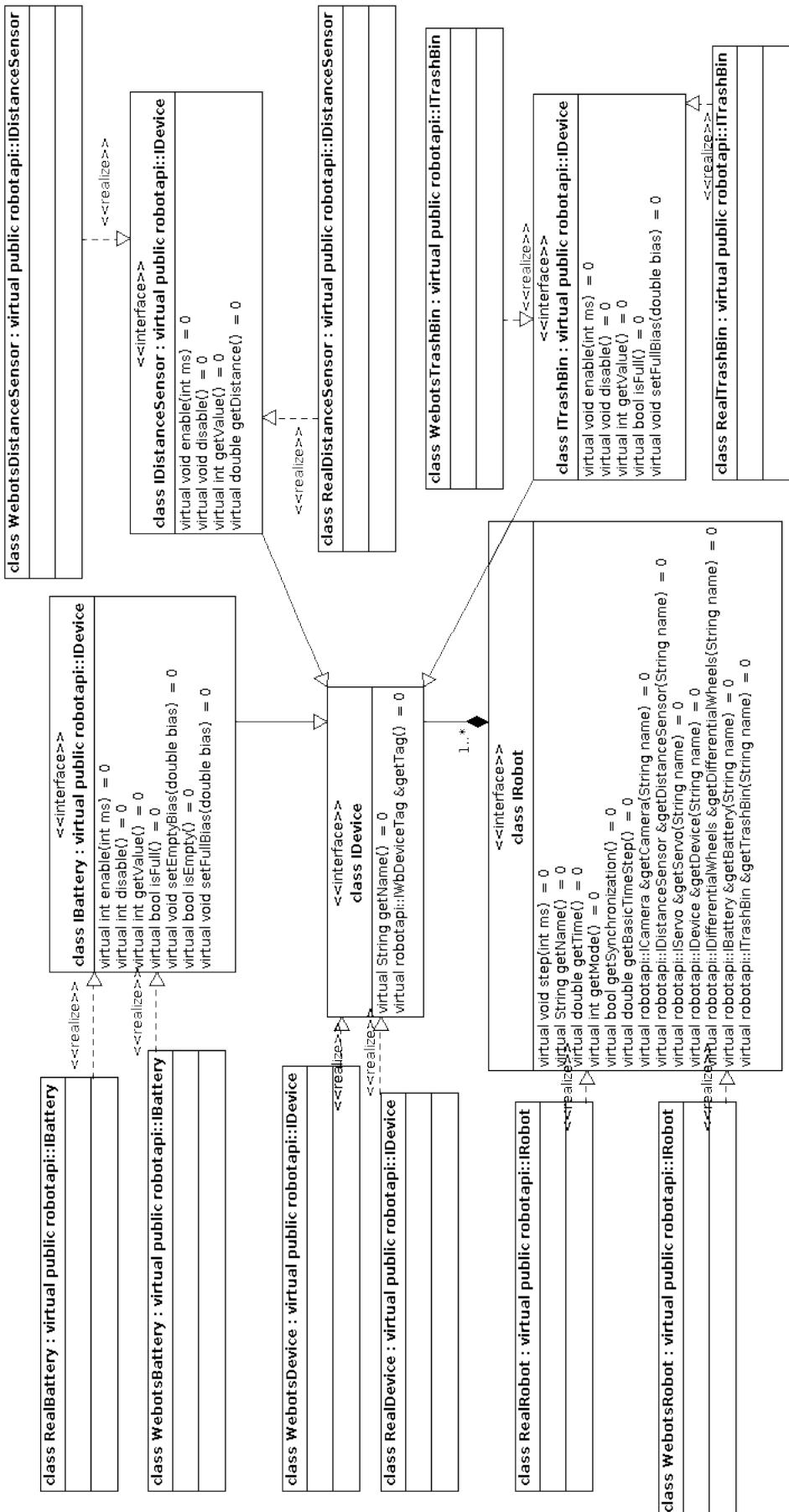


Figura 99: Diagrama de clases de la arquitectura de software. Paquete de api de 1 robot

D.3. Reconocimiento de objetos

La figura 100 exhibe la relación entre el módulo de reconocimiento de objetos y el módulo de comportamientos. El módulo de visión queda encapsulado dentro de la clase de *GarbageRecognition*, permitiendo que el módulo de comportamientos funcione en forma independiente a éste. La creación de una interfaz para la cámara nos permite alternar entre el entorno de webots y el mundo real. El comportamiento de las clases *WebotsCamera* y *RealCamera* es el mismo pero sus implementaciones difieren en que, en el caso de la primera, se pide la imagen a webots mientras que en la segunda se toma una imagen del mundo real. Por motivos similares utilizamos una interfaz para la representación de las imágenes. Esto se debe a que OpenCV y Webots codifican las imágenes de maneras distintas y tuvimos que implementar sus respectivas clases.

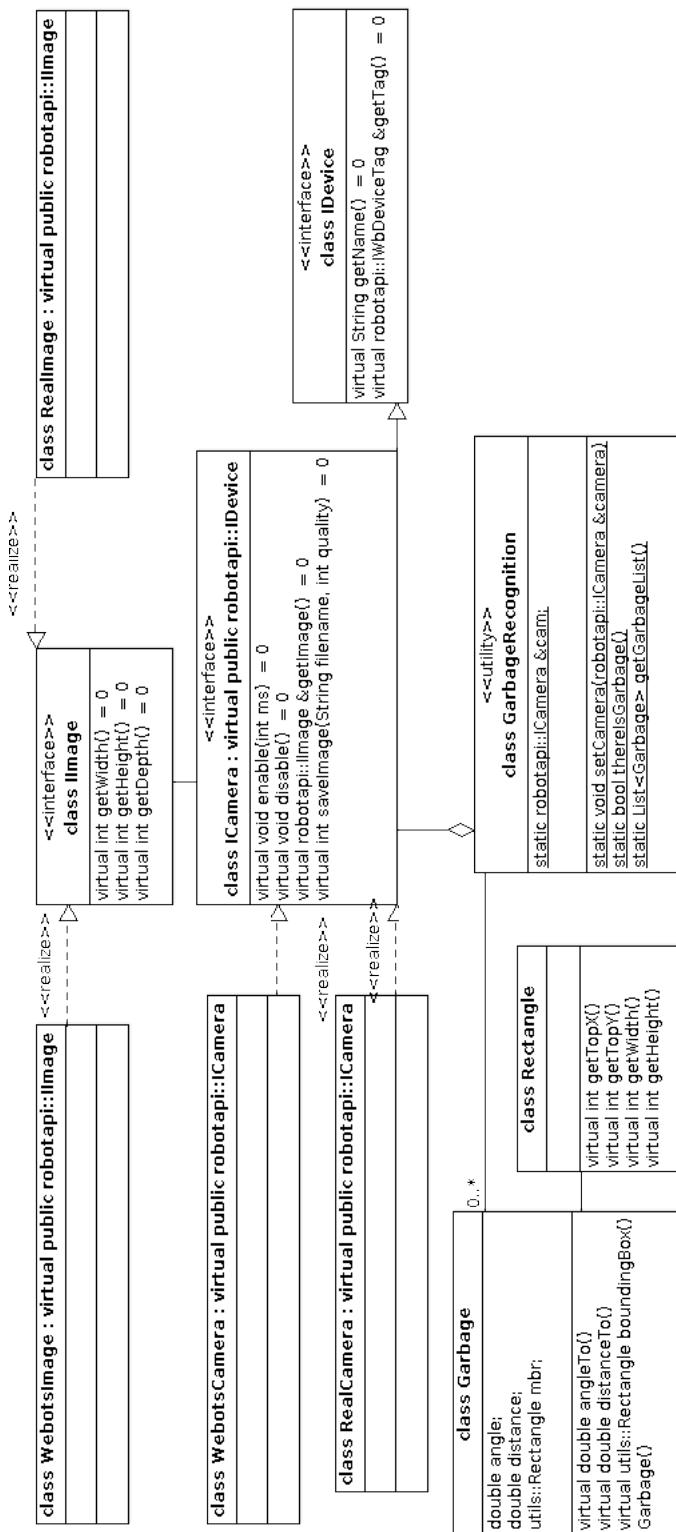


Figura 100: Diagrama de clases de la arquitectura de software. Paquete de reconocimiento de basuras

E. Implementación del protocolo de comunicación del lado de la PC

E.1. Packet Server

La implementación del protocolo en la PC la hicimos en C++, al igual que el controlador y el módulo de reconocimiento de basuras. Para la misma implementamos los paquetes descriptos en el protocolo y un servidor de envío y recepción de los mismos a través del puerto serial. También implementamos lo que llamamos handlers, quienes son los responsables de convertir los comandos de la api del robot en paquetes y enviarlos al servidor, así como también de recibir los paquetes de respuesta que le incumben y proveerlos a la api de una forma que ésta los entienda. Mostramos el diagrama de ésta relación en la figura 101.

Como mencionamos anteriormente, hay un servidor que denominamos packet server encargado del envío y recepción de paquetes. El servidor sólo se encarga de mandar una serie de bytes y de recibir los mismos, sin inspeccionar de qué tipo de paquete se trata. Provee dos métodos para realizar el envío y recepción:

- sendPacket: Su función es recibir un paquete y encollarlo para que el servidor lo envíe. Como el servidor es un thread diferente al de la api, no se interrumpe la recepción o envío de paquetes.
- registerHandler: Registra un handler para un paquete enviado desde una placa con grupo *groupid* e id de placa *boardid*. Cuando llega un paquete de dicha placa, el servidor invoca el método handlePacket() del handler registrado.

La idea del método handlePacket es que se limite sólo a guardar los nuevos valores recibidos, ya que durante la invocación del mismo el server no recibe ni envía paquetes. La clase packet provee funciones de utilidad para cualquier tipo de paquete, tales como cálculo y verificación de CRC, seteo y obtención de valores de los campos del paquete, entre otras.

E.2. Packets

Como indicamos en la figura 102, group packet extiende a packet, agregando métodos de utilidad para los comandos que todos los grupos son capaces de recibir. Lo mismo sucede con board packet, extendiendo de group packet ya que es más específico que este último.

En las figuras 103 y 104 mostramos los paquetes que extienden de board packet, específicos para cada sensor o actuador que pueda haber en una placa. Como se puede observar, cada paquete específico tiene funciones que permiten obtener y establecer información propia del sensor/actuador en cuestión, respetando el protocolo que propusimos.

Por ejemplo, BatteryPacket permite establecer los umbrales a los cuales informará que la batería se encuentra cargada o en estado crítico, respectivamente. Además permite obtener el valor de carga de la batería.

A continuación damos un ejemplo de uso de un paquete. Supongamos que se quiere saber la carga de la batería. Para ésto instanciamos un BatteryPacket con el id de grupo y placa correspondientes. El mismo se encarga de establecer

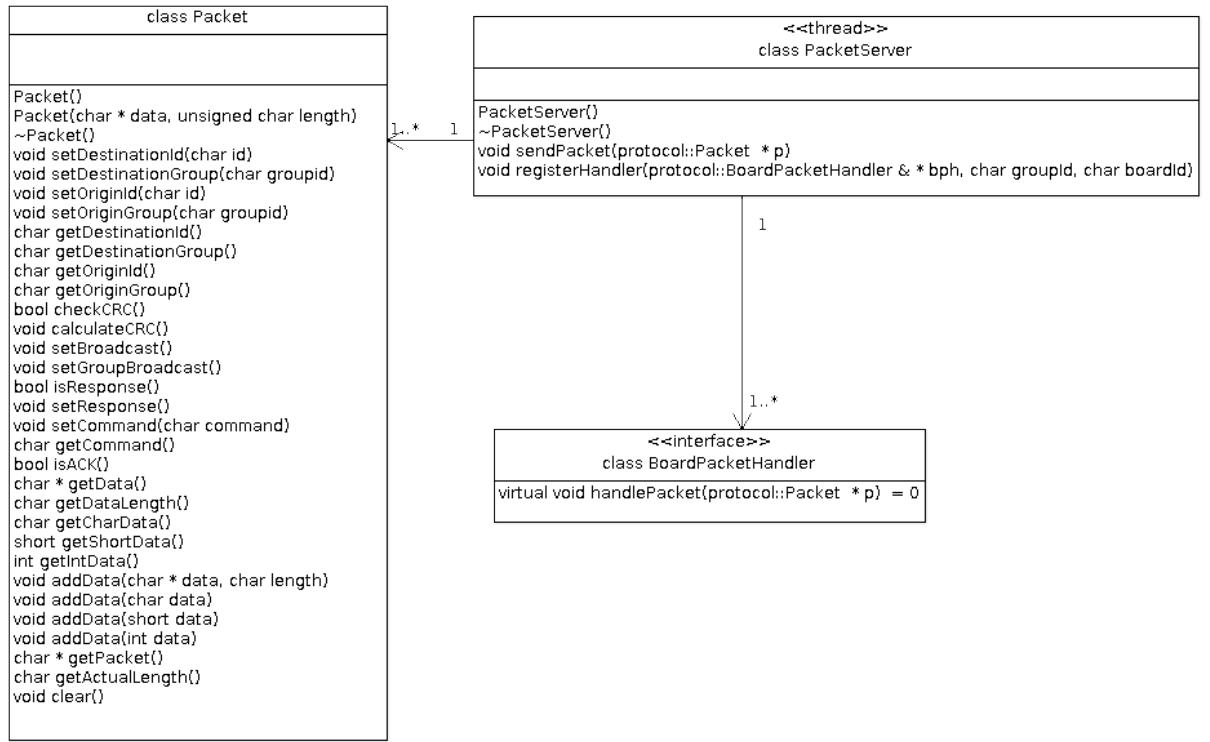


Figura 101: Diagrama de clases. Estructura general del servidor de paquetes e interfaz para los handlers.

los campos de grupo y placa de origen y destino, tamaño del paquete y comando. Le indicamos que queremos el valor de carga con `senseBattery()`, que pone el comando correspondiente en el campo del paquete. Una vez que tenemos el paquete armado, llamamos a la función `sendPacket()` del packet server. Suponiendo que registramos un handler para ese id de grupo y de placa, cuando el server reciba la respuesta de la placa va a llamar al método `handlePacket()` del handler. En la función, instanciamos un `BatteryPacket` y llamamos a la función `analysePacket()` con el paquete que recibimos. Finalmente, nos queda invocar al método `getBatteryValue()` de este paquete.

E.3. Handlers

En las figuras 105 y 106 mostramos los handlers implementados para que las implementaciones de los sensores de la api del robot pudieran manejar los mismos, sin tener conocimiento sobre el protocolo. Hay un handler por default, `DefaultBoardPacketHandler`, para el caso en que no haya registrado un handler para determinado id de grupo y placa. La función `handlePacket` del mismo se encarga de imprimir por salida estándar cada campo del paquete en formato hexadecimal.

Se puede observar que la cantidad de métodos en los paquetes y en los handlers no es la misma, y en algunos casos, hay métodos que en el otro no

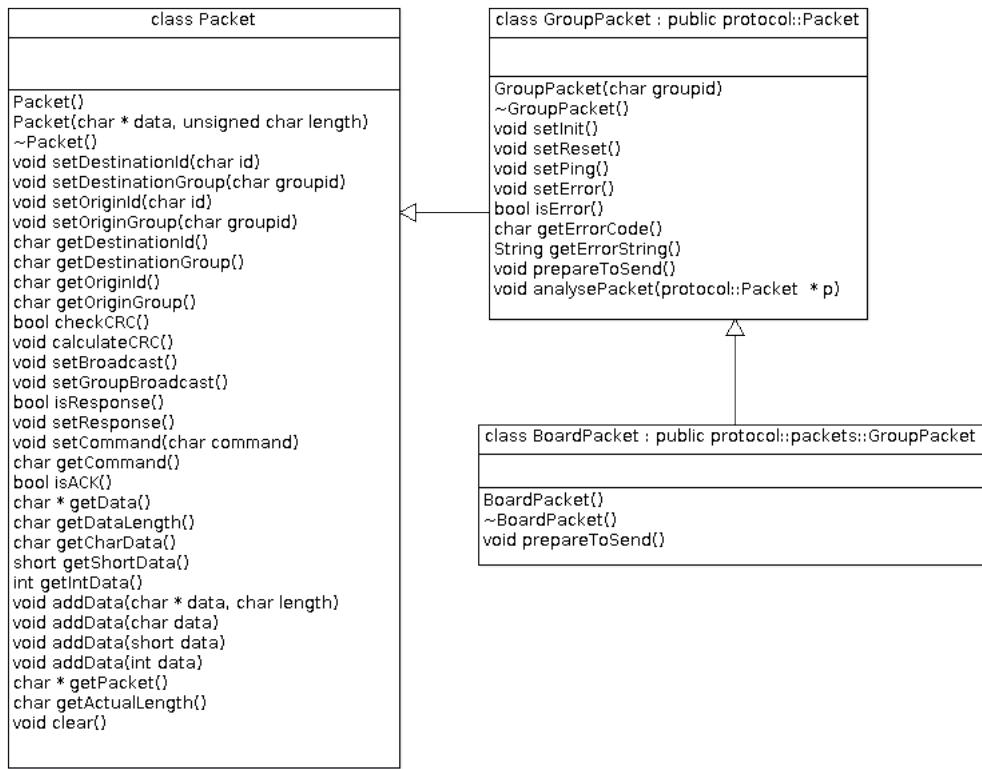


Figura 102: Diagrama de clases. Estructura general de los paquetes.

están. Ésto se debe a que los handlers son intermediarios entre el protocolo y la api del controlador encargado de la ejecución de los comportamientos. En algunos casos, el protocolo brinda más opciones de las que realmente utiliza el controlador para funcionar.

E.4. Modificación del protocolo

En el caso que haya una modificación en el protocolo, se deberán actualizar las clases afectadas por los cambios, y eventualmente, los handlers correspondientes.

Supongamos el caso en el que se agrega un campo de 1 byte luego del tamaño total del paquete. Éste cambio, en principio, sólo afecta la clase Packet. En el caso que el campo indique algo sobre el grupo o placa, también afectará a group packet o board packet, respectivamente. En el caso que se amplíe el set de comandos de un determinado sensor o actuador, es necesario modificar el packet y handler asociados a los mismos. Lo importante de ésto es que los cambios están localizados. Es decir, un cambio en el protocolo llega como mucho hasta los handlers, aislando a la api del controlador de posibles cambios.

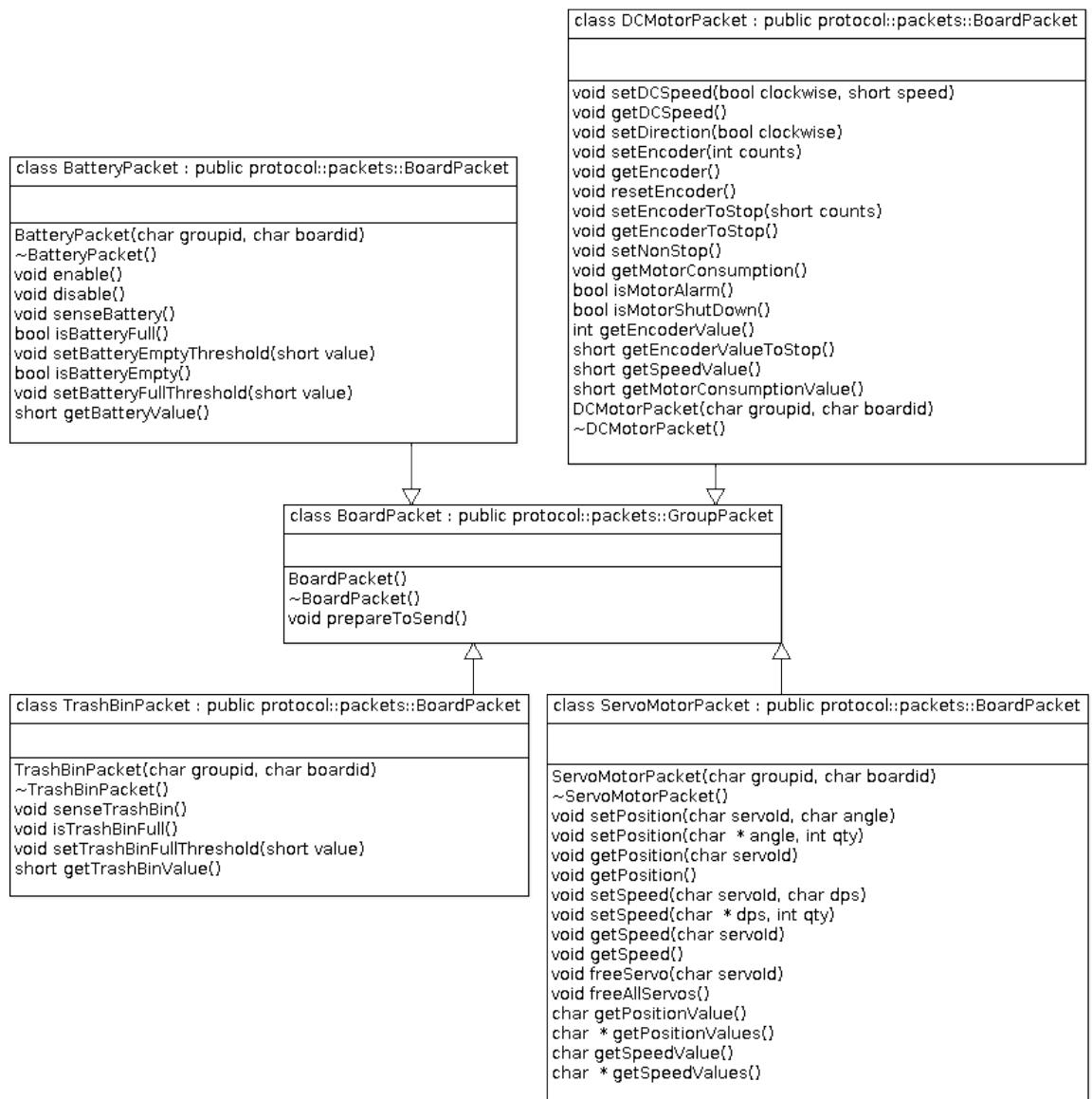


Figura 103: Diagrama de clases. Estructura de los paquetes de motor, recipiente de basura, servos y batería.

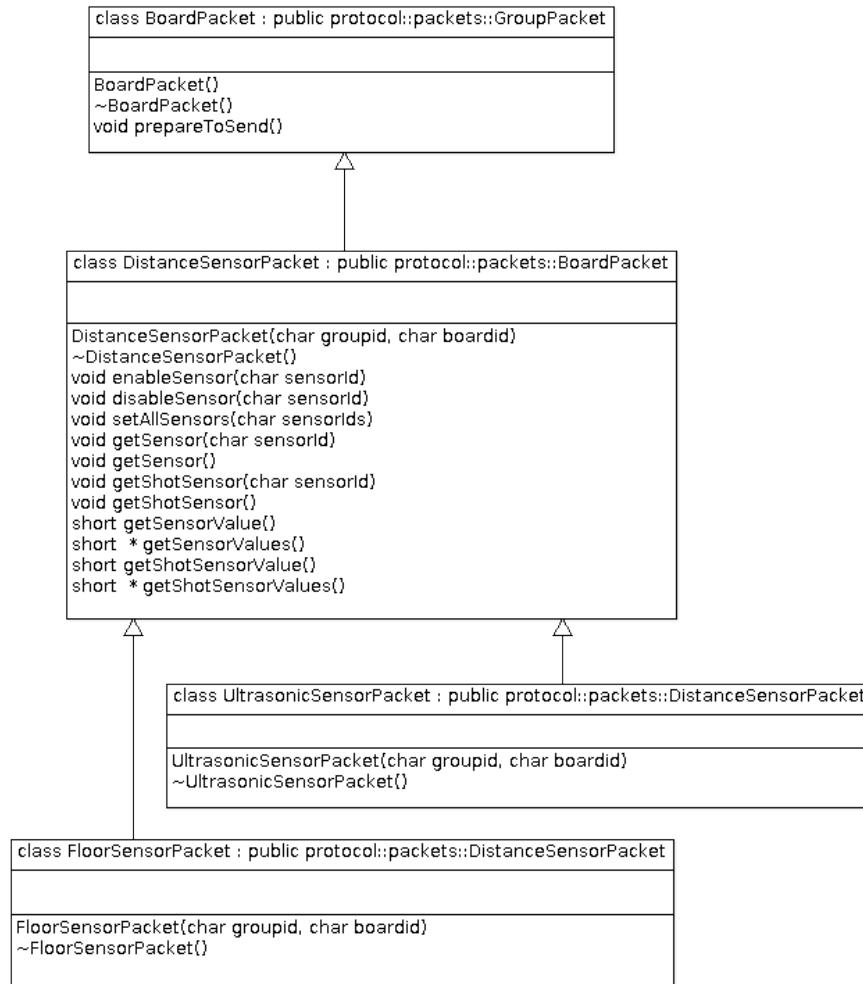


Figura 104: Diagrama de clases. Estructura general de los paquetes para sensores de distancia.

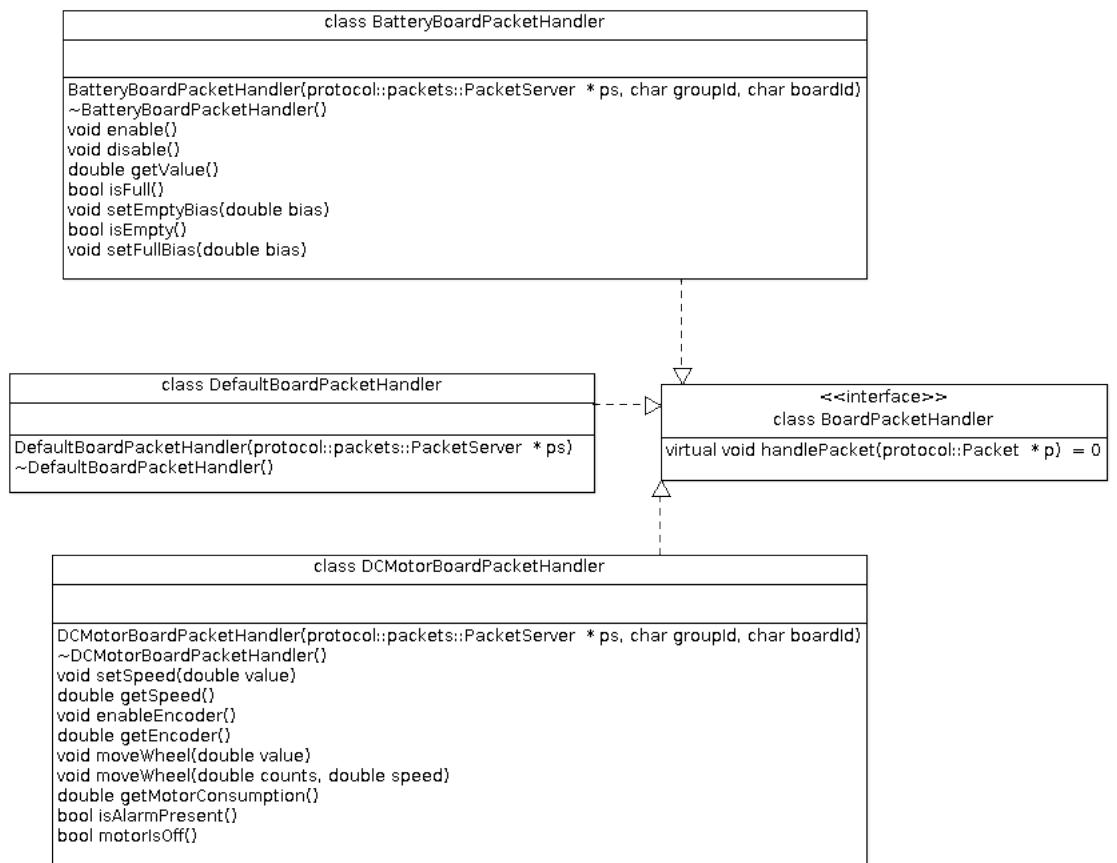


Figura 105: Diagrama de clases. Handlers de paquetes default y para las placas de motor y batería.

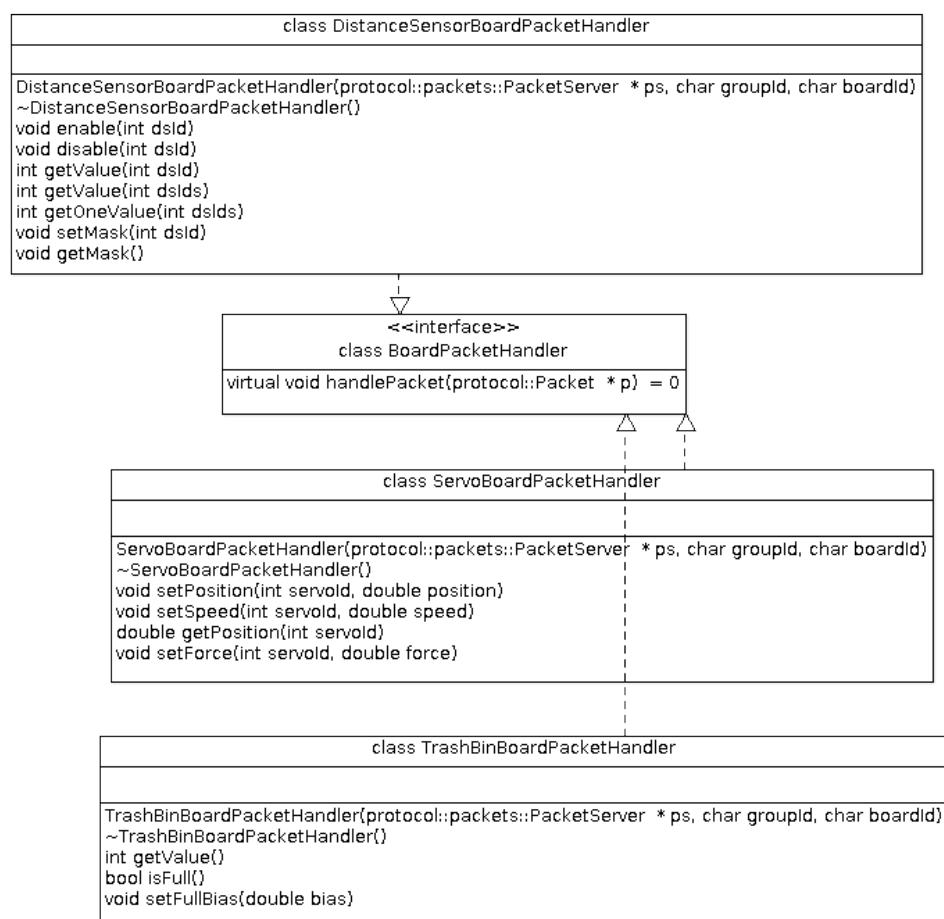


Figura 106: Diagrama de clases. Handlers de paquetes para las placas de sensores de distancia, servos y recipiente de residuos.