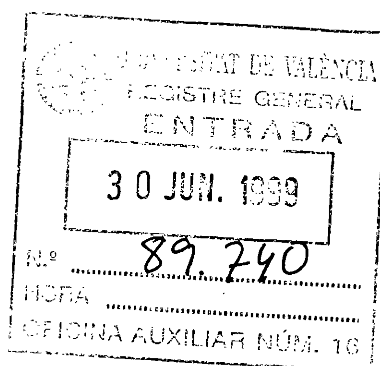


UNIVERSITAT DE VALÈNCIA

# Comparación, Test y Diseño de Arquitecturas de Control para la Generación de Comportamientos Eficientes por Robots Móviles en Entornos No Estructurados.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR PRESENTADA
AL DPTO. DE INFORMÁTICA Y ELECTRÓNICA
FACULTAD DE FÍSICA
UNIVERSITAT DE VALÈNCIA

Francisco Vegara Meseguer

Dirección

Juan de Mata Domingo Esteve

©Francisco Vegara Meseguer, 28 de junio de 1999

UMI Number: U607751

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U607751
Published by ProQuest LLC 2014. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

UNIVERSITAT DE VALÈNCIA
BIBLIOTECA CIÈNCIES

Nº Registre 13.546
DATA 29.X.1999
SIGNATURA T.D.366 FISICAS

Nº LIBIS: i20127364

29 cm.

D. JUAN DE MATA DOMINGO ESTEVE, profesor titular de Ciencias de la Computación e Inteligencia Artificial de la Universitat de Valencia,

**CERTIFICO** que la presente memoria

"Comparación, test y diseño de arquitecturas de control para la generación de comportamientos eficientes por robots móviles en entornos no esctructurados"

ha sido realizada bajo mi dirección, en el Departamento de Informática y Electrónica de la Universitat de Valencia por D. Francisco Vegara Meseguer, y constituye su tesis para optar al grado de Doctor en Ingeniería Informática.

Y para que conste, en cumplimiento de la legislación vigente, presentamos ante la Facultad de Física de la Universitat de Valencia, a 30 de Junio de 1999.

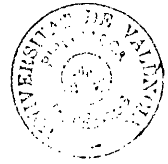
EL DIRECTOR

Juan de M. Domingo Esteve

# Agradecimientos

Muchas son las personas e Instituciones a las que debo agradecimiento por su ayuda o colaboración. Entre ellas cabe citar:

- A mi director de tesis Dr. Juan Domingo. La competencia y la capacidad de trabajo son dos cualidades que (al menos en el ámbito científico) juntas en una misma persona suelen resultar en un gran hombre. No tengo ninguna duda de que él ya lo es.
- Al Dr. Joan Pelechano por ayudarme, apoyarme y poner a mi disposición todo lo necesario para poder realizar mi trabajo sin demoras durante estos años.
- Al Catedrático de CC. e IA. y Director del Instituto de Robótica Dr. Gregorio Martín, por darme la posibilidad de entrar a formar parte de la Comunidad Universitaria y por confiar en mi a lo largo de estos años en el desarrollo de varios proyectos de I+D. Gracias.
- A la Comisión Interministerial de Ciencia y Tecnología (CICYT), por subvencionar los proyectos "Planificación de movimientos para vehículos móviles autónomos en entornos industriales basados en percepción sensorial" perteneciente al Area de Tecnologías avanzadas de la Producción, con referencia TAP95-1086-CO2-02 y con fecha de realización desde 1-7-95 hasta 1-7-98 y "Algoritmos de Visión para la Navegación de Robots Móviles" perteneciente al Area de Tecnologías de la Información y las Comunicaciones, con referencia TIC98-1026 y con fecha de realización desde 1-10-98 hasta 30-9-00. Buena parte del material utilizado en el desarrollo del robot ha sido posible gracias a esta ayuda.
- A mis compañeros del LISITT, del Instituto de Robótica y del Departamento de Informática y Electrónica por su ánimo, y en algunos casos también por su ayuda y sugerencias.
- A mi hermana por utilizar su experiencia como Física y gran parte de su paciencia, en la verificación de la corrección analítica de las ecuaciones pertenecientes al capítulo de Control.
- A Jose Pino y Rafa Vendrell por pasar a soporte informático (PCB sobre Tango), los esquemas electrónicos de algunas de las tarjetas del robot.
- A todas las personas e Instituciones que, desinteresadamente, ponen al servicio de los demás sus conocimientos e información por el bien y el desarrollo de todos.
- A mis padres, quienes con su ayuda y esfuerzo hicieron posible que descubriera el maravilloso Mundo de la Ciencia.
- A mis dos hijas, Rebeca y Sofía por conseguir despejar mi cabeza en sólo unos minutos.



*A Maribel, mi compañera y amiga*
*con quien todo comparto.*

# Resumen

El presente trabajo tiene por objeto el desarrollo de una herramienta que pueda ser utilizada para la implementación y comparación de arquitecturas software para el control de robots móviles.

La idea de partida nace de la constatación de la existencia de un conjunto no muy numeroso, pero sí significativo de concepciones organizativas entre los módulos software o software-hardware que conforman un robot móvil en la actualidad. Cada una de estas concepciones se desarrolla normalmente sobre una determinada plataforma hardware y existe una dificultad notable a la hora de modificar las interrelaciones entre los distintos módulos con el fin de poder realizar comparaciones de eficiencia.

La herramienta desarrollada es un pseudo-planificador sobre Linux con una política similar a la de Round-Robin, pero con repartos variables para los diferentes procesos. Así, dotando a cada uno de ellos (o únicamente a los permitidos) de la capacidad de asignar al resto (o también sólo a los permitidos) una fracción del periodo de pseudo-planificación (la cual también puede ser nula), se consigue emular cualquiera de las concepciones actuales en cuanto a relaciones entre módulos.

Por problemas de accesibilidad en la mayoría de los robots móviles comerciales, se decidió desarrollar una plataforma hardware completamente abierta en la que poder probar nuestras ideas. Para ello y utilizando como base una placa con Pentium a 133 MHz, se han ido diseñando y añadiendo tarjetas electrónicas para dotar al robot tanto de control del movimiento como de capacidad de detección del entorno. También en este hardware se ha buscado la máxima modularidad y accesibilidad, de modo que se facilitara su manejo por la herramienta software propuesta.

Después de decidir la estrategia de movimiento para nuestro robot (movimientos en línea recta y rotaciones sobre su centro geométrico), se decidió utilizar un esquema de control muy utilizado en las máquinas CNC actuales para el control de varios ejes, es decir, un control por referencias cruzadas. Mediante este esquema, cada uno de los lazos no sólo posee realimentación de su propio eje, sino también una referencia del otro. De esta manera, las posibles perturbaciones que pudieran afectar a cualquiera de los bucles de control durante un determinado movimiento tendrán respuesta por parte de ambos ejes, minimizando los posibles errores de orientación.

Una aportación significativa en este capítulo es el desarrollo analítico de las ecuaciones que garantizan la estabilidad del sistema de control frente a perturbaciones de tipo estático (como por ejemplo fuerzas de fricción diferentes entre ejes o distribuciones asimétricas de la carga) o dinámico (como imperfecciones en el suelo o posibles colisiones de cualquier rueda con cualquier pequeño objeto) garantizando además un error nulo tanto en la orientación

como en la posición (sin tener en cuenta los errores debidos a posibles imprecisiones mecánicas ni los debidos a derrapes).

Las conclusiones más importantes son en primer lugar que el diseño de una plataforma de prueba para arquitecturas diversas debe tener en cuenta desde el principio la integración entre el hardware y el software, tanto el de bajo nivel (sistema operativo) como las sucesivas capas que se pretendan implantar con posterioridad, y en segundo lugar que un pseudo-planificador de reparto variable se manifiesta como una herramienta perfectamente válida y flexible para la implementación y comparación de las diversas arquitecturas software de control de robots móviles existentes en la actualidad.

# Índice General


**1 Introducción y objetivos. 1**

**2 Estado de la investigación. 7**
2.1 Introducción. . . . . . . . . . 7
2.2 Evolución histórica en el pensamiento y en la tecnología. . . . . . . . . . 8

**3 El hardware del sistema. 23**
3.1 Introducción. . . . . . . . . . 23
3.2 Descripción genérica. . . . . . . . . . 23
3.3 Sensorización genérica de un robot móvil. . . . . . . . . . 25
3.3.1 Consideraciones generales. . . . . . . . . . 25
3.3.2 Sensores de luz. . . . . . . . . . 27
3.3.3 Sensores de fuerza. . . . . . . . . . 31
3.3.4 Sensores de sonido. . . . . . . . . . 32
3.3.5 Sensores de posición y orientación. . . . . . . . . . 33
3.3.6 Sensores específicos. . . . . . . . . . 35
3.4 El hardware de RODNEY. . . . . . . . . . 36
3.4.1 La tarjeta de control de puertos de E/S. . . . . . . . . . 36
3.4.2 El control de tracción. . . . . . . . . . 37
3.4.3 El sistema odométrico. . . . . . . . . . 41
3.4.4 El módulo sonar. . . . . . . . . . 44
3.4.5 Los microinterruptores. . . . . . . . . . 47
3.4.6 El display. . . . . . . . . . 49

**4 El sistema de control. 51**
4.1 Introducción. . . . . . . . . . 51
4.2 Consideraciones en el diseño de robots móviles. . . . . . . . . . 51
4.3 Análisis del bucle de control de velocidad. . . . . . . . . . 53
4.4 Análisis del bucle de control de posición. . . . . . . . . . 62
4.5 El control del movimiento. . . . . . . . . . 68
4.6 Análisis del sistema de control. . . . . . . . . . 70
4.7 Mejoras en el sistema de control. . . . . . . . . . 78

**5 El pseudo-planificador. Una herramienta para la comparación de arquitecturas de robots móviles. 87**
5.1 Arquitecturas de control como secuencias de procesos. . . . . . . . . . 87

5.2 Descripción del mecanismo de pseudo-planificación. . . . . 89
5.3 El pseudo-planificador y su implantación. . . . . 91

**6 Implementación de diversas arquitecturas de control de robots móviles usando el pseudo-planificador. 101**
6.1 Navegación basada en casos . . . . . 101
6.2 Arquitectura de Subsumción . . . . . 107
6.3 Arquitectura Teleo-Reactiva . . . . . 125

**7 Conclusiones y trabajo futuro 131**

**A Código fuente 137**

**B Esquemáticos 141**
B.1 Esquemático 1 . . . . . 141
B.2 Esquemático 2 . . . . . 142
B.3 Esquemático 3 . . . . . 143
B.4 Esquemático 4 . . . . . 144

**C Características del LS7166 145**

**Bibliografía 147**

# Índice de Figuras

2.1 Modelo general para la descomposición funcional . . . . . . . . . . . . . . . 11
2.2 Modelo general para la descomposición modular horizontal . . . . . . . . . 13
2.3 Procedimiento general para el posicionamiento basado en marcas . . . . . . 17
2.4 Procedimiento general para el posicionamiento basado en mapas . . . . . . 17
2.5 Construcción de la burbuja y detección de bordes virtuales y reales. . . . . 19

3.1 (a)Esquema de fotoresistencia. (b) Esquema de fotodiodo. (c) Esquema de fototransistor. (d) Posible conexión de fotoresistencia. . . . . . . . . . . . . 27
3.2 Posible montaje de sensores de infrarrojos con la pareja emisor-receptor SFH 484-GP1U52X . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.3 Diagrama de señales emisor-receptor . . . . . . . . . . . . . . . . . . . 29
3.4 Señal típica de un sensor . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.5 Circuito básico para microinterruptores . . . . . . . . . . . . . . . . . . 31
3.6 Circuito básico para conexión de micrófono y posible señal de salida . . . . 32
3.7 Circuito para la detección del nivel de batería . . . . . . . . . . . . . . . 35
3.8 Diagrama de bloques del circuito generador de PWM . . . . . . . . . . . 38
3.9 Diagrama de bloques para la etapa de potencia . . . . . . . . . . . . . . 38
3.10 Diagrama de bloques del control del sentido de giro de los motores . . . . . 39
3.11 Diagrama de bloques del LS7166 . . . . . . . . . . . . . . . . . . . . . 43
3.12 Respuesta en frecuencia en emisión y en recepción de las cápsulas ultrasónicas utilizadas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.13 Configuración del NE555 como astable . . . . . . . . . . . . . . . . . . 45
3.14 Esquema genérico del circuito de receptores. . . . . . . . . . . . . . . . 46
3.15 Diagrama temporal de las principales señales para la serie 6500. . . . . . . 48
3.16 Diagrama de bloques del módulo sonar por tiempo de vuelo. . . . . . . . . 48
3.17 Esquema del circuito correspondiente a los visualizadores. . . . . . . . . . 49

4.1 Diagrama del robot móvil en estudio . . . . . . . . . . . . . . . . . . . 52
4.2 Diagrama de bloques del bucle de control de velocidad . . . . . . . . . . . 54
4.3 Esquema genérico de un bucle de control digital . . . . . . . . . . . . . . 54
4.4 Diagrama de bloques considerando la entrada perturbadora y regulador P. 56
4.5 Diagrama de bloques considerando la entrada perturbadora y regulador PI. 59
4.6 Respuesta del bucle de control de velocidad con ganancia insuficiente. . . . 62
4.7 Respuesta del bucle de control de velocidad sobre Rodney. Caso 2. . . . . . 63
4.8 Respuesta del bucle de control de velocidad sobre Rodney. Caso 3. . . . . . 63
4.9 Respuesta del bucle de control de velocidad sobre Rodney. Caso 4. . . . . . 64

4.10 Respuesta del bucle de control de velocidad sobre Rodney. Caso 5. . . . . . 64
4.11 Respuesta del bucle de control de velocidad sobre Rodney. Caso 6. . . . . . 65
4.12 Diagrama de bloques del esquema de control de posición . . . . . . . . . . 65
4.13 Respuesta de la implementación del bucle de control de posición sobre Rodney. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.14 Ampliación de la respuesta del control de posición donde se puede observar el error en pulsos. . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.15 Procedimiento propuesto para viajar desde un punto origen a uno destino . 69
4.16 Bucle de control para el robot móvil . . . . . . . . . . . . . . . . . . . . . . 71
4.17 Diagrama de bloques del sistema de control completo . . . . . . . . . . . 72
4.18 Comportamiento del modulador por anchura de pulso . . . . . . . . . . . 73
4.19 Rango de valores permitido para las ganancias $K_p$ y $K_c$ . . . . . . . . . . 78
4.20 Nuevo esquema de control propuesto . . . . . . . . . . . . . . . . . . . 79
4.21 Evolución de la señal de error con un funcionamiento estable del esquema de control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.22 Evolución de la señal de error con un funcionamiento inestable del esquema de control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

5.1 Posible evolución en los repartos de los procesos con el tiempo. . . . . . . 99

6.1 Evolución de los repartos de los procesos en funcion del tiempo. Fase 1 . . 107
6.2 Evolución de los repartos de los procesos en funcion del tiempo. Fase 2 . . 108
6.3 Evolución de los repartos de los procesos en funcion del tiempo. Fase 3 . . 108
6.4 Evolución de los repartos de los procesos en funcion del tiempo. Fase 4 . . 109
6.5 Evolución de los repartos de los procesos en funcion del tiempo. Fase 5 . . 109
6.6 Evolución de los repartos de los procesos en funcion del tiempo. Fase 6 . . 110
6.7 Evolución de los repartos de los procesos en funcion del tiempo. Fase 7 . . 110
6.8 Evolución de los repartos de los procesos en funcion del tiempo. Fase 8 . . 111
6.9 Evolución de los repartos de los procesos en funcion del tiempo. Fase 9 . . 111
6.10 Evolución de los repartos de los procesos en funcion del tiempo. Fase 10 . . 112
6.11 Evolución de los repartos de los procesos en funcion del tiempo. Fase 11 . . 112
6.12 Evolución de los repartos para el proceso TEST a lo largo de toda la maniobra. 113
6.13 Evolución de los repartos para el proceso ROTAR a lo largo de toda la maniobra. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.14 Evolución de los repartos para el proceso AVANZAR a lo largo de toda la maniobra. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.15 Evolución de los repartos para el proceso MOVER a lo largo de toda la maniobra. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.16 Diagrama de bloques y niveles para una arquitectura de subsumción básica. 116
6.17 Maniobra a realizar como ejemplo de utilización del pseudo-planificador en la implementación de una arquitectura reactiva. . . . . . . . . . . . . . 117
6.18 Evolución del reparto para el proceso FEELFORCE. . . . . . . . . . . . 119

6.19 Evolución del reparto para el proceso VAGABUNDEAR. . . . . . . . . . . 120
6.20 Evolución del reparto para el proceso AVOID. . . . . . . . . . . . . . . . . 120
6.21 Evolución del reparto para el proceso RUNAWAY. . . . . . . . . . . . . . . 121
6.22 Disposición de objetos para el experimento de arquitectura teleo-reactiva. . 127

7.1 Diagrama de bloques del esquema de control propuesto incluyendo el módulo de giro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134

# Capítulo 1

# Introducción y objetivos.

La Robótica móvil se puede considerar como la parte de la Robótica que se encarga del estudio de los vehículos móviles y su relación con el entorno.

Si bien en sus inicios, la idea de la construcción de vehículos móviles con capacidad suficiente como para desenvolverse satisfactoriamente en entornos relativamente estructurados se plantea como una meta idealista especialmente por causas relativas al hardware (pesado y lento), a lo largo de la última década la evolución ha sido espectacular, sin duda animada por el desarrollo tecnológico en especial en lo referente al capítulo sensorial.

En la actualidad y como consecuencia de tratarse de una materia multidisciplinar, en la gran mayoría de las Universidades y Centros de Investigación a nivel mundial se trabaja de una u otra manera con temas relacionados con esta disciplina. Así, los tópicos más comúnmente tratados van desde aspectos de tipo mecánico como pueden ser posibles configuraciones físicas de robot, grados de libertad o incluso distintos medios de desplazamiento (aéreos, submarinos,...) hasta la evolución en el comportamiento de colonias de robots, pasando por los métodos para adquirir y representar la información del entorno, el aprendizaje, los sistemas de guiado y la estimación de la posición dentro del entorno de trabajo, las diferentes maneras de relacionar los distintos módulos que conforman el robot completo (arquitecturas) y muchos otros.

Dentro de este inmenso campo de estudio y realizando una visión retrospectiva de nuestro trabajo, la posición de partida y los desarrollos posteriores han sido los siguientes:

Observando a través de la bibliografía la evolución de las distintas configuraciones organizativas para los diferentes módulos que conforman el sistema software completo (e incluso software-hardware) para un robot móvil, podemos llegar a la conclusión de la existencia de diferentes puntos de vista a la hora de configurar las relaciones entre éstos.

Más aún, podemos observar que en muchos casos, una arquitectura en particular debe (o al menos es conveniente que debiera) estar implementada aprovechando determinadas características hardware.

Existe un número no muy grande, pero aún considerable de arquitecturas, cada una (o al menos algunas de ellas) asociadas a determinadas necesidades hardware (lo cual nos lleva a plataformas particulares). Esto puede dificultar considerablemente la comparación entre ellas respecto a su eficiencia en presencia de restricciones a la hora de realizar una determinada tarea.

Esta es, esencialmente una tesis sobre arquitecturas para el control de robots, por ello, para fijar desde el principio las ideas básicas sobre las que se sustenta, es necesario dar una definición concreta de arquitectura. Diremos que una **arquitectura** de control de robots es un conjunto de reglas expresables algorítmicamente, que establecen las relaciones entre los procesos elementales que controlan al robot, indicando en función de directrices internas, de información del mundo externo y de la historia anterior, qué módulo(s) debe(n) estar activo(s) y qué señales debe(n) enviarse a los actuadores en cada momento.

De acuerdo con esta definición, el objetivo principal del presente trabajo fue encontrar una herramienta, la cual pudiese ser utilizada para la implementación de cualquiera de las configuraciones modulares actuales. Así, puesto que, según argüiremos después, las diferencias entre ellas siempre pueden ser descritas en función del orden en la activación de los procesos elementales que la conforman, del tiempo de procesador asignado a cada uno y de las prioridades entre ellos, el desarrollo de un ejecutor cíclico utilizando una variante del algoritmo de Round-Robin que permitiera a todos o a algunos de los procesos el cambio de los repartos de forma dinámica durante su ejecución podía ser la solución buscada. El fundamento se basa en la existencia de un proceso supervisor, el cual se activa periódicamente. Este proceso mantiene una tabla con el reparto asignado a cada uno de los procesos que puede ejecutarse, además de una cuenta de los pulsos de reloj transcurridos desde la última llamada, un registro del proceso que actualmente está en ejecución y un flag indicando si la tabla de repartos ha sido modificada.

El primer problema con el que nos encontramos a la hora de realizar nuestro trabajo es el hecho de que los robots móviles disponibles comercialmente en la actualidad, en su gran mayoría o no eran completamente accesibles en cuanto al hardware y/o software o poseían unos precios prohibitivos para nosotros, de manera que (y sobre todo) por la primera razón, nos embarcamos en el diseño y construcción de una plataforma genérica y de muy bajo coste.

El robot móvil construido, aún siendo un prototipo, es el resultado de la implementación de distintos tipos de sistemas de tracción, sensorial y de proceso. Se trata de una base con un hardware conocido en su intimidad (puesto que ha sido diseñado y construido

por nosotros mismos), completamente accesible y flexible, de manera que el programador puede utilizar a voluntad cualquiera de los recursos disponibles.

Después de disponer de un robot sobre el que trabajar, había que elegir un sistema operativo con garantías bajo el cual pudiésemos realizar nuestro pseudo-planificador. Este debía ser robusto, estar abierto, bien documentado y fácilmente modificable. Todos estos requisitos nos hicieron decidirnos por Linux.

Así, el objetivo principal del trabajo trata del desarrollo de un pseudo-planificador sobre Linux (para lo cual habría que realizar las modificaciones pertinentes sobre dicho sistema operativo) mediante el cual poder realizar la implementación de diversas arquitecturas con el fin de facilitar la comparación entre ellas en la realización de diferentes tipos de tareas, utilizando nuestra plataforma hardware.

La evolución de los acontecimientos ha hecho que determinadas partes del desarrollo se hicieran lo suficientemente extensas e importantes como para ocupar por sí mismas un capítulo completo dentro del trabajo final. En particular, los capítulos referentes al hardware y al sistema de control.

Después de haber explicado la motivación y los objetivos deseados, a continuación se comenta brevemente cada uno de los capítulos de que consta el presente trabajo.

En el capítulo 2 se pretende presentar una visión de la evolución histórica desde dos de los aspectos principales que confluyen en la robótica móvil: por un lado la evolución de las ideas y los procedimientos para resolver los diferentes tópicos que se presentan en el estudio de esta materia, y por otro la evolución tecnológica especialmente en el desarrollo y aplicación de dispositivos sensoriales. La conclusión final a la que se puede llegar fácilmente respecto a esto último es que la gran cantidad de sistemas de sensorización y de tipos distintos de materiales asociados a ellos existentes en la actualidad, no hacen más que demostrar la inexistencia de un tipo de sensor que por sí sólo represente la información del entorno con la suficiente riqueza y fiabilidad como para ser utilizado como único elemento sensor.

En este capítulo también se pretende insinuar el hecho de que en la comunidad científica existe cierta divergencia (divergencia histórica) no sólo a la hora de resolver determinado tipo de problemas, sino incluso a la hora de plantearlos (por ejemplo, ¿es necesario que el robot conozca su posición y orientación exacta dentro de su entorno de trabajo? o ¿es posible realizar cualquier estudio sobre estos temas mediante simulación exclusivamente?). Las principales discrepancias se dan entre los dos principales grupos de investigación: el relacionado con la IA por un lado y el de los ingenieros robóticos por el otro.

En el capítulo 3 se presenta el sistema de control implementado en el robot. El sistema de control trata de implementar la estrategia de movimiento deseada. En particular, se decidió dotar a nuestro robot de dos tipos de movimiento: movimientos en línea recta y movimientos rotacionales sobre su centro geométrico.

En primer lugar se analizan los bucles de control de velocidad y de posición para motores de CC. y los resultados obtenidos sobre nuestro robot en particular. A continuación se presenta el esquema de control por referencias cruzadas utilizado inicialmente po J.Borenstein y Y. Koren [BK87] en su robot enfermera y que trata de analizar la estabilidad del sistema de control para seguir trayectorias rectas o rotaciones sobre su centro geométrico frente a perturbaciones externas o internas (como distribuciones asimétricas en la carga). Dicho sistema de control garantiza un error de orientación nulo en régimen permanente, pero no garantiza otra restricción que puede ser importante y también en régimen permanente que es el error de posición (básicamente, que el robot viaje a la velocidad deseada, además de no desviarse de su trayectoria).

Por último se presenta un nuevo esquema de control que complementa al anterior, de manera que se realizan los desarrollos analíticos necesarios para obtener el conjunto de ecuaciones que deben cumplirse para garantizar la estabilidad del sistema frente a perturbaciones temporales o contínuas garantizando un error nulo tanto en la posición como en la orientación.

Puesto que no está realizado el proceso de identificación exacto de todos los bloques que conforman el sistema de control, dichas ecuaciones no están acotadas numéricamente (esto se plantea como trabajo futuro), pero en la práctica es relativamente fácil encontrar unos valores para los coeficientes de los reguladores que hagan estable al sistema. Otro aspecto interesante a comentar es el hecho de la conveniencia de elegir valores para los reguladores que situen el punto de funcionamiento lejos de las rectas frontera de estabilidad, puesto que de esta forma, la respuesta frente a las perturbaciones se aleja del comportamiento oscilatorio.

Por último en este capítulo se puede observar que el hecho de añadir la restricción en régimen estático de un error de posición nulo frente a una entrada escalón (velocidad de referencia) aumenta notablemente el número de ecuaciones que deben cumplirse para obtener un sistema estable, de manera que, además de descartar el añadir restricciones de tipo dinámico, incluso para la mayoría de los casos puede descartarse esta restricción de tipo estático (sobre todo cuando el error de posición puede hacerse suficientemente pequeño eligiendo un valor adecuado de la constante de ganancia proporcional).

En el capítulo 4 se presentan las ideas que pensamos pueden desembocar en el desarrollo de una herramienta para facilitar la programación de distintos tipos de arquitecturas de robots móviles y la herramienta misma: el pseudo-planificador.

Se trata de un ejecutor cíclico con una política similar a la del algoritmo de Round-Robin, pero con reparto variable entre procesos sobre el sistema operativo Linux. También se presentan las estructuras principales de datos entre las que cabe citar fundamentalmente la tabla en la que se almacenan los datos referentes a cada uno de los procesos con su tipo asociado, además de su reparto temporal actual y el deseado para el siguiente o los siguientes ciclos del pseudo-planificador.

Mediante esta herramienta, comprobamos que nuestras ideas iban bien encaminadas, de manera que tras el desarrollo del pseudo-planificador, verificamos la facilidad de programar diversas políticas de organización modular sobre la misma plataforma hardware.

En el quinto y penúltimo capítulo, se trata de mostrar cómo utilizar el pseudo-planificador en el proceso de implementación de diversas arquitecturas software sobre robots móviles. En particular se muestran tres concepciones organizativas modulares con sus correspondientes ejemplos prácticos de aplicación. En la primera de ellas se trata de implementar una organización eminentemente jerárquica (la navegación basada en casos). El ejemplo práctico que se presenta para implementar este tipo de organización es el del robot avanzando a lo largo de un pasillo. El robot debe ser capaz de moverse a lo largo del pasillo sin chocar con las paredes y siempre que encuentre una puerta abierta a su izquierda, debe atravesarla. El proceso acaba cuando encuentra un obstáculo frontal a una distancia de 45 cm (aproximadamente).

La segunda aplicación práctica que se presenta es el de la implementación de una arquitectura de tipo reactivo, en particular los niveles 0 y 1 de la arquitectura de subsumción de Brooks [Bro85], donde el nivel 0 trata de evitar la colisión del robot con los obstáculos circundantes y el nivel 1 trata de guiar al robot desde una posición y orientación inicial hasta una posición y orientación final. El resultado final, debe ser el de llevar al robot desde un punto hasta otro sin chocar.

Por último se implementa la arquitectura teleo-reactiva propuesta por Nilsson [Nil94] mediante la cual se trata de guiar al robot hacia un objetivo determinado, a la vez que se tienen en cuenta las circunstancias de cambio en el entorno. La aplicación práctica que implementa este tipo de organización mediante nuestro pseudo-planificador es la del seguimiento por parte del robot de cualquier objeto móvil detectado por los sensores ultrasónicos frontales y siempre manteniendo una distancia de seguridad.

Por último, en el capítulo 6 se presentan las conclusiones y el trabajo previsto para el futuro próximo. En cuanto a las conclusiones, cabe destacar el hecho de que el pseudo-planificador desarrollado se manifiesta como una herramienta potente y fácil de utilizar a la hora de programar la implementación de cualquiera de las arquitecturas software existentes en la actualidad sobre una plataforma hardware estandar, abierta y flexible como es la nuestra. En cuanto al apartado de trabajos futuros cabe destacar la propuesta

de utilización de algoritmos de tipo evolutivo (algoritmos genéticos) con el fin de poder realizar el aprendizaje del sistema de repartos entre módulos con fines de optimización en función (entre otros aspectos) de la misión a realizar.

# Capítulo 2

# Estado de la investigación.

## 2.1 Introducción.

En este capítulo se pretende dar una breve descripción de los trabajos que a nuestro criterio, han presentado ideas o implementaciones más o menos relevantes con respecto a nuestros objetivos dentro del campo de la investigación robótica en general y de la robótica móvil en particular.

De entre la gran cantidad de publicaciones que se realizan en la actualidad relacionadas con este campo de la ingeniería, podríamos distinguir dos principales tendencias o líneas de investigación. Una de ellas trata fundamentalmente con la estructura lógica y funcional de los principales módulos que pueden componer un robot (robot móvil). Bajo este grupo se incluyen principalmente las presentaciones que muestran las principales concepciones organizativas de un robot (arquitecturas). Los principales grupos de investigación que se han encargado (y continúan encargándose) de engrosar este bloque son por una parte los teóricos de la Inteligencia Artificial y por otro los Ingenieros robóticos, cada uno de ellos con su visión más o menos particular de planteamiento y resolución de los problemas y con posturas no siempre convergentes.

La otra fuente principal de publicaciones está relacionada con el desarrollo tecnológico (fundamentalmente con la evolución en tecnología sensorial) y su aplicación en el desarrollo y mejora de los algoritmos que resuelven los problemas típicos con los que se enfrenta un robot móvil (evitación de colisiones, detección de obstáculos, planificación de trayectorias, generación de mapas del entorno, etc).

## 2.2 Evolución histórica en el pensamiento y en la tecnología.

Los robots móviles se utilizan de manera todavía muy escasa en la industria actual. De todas formas, aunque este tipo de máquinas suelen emplearse mayoritariamente en entornos cerrados y con unas condiciones particularmente buenas principalmente debido a que la inmensa mayoría de los tópicos con que se trata no están resueltos o al menos no lo están de manera definitiva, también es cierto que en determinadas industrias y especialmente para algunas labores específicas ya se están comenzando a utilizar algoritmos y sistemas más flexibles que puedan afrontar variaciones más o menos restringidas de las condiciones externas. Así, paralelamente a la utilización de los robots industriales en tareas esencialmente de montaje, pintura y soldadura donde la característica principal es la repetición de las acciones preprogramadas sin ningún tipo de variación (o a lo sumo con el uso de sensores cuya información detiene el robot en caso de colisión o ajusta la fuerza del brazo), los robots móviles se utilizan principalmente en el campo de la fabricación, en el de defensa y en el de exploración remota (inspección en centrales nucleares,...) normalmente para la realización de tareas especialmente de transporte y almacenaje de objetos localizados en posiciones previamente conocidas y viajando a lo largo de caminos generalmente fijos, casi siempre formados por carriles filoguiados utilizando algún tipo de señal eléctrica o magnética. Muchos de los problemas relacionados con el control de bajo nivel (modelización,regulación,robustez,estabilidad,etc) han sido planteados y resueltos mediante muy diversos métodos con el fin de poder ser utilizados en estos sistemas. Esto ha permitido una mejora técnica importante. Pueden verse ejemplos de ello en [Mea82]. No obstante, tal clase de robots carece por completo de cualquier comportamiento que pudiéramos llamar inteligente y en este sentido están más próximos a las máquinas-herramienta que a la moderna concepción de robot.

Normalmente, cuando nos referimos a robots en general y a robots móviles en particular, nos referimos a ellos como si se tratara de sistemas autónomos inteligentes, porque desde sus comienzos, más allá de cualquier intento de construir un robot, siempre se ha tenido la presunción de que de hecho era posible realizar tal cosa. Así, los grandes debates desde siempre han sido y siguen siendo, en primer lugar qué se entiende por inteligencia, en segundo lugar si los humanos somos capaces de crear (construir) sistemas inteligentes y por último y llegado el caso, cómo implementar el comportamiento inteligente en una máquina construida por el hombre.

En los comienzos de la investigación en inteligencia artificial (IA), muchos de los pioneros creyeron que la IA y la robótica estaban íntimamente unidas. Por ejemplo, tanto Marvin Minsky como John McCarty trabajaron en proyectos relacionados con robots.

A finales de los 50 Minsky, junto con Richard Greemblatt y William Gosper se embarcaron en un proyecto para la construcción de un robot que jugara al ping-pong, pero pronto consideraron que el proyecto era impracticable porque su brazo robot era demasiado lento. Posteriormente, Minsky y sus colegas abordaron el problema de la manipulación en un entorno formado por bloques, hecho por el cual se les conoce como los precursores en el problema del ensamblaje robótico. Un aspecto importante de este tipo de problemas era que favorecían la descomposición en subtareas de manera que varios aspectos del robot en su totalidad podían ser estudiados por simulación en ausencia de algunas de las partes. Realizando una visión retrospectiva, muchos investigadores actuales coinciden en que aunque en un principio este hecho podría haber sido una buena razón para tratar el estudio del sistema robótico en su conjunto, finalmente todo quedó reducido a un desarrollo relativo únicamente a problemas relacionados con la ciencia del conocimiento: razonamiento, representación, etc.

Otros precursores en la investigación en IA tuvieron experiencias similares. La conclusión general fue que la construcción de robots reales (la conjunción entre mecánica y electrónica) era demasiado difícil y en cualquier caso dependiente en gran medida de la tecnología. La pregunta científica importante era cómo hacer inteligente al robot. Una concepción comúnmente admitida fue que la inteligencia del robot se implementara como un determinado mecanismo de procesamiento de información ('programa') ejecutándose en un ordenador; así, el hardware del computador sería el cerebro mientras que el software sería la mente. Consecuentemente los investigadores en IA se concentraron en el estudio de la representación simbólica pura mientras que los ingenieros roboticistas se dedicaron al desarrollo e implementación de las nuevas tecnologías (especialmente sensorial) y a su aplicación a la resolución de los problemas que la industria del momento les pedía.

Más allá de esta divergencia entre IA y robótica estaba la idea de que la inteligencia de un robot debía residir en un determinado esquema para el procesamiento de la información, mientras que la manipulación real de objetos físicos sería una mera cuestión tecnológica, cuando se dispusiera del hardware adecuado. Este modelo de la mente que la IA construyó estaba basado en en el razonamiento simbólico sobre modelos del mundo, conocido a veces como GOFAI. [1] La idea básica subyacente a esta visión (o al menos a la parte más pura de ella) se enuncia en la hipótesis del Sistema Físico de Símbolos (PSSH) de Newell y Simon [NS89]

> Un sistema físico de símbolos posee los medios necesarios y suficientes para producir acción inteligente general.

Esto conduce a la noción de Inteligencia Artificial como

[1] Good Old Fashioned Artificial Intelligence (la buena y vieja Inteligencia Artificial)

El estudio de los sistemas de símbolos con el propósito de entender e implantar en ellos una búsqueda inteligente [LS89].

Lo anterior quiere significar que, si contruimos un sistema de símbolos (proposiciones u otros) capaz de modelizar suficientemente bien un determinado aspecto de la realidad, podremos razonar sobre los modelos, de manera que el resultado nos indique no sólo el comportamiento del sistema real, sino también cómo influirá sobre él cualquiera de nuestras actuaciones. De este modo, en el caso más general de un modelo del mundo físico en su conjunto, podríamos obtener inteligencia general (en el sentido humano del término).

Así, el primer paradigma en la investigación robótica nace a raiz de esta concepción del modelo de mente y la consecuencia fundamental es la necesidad de construir una representación abstracta (básicamente un conjunto de símbolos) que sea propuesto como un modelo del mundo real en el que el robot trate de operar. Es lógico pensar, (al menos en un primer momento,) que un modelo geométrico puede ser adecuado, dado que la mayoría de los objetos que nos rodean son objetos construidos por el hombre.

El problema esencial que entonces se plantea es pues, cómo ir desde los datos adquiridos por los sensores a una descripción de los objetos, ya sea geométrica o topológica. Se supone que un planificador de alto nivel debería trabajar con estos datos y tendría que generar una secuencia de acciones, normalmente dadas en forma de movimientos del robot, que describieran las subtareas ordenadas que constituyen la tarea completa. Un paso adicional convertiría estas especificaciones de la tarea en movimientos del robot.

Esta primera visión organizativa de los procesos a realizar (arquitectura) viene formada explícitamente por tres subsistemas: percepción, planificación y control, ciclo al cual se le refiere en la literatura como ciclo sentir-pensar-actuar. Este tipo de descomposición vertical es conocido como descomposición funcional y puede verse un esquema de él en la figura 2.1. Pat Hayes describe ya a finales de los 60 este punto de vista [Hay69].

Los inconvenientes en la construcción de un modelo simbólico para este tipo de descomposición parecen evidentes simplemente observando la longitud y complejidad del proceso a seguir. En primer lugar, los datos son tomados con ruido (la distancia medida por los sensores ultrasónicos puede estar afectada entre otras cosas por rebotes de señales procedentes de sensores vecinos, las medidas odométricas pueden no describir exactamente (de hecho no lo pueden hacer por sí mismas) la posición y orientación del robot respecto de un sistema de coordenadas de referencia a causa de (entre otras cosas) cualquier pequeño derrape, etc). Lógicamente, datos imprecisos generan descripciones simbólicas erróneas, y éstas son difíciles de encajar en los modelos almacenados. Además, una descomposición secuencial implica que cada paso no puede comenzar hasta que el anterior

Figura 2.1: Modelo general para la descomposición funcional

no haya terminado y que el fallo de uno sólo de ellos, hace fracasar el proceso completo.

Es a finales de los 80 cuando comienza a tomar fuerza una nueva concepción de la implementación de los modelos inteligentes, aunque bien es cierto que la idea ya aparece de modo minoritario e incluso un tanto herético casi desde los principios de la Inteligencia Artificial (por ejemplo la discusión de Minsky de 1961 relativa a la naturaleza de la inteligencia expuesta en [Min61]). Este nuevo paradigma que emerge al cual se le denomina sintético por oposición al clásico, analítico, establece que las facultades mentales son el resultado de comportamientos, entendidos éestos como los actos que pueden conseguir la ejecución correcta de una determinada acción, resultando tal ejecución en la interacción entre el robot (o el ser vivo) y su entorno. Estos comportamientos pueden ser o bien directamente implantados en el sistema como partes de él, (a los cuales se les denomina módulos comportamentales) o bien pueden resultar de la acción concurrente de varios de tales módulos. Una buena referencia sobre la toma de conciencia de la emergencia de este nuevo paradigma (nueva arquitectura) se presenta en [MSH89].

Como parece obvio, este nuevo punto de vista concede una importancia excepcional a las relaciones con el entorno, las cuales se realizan a través de los órganos sensoriales. En el aspecto práctico, a la hora de construir un robot, la Robótica Comportamental prescindirá de programar explícitamente un comportamiento orientado a un determinado propósito. Más bién, programará reflejos o comportamientos muy simples, pero eficaces,

que en conexión directa con los sensores y los actuadores, generen mediante su interacción, el comportamiento más adecuado para el caso. Los ejemplos más espectaculares son los trabajos de Brooks, tanto con sus robots insectoides [Bro89] como con su robot móvil para coger vasos vacíos [BCH88].

De esta manera, los programas pueden ser construidos usando módulos comportamentales, los cuales deben estar formados no simplemente por programas, sino más bien por bloques programa-soporte físico (sensores y actuadores) para usar y (en su caso) cambiar el entorno, que de este modo pasa a formar también parte del módulo. Sus entradas serán no sólo parámetros, sino también lecturas de los sensores. Y similarmente, sus salidas serán señales para los diferentes actuadores los cuales podrán realizar cambios en el mundo. A su vez, estos cambios eventualmente pueden provocar diferentes lecturas en determinados sensores, con lo cual es posible que alguno o algunos de los módulos que permanecían en estado latente se activen mientras que otro u otros que estaban activos pasen a estado latente. De esta manera es posible la emergencia de nuevos comportamientos.

La mayor parte de las veces el sistema puede incluso ser perfectamente ignorante sobre lo que ha conseguido. El único requerimiento importante es que cada módulo sea eficiente y tan autocontenido como sea posible. Es dentro del módulo donde se realiza la fusión de los datos de diferentes sensores (si los hay) y la contención del ruido. Pues bien, a la arquitectura de control distribuido que conecta los distintos módulos se la denomina *arquitectura de subsumción* [Bro91b], en el sentido de que cada módulo en muchas ocasiones subsume los comportamientos del anterior, de manera que si alguno falla, es posible que el robot todavía sea capaz de ejecutar algunos de sus módulos, (es decir, pueda presentar algunos comportamientos) o al menos no pararse completamente; esto se conoce como degradación suave ("graceful degradation"). De esto se puede encontrar más información en [Mal91].

En lo referente a la construcción de modelos, existen diferentes posiciones más o menos extremistas entre los partidarios de la Robótica Comportamental. Desde los que defienden, como Brooks, que deben ser evitados al máximo debido a que la mayoría de las tareas normales en un robot pueden ser ejecutadas sin ellos, (vease [Bro91b]) hasta los que transigen en su uso, especialmente si se trata de modelos mínimos [Mal91].

Por todo lo dicho anteriormente, se entiende que la Robótica Comportamental busque habitualmente su fuente de inspiración en la biología. Sus partidarios piensan que es mejor estudiar primero criaturas completas intentando reproducir cada una de sus capacidades senso-motoras para hacer que trabajen o bien separadamente o bien en conjunto. Un esquema de cómo organiza la Robótica comportamental una aplicación típica en un robot móvil puede verse en la figura 2.2.

Por último, a lo largo de la última década han ido surgiendo posiciones más o menos

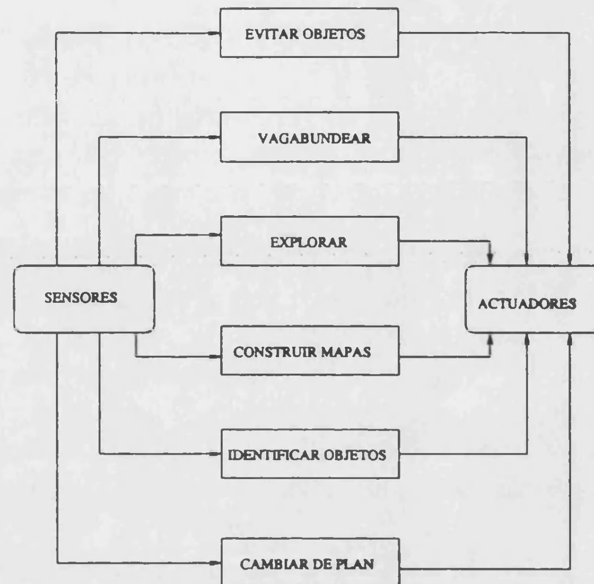



Figura 2.2: Modelo general para la descomposición modular horizontal

intermedias entre estas dos posturas. Así, mientras los partidarios del punto de vista clásico continúan defendiéndolo como una concepción perfectamente válida (véase la réplica a Brooks dada en [Etz93] a raiz de los artículos de éste [Bro91b],[Bro91a]) y los partidarios del punto de vista comportamental continúan desarrollando aplicaciones de su concepción en diversas configuraciones de robot (pueden verse algunos de tales ejemplos en [Arka],[Arkb],[BA]), también es cierto que cada vez son más los grupos de investigación partidarios de optar por soluciones mixtas.

En este caso, se trata de tomar las dos arquitecturas anteriores y generar una arquitectura híbrida de manera que se puedan aprovechar las ventajas de cada una de ellas. Consecuentemente, en este tipo de arquitecturas, continúa manteniéndose una organización jerárquica en cuanto a la planificación de trayectorias, generación de mapas del entorno y en general en todo lo que se refiere a las tareas de alto nivel, mientras que las tareas de más bajo nivel (básicamente el control de la sensorización, la detección de obstáculos, la evitación de colisiones y el control del movimiento) están formadas por un conjunto de módulos cuyo comportamiento es puramente reactivo. Como claro ejemplo de esta visión de implementación, está la arquitectura AuRA, la cual se desarrolló a mediados de los 80 como una aproximación híbrida a la navegación de robots. La hibridación se realiza por la presencia de dos componentes distintos: un planificador jerárquico o deliberativo basado en las técnicas tradicionales de la IA y un controlador reactivo encargado del gobierno de los motores y la sensorización [AB].

Otro tipo de construcciones pueden estar más o menos cercanas a cada una de las

vistas anteriormente. Así, A.D. Webler y D.L. Bisset [WB] desarrollan un modelo para controladores comportamentales cuya estructura es idéntica a la reactiva, pero en la que el mecanismo de selección de comportamientos está basado en la suma de los pesos aportados por los módulos que cooperan frente a los que compiten con respecto a un determinado comportamiento (en la visión comportamental pura todos los módulos compiten entre si).

Por último comentar la arquitectura Teleo-reactiva propuesta por Nilson (Nil94) en la que la organización de los módulos se parece a un sistema de reglas de producción. Así, se comprueban las condiciones de los antecedentes y si se cumplen, se ejecuta la acción asociada. Los módulos se organizan de forma jerárquica y el cumplimiento de las condiciones de uno de ellos causa no sólo la ejecución de su propia acción asociada, sino la inhibición de las acciones de todos los módulos que estén por debajo en el nivel de jerarquía. Esto significa que en un momento dado, sólo una acción puede estar en ejecución.

En lo que respecta a la evolución tecnológica y procedural y su aplicación a la robótica móvil, podemos destacar :

Las necesidades referentes a la aplicación de algoritmos de control más o menos sofisticados van relacionadas con el esquema del control del movimiento a implementar. Por ejemplo, a partir de un planteamiento en el que lo importante es 'lo sentido' por los sensores más que las coordenadas espaciales del robot respecto de un sistema de referencia fijo, las necesidades en el ámbito del control pueden estar reducidas e incluso eliminadas (en respuesta a la detección de un obstáculo girar *un poco* en *una determinada* dirección). Además tampoco existen restricciones en cuanto a la trayectoria a seguir, velocidad, puntos de consigna, etc). Sin embargo en la mayoría de los casos, es necesario implementar algún esquema de control a fin de poder aplicar una determinada estrategia de movimiento.

Aunque la descripción detallada del esquema de control utilizado en un determinado sistema robótico tiende a omitirse, normalmente se trata de implementar el bucle típico de control de posición y/o velocidad de motores de c.c. modificando el comportamiento en régimen estático y/o dinámico del sistema (en caso necesario, que además es lo típico) mediante algún esquema de regulación (normalmente PIDs).

Una mejora importante en el esquema de control se produjo cuando se aplicaron las técnicas de control empleadas en máquinas de control numérico a los robots móviles. En este caso se trataba de coordinar el movimiento de varios ejes de manera que un determinado bucle de control estuviera afectado no sólo por su propio lazo, sino también por una referencia de los bucles correspondientes al resto de ejes (acoplo cruzado)[Kor80]. Un ejemplo de aplicación de este esquema de control es el sistema de control de un robot utilizado para maniobrar en el interior de hospitales (robot enfermera) realizado por J.

Borenstein y Y. Koren [BK87]. El robot aplica una estrategia de control del movimiento que intenta evitar el derrape y minimizar los errores de posición.

Un ejemplo de aplicación utilizando otra filosofía es el diseño de un controlador híbrido desarrollado para sistemas no holonómicos. En el momento en que se detecta un error tanto en la posición como en la orientación del vehículo (un coche), el regulador genera una serie de movimientos adelante-atrás gobernando la dirección a fin de eliminar o minimizar dicho error [ea95].

En lo referente a los métodos de navegación para el posicionamiento de un robot, la odometría es el sistema más ampliamente utilizado. Debido a que el conocimiento de la posición mediante esta técnica se realiza a través de la integración de la información, los errores son acumulativos y proporcionales a la distancia viajada. Por esta razón, Cox [Cox91], Byrne et al. [R.H93] y Chenavier y Crowley [CC92] propusieron métodos para refundir los datos odométricos con medidas de la posición absoluta para mejorar la estimación de la posición. En otros casos, se han desarrollado algoritmos para la estimación de la incertidumbre en la posición (por ejemplo [ea94c] y Ko94) mediante los cuales la posición del robot viene asociada con una elipse de incertidumbre. Normalmente esta elipse crece con la distancia viajada hasta que una medida de la posición absoluta reduce o incluso elimina dicha elipse [ea94b].

Los errores odométricos pueden clasificarse en dos categorías principales: sistemáticos y no sistemáticos [BF96], relacionados los primeros con imperfecciones cinemáticas del robot y los segundos por el contacto entre las ruedas y el suelo. La reducción de ambos tipos de errores llevaron a Borenstein y Feng [BF95] a desarrollar un método para la medida cuantitativa de los errores odométricos sistemáticos y no sistemáticos. Este método llamado UMBmark (University of Michigan Benchmark) necesita que el robot móvil se programe para seguir un cuadrado de 4x4 metros y 4 giros de 90° sobre su centro geométrico. Esto se debe realizar 5 veces en el sentido de las agujas del reloj y otras 5 en el sentido contrario. Basado en el test UMBmark, Borenstein y Feng [BF94] desarrollaron un procedimiento de calibración para reducir los errores odométricos sistemáticos.

Los robots con una distancia reducida entre las ruedas motrices son más propensos a los errores de orientación. Como ejemplo, Gourley y Trivedi [GT94] analizan el sistema odométrico del robot LabMate (el cual posee una distancia entre ruedas motrices de 340 mm.) llegando a la conclusión de que para este robot en particular, es aconsejable corregir las medidas odométricas cada 10 metros de viaje para que los errores de orientación no sean excesivamente grandes.

Un método alternativo para estimar la posición en la problemática de la navegación es la utilización de acelerómetros y giróscopos (navegación inercial). Los acelerómetros se utilizan para medir la aceleración respecto del suelo y los giróscopos para medir la

velocidad de rotación. Así, integrando una o dos veces (según se trate de la velocidad o de la aceleración) se puede obtener una estimación de la posición. El principal atractivo de este método de posicionamiento es el hecho de no necesitar información del movimiento respecto del exterior (es decir, se trata de un sistema sensorial autocontenido). Resultados experimentales llevados a cabo tanto en la Universidad de Montpellier en Francia ([ea93a],[ea93b]) como en la Universidad de Oxford en U.K. ([BW93],[BW95]) concluyen que la navegación puramente inercial no es conveniente para aplicaciones de robots móviles (por ejemplo es muy cara, además de que este tipo de sensores son muy sensibles al ruido [PG93]).

Otro sistema que se ha utilizado para la navegación de robots móviles es el denominado por guias (Active Beacon). Si bien este sistema de navegación presenta una alta precisión en cuanto al posicionamiento, también es cierto que posee algunos inconvenientes, siendo los principales el del costo de instalación y el de mantenimiento [Mad94]. Se pueden distinguir dos tipos diferentes de procedimientos de guiado: trilateralización y triangulación. En el primero de ellos, la posición del vehículo se basa en la medida de la distancia a tres o más fuentes luminosas conocidas (normalmente calculando el tiempo de vuelo), estando el transmisor montado sobre el robot. En la triangulación existen tres o más transmisores (normalmente infrarojos) colocados en posiciones conocidas del entorno. Un sensor montado en una torreta giratoria sobre el robot mide los ángulos con que éste ve a cada uno de los transmisores. A partir de esta información se puede calcular la posición y orientación del vehículo.

Cohen y Koss [CK92] realizan un análisis detallado de los algoritmos para la triangulación por tres puntos dando como resultado el rendimiento de cada uno de ellos. Por otro lado, Betcke y Gurvits [BC94] desarrollan un algoritmo denominado de estimación de la posición que resuelve el problema genérico de la triangulación, es decir, dadas N marcas y sus correspondientes M medidas de ángulos, estimar la posición del robot respecto del sistema de coordenadas global.

La navegación por marcas también es un método relativamente extendido en el campo de la navegación de robots. Las marcas pueden ser distintas características que el robot puede reconocer con sus sensores, normalmente formas geométricas. Estas marcas se eligen cuidadosamente para poder ser fácilmente distinguibles del resto de elementos del entorno. Existen dos tipos de marcas principalmente: naturales y artificiales. Las marcas naturales son las más adecuadas en entornos altamente estructurados (como pasillos de hospitales, interiores de laboratorios, etc) y están formadas por objetos o características que ya pertenecen al entorno y que evidentemente poseen otra función principal que la de navegación del robot, mientras que las marcas artificiales están específicamente diseñadas para el guiado de éste. Un ejemplo reciente de navegación basada en marcas naturales es el proyecto ARK desarrollado por el Departamento de Energía Atómica de Canadá y la Universidad de Toronto [ea93c]. El procedimiento general para el posicionamiento basado

en marcas se presenta en la figura 2.3.

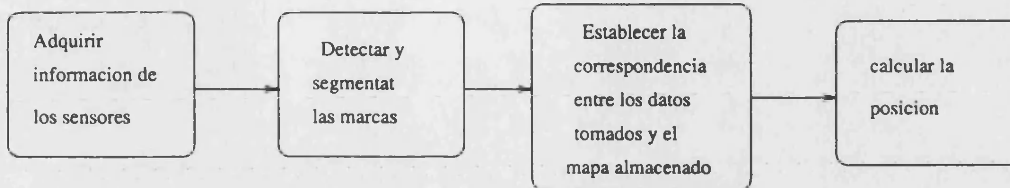



Figura 2.3: Procedimiento general para el posicionamiento basado en marcas

La detección de marcas artificiales es normalmente más sencilla que la de marcas naturales [AH93]. Analizando la bibliografía, tanto los procedimientos de detección, como los tipos de marcas son de lo más variado. Así, suele utilizarse mucho la visión artificial como método de detección. Otros sistemas usan materiales reflectantes para facilitar la extracción de parámetros en la segmentación [MM92]. Por último, un interesante compendio tanto de sensores como de marcas para autolocalización puede verse en [Eve95]. Esta cantidad de sistemas de detección, así como la gran cantidad de materiales y métodos utilizados es la mejor manera de constatar que aunque se ha avanzado mucho en la detección de las características del entorno por parte de un robot móvil, el problema continúa sin estar del todo resuelto.

Otro de los sistemas de posicionamiento más utilizados en robótica móvil es el posicionamiento basado en mapas. Esta es una técnica mediante la cual el robot usa sus sensores para generar un mapa del entorno local. Entonces, este mapa local puede ser comparado con un mapa global previamente almacenado en memoria. Si se encuentran coincidencias, se puede calcular la posición y orientación del robot respecto de su entorno. El mapa prealmacenado puede ser o bien un modelo CAD del entorno o bien un modelo contruido anteriormente con los datos de los sensores.

El procedimiento básico para el posicionamiento basado en mapas se muestra en la figura 2.4.

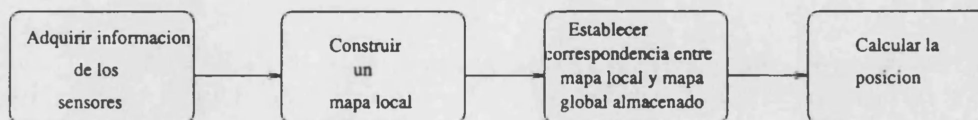



Figura 2.4: Procedimiento general para el posicionamiento basado en mapas

Rencken [Ren93] plantea el problema de la construcción de mapas como:

Dada una posición del robot y un conjunto de medidas: cómo plasmar lo que los sensores están viendo. Obviamente, la capacidad para la construcción de mapas de un

robot, está íntimamente ligada a su capacidad de sentir.

Hoppen et al. [ea90] describen los tres principales pasos necesarios para la construcción de mapas a partir de los datos obtenidos por los sensores:

- Extracción de características a partir de la información sensorial.
- Fusión de datos de los distintos tipos de sensores existentes en el robot.
- Generación automática de un modelo del entorno con diferentes grados de abstracción.

Un problema relacionado con la construcción de mapas es la exploración autónoma. Se supone que normalmente el robot comienza su exploración sin tener ningún conocimiento de su entorno. Entonces, es conveniente seguir cierta estrategia de movimiento a fin de minimizar el tiempo necesario para generar un mapa. Tal estrategia de movimiento se denomina estrategia de exploración y depende en gran medida del tipo de sensores usados. Un interesante ejemplo de estrategia de exploración es la realizada por el robot MOBOT-IV de la Universidad de Kaiserslautern (Alemania) [EP94a]. La idea básica es que el robot está en el centro de una burbuja virtual, inicialmente limitada por la distancia máxima de medida de los sensores. Mientras el robot se mueve, la burbuja se deforma en la dirección del movimiento, de manera que el área completa rastreada por los sensores, permanece sienpre dentro de la burbuja. Cada vez que los sensores detectan un objeto, la superficie del objeto se convierte en un borde real de la burbuja en comparación con los bordes virtuales, los cuales se determinan por la distancia máxima de medida de los sensores. La fase de exploración termina cuando la burbuja no tiene más bordes virtuales (ver figura 2.5). Otro ejemplo de estrategia de exploración simple puede verse en [EP94b].

La mayoría de los investigadores piensan que ninguno de los sensores existentes en la actualidad puede capturar adecuadamente por sí sólo todas las características relevantes de su entorno. Para resolver este problema es necesario combinar los datos de los diferentes tipos de sensores, proceso conocido como fusión de sensores. Jorg [Jor95] desarrolla un mecanismo que utiliza información heterogénea obtenida a partir de laser, radar y sonar para construir un aceptable modelo del mundo.

Para el posicionamiento basado en mapas, existen básicamente dos representaciones: mapas geométricos y mapas topológicos. Un mapa geométrico representa los objetos de acuerdo a sus relaciones geométricas absolutas. El mapa puede estar formado por una representación en forma de malla o una más abstracta como un mapa de lineas o uno poligonal. En el posicionamiento basado en mapas, el mapa geométrico generado a partir de la lectura de los sensores se debe comparar con un mapa global (normalmente de una gran area). Esto, a menudo es de una dificultad enorme debido principalmente a los errores de posición del propio robot. Por el contrario, los mapas topológicos se basan en el registro de información referente a las relaciones entre las características

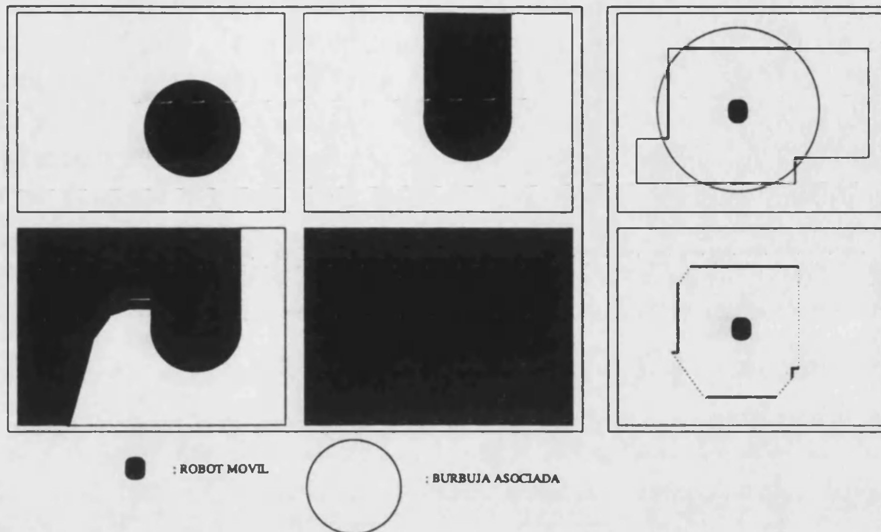



Figura 2.5: Construcción de la burbuja y detección de bordes virtuales y reales.

observadas más que de su posición absoluta con respecto a un sistema de coordenadas de referencia. Así, la representación resultante tiene la forma de un grafo donde los nodos representan las características observadas y los arcos representan las relaciones entre dichas características. Al contrario que los mapas geométricos, los mapas topológicos se pueden contruir y mantener sin ningún conocimiento de la posición absoluta del robot. Esto significa que los errores en esta representación serán independientes de cualquier error en la estimación de su posición [Tay91]. Un buén ejemplo de construcción de un mapa topológico es el dado en [Kur96]. En este trabajo se introduce una aproximación a la generación de mapas del entorno basada en la lectura de sensores ultrasónicos. Así, por medio de un clasificador, las lecturas de los datos se condensan en conceptos abstractos, los cuales ayudan al robot móvil a diferenciar situaciones. Como consecuencia de esto, el espacio libre se particiona en áreas de situación, las cuales se definen como las regiones donde se reconoce una situación específica. De esta manera, tales áreas de situación, pueden asociarse a los nodos de un grafo para generar un mapa del espacio libre en forma de grafo de representación.

Existen diferentes formas para representar mapas geométricos. La forma más simple es un mapa basado en una malla de ocupación. El primero de tales mapas fue la malla de certidumbre desarrollada por Moravec y Elfes, [ME85] y la fusión de sensores en mallas de certidumbre para robots móviles [Mor88]. En la representación por malla de certidumbre, las lecturas de los sensores se proyectan sobre una retícula bidimensional, utilizando determinadas distribuciones de probabilidad para describir la certidumbre que se tiene de la existencia de objetos en una determinada celda. Basada en la aproximación de la malla de certidumbre, Borenstein y Koren [BK91b],[BK91a] refinaron el método con la

malla histograma, la cual aplica una pseudo-distribución de probabilidad independiente del movimiento del robot. Esta pseudo-distribución de probabilidad consiste en actualizar una única celda de la malla en vez de todas las celdas que en principio están afectadas por una determinada lectura de sonar. La celda afectada es la que cayendo sobre el eje acústico, está a la distancia medida por este. Este método reduce considerablemente el tiempo de cálculo necesario para actualizar el mapa, de manera que puede usarse para la evitación de obstáculos en tiempo real; por esta razón, este método es ampliamente usado en muchos robots móviles (por ejemplo en [CJ94],[ea94a]).

Un ejemplo referente al problema de la comparación de mapas basados en mallas de ocupación es el sistema que se implementó en el robot Blanche [Cox91]. Este sistema de posicionamiento se basaba en la comparación entre un mapa local (malla) y un mapa global formado por segmentos lineales. Otro ejemplo típico de aplicación basada en un mapa, esta vez topológico, es el presentado por Courtney y Jain [CJ94]. En este trabajo, la posición del robot se determina clasificando un mapa de descripciones. Tal clasificación permite reconocer la región del espacio relacionada con un determinado mapa, usando los datos sensoriales recogidos de 10 habitaciones diferentes y 10 puertas distintas de su espacio de trabajo. En este estudio, el área de trabajo del robot se representa como un conjunto de mapas basados en mallas interconectados a través de relaciones topológicas. Cada región se representa por un conjunto de mapas sensoriales en malla y la fusión de sensores a nivel de características se realiza extrayendo descripciones espaciales de estos mapas. En la fase de navegación, el robot se autolocaliza comparando las características extraidas de su mapa de espacio actual con las características representativas de locales conocidos del entorno.

En cuanto al problema de la planificación de trayectorias, existen tres aproximaciones principales [yJT97]:

- Roadmap: este método inserta conectivamente puntos extraidos del espacio libre de colisiones del robot en una red de curvas. Una vez que la red ha sido construida, el problema se reduce a conectar las configuraciones inicial y final a puntos del roadmap y buscar por la red un camino que una dichos puntos. Basados en este método podemos citar las técnicas del grafo de visibilidad, el diagrama de Voronoi, la red de caminos libres, etc.
- Descomposición celular: este método está basado en la descomposición del espacio libre de colisiones del robot en una serie de regiones simples denominadas celdas, de tal modo que el camino que une dos celdas consecutivas pueda ser generado fácilmente. De este modo, se puede construir un grafo no dirigido que represente la relación de adyacencia entre celdas. Este grafo se conoce con el nombre de "grafo conectivo".
- Campos potenciales artificiales: Mediante este método se plantea que el robot se encuentra sometido a la influencia de un campo potencial U, cuya estructura depende

de la configuración espacial del entorno del robot, así como del punto destino al que se dirija el mismo. La idea se basa en la superposición de dos tipos de campos: uno atractivo hacia el punto de destino y otro repulsivo ejercido por los obstáculos. La planificación de la trayectoria se realiza de forma iterativa, de manera que en cada iteración, el vehículo está sometido a una fuerza artificial introducida por el campo potencial que hace que el robot se desplace en la dirección decreciente del campo.

La idea de tener obstáculos conceptualmente ejerciendo fuerzas sobre el robot fue introducida por Khatib [O.85]. Krogh [Kro85] reforzó esta idea, teniendo además en cuenta la velocidad del robot en las proximidades de los obstáculos. Se han utilizado otras muchas aproximaciones aprovechando esta idea, pero quizá una de las más interesantes sea la utilizada por Borenstein y Coren [BK89]. Su aproximación al problema pasa por la combinación del método de Campos Potenciales con la división del espacio en una malla (malla de certidumbre). Mientras el robot se mueve, las lecturas de los sensores se proyectan en la malla de certidumbre. Simultaneamente, el algoritmo rastrea una pequeña ventana cuadrada de la malla la cual está localizada de manera que el robot siempre está en su centro. Cada celda ocupada del interior de la ventana ejerce una fuerza repulsiva sobre el robot que tiende a alejarlo de ella. La magnitud de esta fuerza es proporcional al contenido de la celda e inversamente proporcional al cuadrado de la distancia entre la celda y el robot. La fuerza repulsiva resultante $F_r$ es la suma vectorial de las fuerzas individuales de todas las células. En cualquier momento durante el movimiento, una fuerza atractiva de magnitud constante $F_t$ empuja al robot hacia el punto destino. Por fin, la suma vectorial de la resultante repulsiva y la atractiva, produce el vector de fuerza resultante $R = F_t + F_r$.

Mediante la técnica de Campos Potenciales suelen ocurrir dos tipos de problemas principalmente [KB91], los cuales deben ser resueltos de alguna manera. Por un lado, pueden ocurrir movimientos de tipo oscilatorio cuando el robot se mueve en presencia de obstáculos. Una posible solución es incrementar la fuerza repulsiva cuando el robot se mueve hacia un obstáculo y reducirla cuando se mueve a lo largo del obstáculo. Metodológicamente se trata de variar la magnitud de la fuerza repulsiva en función de la dirección de dicha fuerza y de la velocidad. El segundo tipo de problemas que pueden ocurrir es que la fuerza atractiva y la resultante de las fuerzas repulsivas sean de la misma magnitud y de sentido contrario (puntos muertos). En este caso, el robot se quedaría bloqueado en una situación inestable. La solución más común que se suele utilizar para resolver este problema (aunque en nada elegante), es introducir en estos casos una nueva fuerza de magnitud constante y dirección más o menos aleatoria para sacar al robot de dicho estado.

Por último comentar la existencia de un último gran grupo de sistemas de posicionamiento, el basado en visión. El procedimiento general seguido por este tipo de sistemas

suele ser el análisis de las imágenes tomadas por cámaras CCD para obtener a partir de ellas la máxima información posible a través de mecanismos como la segmentación y las operaciones lógicas entre imágenes.

# Capítulo 3

# El hardware del sistema.

## 3.1 Introducción.

En este capítulo se describe el hardware del sistema, tanto desde un punto de vista morfológico como funcional; entendemos por hardware no sólo los circuitos estrictamente considerados, sino también todo el soporte físico y mecánico que usamos para construir el robot. La decisión de construirlo, en lugar de recurrir a una plataforma comercial, no ha sido, como ya comentamos en la introducción, arbitraria, sino que responde a la necesidad de disponer de una plataforma donde todas las ideas que proponemos pudieran ser probadas. Trataremos de ver en este capítulo cómo esto ha sido desarrollado.

## 3.2 Descripción genérica.

El sistema está formado por una plataforma base de forma octogonal inscrita en una circunferencia de 60 cm de diámetro. El control de tracción está formado por dos ruedas motrices a ambos lados del punto central de la plataforma (C) y dos ruedas de movimiento libre en la parte anterior y posterior de éste. Las ruedas motrices están constituidas por un nucleo de hierro y un recubrimiento de goma. El diámetro de las mismas es de 10 cm. Cada una de estas ruedas, va gobernada por un motor de cc más un reductor más un codificador óptico incremental. El motor es de tipo genérico de cc de 12 V. El reductor es de tipo planetario con una reducción de 43:1 y por último, el codificador óptico posee dos salidas desfasadas $\pi/2$ con una resolución de 500 pulsos por vuelta de su eje.

Con una configuración como la descrita anteriormente, la cinemática del robot se define de una manera simple, puesto que la posición y orientación de la platafoma base puede darse en función de los pulsos contados por los codificadores ópticos asociados a las ruedas motrices, usando simples ecuaciones geométricas a partir de una determinada posición de inicio.

Las ecuaciones para conocer la posición y orientación del punto central de una plataforma con tracción diferencial respecto de un sistema de coordenadas fijo se describen a continuación (ver [Kla88],[CR92]):

Supongamos que en el instante de muestreo i, los contadores asociados a los codificadores ópticos de las ruedas izquierda y derecha, muestran un incremento de pulsos de $N_L$ y $N_R$ respectivamente. Si además definimos

- $D_n$: diámetro nominal de las ruedas en mm.
- $C_e$: resolución del encoder en pulsos por revolución.
- n: relación de reducción entre la salida del motor (al cual está unido el codificador óptico) y la salida del reductor.

entonces, podemos definir

$$c_m = \frac{\pi D_n}{n C_e} \tag{3.1}$$

donde $c_m$ se define como el factor de conversión que relaciona los pulsos de encoder con el desplazamiento lineal de las ruedas.

Si definimos el incremento de desplazamiento lineal de las ruedas izquierda y derecha como $\triangle U_{L,i}$ y $\triangle U_{R,i}$, podemos poner que

$$\triangle U_{L,i} = c_m N_{L,i} \tag{3.2}$$
$$\triangle U_{R,i} = c_m N_{R,i} \tag{3.3}$$
$$\tag{3.4}$$

y si definimos el desplazamiento del punto central del robot (C) como $\triangle U_i$, a raiz de los resultados anteriores podemos poner

$$\triangle U_i = \frac{\triangle U_{R,i} + \triangle U_{L,i}}{2} \tag{3.5}$$

Seguidamente, podemos calcular el cambio incremental en la orientación del robot como

$$\Delta\theta_i = \frac{\Delta U_{R,i} - \Delta U_{L,i}}{b} \tag{3.6}$$

donde b es la distancia entre las ruedas directoras (idealmente medida como la distancia entre los dos puntos de contacto de las ruedas con el suelo).

Así, la nueva orientación relativa del robot, puede ser actualizada como

$$\theta_i = \theta_{i-1} + \Delta\theta_i \tag{3.7}$$

y la nueva posición relativa del punto central como

$$x_i = x_{i-1} + \Delta U_i \cos\theta_i \tag{3.8}$$

$$y_i = y_{i-1} + \Delta U_i \sin\theta_i \tag{3.9}$$

donde $x_i$ e $y_i$ son las coordenadas que definen la posición relativa del punto central del robot (C) en el instante de muestreo i.

## 3.3 Sensorización genérica de un robot móvil.

### 3.3.1 Consideraciones generales.

Existen dos formas básicas de interacción del robot con el entorno que le rodea. Una de ellas, le da la posibilidad de recibir información procedente del medio. Esta información, como por ejemplo la existencia o no de objetos cercanos, la forma de estos, la orientación, etc debe ser captada y procesada a fin de poder tomar decisiones, que en su mayor parte involucran a la segunda forma de interacción con el medio, es decir, con la posibilidad de movimiento dentro del entorno o de variación de la disposición de los objetos dentro de éste.

A los dispositivos físicos que se encargan de adquirir la información procedente del medio se les suele denominar dispositivos sensores, mientras que a los que se encargan de generar el movimiento o a modificar el entorno se les denomina dispositivos actuadores.

Dos conceptos importantes a tener en cuenta cuando se analiza cualquier sensor son su sensitividad y su rango. La sensitividad es una medida del grado de cambio de la señal de salida en función del cambio de la cantidad medida. Así, si llamamos R a la salida del sensor y X a la cantidad física medida, la sensitividad del sensor viene definida por

$$\frac{\triangle R}{R} = S\frac{\triangle X}{X} \tag{3.10}$$

Así, un pequeño cambio en la cantidad medida, $\triangle X$ se traduce en un pequeño cambio en la respuesta del sensor $\triangle R$ por la sensitividad S.

Por rango de medida de un sensor, se entiende normalmente el intervalo de variación física que es capaz de detectar.

Cualquier dispositivo sensor, reacciona a los niveles de variación de algún estímulo físico generando una determinada variación eléctrica (tensión, corriente, frecuencia, etc). Normalmente, el circuito asociado con el sensor adapta la señal eléctrica de salida a una tensión, la cual es introducida en un conversor A/D para poder ser procesada por el microprocesador. El conversor A/D es sensible sólo a un rango limitado de tensiones (a menudo de 0 a 5 V). En el caso de un conversor A/D de 8 bits, la tensión de entrada es convertida en un valor digital de los 256 posibles. Esta es la ventana del mundo para el microprocesador.

Así pues, es importante considerar cuidadosamente cómo una cantidad física es transformada en un valor digital accesible al microprocesador (mapeado). Los mapeados más comunmente utilizados son el lineal y el logarítmico [JF93]. En el mapeado lineal, el rango de valores de entrada al sensor se divide por el número de posibles códigos digitales de salida. De esta forma, a cada uno de estos incrementos (constantes) en la cantidad física a medir le corresponde un código digital de salida. En el mapeado logarítmico, la traslación de la variación de la señal física a medir en un determinado código digital se realiza en una escala logarítmica, de manera que el rango de medida física por código no es constante.

Existe una gran variedad de sensores, cada uno especializado en la detección de una característica determinada. Los más comúnmente usados en el campo de la Robótica Móvil son descritos brevemente a continuación:

UNIVERSITAT DE VALÈNCIA BIBLIOTECA DE CIÈNCIES



### 3.3.2 Sensores de luz.

Los sensores de luz son aquellos que varían una magnitud eléctrica en función de la variación de intensidad luminosa. Los más usuales se describen a continuación

**Fotoresistencias, fotodiodos y fototransistores.**

Las fotoresistencias son simplemente resistencias variables similares a los potenciómetros, excepto en que el cambio de resistencia se realiza por una variación en la intensidad de la radiación luminosa recibida y no por un giro mecánico en la rotación de su eje. Los fototransistores conectan su colector con su emisor (es decir, conducen la corriente) de manera proporcional a la intensidad de radiación luminosa detectada por su base. Por último, los fotodiodos varían su resistencia directa (de ánodo a cátodo) también de forma proporcional a la intensidad luminosa recibida.

Los fototransistores proporcionan generalmente mayor sensitividad a la luz que las fotoresistencias. En cuanto a los fotodiodos, éstos poseen una magnífica sensitividad. Normalmente producen una salida bastante lineal sobre un rango muy amplio de niveles de luz y además, responden rápidamente a los cambios en la radiación recibida.

En la figura (3.1) se puede observar el esquema eléctrico que se suele presentar para identificar a los sensores de luz más frecuentes

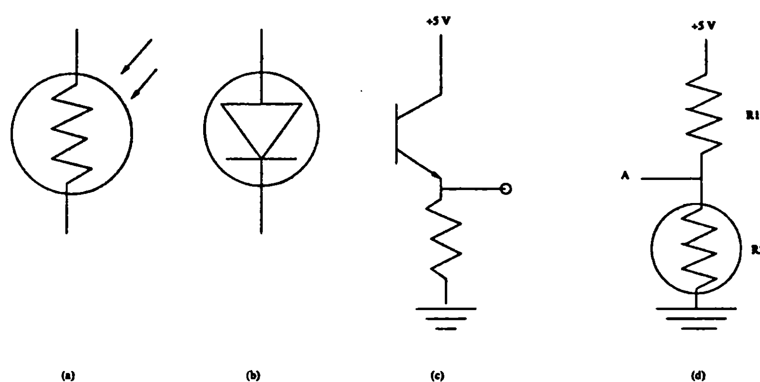



Figura 3.1: (a)Esquema de fotoresistencia. (b) Esquema de fotodiodo. (c) Esquema de fototransistor. (d) Posible conexión de fotoresistencia.

En el apartado (d) de dicha figura se puede observar un simple circuito que utiliza fotoresistencias para detectar las variaciones en la intensidad luminosa. La fotoresistencia

($R_2$) varía su valor en función de la luz recibida. El circuito es un simple divisor de tensión, en donde la tensión en el punto (A) viene dada por

$$V_A = \frac{R_1}{R_1 + R_2} V \quad (3.11)$$

Así, la entrada de tensión $V_A$ puede ser introducida en un conversor A/D para obtener un determinado código digital. Un pequeño inconveniente es que el divisor de tensión no realiza un mapeado logarítmico sobre el rango de códigos del conversor (como suele ser conveniente con los sensores lumínicos), pero la conversión puede realizarse por software y la señal de salida suele ser muy útil.

Normalmente, se puede conseguir un buen compromiso entre sensitividad y rango, tomando el valor de $R_1$ del mismo valor que la resistencia que presenta la fotoresistencia, cuando se expone a un nivel de luz que está en la mitad del rango de los niveles de luz con los que se quiere trabajar.

### Detectores de proximidad por infrarojos.

Estos sensores están formados normalmente por un par emisor-receptor, de manera que el primero de ellos, fabricado normalmente con cristales de arseniuro de galio, emite radiación infrarroja (en torno a los 880 nm). Esta radiación puede ser reflejada por los objetos circundantes siempre que estén dentro del radio de detección. Así, la energía reflejada puede ser detectada por el receptor. Normalmente, la señal de salida del receptor es binaria, de manera que la información proporcionada por este tipo de dispositivos no es continua y proporcional a la distancia, sino más bién sirve para informar de la existencia de un determinado objeto dentro del radio de detección de dicho sensor.

Una posible pareja de sensores infrarrojos emisor-receptor pueden ser el led emisor SFH 484 de Siemens y el detector GP1U52X de Radio Shack y un posible montaje de ambos se presenta en la figura 3.2.

Según se ve en la figura y según las especificaciones de los fabricantes de los dispositivos, el circuito funciona de manera que las entradas de control gobiernan el funcionamiento de un oscilador de 40 KHz donde están los diodos emisores de infrarrojos. La señal de salida del oscilador debe modularse a una frecuencia menor por las características del receptor (1.66 KHz). El receptor detecta un obstáculo cuando estando el emisor funcionando, el receptor tiene un nivel bajo en su salida y cuando estando el emisor desconectado, el receptor tiene un nivel alto en su salida. También es interesante resaltar

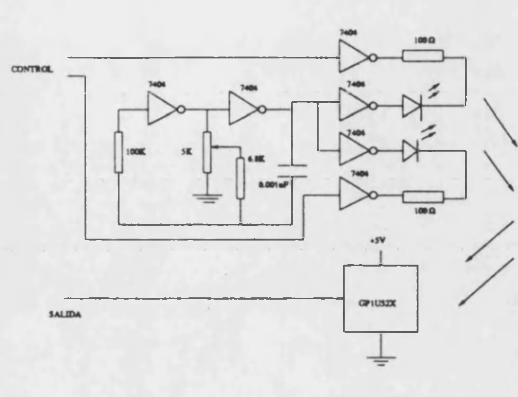



Figura 3.2: Posible montaje de sensores de infrarrojos con la pareja emisor-receptor SFH 484-GP1U52X

el hecho de que existe un retraso entre la activación y/o desactivavión de los diodos emisores y la activación y/o desactivación de la señal del receptor.

Por último, un posible diagrama de senáles puede observarse en la figura 3.3.

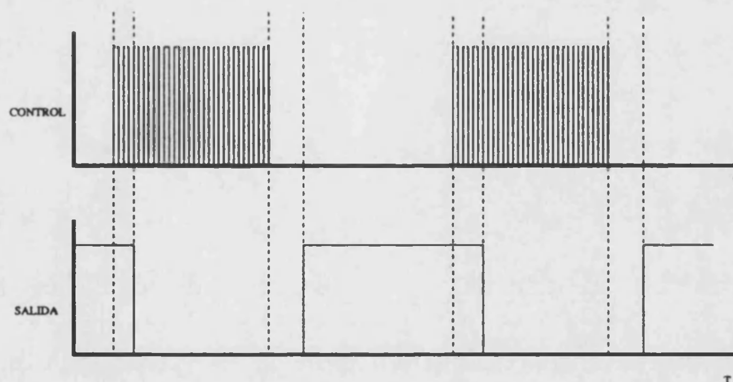



Figura 3.3: Diagrama de señales emisor-receptor

### Sensores piroeléctricos.

La salida de un sensor piroeléctrico cambia en función de los pequeños cambios de temperatura que puedan ocurrir en su superficie. El elemento activo en este tipo de sensores es un cristal de tantalato de litio. La carga en el cristal se induce cuando éste se calienta. Normalmente se usan para la detección de personas en el entorno, de manera

que normalmente están especialmente diseñados para detectar la radiación energética infrarroja emitida por éstos que está en el rango de 8-10 $\mu m$.

La salida típica de un sensor piroeléctrico tal como el Eltec 422-3 es la que se puede observar en la figura 3.4. Así, introduciendo dicha salida a la entrada de un conversor A/D, se puede detectar el movimiento de humanos (vida en general) en sus proximidades.

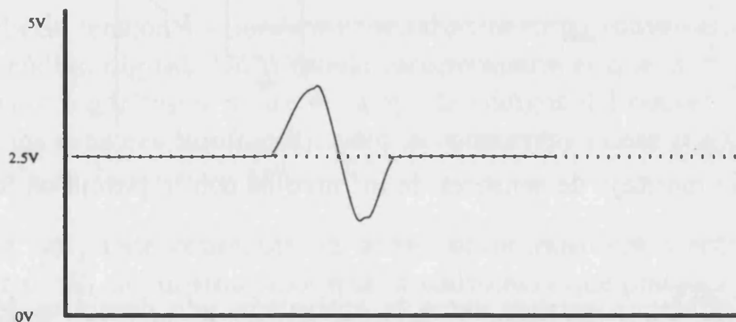



Figura 3.4: Señal típica de un sensor

### Cámaras.

Las cámaras de video, usualmente denominadas CCD también pueden ser usadas como sensores luminosos. Básicamente se trata de una matriz de mxn celdas sensibles a la luz. La imagen es almacenada en una memoria de manera que la carga de cada celda individual (pixel) es proporcional a la intensidad luminosa recibida.

La imagen almacenada en memoria puede ser procesada según las características que se quieran resaltar, de manera que a partir de la imagen obtenida se pueden extraer numerosas características del entorno. El principal inconveniente de este tipo de dispositivos es el tiempo necesario para el procesado de la información.

Existe ya un tipo de sensores con una filosofía de diseño diferente a las clásicas cámaras CCD denominados sensores retínicos espacio-variantes (o log-polar). Este tipo de sensores ofrecen numerosas ventajas frente a las clásicas matrices CCD. Básicamente ofrecen una mayor resolución en el centro de la imagen (fóvea) y una menor reslución en la periferia, de manera que analizando únicamente el centro de la imagen, se tiene la mayor parte de la información almacenada en la imagen total. Además, almacenando la intensidad de luz recibida en forma logarítmica, la imagen nunca se satura y siempre puede ser procesada con independencia de las diferencias de luz recibidas. Además, desplazamientos radiales o rotaciones respecto al eje óptico de la cámara dan como resultado desplazamientos lineales de la imagen, con la consiguiente reducción del tiempo de proceso.

### 3.3.3 Sensores de fuerza.

Los sensores de fuerza son aquellos que son capaces de dar una daterminada señal de salida cuando son sometidos a presión. La salida puede ser tanto analógica (señal proporcional a la presión ejercida) como digital.

#### Microinterruptores.

Los microinterruptores son sensores de fuerza de dos posiciones (abierto-cerrado). La salida es obviamente digital con sólo los valores de nivel lógico alto y nivel lógico bajo. Se usan normalmente para la detección de colisiones, de manera que en un choque, el dispositivo se cierra (o abre) mecánicamente, abriendo o cerrando un simple circuito eléctrico.

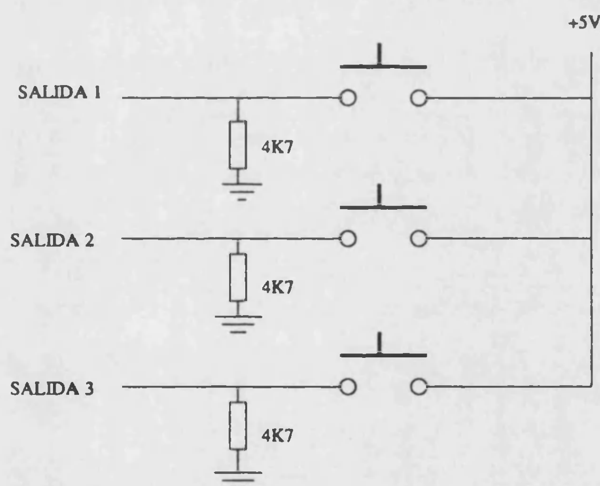



Figura 3.5: Circuito básico para microinterruptores

Una posible forma de conectar un microinterruptor para dar una salida lógica es la que se presenta en la figura 3.5. En ella, cuando se cierra o se abre mecánicamente el interruptor, se obtiene una salida lógica alta o baja respectivamente.

#### Sensores de deformación.

Otro sensor útil para la detección del contacto es el sensor de deformación. Este tipo de sensor es básicamente una resistencia variable, la cual se modifica según la deformación

sufrida. Este sensor puede utilizarse en un circuito de manera similar a una fotoresistencia, por ejemplo utilizando un divisor de tensión con la señal de salida conectada a un conversor A/D.

### 3.3.4 Sensores de sonido.

Son sensores para la detección tanto de las frecuencias audibles por el ser humano como las inmediatamente superiores e inferiores.

#### Micrófonos.

Un micrófono es un dispositivo que devuelve una salida analógica proporcional a la intensidad sonora recibida. Esta señal normalmente debe ser amplificada, muestreada y convertida en señal digital para su comparación con una serie de patrones. Un posible circuito para detectar la señal de salida de un micrófono y una posible señal de salida es el que se presenta en la figura 3.6.

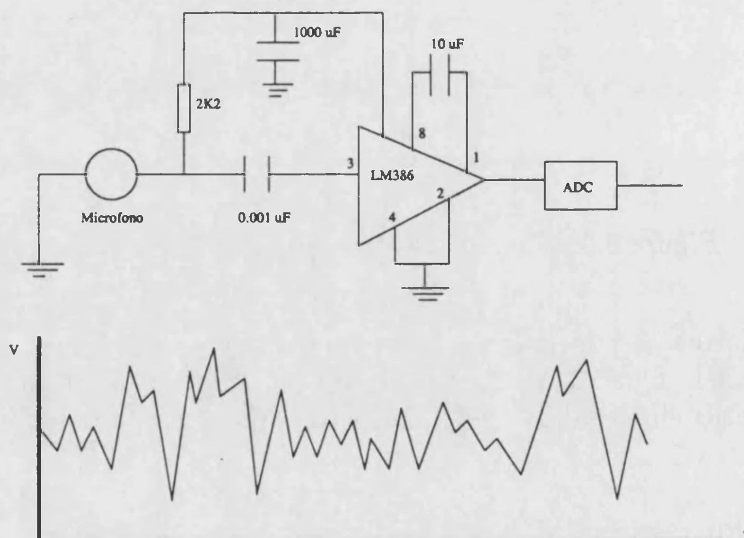



Figura 3.6: Circuito básico para conexión de micrófono y posible señal de salida

**Sónares.**

Estos dispositivos son capaces de medir la distancia desde ellos mismos hasta un determinado objeto. Existen varios modos de funcionamiento para realizar la medida de distancias; el más normal es el denominado por tiempo de vuelo. Mediante este método, se emite a una determinada frecuencia un tren de pulsos ultrasónicos y a continuación se espera recibir su eco. Así, conociendo la velocidad del sonido en el aire, es inmediato conocer la distancia.

Otro posible método consiste en emitir una señal a frecuencia ultrasónica y medir en el receptor la potencia de la señal reflejada. Así, la potencia recibida es aproximadamente proporcional a la distancia al objeto en cuestión.

A pesar de tratarse de los dispositivos sensores más utilizados en robótica móvil para la detección de obstáculos, presentan ciertos inconvenientes que cabe resaltar:

- Pobre direccionalidad. No se puede saber con precisión la posición de un obstáculo detectado por un determinado sensor dentro de su campo de acción, que suele ser un cono de unos 30°, aunque la mayor probabilidad es que se encuentre sobre su eje acústico.
- Frecuentes lecturas erróneas. Causadas por ruido ultrasónico de fuentes externas o por reflexiones de los sensores vecinos.
- Reflexiones especulares. Ocurren cuando el ángulo entre el frente de la onda y la perpendicular a la superficie del objeto es demasiado grande. En este caso, la señal reflejada es enviada lejos de la fuente receptora, dando una información erronea de la distancia o tamaño real del objeto.

El sistema sonar utilizado por excelencia es comercializado por la firma POLAROID y está formado básicamente por una tarjeta de control y un sensor. El principio de funcionamiento es el de la medida de distancia por tiempo de vuelo y sigue un protocolo sencillo que se explica posteriormente.

### 3.3.5 Sensores de posición y orientación.

**Codificadores ópticos.**

Estos dispositivos se conectan directamente al eje de la rueda del robot y generan dos ondas cuadradas desfasadas entre si $\frac{\pi}{2}$. La sensibilidad del sistema rueda-encoder depende de la resolución de este. Valores usuales de resolución van desde 1 pulso por revolución

hasta varios miles de pulsos por revolución.

Así, el desplazamiento angular sufrido por la rueda es directamente proporcional al número de pulsos generados y la velocidad angular, directamente proporcional a la frecuencia de dichos pulsos. Además, el hecho de tener dos señales desfasadas nos da la posibilidad de detectar el sentido del giro.

Existen dos tipos básicos de encoders ópticos, a saber, encoders ópticos incrementales y absolutos. En los primeros, es necesario un hardware asociado para almacenar los pulsos desde el inicio, mientras que en los segundos, la información de salida ya es directamente el código que corresponde a la posición angular actual.

### Giróscopos.

El giróscopo es un sensor que usa el principio de la conservación del momento angular para mantener uno o más ejes internos apuntando en la misma dirección. Así, un giróscopo unido a un robot nos puede permitir determinar tanto la velocidad de rotación como el ángulo girado.

La empresa Futaba comercializa un pequeño giróscopo que se usa normalmente en aviación. Su entrada debe ser una señal modulada en anchura de pulso y la salida es un incremento o disminución de la anchura de pulso respecto de la señal de entrada dependiendo de la señal de rotación.

### Inclinómetros.

El inclinómetro es un sensor que detecta el ángulo relativo entre un eje perpendicular al cuerpo del sensor y el vector gravitacional terrestre. El inclinómetro más simple es el interruptor de mercurio. Dos electrodos dispuestos en el interior de una burbuja de vidrio donde se introduce mercurio. Cuando la burbuja se gira, el mercurio abre o cierra el circuito unido a los electrodos. Este sensor nos daría una señal digital.

La empresa U.S. Digital Corp. comercializa un inclinómetro formado por un codificador óptico absoluto, de manera que el eje mantiene la dirección del vector gravitacional, mientras que el cuerpo exterior puede inclinarse. La salida es un código digital que indica el ángulo de inclinación en un rango de $\pm 15^o$.

**Brújulas.**

Las brújulas son sensores que detectan el débil campo magnético terrestre. De esta manera se puede saber en todo momento el ángulo del sensor respecto de este. La utilización de este tipo de sensores en laboratorios es muy delicada, puesto que en numerosas ocasiones, los campos magnéticos de aparatos o redes eléctricas, las estructuras de acero, etc pueden producir grandes errores en la lectura del sensor.

### 3.3.6 Sensores específicos.

**Detección del nivel de las baterías.**

Uno de los comportamientos que más comúnmente se suelen implementar en robots móviles es el de recargar su nivel energético cuando éste está bajo. Para ello, la máquina debe ser capaz de detectar el nivel de tensión en sus baterías. En la figura 3.7 se puede observar un posible circuito que resuelve este problema.

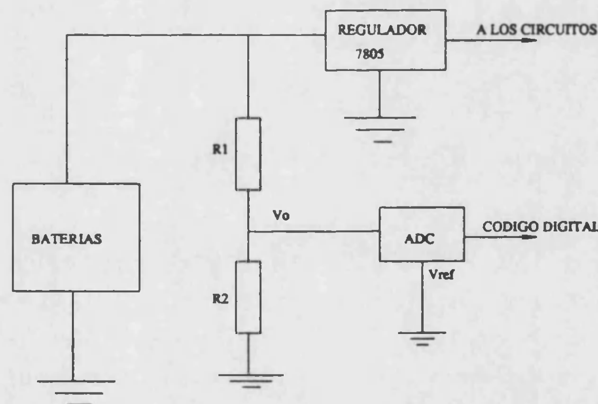



Figura 3.7: Circuito para la detección del nivel de batería

En este circuito, la salida de tensión de las baterías se conecta tanto a la salida de tensión regulada como a un divisor de tensión. Si se diseña el divisor de manera que con las baterías completamente cargadas, éste entregue la tensión máxima de entrada a un conversor A/D (por ejemplo $V_0 = 5V$ en la figura), entonces, cuando las baterías se descarguen, la salida del divisor irá disminuyendo progresivamente, hecho que podrá ser detectado por microprocesador a través del ADC.

Si denominamos $V_{bmax}$ a la tensión máxima de las baterías cuando están cargadas y $V_b$ a la tensión que presentan en un momento determinado, entonces para una tensión

máxima de salida supuesta de $T_1$ Volt en el divisor, tenemos

$$V_0 = \frac{R_1}{R_1 + R_2} V_{bmax} = T_1 \tag{3.12}$$

para el diseño es importante que $R_1 + R_2$ sea lo suficientemente elevada como para que la corriente que circule por ellas sea despreciable frente a la corriente de consumo de los circuitos y lo suficientemente pequeña como para presentar una impedancia despreciable frente a la del conversor A/D.

Si suponemos que $R_1 + R_2 = X$, entonces

$$\begin{aligned} R_1 &= \frac{T_1}{V_{bmax}} X \\ R_2 &= X - R_1 \\ V_b &= \frac{X}{R_1} V_0 \end{aligned}$$

## 3.4 El hardware de RODNEY.

Siguiendo una práctica común en el diseño de hardware para conectarse a una arquitectura INTEL, y más específicamente a un PC con bus ISA, el control tanto de los dispositivos sensores como de los actuadores se realiza leyendo o escribiendo registros en el espacio reservado a puertos de E/S. Por ello, es necesario disponer de una placa que, actuando de interface con el bus, provea un número suficiente de señales para lectura y/o escritura en dichos registros.

### 3.4.1 La tarjeta de control de puertos de E/S.

Mediante esta tarjeta, se generan las señales que gobiernan las lecturas y escrituras en los registros de E/S. Se trata de una tarjeta diseñada para conectarse al bus ISA de la placa base del computador principal. Se utilizan las lineas correspondientes al bus de datos, al bus de direcciones y determinadas líneas de control. En el apéndice B.3 se puede observar el fichero esquemático correspondiente a la parte de la placa encargada de generar las señales de control de lectura/escritura en los registros de E/S. Las lineas de

salida marcadas como $IOW_0..IOW_7$ son las señales de control de escritura en registros, mientras que las marcadas como $IOR_0..IOR_7$ son las señales de control de lectura en los registros de E/S.

### 3.4.2 El control de tracción.

#### El generador PWM.

El generador PWM es la parte del control de tracción encargado de controlar la potencia que se debe suministrar a los motores. La modulación por anchura de pulso es una técnica muy utilizada para controlar la velocidad angular de motores de cc.

La idea principal es la de poder modificar la relación entre el tiempo del nivel alto y el tiempo del nivel bajo que caracterizan a una onda cuadrada de frecuencia fija. Así, si la onda está todo el tiempo a nivel alto, la etapa de potencia estará enviando la máxima tensión al motor y si la onda está todo el tiempo a nivel bajo, la etapa de potencia estará enviando tensión nula al motor.

Existen diversos circuitos integrados comerciales que se encargan de generar una salida modulada en anchura de pulso, normalmente a partir de una tensión analógica de entrada. Por motivos autodidactas, en este caso se diseñó un circuito para generar el PWM.

Los bloques fundamentales del circuito son un registro latch (74LS273), un contador binario de 8 bits formado a partir de 2 contadores binarios de 4 bits (74LS161), un comparador para 2 entradas de 8 bits (74LS688), un generador de señal de reloj a partir de un cristal de cuarzo y un oscilador (9501), biestables tipo D para controlar la conmutación de la onda y un amplificador operacional para amplificar la señal de ataque al circuito de potencia. Todo esto puede apreciarse en el diagrama de bloques de la figura 3.8.

El funcionamiento es muy simple. En el registro latch se escribe un código de 8 bits que va a controlar la anchura del pulso. El contador va contando ciclicamente, de manera que a cada inicio de ciclo, los biestables ponen la señal de salida a un nivel lógico bajo. Cuando el comparador detecta que el código almacenado en el registro es igual al código de cuenta del contador, la lógica combinacional hace conmutar la salida de los biestables a un nivel lógico alto. Esto permanecerá así hasta el inicio del ciclo siguiente del contador. De esta manera se puede disponer de 256 divisiones de tensión para el motor. Puesto que la tensión máxima que se va a enviar a los motores es de 12V, se van a poder obtener variaciones de tensión de 0.0468V. El fichero esquemático de la placa para la generación del PWM se muestra en el apéndice B.3.

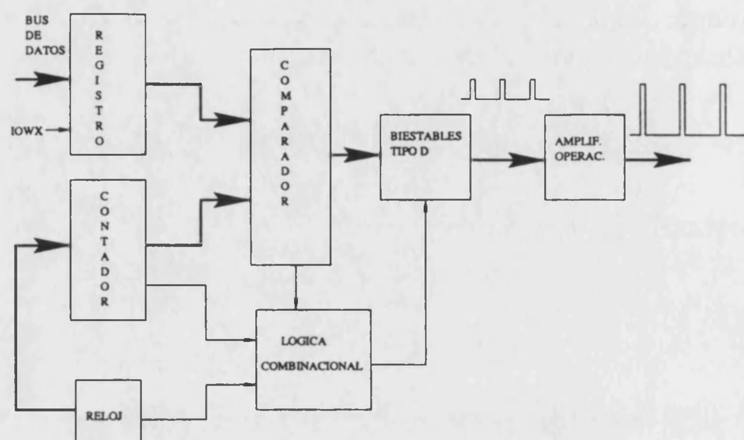



Figura 3.8: Diagrama de bloques del circuito generador de PWM

**La etapa de potencia.**

La etapa de potencia es la parte electrónica encargada de suministrar la corriente suficiente a los motores para que éstos, a partir de la señal generada por el PWM, se puedan mover con la velocidad angular deseada.

La etapa de potencia está formada por un regulador estático de continua en configuración de puente H completo. El diagrama de bloques del circuito puede observarse en la figura 3.9.

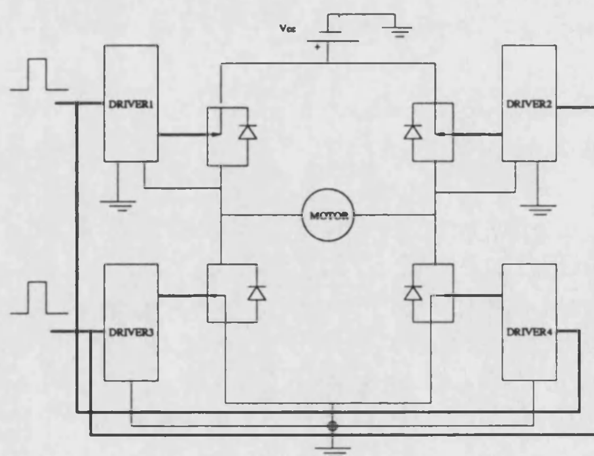



Figura 3.9: Diagrama de bloques para la etapa de potencia

Los conmutadores utilizados son transistores MOSFET capaces de entregar corrientes

nominales de 4A y corrientes de pico de hasta 8A. Los controladores encargados de disparar los transistores son los IR2110. Estos controladores aceptan en sus entradas señales moduladas en anchura de pulso y adaptan los niveles y tiempos de sus salidas de control para disparar hasta un número máximo de 2 transistores al mismo tiempo cada uno. Así, el tiempo que la etapa de potencia aplica la tensión de suministro a los terminales del motor, viene gobernada por la entrada de control, es decir la señal generada por el PWM.

### El sentido de giro.

El control del sentido de giro de los motores se realiza conmutando mediante reles de potencia las entradas de control de la etapa de potencia (salidas del generador PWM).

Básicamente, se trata de que a partir de señales digitales almacenadas en un determinado registro y convenientemente amplificadas, se controle la conmutación de los relés cuyas entradas son las señales PWM y cuyas salidas son o la propia señal de entrada o la señal invertida. Con motivo de evitar sobretensiones inversas en los terminales de las bobinas de los relés, se disponen de unos diodos de protección (diodos de flyback).

Un diagrama de bloques del circuito puede verse en la figura 3.10.

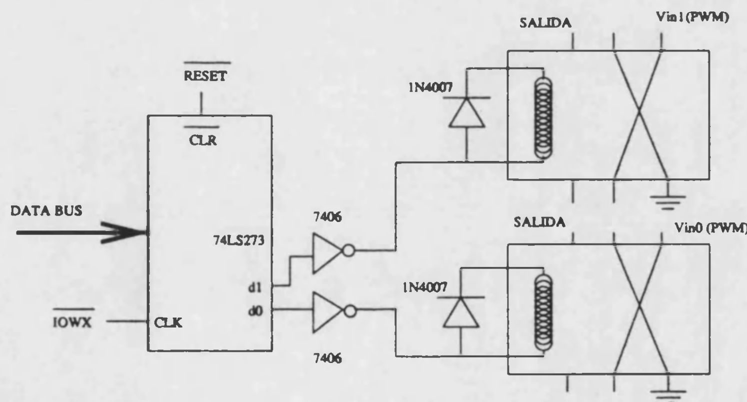



Figura 3.10: Diagrama de bloques del control del sentido de giro de los motores

### Los motores.

El conjunto encargado del movimiento del robot está formado por un motor, un reductor y un codificador óptico unidos solidariamente para cada rueda tractora. Antes de la

construcción del robot móvil, es importante obtener un modelo del vehículo, a fin de poder dimensionar convenientemente la potencia que los motores deben suministrar para poder realizar determinados movimientos (como por ejemplo, subir una determinada pendiente a la velocidad deseada).

Si, por ejemplo ponemos como requisito que nuestro robot sea capaz de subir una determinada pendiente de $\theta$ grados a una velocidad constante $v_x$, a partir de las ecuaciones de la física elemental, la fuerza aplicada a las ruedas actuando en la dirección de avance ($F_{apli}$), debe contrarestar a las fuerzas que se oponen al movimiento, a saber, la fuerza debida al rozamiento ($F_r$) y la fuerza debida al propio peso del vehículo ($F_w$).

$$F_{apli} = F_r + F_w \tag{3.13}$$

A su vez, la fuerza debida al rozamiento es el producto del coeficiente de rozamiento por la fuerza normal al peso, es decir

$$F_r = \mu F_N = \mu mg\cos(\theta) \tag{3.14}$$

y $F_w$ es $mg\sin(\theta)$ (donde m es la masa del vehículo y g la aceleración gravitatoria). Entonces, a partir de estas ecuaciones tenemos que

$$F_{apli} = \mu mg\cos(\theta) + mg\sin(\theta) \tag{3.15}$$

La potencia que deben tener los motores, es el producto de la fuerza que necesita ser aplicada a las ruedas por la velocidad a la que el robot debe viajar, o sea

$$P_m = F_{apli}v_x \tag{3.16}$$

El par y la velocidad necesaria en cada motor se puede calcular a partir de

$$\begin{aligned} \omega &= \frac{v_x}{r} \\ \frac{P_m}{N_m} &= T\omega \end{aligned}$$

donde r es el radio nominal de las ruedas tractoras y $N_m$ es el número de motores que se van a utilizar para mover el robot. Además, el tiempo que el robot puede estar

trabajando, depende de la energía almacenada en las baterías. Así, si las baterías son de E Joules, entonces el tiempo de vida de estas, será de $t = \frac{E}{P_m}$ y la distancia que el robot habrá viajado será de $D = v_x t$. Normalmente la capacidad de las baterías no se da en Joules, sino en Amperios-hora. Así, para conocer la energía contenida en una batería hay que multiplicar su capacidad por la tensión nominal de salida (recordando que 1 Joule es igual a 1 Coulomb-Volt y 1 Amperio es igual a 1 Coulomb por segundo).

En nuestro caso, para un peso aproximado del robot de unos 20 Kg y una batería de 12 Amp-hora y 12 Volt, se han dispuesto 2 motores de 100 mNw.m y 12 Volt, con una velocidad angular máxima de 5000 rpm. Además, la salida de los motores se conecta a sendos reductores planetarios con una reducción de 43:1, lo cual eleva el par de salida a 4.3 Nw.m y una velocidad angular de aproximadamente $4\pi\frac{rad}{s}$. La energía almacenada en las baterías será de aprox. 518400 Joules y teniendo un consumo de aprox. 2.5 Amp-hora, podemos estar trabajando una media de 4-5 horas antes de tener que volver a recargarlas.

### 3.4.3 El sistema odométrico.

La odometría es el método de navegación más ámpliamente utilizado en la actualidad para realizar el posicionamiento de un robot móvil. Se sabe que mediante sistemas odométricos se obtiene una buena precisión de la posición, son (relativamente) baratos y permiten altas velocidades de muestreo. De todas formas la idea básica del funcionamiento de la odometría es la integración de los incrementos del movimiento a lo largo del tiempo, lo cual nos lleva inevitablemente a la acumulación de errores. Especialmente grave es la acumulación de errores de orientación, puesto que esto causará grandes errores de posición conforme vaya aumentando la distancia viajada por el robot. A pesar de estas limitaciones, la gran mayoría de la comunidad científica reconoce que la odometría es una parte muy importante en la tarea del posicionado de un robot móvil. Este método de posicionamiento se usa en casi todos los robots móviles por varias razones:

- Los datos odométricos pueden ser fusionados con las medidas de la posición absoluta para proporcionar una mejor estimación de la posición [Cox91], [Hol91], [R.H93], [CC92], [Eva94].
- La odometría puede usarse entre dos actualizaciones de la posición absoluta mediante marcas, de manera que se necesitará colocar menos marcas en el entorno para un recorrido determinado.
- En algunos casos, la odometría es la única fuente de información de navegación disponible; por ejemplo cuando no se dispone de marcas externas o cuando cualquier otro subsistema sensorial de navegación deja de funcionar.

La odometría es la parte encargada de contar los pulsos generados por los codificadores ópticos unidos a los motores. De esta manera se puede tener una referencia de los movimientos relativos del robot, es decir, dada una posición y orientación del centro geométrico del sistema, a partir de la lectura de los pulsos asociados a cada una de las ruedas para un determinado movimiento, es posible conocer la nueva posición y orientación.

Esto es sólo de validez limitada, puesto que está basado en la suposición de que la revolución de las ruedas se puede trasladar directamente en desplazamiento lineal relativo al suelo. Un ejemplo extremo en el que esto no se cumple es el derrape. En este caso, el desplazamiento lineal sufrido no se ve reflejado (al menos fielmente) en los pulsos almacenados para dicho movimiento.

Todas las fuentes de error se pueden dividir en dos categorías: errores sistemáticos y errores no sistemáticos. En cuanto a los errores sistemáticos (es decir, acumulativos) más frecuentes cabe citar

- Diámetros de las ruedas desiguales.
- Diámetros reales de las ruedas distintos de los nominales.
- Distancia entre ruedas de tracción distinta de la nominal.
- Desalineamiento de las ruedas tractoras.
- Resolución finita del encoder.

mientras que los errores no sistemáticos más frecuentes suelen ser

- Desplazamiento sobre suelos rugosos.
- Desplazamiento sobre objetos inesperados en el suelo.
- Derrapes.

Normalmente son mucho más importantes los errores sistemáticos que los no sistemáticos por su carácter acumulativo, aunque a veces, un error no sistemático puede causar mayor perjuicio que los errores sistemáticos acumulados durante cierto tiempo.

Los codificadores ópticos utilizados son de tipo incremental y poseen dos salidas formadas por ondas cuadradas desfasadas entre si $\frac{\pi}{2}$. La resolución es de 500 pulsos por revolución. Puesto que el eje del encoder va directamente unido al eje del motor y este posee en su salida un reductor con una relación de 43:1, entonces una vuelta del eje de salida del reductor (donde va directamente unida la rueda) equivale a 21500 pulsos, o lo que es lo mismo, cada pulso equivale a un desplazamiento angular de 0.0167 grados, lo que equivale a un desplazamiento lineal de $\frac{100\pi}{21500} = 0.0146mm$. De todas maneras,

esta es la resolución seleccionada, puesto que en realidad, debido a las características del LS7166, la resolución del encoder puede ser multiplicada hasta por 4, de forma que el desplazamiento lineal mínimo que podría conseguirse es de $\frac{100\pi}{86000} = 0.00365$ mm.

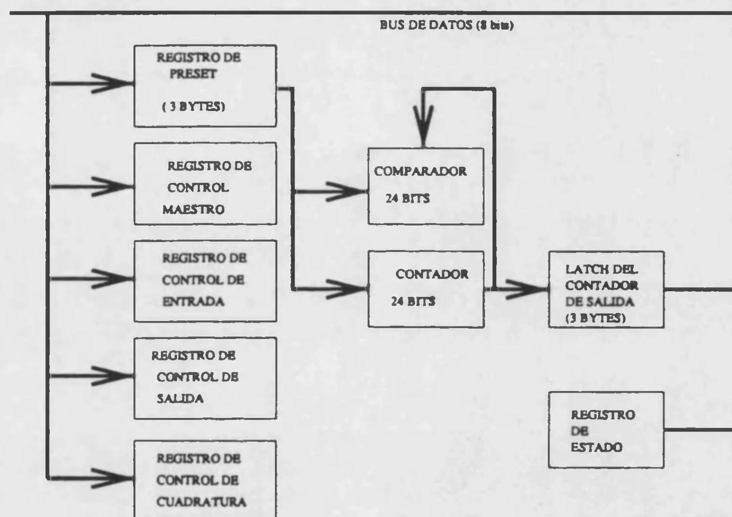



Figura 3.11: Diagrama de bloques del LS7166

El dispositivo encargado de realizar el contaje de los pulsos de los codificadores ópticos es el circuito integrado LS7166 de US Digital Corporation. El LS7166 es un contador de 24 bits con tecnología CMOS que se puede programar para operar de varios modos diferentes. El modo de operación se selecciona escribiendo palabras de control en los registros internos. Existen 3 registros de control de 6 bits y 1 de 2 bits para seleccionar las características funcionales. Además de los registros de control, existe un registro de salida de estado (OSR) que indica el estado actual del contador. El LS7166 se comunica con la circuitería externa a través de un bus de E/S de 8 bits triestado. Las palabras de control y los datos se escriben en el integrado a través del bus. Además del bus de E/S existe un número de entradas y salidas discretas para facilitar funciones de control por hardware e indicaciones de estado instantáneas. En la figura 3.11 se puede observar un diagrama de bloques del integrado en cuestión. Las especificaciones detalladas de funcionamiento y programación se añaden en el apéndice C. Por último en el apéndice B.3 se puede observar un diagrama esquemático de la placa que gobierna el sistema odométrico.

### 3.4.4 El módulo sonar.

#### Detección mediante variación de amplitud.

La técnica de detección de obstáculos mediante variación de amplitud es un procedimiento raramente usado. Se trata de emitir una señal a una frecuencia ultrasónica y recibir la parte de dicha señal que se refleja en los objetos circundantes. De esta manera, suponiendo que la amplitud de la señal reflejada es proporcional a la distancia sensor-obstáculo, es posible conocer la distancia del sensor al objeto en cuestión.

Los dispositivos ultrasónicos utilizados para la detección de objetos están formados por dos elementos separados, una cápsula emisora y una cápsula receptora. La frecuencia de resonancia de la cápsula emisora es de $40 \pm 1KHz$, e igual a la de la cápsula receptora. El ángulo de direccionalidad es de $\pm 20^o$ alrededor del eje acústico. En la figura 3.12 se puede observar la respuesta en frecuencia en la transmisión y en la recepción.

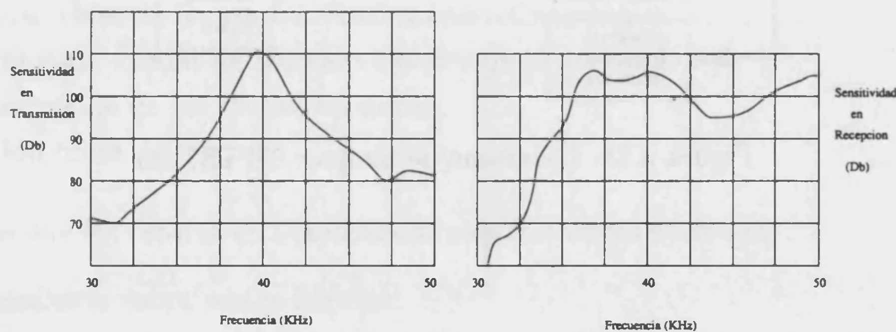



Figura 3.12: Respuesta en frecuencia en emisión y en recepción de las cápsulas ultrasónicas utilizadas

El robot, dispone de 2 pares de emisores-receptores en cada una de las caras. Así, es necesario disponer un circuito con 16 osciladores, uno para cada cápsula emisora. Además, cada emisor lleva asociado un led de color verde que está iluminado siempre que el emisor está funcionando correctamente.

La tarjeta de osciladores para los emisores está compuesta básicamente por circuitos integrados NE555 que son timers de precisión, funcionando en configuración de astable. El esquema básico para este tipo de configuración se presenta en la figura 3.13.

La duración $t_h$ de la salida a nivel alto viene dada por la ecuación

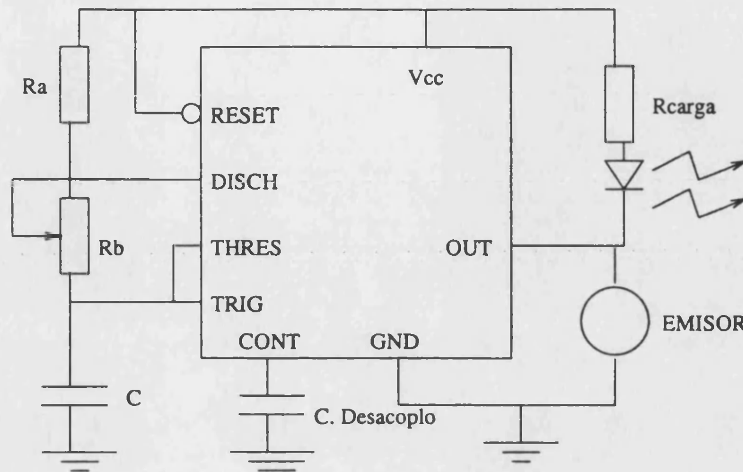



Figura 3.13: Configuración del NE555 como astable

$$t_h = 0.693(Ra + Rb)C \tag{3.17}$$

mientras que la duración de la salida a nivel bajo,$t_l$ viene dada por

$$t_l = 0.693(Rb)C \tag{3.18}$$

y la frecuencia de la señal será

$$Frec = \frac{1.44}{(Ra + 2Rb)C} \tag{3.19}$$

Así, eligiendo adecuadamente los valores para Ra, Rb y C, se puede conseguir una señal cuadrada de unos $40KHz$ para atacar a cada uno de los emisores. Estos emiten una onda senoidal de $40KHz$ la cual al ser reflejada por un objeto debe ser detectada por el circuito de la cápsula receptora.

Hemos de tener en cuenta que el tamaño de los objetos para que la señal pueda ser reflejada depende de la longitud de onda de la señal emitida. Así, teniendo en cuenta que la velocidad del sonido en el aire es de aproximadamente 340 $m/s$ y que la frecuencia de la señal emitida es de $40KHz$, la longitud de onda será de

$$\lambda = \frac{v}{N} = \frac{340}{40.000} = 8.5mm \tag{3.20}$$

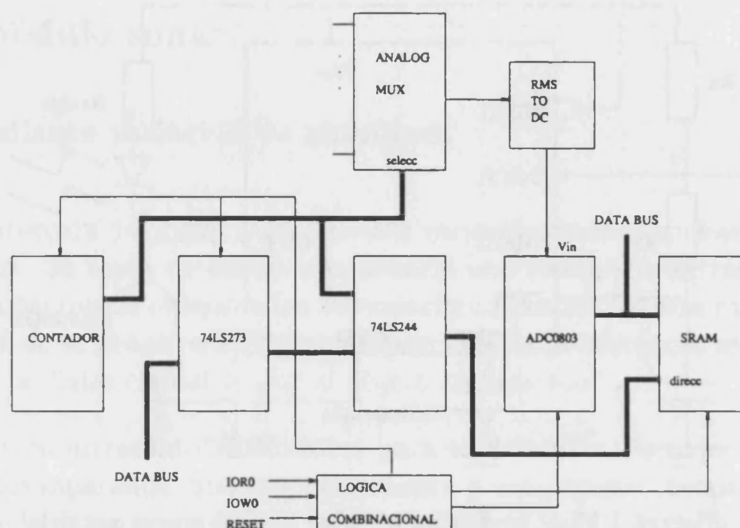



Figura 3.14: Esquema genérico del circuito de receptores.

con lo cual, objetos de un tamaño menor de $8.5mm$ nunca serán detectados.

El esquema de funcionamiento global de la parte de recepción es el siguiente (ver figura 3.14). Cada una de las 16 entradas senoidales procedentes de cada una de las cápsulas receptoras son introducidas en un multiplexor analógico. Un contador va seleccionando secuencialmente cada una de las entradas. La señal seleccionada (senoidal de pequeña amplitud) es pasada a través de un conversor RMS-DC. El nivel de continua obtenido se convierte mediante un conversor ADC (ADC0803) a un código digital, el cual es almacenado en una memoria local. El procedimiento es secuencial y continuo, de manera que el hardware se encarga de manera autónoma de almacenar en la memoria local la última información procedente de dichos sensores. De esta manera se descarga al procesador principal de este tipo de tareas. Así, cuando se quiere conocer la información de los sensores, se puede direccionar cualquiera de los bytes en los que estos están mapeados y leerla. Está claro que cuando se está leyendo la memoria, no se puede actualizar la información de los sensores.

Después de la implementación de este procedimiento para la detección de obstáculos en nuestro robot, este fue descartado debido a la escasa y poco fiable información que se recibía en relación al hardware necesario para su uso. Básicamente, sólo podía usarse como sensor binario de detección de obstáculos y para distancias de detección máximas de aprox. 40 cm (dependiendo del tipo y tamaño de la superficie). Además, existía demasiado ruido que proporcionaba numerosas lecturas erroneas.

**El sistema de medida por tiempo de vuelo.**

El procedimiento sonar más ámpliamente utilizado para la detección de obstáculos es el denominado por tiempo de vuelo. El procedimiento a seguir es muy simple. A una determinada frecuencia se envía una ráfaga de impulsos ultrasónicos. Estos impulsos pueden ocasionalmente chocar con los objetos circundantes y devolver una señal de eco. Así, conociendo la velocidad del sonido en el aire y el tiempo transcurrido entre la activación de la ráfaga de emisión y la recepción de la señal de eco se puede calcular la distancia sensor-objeto.

Polaroid comercializa varios sistemas sonar. En particular en este caso se ha utilizado el módulo sonar de la serie 6500. Este módulo está formado por una tarjeta de control y un transductor electrostático, capaz de medir distancias de hasta 6m con una precisión absoluta del $\pm 1\%$ sobre todo el rango de medidas.

Existen dos modos básicos de operación para el módulo de la serie 6500: el modo de simple eco y el modo de múltiple eco. En el modo de simple eco, que es el utilizado en nuestro caso, el diagrama temporal de las señales involucradas es el siguiente (ver figura 3.15). Después de aplicar la alimentación (Vcc) deben pasar un mínimo de 5 ms antes de que la entrada INIT pueda ponerse a nivel alto. Durante este tiempo se inicializa toda la circuitería interna y se estabiliza el oscilador interno. Cuando se activa la señal INIT, se genera la salida hacia el transductor (XDCR), emitiendo una ráfaga de 16 pulsos a 49.4 KHz con 400 V de amplitud. Con el fin de eliminar la posibilidad de recibir ecos erroneos durante esta etapa, la entrada de recepción (REC) se inhibe por una señal interna durante 2.38 ms. Si se quiere reducir este tiempo con el propósito de poder detectar obstáculos muy cercanos, se puede hacer activando la señal BINH. A continuación se debe esperar la posible activación de la señal de eco (ECHO). En caso de producirse este evento, si se ha medido el tiempo transcurrido entre la activación de INIT y la activación de ECHO, puede deducirse fácilmente la distancia de separación. Por último en la figura 3.16 se puede observar un diagrama de bloques del módulo sonar utilizado.

### 3.4.5 Los microinterruptores.

Rodney posee 8 microinterruptores dispuestos cada uno en uno de los vértices del octógono que forma. De esta manera se pretende detectar cualquier posible colisión tanto de cualquier esquina (con lo cual se activaría un único microinterruptor) como de cualquier cara (con lo cual se activarían los dos microinterruptores situados en sus esquinas) siempre que el obstáculo fuese lo suficientemente grande.

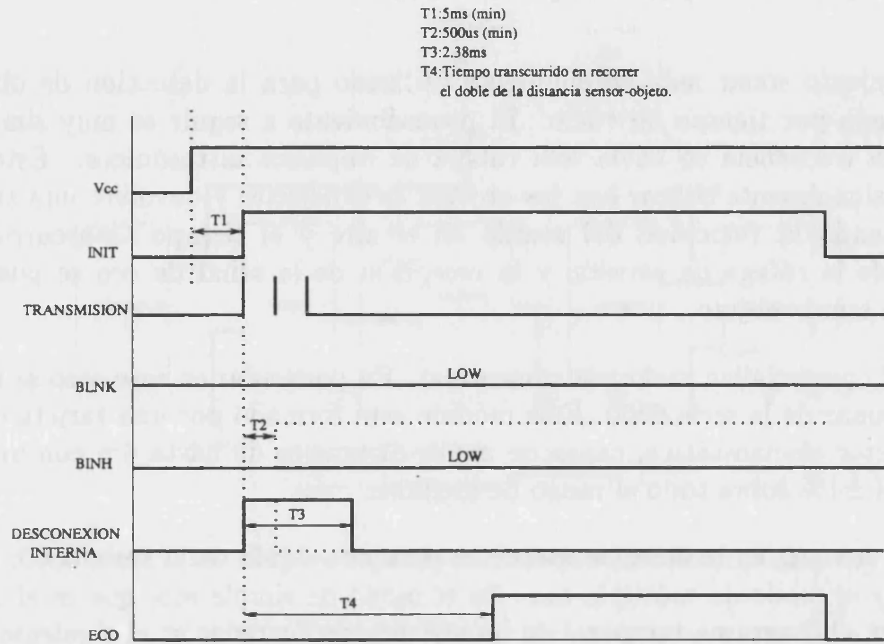



Figura 3.15: Diagrama temporal de las principales señales para la serie 6500.

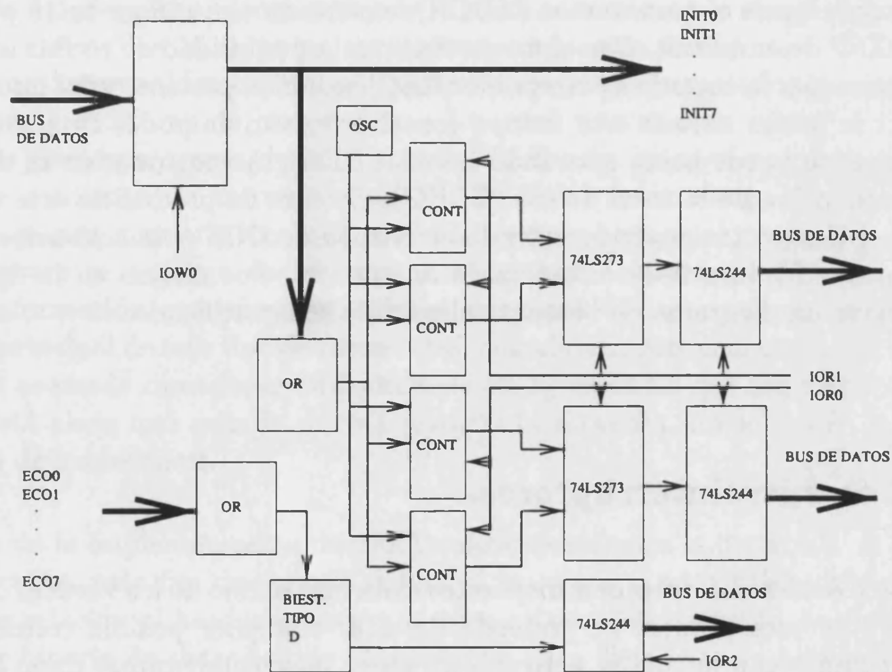



Figura 3.16: Diagrama de bloques del módulo sonar por tiempo de vuelo.

El circuito está diseñado de manera que cada microinterruptor genera la entrada de 1 bit a un registro de 1 byte. De esta manera, cada bit del registro puede tener un valor lógico de 1 ó 0 según el microinterruptor esté cerrado o abierto (colisión o no colisión). Así, leyendo dicho registro, podemos conocer el estado del conjunto de microinterruptores y por lo tanto si ha habido o no colisión.

### 3.4.6 El display.

A fin de poder conocer el estado en el que se encuentra el robot en un momento determinado, se ha dispuesto un display con dos visualizadores de 7 segmentos de cátodo común. Asociado a estos dos visualizadores, se ha dispuesto un registro que almacenará el código que quiere ser enviado al display; así, el código escrito, después de pasar por sendos decodificadores BCD-7 segmentos, generarán las señales oportunas para atacar a los visualizadores. Po último, un esquema del circuito se puede ver en la figura 3.17.

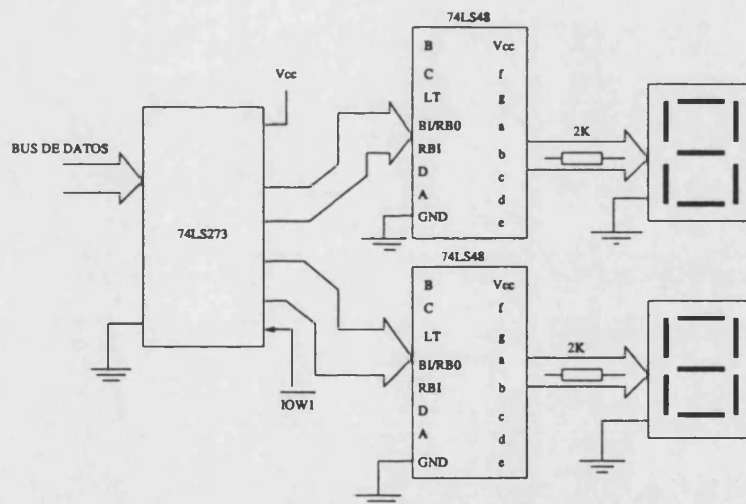



Figura 3.17: Esquema del circuito correspondiente a los visualizadores.

# Capítulo 4

# El sistema de control.

## 4.1 Introducción.

En este capítulo se abordará el problema del movimiento del robot, es decir, qué órdenes y en qué momentos son enviadas a los motores, para que estos o giren a la velocidad deseada o giren el ángulo requerido y todo ello cumpliendo además determinadas especificaciones de control. Así, partiendo de las consideraciones que conviene tener en cuenta a la hora de realizar el diseño de un robot móvil, se seguirá en las siguientes dos secciones analizando primero el bucle de control para que cada motor gire a la velocidad deseada de forma constante (análisis del bucle de control de velocidad) y después el bucle de control para que cada motor gire independientemente un determinado ángulo (análisis del bucle de control de posición). A continuación se presentará una posible estrategia de control para el movimiento del robot, es decir, la forma en la que el vehículo se aproxima a la localización deseada y la implementación de dicha estrategia, pasando a continuación al análisis del sistema de control que garantiza su cumplimiento. Por fin, en la última sección se intentará mejorar el sistema de control para garantizar un error de posición nulo (es decir que la velocidad real del vehículo coincida con la de referencia en cualquier caso).

## 4.2 Consideraciones en el diseño de robots móviles.

A la hora de la construcción de un robot móvil, es deseable colocar las dos ruedas motrices lo más alejadas que nos sea posible entre ellas por las siguientes razones:

- Se mejora considerablemente la estabilidad del vehículo.

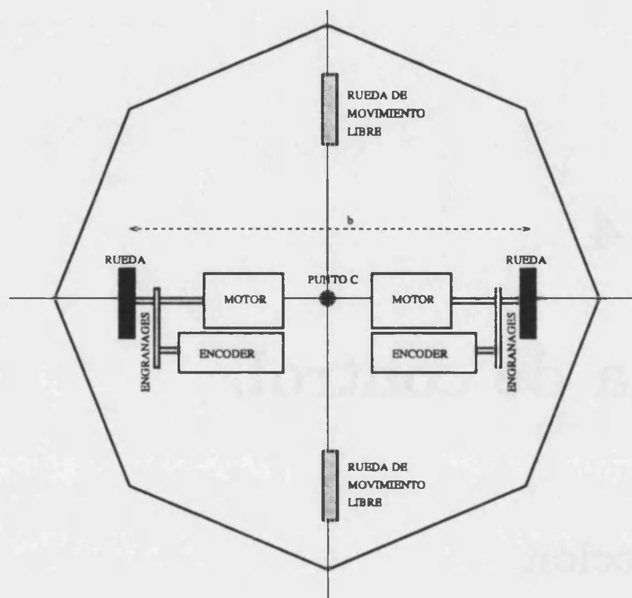



Figura 4.1: Diagrama del robot móvil en estudio

- Decrece el efecto de la resolución limitada del encoder en el error de orientación, puesto que un único pulso de encoder tendrá menos influencia en la orientación de la plataforma.
- Durante movimientos en linea recta, perturbaciones mecánicas pueden causar que los motores giren temporalmente a velocidades angulares diferentes, con lo cual el movimiento resultante será un camino curvo. Se puede observar fácilmente por trigonometría que el radio de la curva trazada es directamente proporcional a la distancia de separación entre las ruedas.
- Diferencias en los diámetros de las dos ruedas, también darán como resultado un camino curvado con un radio proporcional a la distancia entre las ruedas directoras. Si ya de por sí es difícil encontrar ruedas con exactamente el mismo diámetro nominal, las posibles distribuciones de carga asimétricas en el robot pueden hacer que una rueda varíe ligeramente su diámetro respecto de la otra. Ruedas con diferentes diámetros, harán que el robot trace una trayectoria en forma de arco en vez de una recta incluso aunque el algoritmo de control funcione perfectamente y su cometido sea mantener velocidades iguales en los motores asociados a cada una de las ruedas motrices. Así, el radio del arco trazado y el error de orientación se pueden calcular fácilmente por trigonometría ([Bk85]):

$$R = \frac{b}{u}(D + u) \simeq \frac{bD}{u} \tag{4.1}$$

$$\alpha \simeq \frac{L}{R} \simeq \frac{Lu}{Db} \tag{4.2}$$

donde

- R = radio del camino curvado debido a los diferentes diámetros de las ruedas directoras.
- $\alpha$ = error de orientación en radianes.
- L = distancia recorrida.
- u = diferencia de diámetros entre ambas ruedas.
- b = distancia entre ruedas directoras.
- D=diámetro nominal de las ruedas directoras.

Por otro lado, una anchura exagerada de la plataforma base afectará contrariamente a la movilidad del vehículo dentro de una superficie cerrada.

El derrape de las ruedas puede ocurrir principalmente durante dos tipos de movimiento:

- Aceleraciones y deceleraciones.
- Movimientos curvos donde las fuerzas centrífugas pueden causar derrapes laterales.

Este último efecto se puede minimizar limitando los movimientos curvos a rotaciones sobre el centro del vehículo (punto C) donde ambas ruedas directoras giran a la misma velocidad, pero en direcciones opuestas. En este caso, y con distribuciones de carga simétricas en la plataforma, no se generan fuerzas laterales en las ruedas. El derrape durante las fases de aceleración y deceleración se reduce empleando bajos valores de éstos.

## 4.3 Análisis del bucle de control de velocidad.

El esquema genérico del control de velocidad de un motor de cc típico es el representado en la figura 4.2. En él, se pueden observar los bloques principales que lo constituyen. Así, la señal de error se genera como la diferencia entre la velocidad de entrada deseada (dada en pulsos/T) y la referencia de velocidad real obtenida por la lectura de un registro contador asociado a un codificador óptico incremental unido al eje de la rueda. Dicha señal de error es la entrada a un regulador, el cual gobierna al modulador por anchura de pulso (PWM). Este genera la onda cuadrada deseada, la cual, tras pasar por la etapa de potencia es enviada como tensión al motor. A cada periodo de muestreo (T) se lee el contador hardware, el cual nos indica un número de pulsos que representa el ángulo girado

por el eje del encoder (y por lo tanto, también por el eje del motor). Inmediatamente después de leer el contador hardware, éste es puesto a cero. De esta manera, el valor leido, se puede considerar como una aproximación discreta de primer orden a la velocidad angular (en pulsos/T).

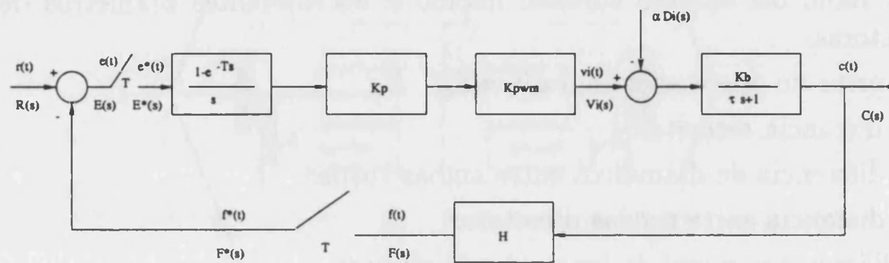



Figura 4.2: Diagrama de bloques del bucle de control de velocidad

El motor se modela como un sistema de primer orden, con una ganáncia estática $K_b$ y una constante de tiempo $\tau$. El encoder tiene una resolución de H pulsos/revolución. Por último el PWM tiene una salida cuasi-lineal con una ganancia $K_{pwm}$.

Según el esquema de un sistema de control digital como el presentado en la figura 4.3, la función de transferencia (fdt) en lazo cerrado en ausencia de preturbaciones $D_i(s)$ puede deducirse a partir de las siguientes ecuaciones:

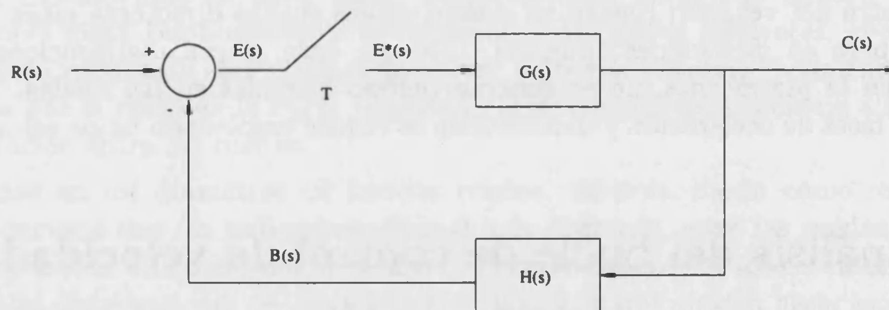



Figura 4.3: Esquema genérico de un bucle de control digital

$$
\begin{aligned}
B(s) &= H(s)G(s)E^*(s) \\
B^*(s) &= [G(s)H(s)]^* E^*(s) \\
E(s) &= R(s) - B(s) \\
E^*(s) &= R^*(s) - B^*(s) = R^*(s) - [H(s)G(s)]^* E^*(s)
\end{aligned}
$$

UNIVERSITAT DE VALÈNCIA BIBLIOTECA CIÈNCIES



$$\begin{aligned} E^*(s) &= \frac{R^*(s)}{1+[G(s)H(s)]^*} \\ C^*(s) &= G^*(s)\frac{R^*(s)}{1+[G(s)H(s)]^*} \\ \frac{C(z)}{R(z)} &= \frac{G(z)}{1+GH(z)} \end{aligned} \tag{4.3}$$

Así, observando que los valores para G(s) y H(s) en nuestro bucle de control son

$$\begin{aligned} G(s) &= K_p K_{pwm} \frac{K_b}{\tau s+1} \\ H(s) &= H \end{aligned} \tag{4.4}$$

donde $K_p$ es la ganancia del regulador (supuesto un regulador proporcional), podemos obtener los valores para G(z) y GH(z) aplicando la transformada z a G(s) y G(s)H(s) respectivamente. Así,

$$G(z) = Z\{G(s)\} = (1-z^{-1})K_b K_p K_{pwm} Z\left\{\frac{1}{s(1+\tau s)}\right\} = K\left[1-\frac{z-1}{z-a}\right] \tag{4.5}$$

donde $K \equiv K_p K_b K_{pwm}$ y $a = e^{-\frac{T}{\tau}}$

$$GH(z) = (1-z^{-1})K_p K_b K_{pwm} H Z\left\{\frac{1}{s(1+\tau s)}\right\} = K_1\left[1-\frac{z-1}{z-a}\right] = \frac{K_1(1-a)}{z-a} \tag{4.6}$$

siendo $K_1 \equiv KH$

De esta manera podemos obtener la fdt total M(z) como:

$$M(z) = \frac{C(z)}{R(z)} = \frac{G(z)}{1+GH(z)} = \frac{K(1-a)}{z-a+K_1(1-a)} = \frac{q_0}{z-p_0} \tag{4.7}$$

donde

$$\begin{aligned} q_0 &= K(1-a) \\ p_0 &= a - K_1(1-a) \end{aligned}$$

En el caso más genérico en que consideremos una posible entrada de perturbación, añadida a la entrada de control al motor, se puede aplicar el principio de superposición, de manera que la salida total se obtiene como la suma de las salidas obtenidas considerando sólo una de las entradas y suponiendo nula la restante.

Así, la respuesta total $\frac{C_t(z)}{R_t(z)}$ se puede obtener como

$$\frac{C_t(z)}{R_t(z)} = \frac{C(z)}{R(z)} + \frac{C_1(z)}{R_1(z)} \tag{4.8}$$

donde $\frac{C_1(z)}{R_1(z)}$ es la fdt considerando la entrada de perturbación y suponiendo nula la entrada R(z). Entonces, observando la figura 4.4, la fdt para la entrada de perturbación, es:

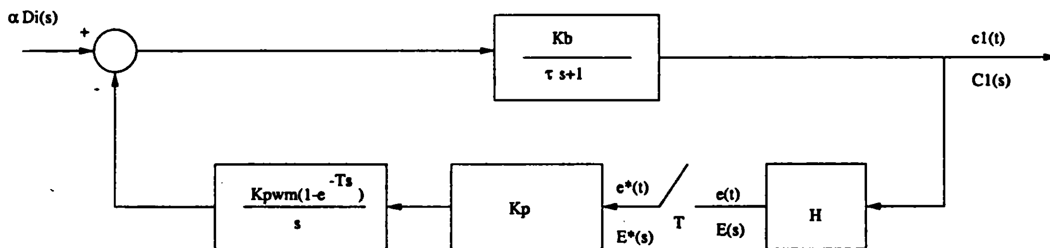



Figura 4.4: Diagrama de bloques considerando la entrada perturbadora y regulador P.

$$\begin{aligned} E(s) &= \frac{HK_b}{1+\tau s}\left[\alpha D_i(s) - \frac{K_p K_{pwm}(1-e^{-Ts})}{s}E^*(s)\right] = \\ &= \frac{HK_b}{1+\tau s}\alpha D_i(s) - \frac{HK_b}{1+\tau s}\frac{K_p K_{pwm}(1-e^{-Ts})}{s}E^*(s) \end{aligned} \tag{4.9}$$

$$E^*(s) = \left[\frac{HK_b}{1+\tau s}\alpha D_i(s)\right]^* - \left[\frac{HK_b}{1+\tau s}\frac{K_p K_{pwm}(1-e^{-Ts})}{s}\right]^* E^*(s) \tag{4.10}$$

y sacando factor común a $E^*(s)$ obtenemos

$$E^*(s) = \frac{\left[\frac{HK_b}{1+\tau s}\alpha D_i(s)\right]^*}{1+\left[\frac{HK_bK_pK_{pwm}(1-e^{-Ts})}{s(1+\tau s)}\right]^*} \tag{4.11}$$

Además, tenemos que

$$E(s) = HC_1(s) \tag{4.12}$$

$$E^*(s) = HC_1^*(s) \tag{4.13}$$

$$C_1^*(s) = \frac{E^*(S)}{H} = \frac{\left[\frac{HK_b}{1+\tau s}\right]^* [\alpha D_i(s)]^* \frac{1}{H}}{1+\left[\frac{HK_bK_pK_{pwm}(1-e^{-Ts})}{s(1+\tau s)}\right]^*} \tag{4.14}$$

y aplicando la transformada z, obtenemos finalmente,

$$\frac{C_1(z)}{\alpha D_i(z)} = Z\left\{\frac{C_1^*(s)}{\alpha D_i^*(s)}\right\} = \frac{K_b Z\left\{\frac{1}{1+\tau s}\right\}}{1+HK_bK_pK_{pwm}(1-z^{-1})Z\left\{\frac{1}{s(1+\tau s)}\right\}} \tag{4.15}$$

$$\frac{C_1(z)}{\alpha D_i(z)} = \frac{K_b(1-a)z}{z^2+[K_1K_b(1-a)-a-1]\,z+[a-K_1K_b(1-a)]} \tag{4.16}$$

Así, la respuesta total del sistema de control de velocidad (suponiéndose un regulador de tipo proporcional) puede ponerse como

$$C_t(z) = \frac{q_0}{z-p_0}C(z) - \frac{m_0 z}{z^2+n_0 z+s_0}\alpha D_i(z) \tag{4.17}$$

donde

$$\begin{aligned} K &= K_pK_bK_{pwm} \\ K_1 &= KH \end{aligned}$$

$$\begin{aligned} a &= e^{-\frac{T}{\tau}} \\ q_0 &= K(1-a) \\ p_0 &= a - K_1(1-a) \\ m_0 &= K_b(1-a) \\ n_0 &= K_1 K_b(1-a) - a - 1 \\ s_0 &= a - K_1 K_b(1-a) \end{aligned}$$

Suponiendo normalmente despreciable la entrada de perturbación, se puede observar que $G(s)H(s)$ es un sistema de tipo 0 (salvo en el caso de que $p_0$ tomase el valor 1, para lo cual $K_1$ tendría que valer -1 lo cual no puede darse nunca) con lo que el sistema de control de velocidad con un regulador de tipo proporcional presentará error de posición frente a una entrada de tipo escalón de velocidad [1].

Para subsanar esta deficiencia en régimen permanente, es preciso aumentar el tipo del sistema, lo cual se puede solucionar fácilmente introduciendo un polo en el origen en $G(s)H(s)$. Esto puede hacerse sustituyendo el regulador proporcional por uno proporcional-integral (PI).

Si calculamos ahora las expresiones equivalentes para G(z) y GH(z) usando un regulador PI cuya función de transferencia discreta se puede poner como $K_p+\frac{z}{z-1}K_c$, obtenemos:

$$G(z) = \left[\frac{K(1-a)}{z-a}\right]\left[K_p + \frac{z}{z-1}K_c\right] = \frac{(K_{x1}+K_{x2})z - K_{x2}}{(z-1)(z-a)} \tag{4.18}$$

donde

$$\begin{aligned} K_{x1} &= K_b K_{pwm} K_c(1-a) \\ K_{x2} &= K_b K_{pwm} K_p(1-a) \end{aligned}$$

y para $GH(z)$ obtenemos:

$$\begin{aligned} GH(z) &= (1-z^{-1})HK_bK_{pwm}\left(K_p + \frac{z}{z-1}K_c\right) Z\left\{\frac{1}{s(\tau s+1)}\right\} = \\ &= \frac{K_1(1-a)}{z-a}\left(K_p + \frac{z}{z-1}K_c\right) = \frac{(K_{x3}+K_{x4})\,z - K_{x4}}{(z-1)(z-a)} \end{aligned} \tag{4.19}$$

[1] La entrada es un escalón, pero representa la velocidad de referencia

donde

$$\begin{aligned} K_{x3} &= HK_bK_{pwm}K_c(1-a) \\ K_{x4} &= HK_bK_{pwm}K_p(1-a) \end{aligned}$$

de manera que la fdt considerando únicamente la entrada de control se puede obtener como

$$M(z) = \frac{C(z)}{R(z)} = \frac{G(z)}{1+GH(z)} = \frac{(K_{x1}+K_{x2})\,z - K_{x2}}{z^2 + (K_{x3}+K_{x4}-a-1)\,z - K_{x4}} \tag{4.20}$$

De la misma manera que para el caso de utilizar un regulador proporcional, si consideramos ahora la salida, teniendo en cuenta únicamente la entrada de perturbación y suponiendo nula la de control, observando la figura 4.5, obtenemos:

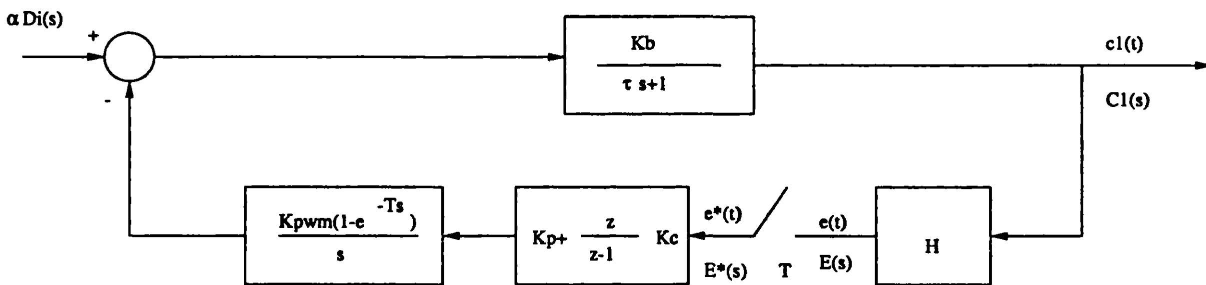



Figura 4.5: Diagrama de bloques considerando la entrada perturbadora y regulador PI.

$$\begin{aligned} E(s) &= \frac{HK_b}{\tau s+1}\left\{\alpha D_i(s) - \left[\frac{K_{pwm}\left(1-e^{-Ts}\right)}{s}\right]\left(K_p + \frac{z}{z-1}K_c\right)E^*(s)\right\} = \\ &= \frac{HK_b\alpha D_i(s)}{\tau s+1} - \frac{HK_b}{\tau s+1}\left[\frac{K_pK_{pwm}\left(1-e^{-Ts}\right)}{s} + \left(\frac{K_{pwm}K_c\left(1-e^{-Ts}\right)}{s}\right)\left(\frac{z}{z-1}\right)\right]E^*(s) = \\ &= \frac{HK_b\alpha D_i(s)}{\tau s+1} - \frac{HK_bK_pK_{pwm}\left(1-e^{-Ts}\right)}{s(\tau s+1)}E^*(s) - \frac{HK_bK_cK_{pwm}\left(1-e^{-Ts}\right)}{s(\tau s+1)}\frac{z}{z-1}E^*(s) \end{aligned}$$

Muestreando ahora la señal de error $E(s)$, obtenemos:

$$E^*(s) = \left[\frac{HK_b\alpha D_i(s)}{\tau s+1}\right]^* - \left[\frac{HK_bK_pK_{pwm}\left(1-e^{-Ts}\right)}{s(\tau s+1)}\right]^* E^*(s) - $$
$$- \left[\frac{HK_bK_cK_{pwm}(1-e^{-Ts})}{s(\tau s+1)}\right]^* \frac{z}{z-1}E^*(s) \quad (4.21)$$

De donde sacando factor común y despejando, llegamos a

$$E^*(s) = \frac{\left[\frac{HK_b}{\tau s+1}\right]^* [\alpha D_i(s)]^*}{1+\left[\frac{HK_bK_pK_{pwm}(1-e^{-Ts})}{s(\tau s+1)}\right]^* + \left[\frac{HK_bK_cK_{pwm}(1-e^{-Ts})}{s(\tau s+1)}\right]^* \frac{z}{z-1}} \quad (4.22)$$

Además,

$$E(s) = HC_1(s) \quad (4.23)$$

con lo que

$$E^*(s) = HC_1^*(s) \quad (4.24)$$

De esta forma,

$$C_1^*(s) = \frac{E^*(s)}{H} = \frac{\left[\frac{HK_b}{\tau s+1}\right]^* [\alpha D_i(s)]^* \frac{1}{H}}{1+\left[\frac{HK_bK_pK_{pwm}(1-e^{-Ts})}{s(\tau s+1)}\right]^* + \left[\frac{HK_bK_cK_{pwm}(1-E^{-Ts})}{s(\tau s+1)}\right]^* \frac{z}{z-1}} \quad (4.25)$$

y la función de transferencia considerando únicamente la entrada de perturbación, se obtendrá como:

$$\frac{C_1(z)}{\alpha D_i(z)} = Z\left\{\frac{C_1^*(s)}{\alpha D_i^*(s)}\right\} =$$
$$= \frac{K_bZ\left\{\frac{1}{\tau s+1}\right\}}{1+HK_bK_pK_{pwm}(1-z^{-1})Z\left\{\frac{1}{s(\tau s+1)}\right\}+HK_bK_cK_{pwm}(1-z^{-1})Z\left\{\frac{1}{s(\tau s+1)}\right\}\frac{z}{z-1}}$$

Desarrollando convenientemente la expresión anterior, podemos llegar a la forma final de la fdt formada por la salida y la entrada perturbadora,

$$\frac{C_1(z)}{\alpha D_i(z)} = \frac{[K_b(1-a)a]\,z}{[a(1+K_1)-K_1a^2]\,z^2+[a(K_2-K_1-1)+a^2(K_1-K_2)-1]\,z+1} = \\ = \frac{mz}{z^2+nz+s} \tag{4.26}$$

donde

$$\begin{aligned} K_1 &= HK_bK_pK_{pwm} \\ K_2 &= HK_bK_cK_{pwm} \\ m &= \frac{K_b(1-a)}{1+K_1(1-a)} \\ n &= \frac{a(K_2-K_1-1)+a^2(K_1-K_2)-1}{a(1+K_1)-K_1a^2} \\ s &= \frac{1}{a(1+K_1)-K_1a^2} \end{aligned}$$

De esta forma, la fdt final obtenida como superposición de entradas (la de control y la perturbadora) suponiendo nula la entrada restante, será la suma de las respuestas calculadas en las expresiones anteriores, es decir,

$$C_t(z) = \frac{q_1z+q_0}{z^2+p_1z+p_0}C(z) + \frac{mz}{z^2+nz+s}\alpha D_i(s) \tag{4.27}$$

donde

$$\begin{aligned} q_1 &= K_{x1}+K_{x2} \\ q_0 &= -K_{x2} \\ p_1 &= K_{x3}+K_{x4}-a-1 \\ p_0 &= -K_{x4} \end{aligned}$$

Por último, en las figuras desde la 4.6 hasta la 4.11 se puede observar el comportamiento del bucle de control de velocidad frente a diversos valores para los coeficientes del regulador en el caso de entrada escalón de 100 $\frac{pulsos}{T}$.

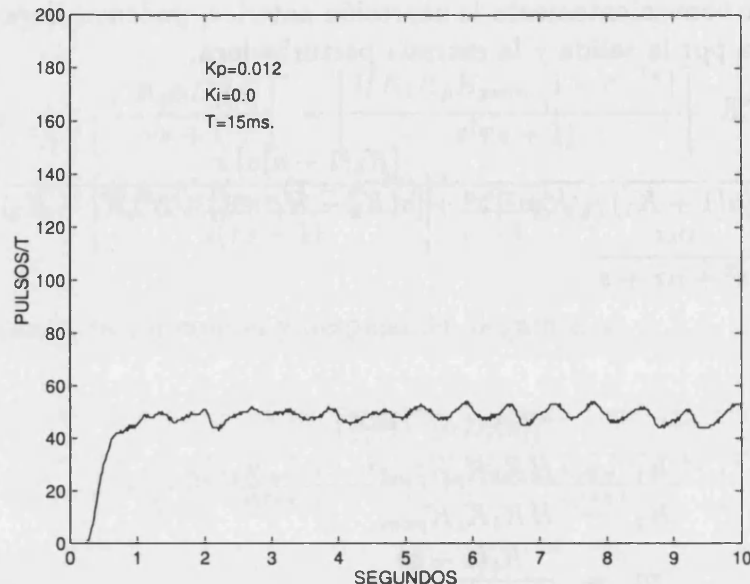



Figura 4.6: Respuesta del bucle de control de velocidad con ganancia insuficiente.

# 4.4 Análisis del bucle de control de posición.

El esquema utilizado para el control de la posición de cada uno de los ejes motrices es el típico control de posición digital con realimentación de velocidad. La realimentación de velocidad mejora la respuesta dinámica del sistema, puesto que con un valor adecuado de la constante de realimentación, se puede conseguir guiar a un sistema con respuesta subamortiguada a uno con respuesta críticamente amortiguada. La razón por la que se consigue esto es porque aumentando el valor de la constante de realimentación (Kg), conseguimos que la señal de error, se anule antes de que los ejes estén alineados. De esta manera se adelanta la corrección de la posición, consiguiendo tiempos de estabilización menores. El esquema genérico puede observarse en la figura 4.12.

En el análisis de este bucle, podemos observar que la señal de error es

$$E(s) = R(s) - \left[1 + K_g \frac{z-1}{z}\right] C(s) \tag{4.28}$$

donde

$$C(s) = \frac{z}{z-1} F^*(s) \tag{4.29}$$

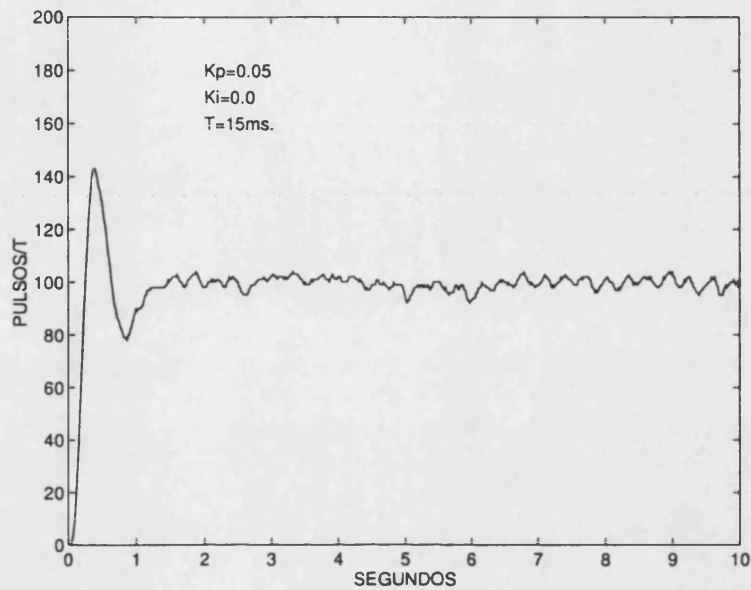



Figura 4.7: Respuesta del bucle de control de velocidad sobre Rodney. Caso 2.

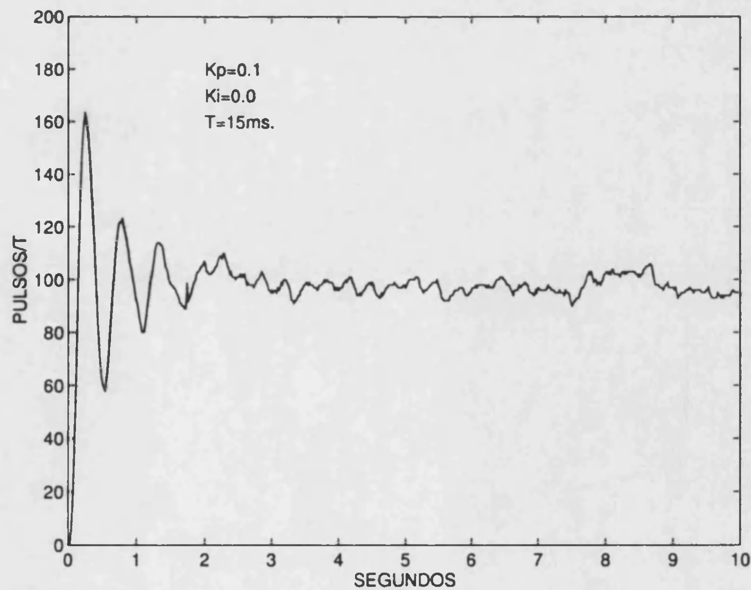



Figura 4.8: Respuesta del bucle de control de velocidad sobre Rodney. Caso 3.

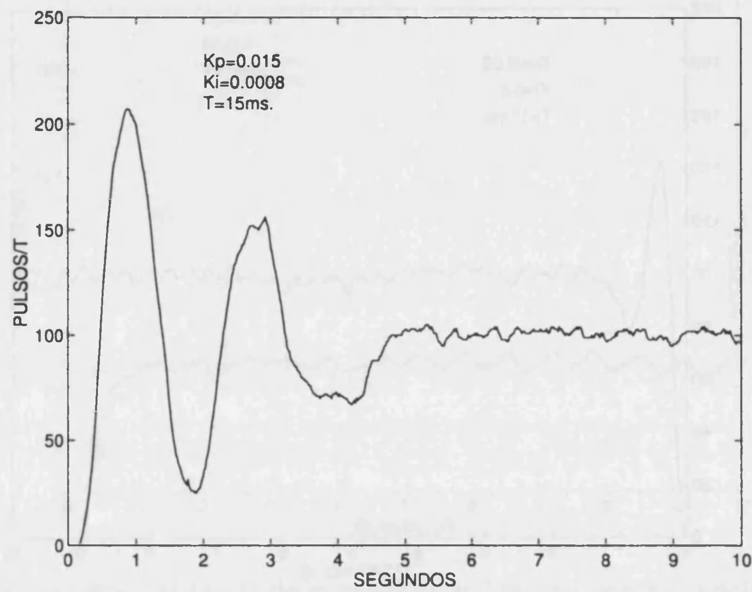



Figura 4.9: Respuesta del bucle de control de velocidad sobre Rodney. Caso 4.

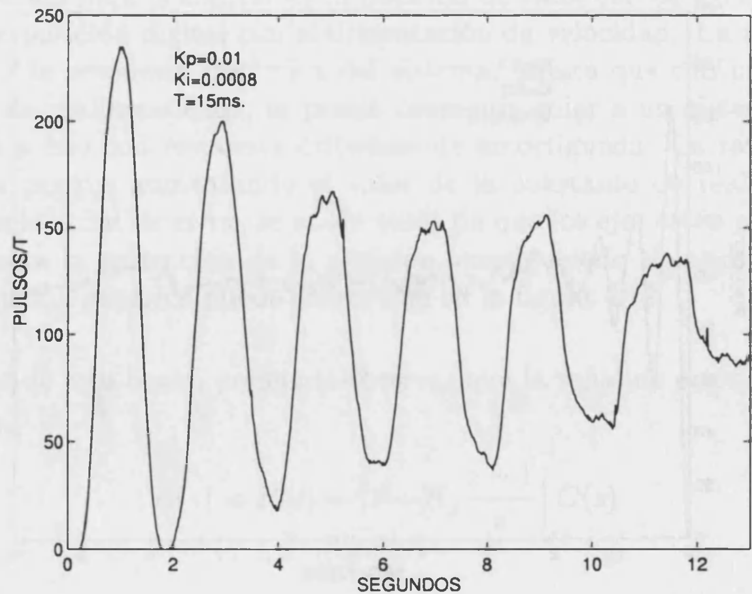



Figura 4.10: Respuesta del bucle de control de velocidad sobre Rodney. Caso 5.

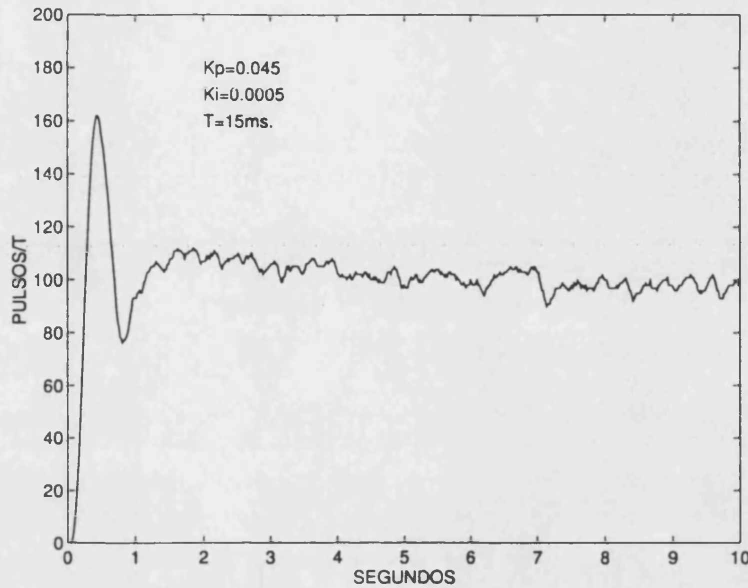



Figura 4.11: Respuesta del bucle de control de velocidad sobre Rodney. Caso 6.

A su vez,

$$F(s) = \frac{HK_pK_{pwm}K_b(1 - e^{-Ts})}{s(1 + \tau s)}E^*(s) \tag{4.30}$$

Muestreando la señal de error, tenemos

$$E^*(s) = R^*(s) - \left[1 + K_g\frac{z-1}{z}\right]C^*(s) \tag{4.31}$$

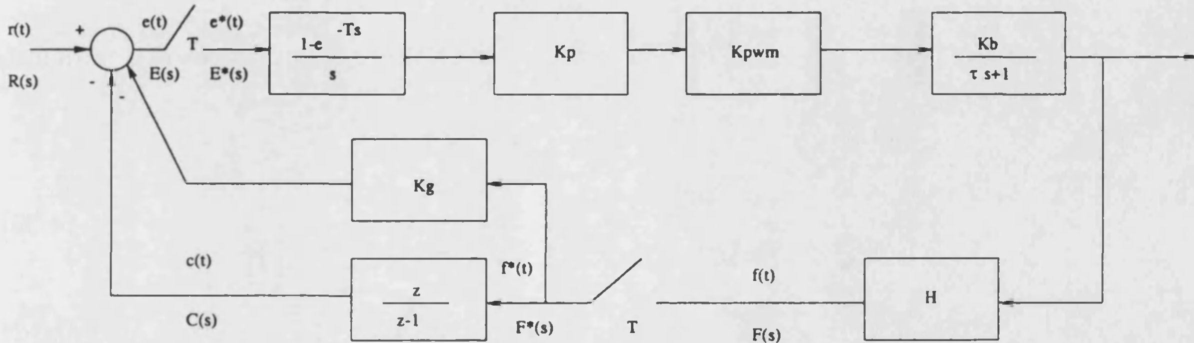



Figura 4.12: Diagrama de bloques del esquema de control de posición

Si muestreamos también la señal de salida del encoder, tenemos

$$F^*(s) = \left[\frac{HK_pK_{pwm}K_b(1-e^{-Ts})}{s(1+\tau s)}\right]^* E^*(s) \tag{4.32}$$

Substituyendo la ecuación (4.31) en la (4.32) y teniendo en cuenta que

$$C^*(s) = \frac{z}{z-1}F^*(s) \tag{4.33}$$

podemos poner

$$\begin{aligned} C^*(s) &= \frac{z}{z-1}\left[\frac{HK_pK_{pwm}K_b(1-e^{-Ts})}{s(1+\tau s)}\right]^* E^*(s) = \\ &= \frac{z}{z-1}\left[\frac{K_1(1-e^{-Ts})}{s(1+\tau s)}\right]^* \left[R^*(s) - \left(1+K_g\frac{z-1}{z}\right)C^*(s)\right] = \\ &= \frac{z}{z-1}\left[\frac{K_1(1-e^{-Ts})}{s(1+\tau s}\right]^* R^*(s) - \\ &- \frac{z}{z-1}\left[\frac{K_1(1-e^{-Ts})}{s(1+\tau s}\right]^* \left(1+K_g\frac{z-1}{z}\right)C^*(s) \end{aligned} \tag{4.34}$$

Sacando factor común en la expresión anterior, podemos obtener la relación entre la salida y la entrada muestreada, esto es:

$$\frac{C^*(s)}{R^*(s)} = \frac{\frac{z}{z-1}\left[\frac{K_1(1-e^{-Ts})}{s(1+\tau s)}\right]^*}{1+\left\{\frac{z}{z-1}\left[\frac{K_1(1-e^{-Ts})}{s(1+\tau s)}\right]^*\left(1+K_g\frac{z-1}{z}\right)\right\}} \tag{4.35}$$

Por último, aplicando la Transformada z al cociente de señales muestreadas, podemos obtener la fdt discreta, de manera que

$$Z\left\{\frac{C^*(s)}{R^*(s)}\right\} = \frac{C(z)}{R(z)} = \frac{Z\left\{\frac{K_1}{s(1+\tau s)}\right\}}{1+\left[\left(1+K_g\frac{z-1}{z}\right)Z\left\{\frac{K_1}{s(1+\tau s)}\right\}\right]} \tag{4.36}$$

Así, desarrollando la expresión anterior, podemos obtener la fdt para el sistema de control de posición con realimentación de velocidad en el sistema de tracción del robot, la cual nos da

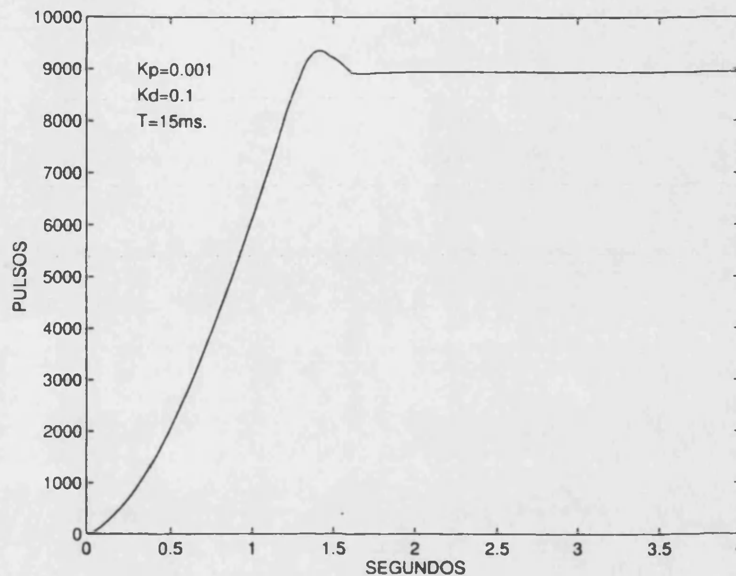



Figura 4.13: Respuesta de la implementación del bucle de control de posición sobre Rodney.

$$M(z) = \frac{K_1(1-a)z}{z^2 + [K_1(1-a)(1+K_g) - (1+a)]\,z + [a - K_1K_g(1-a)]} \tag{4.37}$$

donde como ya estaba definido, $a = e^{-\frac{T}{\tau}}$ y $K_1 \equiv HK_pK_bK_{pwm}$

Sobre nuestro sistema, la aplicación de dicho esquema de control para la realización de rotaciones sobre el centro geométrico, presenta una buena respuesta para valores de la constante de realimentación cercanos a $K_d = 0.1$. En las figuras 4.13 y 4.14 se puede observar la respuesta del robot frente a una rotación de $30^o$ lo cual equivale a una cuenta de 8958 pulsos. El algoritmo tolera errores de $\pm 25$ pulsos, lo cual equivale a un error de orientación de

$$Err_{ang} = \pm\frac{N_{pulsos}\pi d}{rNR} = \pm\frac{25 * \pi * 100.0}{250.0 * 500 * 43} = \pm 0.0014 Rad. \tag{4.38}$$

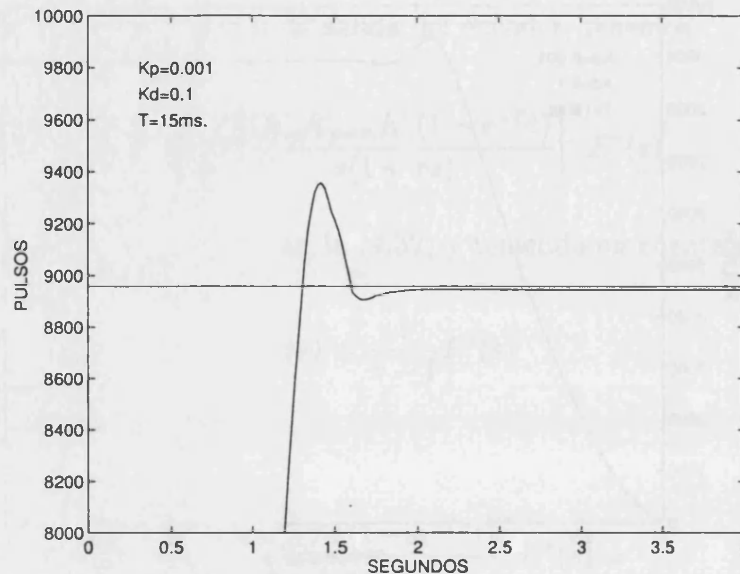



Figura 4.14: Ampliación de la respuesta del control de posición donde se puede observar el error en pulsos.

## 4.5 El control del movimiento.

El control del movimiento se refiere a la estrategia mediante la cual el vehículo se aproxima a la localización deseada, y la implementación de dicha estrategia.

Si bien la plataforma móvil está diseñada de forma genérica, de manera que se puede implementar cualquier estrategia para su movimiento, para el presente estudio, se generarán únicamente dos tipos distintos de movimiento: movimientos en linea recta, donde ambos motores giran a la misma velocidad y en la misma dirección, y rotaciones del vehículo sobre su punto central (punto C), donde ambos motores giran a la misma velocidad, pero en direcciones opuestas. Esta solución tiene el inconveniente de que en la unión entre dos tramos de una determinada trayectoria, el robot debe parar su movimiento, pero también presenta ventajas por varias razones:

- Se minimiza el derrape de las ruedas debido a la acción simultanea de ambas ruedas y a la rotación sobre su centro de gravedad para los giros.
- Se puede diseñar un sistema de control relativamente simple, puesto que la tarea del controlador siempre será mantener iguales las velocidades angulares de ambos motores.
- El vehículo siempre viaja a través del camino más corto posible.

Para poder representar la localización del vehículo relativa a un sistema de coordenadas

fijo, se deben dar tres valores: las coordenadas X e Y del punto central, C, y el ángulo $\theta_0$ entre el eje longitudinal del vehículo y el eje X.

Si se tiene que viajar desde una localización conocida $(x_0, y_0, \theta_0)$ a una nueva $(x_f, y_f, \theta_f)$, para determinar la trayectoria se sigue el siguiente procedimiento. Primeramente se calcula la distancia L y el ángulo $\phi$ que forman el eje X y la recta que une los puntos inicial y final de la trayectoria como (ver figura (4.15):

$$\phi = \arctan\left(\frac{y_f - y_0}{x_f - x_0}\right) \tag{4.39}$$

$$L = \sqrt{(x_f - x_0)^2 + (y_f - y_0)^2} \tag{4.40}$$

Entonces el vehículo gira $\theta_1$ grados sobre su punto central, donde $\theta_1 = \phi - \theta_0$. A continuación viaja a lo largo de una trayectoria recta de longitud L, de forma que su punto central coincidirá con $(x_f, y_f)$. Finalmente el vehículo gira $\theta_2$ grados sobre su punto central, donde $\theta_2 = \theta_f - \phi$.

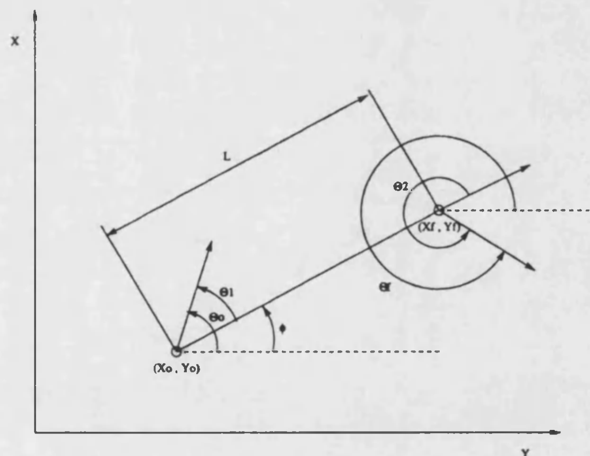



Figura 4.15: Procedimiento propuesto para viajar desde un punto origen a uno destino

Para cada uno de estos pasos, se calcula el número de pulsos que cada motor debe producir para completar el comando. Este número siempre es igual para ambos motores. Durante rotaciones del vehículo, ambos motores giran en direcciones opuestas y durante movimientos en linea recta, ambos giran en la misma dirección. Los cálculos de $L, \phi, \theta_1$ y $\theta_2$ se realizan por el proceso encargado del sistema de control en tiempo real, de manera que la ejecución del resto de tareas no debe afectar al periodo de muestreo en el bucle de control.

## 4.6 Análisis del sistema de control.

Normalmente, el sistema de control del movimiento de un robot móvil está formado por dos bucles de control independientes, uno para cada motor. La coordinación de movimientos en estos sistemas, se realiza ajustando las velocidades de referencia de ambos bucles. El inconveniente de usar un esquema de control de este tipo es que cualquier perturbación dinámica de carga en uno de los ejes producirá un error, que será corregido únicamente por su propio bucle, mientras que el otro eje, a partir de entonces, funcionará seguramente descoordinado. Esta ley de coordinación, normalmente causa un error en el camino resultante.

Una mejora en el cálculo del camino resultante consiste en proporcionar información cruzada entre los dos ejes (referencia cruzada). De esta forma, una perturbación en uno de los dos ejes afecta a ambos bucles de control.

El controlador tratado aquí utiliza una filosofía similar a la de los controladores de referencia cruzada. En este diseño, se calcula el error de dirección y se introduce como una señal de corrección a ambos bucles. Además, para el presente análisis, se supone que las velocidades de referencia absolutas para ambos ejes serán siempre iguales.

Cada contador software asociado a un eje motriz, contiene un número que representa el número total de pulsos generados desde el comienzo de un cierto movimiento. La comparación entre los valores absolutos de ambos contadores software produce una señal de error E :

$$E(i) = |P_1(i)| - |P_2(i)| \tag{4.41}$$

Un valor de E distinto de cero significa que un motor ha girado más rápido que el otro, y el signo de E identifica al motor implicado.

La señal de error genera una variable de corrección M que se puede utilizar o bién para reducir la velocidad del motor más rápido o bién para aumentar la del motor más lento o ambas cosas a la vez (dependiendo de los valores de $\beta_1$ y $\beta_2$). Consecuentemente, las velocidades de ambos motores tienden a igualarse. En (4.41) se usan valores absolutos en $P_1$ y $P_2$ para el caso en que el movimiento sea una rotación (para tener en cuenta el sentido de giro de cada uno de los motores).

Cualquier perturbación temporal de las velocidades en estado estacionario será suficientemente corregida por un controlador proporcional P. De todas formas, a fin de poder corregir perturbaciones continuas causadas por ejemplo por fuerzas de fricción diferentes

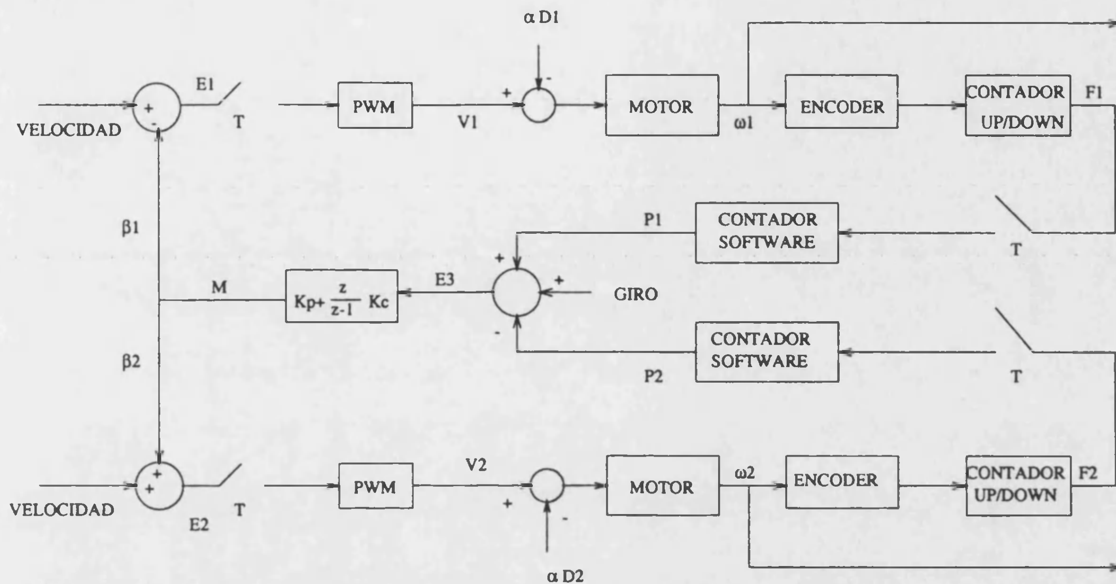



Figura 4.16: Bucle de control para el robot móvil

producidas por distribuciones de carga asimétricas, se añadirá un control integral I. El controlador PI diseñado no sólo iguala las velocidades de ambos motores, sino que además mantiene igual el número de pulsos desde el principio de cada movimiento, por lo tanto, este controlador garantiza un error de orientación nulo del vehículo en estado estacionario para cualquier perturbación contínua constante. Las únicas fuentes de perturbación que no se podrán subsanar serán las debidas a imprecisiones mecánicas o las debidas al derrape de alguna de las ruedas.

Las ecuaciones en diferencias para el controlador PI (suponiendo integración rectangular) son:

$$S(i) = S(i-1) + E(i) \tag{4.42}$$

$$M(i) = K_c S(i) + k_p E(i) \tag{4.43}$$

Donde S(i) almacena el error acumulado en la trayectoria desde el principio del movimiento, $K_c$ es la ganancia del integrador y $K_p$ es la ganancia proporcional. Los rangos de los parámetros $K_c$ y $K_p$ que garantizan la estabilidad se analizarán posteriormente.

El bucle de control para la plataforma móvil se representa en la figura 4.16, mientras que el diagrama de bloques del sistema de control completo se muestra en la figura 4.17.

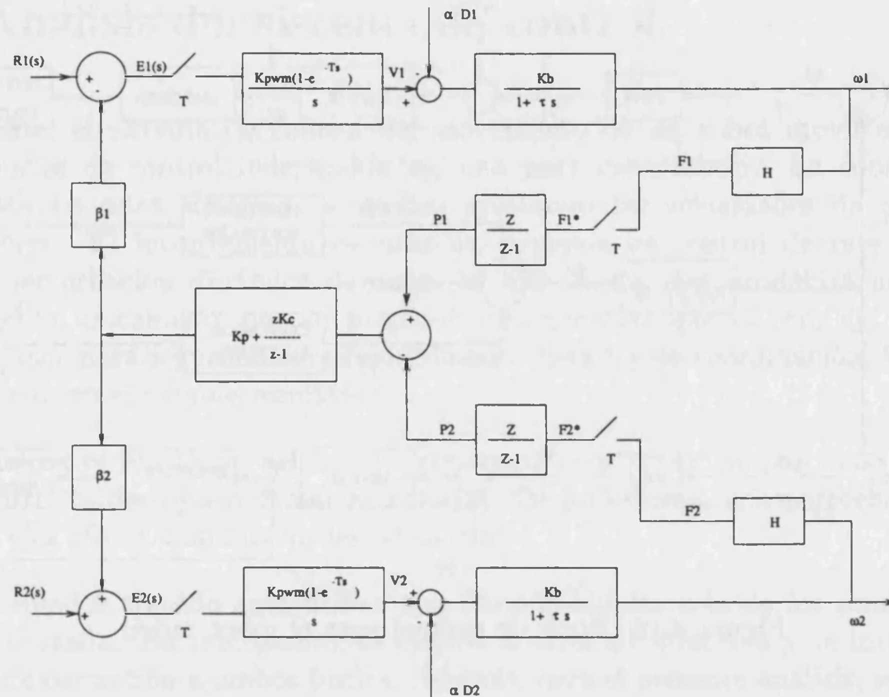



Figura 4.17: Diagrama de bloques del sistema de control completo

El motor se aproxima mediante una función de transferencia de primer orden, la cual en notación de transformada de Laplace (incluyendo el encoder) es:

$$F_i(s) = \frac{HK_b}{1+\tau s}(V_i - \alpha D_i) \qquad i = 1, 2 \tag{4.44}$$

El modulador por anchura de pulso es funcionalmente equivalente a un convertidor DAC en serie con un amplificador. La potencia suministrada a los motores es proporcional al código digital enviado a cada uno de ellos. Este código indica la relación temporal entre el estado alto y el estado bajo de la onda cuadrada que a una frecuencia fija se enviará al sistema de potencia.

El comportamiento teórico del modulador por anchura de pulso (figura 4.18) es tal que la potencia enviada a los motores es directamente proporcional al código digital enviado. En la práctica, existe una alinealidad en las proximidades del envio de potencia nula, de manera que para que el motor comience a girar, normalmente hay que enviar un código, que dependiendo (entre otros factores) del peso del robot y de los coeficientres de fricción entre transmisiones será mayor o menor. Esta alinealidad se puede subsanar por software, de manera que se debe evitar el envio de los códigos digitales que entran dentro del rango

de la zona no lineal.

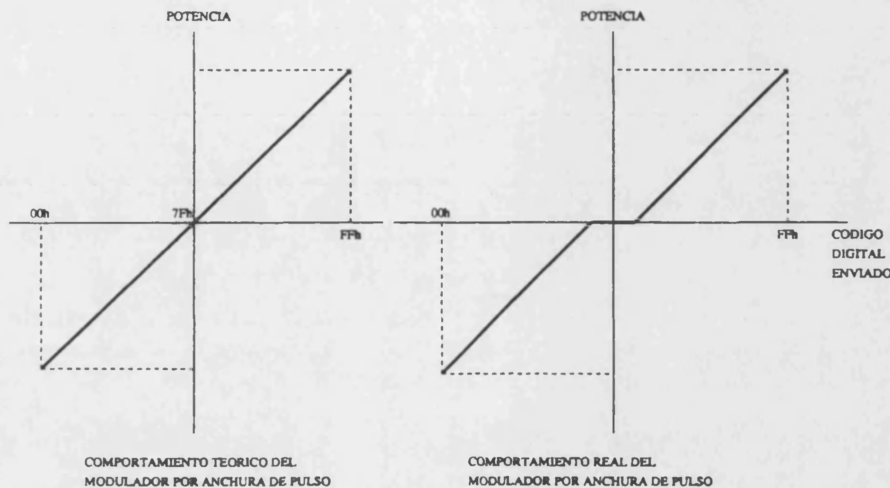



Figura 4.18: Comportamiento del modulador por anchura de pulso

En el modulador por anchura de pulso, la señal se mantiene constante durante el intervalo de muestreo T y por lo tanto, su función de transferencia es la de un retenedor de orden cero (ZOH):

$$\frac{V_i}{E_i}(s) = \frac{K_{pwm}(1 - e^{-sT})}{s} \qquad i = 1, 2 \tag{4.45}$$

Substituyendo la ecuación (4.45) en la (4.44) obtenemos:

$$F_i(s) = \frac{HK_{pwm}K_bE_i(s)(1 - e^{-sT})}{s(1 + \tau s)} - \frac{HK_b\alpha D_i}{1 + \tau s} \qquad i = 1, 2 \tag{4.46}$$

Realizando la transformada z de la ecuación (4.46) obtenemos:

$$\begin{aligned}
F_i(z) &= Z\left\{\frac{HK_{pwm}K_bE_i(s)(1 - e^{-sT})}{s(1 + \tau s)}\right\} - Z\left\{\frac{HK_b\alpha D_i(s)}{1 + \tau s}\right\} = \\
&= \frac{1}{\tau}HK_{pwm}K_bE_i(z)Z\left\{\frac{1 - e^{-sT}}{s(s + \frac{1}{\tau})}\right\} - \frac{1}{\tau}HK_b\alpha D_i(z)Z\left\{\frac{1}{s + \frac{1}{\tau}}\right\} = \\
&= \frac{HK_{pwm}K_bE_i(z)(z - 1)(\tau z)[z - e^{-\frac{T}{\tau}} - z + 1]}{(\tau z)(z - 1)(z - e^{-\frac{T}{\tau}})} - \frac{1}{\tau}HK_b\alpha D_i(z)\left(\frac{z}{z - e^{-\frac{T}{\tau}}}\right) =
\end{aligned}$$

$$= K\left(\frac{1-r}{z-r}\right)E_i(z) - HK_b\left(\frac{zw}{z-r}\right)\alpha D_i(z) \qquad i=1,2 \tag{4.47}$$

donde

- K es la ganancia en bucle abierto $K = K_{pwm}K_bH$.
- $r = e^{-\frac{T}{\tau}}$
- $w = \frac{1}{\tau}$

El esquema del algoritmo de control se puede representar con la ayuda de la transformada z como:

$$P_i(z) = \frac{z}{z-1}F_i(z) \qquad i=1,2 \tag{4.48}$$

$$E(z) = P_1(z) - P_2(z) = \frac{z}{z-1}\left[F_1(z) - F_2(z)\right] \tag{4.49}$$

$$M(z) = \left[K_p + K_c\frac{z}{z-1}\right]E(z) \tag{4.50}$$

$$E_1(z) = R_1(z) - \beta_1 M(z) \tag{4.51}$$

$$E_2(z) = R_2(z) + \beta_2 M(z) \tag{4.52}$$

donde $\beta_1 = 0$ y $\beta_2 = 1$ para M negativo, y $\beta_1 = 1$ y $\beta_2 = 0$ para M positivo. Substituyendo las ecuaciones (4.50), (4.51) y (4.52) en (4.47) obtenemos:

$$F_1(z) = K\left(\frac{1-r}{z-r}\right)\left[R_1(z) - \beta_1\left(K_p + K_c\frac{z}{z-1}\right)E(z)\right] - HK_b\left(\frac{zw}{z-r}\right)\alpha D_1(z) \tag{4.53}$$

$$F_2(z) = K\left(\frac{1-r}{z-r}\right)\left[R_2(z) + \beta_2\left(K_p + K_c\frac{z}{z-1}\right)E(z)\right] - HK_b\left(\frac{zw}{z-r}\right)\alpha D_2(z) \tag{4.54}$$

Para movimientos en linea recta se cumple $R_1 = R_2$ y la diferencia entre estas dos señales es:

$$\begin{aligned}
F_1(z) - F_2(z) &= K\left(\frac{1-r}{z-r}\right)\left[R_1(z) - \beta_1\left(K_p + K_c\frac{z}{z-1}\right)E(z)\right] - HK_b\left(\frac{zw}{z-r}\right)\alpha D_1(z) - \\
&- K\left(\frac{1-r}{z-r}\right)\left[R_2(z) + \beta_2\left(K_p + K_c\frac{z}{z-1}\right)E(z)\right] + HK_b\left(\frac{zw}{z-r}\right)\alpha D_2(z) = \\
&= K\left(\frac{1-r}{z-r}\right)\left[-\left(K_p + K_c\frac{z}{z-1}\right)E(z)\right] + HK_b\left(\frac{zw}{z-r}\right)[\alpha D_2(z) - \alpha D_1(z)] = \\
&= -K\left(\frac{1-r}{z-r}\right)\left(K_p + K_c\frac{z}{z-1}\right)E(z) + HK_b\left(\frac{zw}{z-r}\right)\triangle D(z) \qquad (4.55)
\end{aligned}$$

donde $\triangle D(z) = \alpha D_2(z) - \alpha D_1(z)$

Substituyendo la ecuación (4.55) en la (4.49) obtenemos la solución para E(z):

$$E(z) = \frac{z}{z-1}\left[-K\left(\frac{1-r}{z-r}\right)\left(K_p + K_c\frac{z}{z-1}\right)E(z) + HK_b\left(\frac{zw}{z-r}\right)\triangle D(z)\right] \qquad (4.56)$$

Así, la correspondiente función de transferencia es:

$$\begin{aligned}
\frac{E(z)}{\triangle D(z)} &= \frac{\left(\frac{z}{z-1}\right)HK_b\left(\frac{zw}{z-r}\right)}{1 + \frac{z}{z-1}\left[K\left(\frac{1-r}{z-r}\right)\left(K_p + K_c\frac{z}{z-1}\right)\right]} = \\
&= \frac{\frac{HK_b z^2 w(z-1)}{(z-1)^2(z-r)}}{1 + \frac{Kz(1-r)K_p}{(z-1)(z-r)} + \frac{Kz(1-r)K_c z}{(z-1)^2(z-r)}} = \\
&= \frac{z^2(z-1)HK_b w}{(z-1)^2(z-r) + z(z-1)KK_p(1-r) + z^2KK_c(1-r)} \qquad (4.57)
\end{aligned}$$

con la ecuación característica:

$$P(z) = z^3 + [K(1-r)(K_p + K_c) - (2+r)]z^2 + [1 + 2r - KK_p(1-r)]z - r \qquad (4.58)$$

la cual podemos definir de manera genérica como

$$P(z) = a_3z^3 + a_2z^2 + a_1z + a_0 \qquad (4.59)$$

donde

$$\begin{aligned}
a_3 &= 1 \\
a_2 &= K(1-r)(K_p+K_c)-(r+2) \\
a_1 &= 1+2r-KK_p(1-r) \\
a_0 &= -r
\end{aligned} \tag{4.60}$$

Según el criterio de estabilidad de Jury, para que un sistema sea estable deben cumplirse las siguientes cuatro condiciones:

- $P(z)\mid_{z=1}>0$

$$\begin{aligned}
0 &< [1+K(1-r)(K_p+K_c)-(2+r)+1+2r-KK_p(1-r)-r] \\
0 &< [KK_c(1-r)] \\
0 &< K_c
\end{aligned} \tag{4.61}$$

  Luego esta condición se cumple siempre que $K_c > 0$

- $P(z)\mid_{z=-1}<0$

$$\begin{aligned}
0 &> [K(1-r)(K_p+K_c)-r-2-1-1-2r+KK_p(1-r)-r] \\
[K(1-r)K_c] &< [4(r+1)-2KK_p(1-r)] \\
K_c &< \frac{4(r+1)}{K(1-r)}-2K_p
\end{aligned} \tag{4.62}$$

- $|a_0| < a_3$
  Esta condición siempre se satisface, ya que siempre se cumple en condiciones normales que $e^{\frac{T}{\tau}} > 1$ para $T, \tau > 0$.

- $|b_0| > |b_2|$
  donde

$$b_0 = \begin{vmatrix} a_0 & a_3 \\ a_3 & a_0 \end{vmatrix}$$

$$b_2 = \begin{vmatrix} a_0 & a_1 \\ a_3 & a_2 \end{vmatrix}$$

y así,

$$(1 - r^2) > | - (1 - r^2) + K(1 - r^2)K_p - K(1 - r)rK_c| \qquad (4.63)$$

La ecuación anterior contiene dos casos:

1. $b_2 < (1 - r^2)$
   lo cual da

$$K_c > K_p\left(\frac{1}{r} - 1\right) - \frac{2(1 - r^2)}{K(1 - r)r} \qquad (4.64)$$

2. $b_2 > -(1 - r^2)$
   lo cual da

$$K_c < K_p\left(\frac{1}{r} - 1\right) \qquad (4.65)$$

Las tres desigualdades dadas por (4.62),(4.64) y (4.65) definen una región en el plano $K_cK_p$ en la cual se garantiza la estabilidad del sistema. En la figura 4.19 se muestra el lugar geométrico de valores $K_p, K_c$ que generan una posible región de estabilidad.

A fin de poder demostrar que el error de orientación en estado estacionario $E'_{ss}$ debido a perturbaciones continuas es cero, si suponemos $\triangle D(z)$ como la suma de perturbaciones continuas desde el principio del movimiento ($\triangle D_0$), podemos poner

$$\triangle D(z) = \frac{z}{z - 1}\triangle D_0 \qquad (4.66)$$

y sustituyendo (4.66) en (4.57) obtenemos

$$E'(z) = \frac{z^2(z - 1)HK_bw\frac{z}{z-1}\triangle D_0}{(z - 1)^2(z - r) + z(z - 1)KK_p(1 - r) + z^2KK_c(1 - r)} \qquad (4.67)$$

y aplicando el teorema del valor final se puede ver que el error de orientación en estado estacionario es cero:

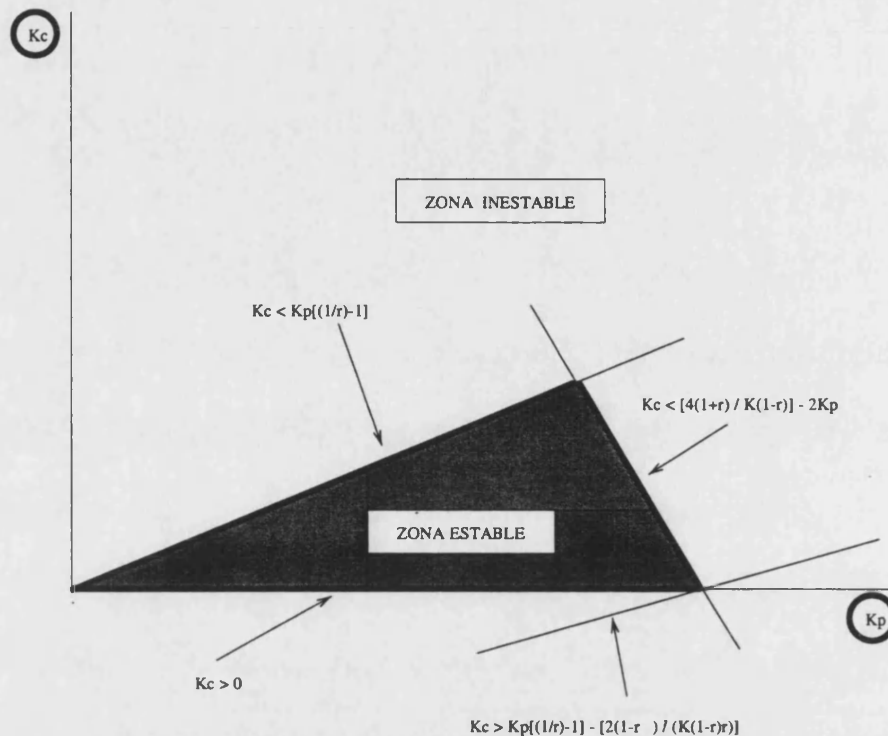



Figura 4.19: Rango de valores permitido para las ganancias $K_p$ y $K_c$

$$E'_{ss} = \lim_{z \to 1}(z-1)E'(z) = 0 \tag{4.68}$$

## 4.7 Mejoras en el sistema de control.

El esquema de control analizado anteriormente, garantiza un error de orientación nulo ante perturbaciones dinámicas de determinada amplitud en cualquiera de los bucles de control, pero no garantiza un funcionamiento apropiado en régimen permanente (básicamente un error de posición nulo). Para garantizar que la velocidad final del vehículo es la velocidad de referencia y que el error de orientación continúa siendo nulo, el esquema de control propuesto es el que se representa en la figura 4.20.

En este esquema, a la realimentación obtenida como diferencia de pulsos entre ambos bucles afectada por un regulador PI, se le añadirán la realimentación de referencia de velocidad de cada bucle y además un regulador PI en cada eje que garantice un error de posición nulo.

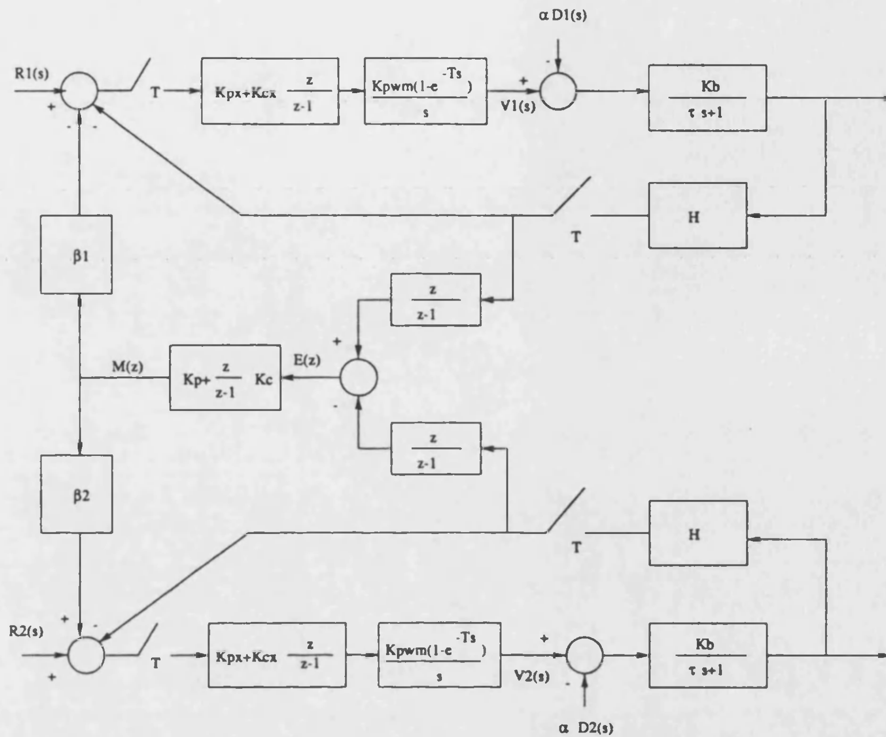



Figura 4.20: Nuevo esquema de control propuesto

El precio que hay que pagar por añadir un error de posición nulo en cada bucle frente a cualquier entrada de tipo escalón es muy alto, puesto que las expresiones resultantes para calcular el rango de valores que hacen estable al sistema son complejas; sin embargo, en la práctica, es relativamente fácil encontrar unos valores adecuados para la estabilidad de todo el sistema. A raiz de lo comentado anteriormente, es obvio despreciar la imposición de restricciones de tipo dinámico en cada uno de los bucles de control, puesto que entonces, determinar expresiones analíticas que nos describan el rango de valores que hacen estable al sistema se convierte una labor poco menos que imposible.

Con nuestro nuevo esquema de control, podemos seguir modelando el motor como un sistema de primer orden, cuya fdt incluyendo el encoder se puede poner en forma de transformada de Laplace como

$$F_i(s) = \frac{HK_b}{\tau s + 1}\left(V_i(s) - \alpha D_i(s)\right) \qquad i = 1, 2 \tag{4.69}$$

La fdt. formada por la relación entre la tensión entregada al motor y la señal de error vendrá ahora dada por

$$\begin{aligned}
\frac{V_i(s)}{E_i(s)} &= \left[\frac{K_{pwm}(1-e^{-Ts})}{s}\right]\left[K_{px}+\frac{z}{z-1}K_{cx}\right] = \\
&= \frac{K_{px}K_{pwm}(1-e^{-Ts})}{s}+\frac{K_{cx}K_{pwm}(1-e^{-Ts})}{s}\frac{z}{z-1} \qquad i=1,2 \quad (4.70)
\end{aligned}$$

y substituyendo la ecuación (4.70) en la (4.69), obtenemos

$$\begin{aligned}
F_i(s) &= \frac{HK_b}{\tau s+1}\left\{\left[\frac{K_{px}K_{pwm}(1-e^{-Ts})E_i(s)}{s}+\frac{K_{cx}K_{pwm}(1-e^{-Ts})E_i(s)}{s}\frac{z}{z-1}\right]-\alpha D_i(s)\right\} = \\
&= \frac{HK_bK_{px}K_{pwm}(1-e^{-Ts})E_i(s)}{s(\tau s+1)}+\frac{HK_bK_{cx}K_{pwm}(1-e^{-Ts})E_i(s)}{s(\tau s+1)}\frac{z}{z-1}-\frac{HK_b\alpha D_i(s)}{\tau s+1}
\end{aligned}$$

y realizando la transformada z de la ecuación anterior, obtenemos

$$\begin{aligned}
F_i(z) &= Z\left\{\frac{K_1(1-e^{-Ts})E_i(s)}{s(\tau s+1)}\right\}+Z\left\{\frac{K_2(1-e^{-Ts})E_i(s)}{s(\tau s+1)}\right\}\frac{z}{z-1}-Z\left\{\frac{HK_b\alpha D_i(s)}{\tau s+1}\right\} = \\
&= K_1E_i(z)(1-z^{-1})Z\left\{\frac{1}{s(\tau s+1)}\right\}+K_2E_i(z)Z\left\{\frac{1}{s(\tau s+1)}\right\}-HK_bZ\left\{\frac{1}{\tau s+1}\right\}\alpha D_i(z) = \\
&= \left[\frac{(1-r)(K_1+K_2)z-K_1(1-r)}{(z-1)(z-r)}\right]E_i(z)-HK_b\left(\frac{\omega z}{z-r}\right)\alpha D_i(z) \qquad (4.71)
\end{aligned}$$

donde

$$\begin{aligned}
\omega &= \frac{1}{\tau} \\
r &= e^{-\frac{T}{\tau}} \\
K_1 &= HK_bK_{px}K_{pwm} \\
K_2 &= HK_bK_{cx}K_{pwm}
\end{aligned}$$

Las ecuaciones para el algoritmo de control de orientación se implementan de la misma manera que en el esquema primitivo, pero ahora, las señales de error en ambos bucles de

control se ven afectadas por las respectivas realimentaciones de velocidad, de manera que tendremos

$$E_1(z) = R_1(z) - \beta_1 M(z) - F_1(z) \tag{4.72}$$

$$E_2(z) = R_2(z) + \beta_2 M(z) - F_2(z) \tag{4.73}$$

Realizando ahora las substituciones adecuadas de manera similar a como se realizaron en el primer esquema, obtenemos para $F_1(z)$ y $F_2(z)$ los siguientes valores

$$\begin{aligned} F_1(z) &= \frac{(1-r)(K_1+K_2)z - K_1(1-r)}{(z-1)(z-r)} \left[R_1(z) - \beta_1 \left(K_p + \frac{z}{z-1}K_c\right) E(z) - F_1(z)\right] - \\ &- HK_b\left(\frac{\omega z}{z-r}\right) \alpha D_1(z) \end{aligned} \tag{4.74}$$

$$\begin{aligned} F_2(z) &= \frac{(1-r)(K_1+K_2)z - K_1(1-r)}{(z-1)(z-r)} \left[R_2(z) + \beta_2 \left(K_p + \frac{z}{z-1}K_c\right) E(z) - F_2(z)\right] - \\ &- HK_b\left(\frac{\omega z}{z-r}\right) \alpha D_2(z) \end{aligned} \tag{4.75}$$

Desarrollando las dos ecuaciones anteriores, podemos poner

$$\begin{aligned} F_1(z) &= \frac{(1-r)(K_1+K_2)z - K_1(1-r)\left[R_1(z) - \beta_1\left(K_p + \frac{z}{z-1}K_c\right)E(z)\right]}{(z-1)(z-r) + (1-r)(K_1+K_2)z - K_1(1-r)} - \\ &- \frac{HK_b \omega z \alpha D_1(z)}{(z-r)\left[(z-1)(z-r) + (1-r)(K_1+K_2)z - K_1(1-r)\right]} \end{aligned} \tag{4.76}$$

$$\begin{aligned} F_2(z) &= \frac{(1-r)(K_1+K_2)z - K_1(1-r)\left[R_2(z) - \beta_2\left(K_p + \frac{z}{z-1}K_c\right)E(z)\right]}{(z-1)(z-r) + (1-r)(K_1+K_2)z - K_1(1-r)} - \\ &- \frac{HK_b \omega z \alpha D_2(z)}{(z-r)\left[(z-1)(z-r) + (1-r)(K_1+K_2)z - K_1(1-r)\right]} \end{aligned} \tag{4.77}$$

y la diferencia entre ellas, nos da como resultado

$$
\begin{aligned}
&F_1(z) - F_2(z) = \\
= &\left[\frac{(1-r)(K_1+K_2)z - K_1(1-r)}{(z-1)(z-r)+(1-r)(K_1+K_2)z - K_1(1-r)}\right]\left[\left(K_p + \frac{z}{z-1}K_c\right)E(z)\right](\beta_1+\beta_2) + \\
+ &\frac{HK_b\omega z\left[\alpha D_2(z) - \alpha D_1(z)\right]}{(z-r)\left[(z-1)(z-r)+(1-r)(K_1+K_2)z-K_1(1-r)\right]}
\end{aligned}
\quad .(4.78)
$$

Substituyendo la ecuación (4.78) en la (4.49), obtenemos

$$
\begin{aligned}
E(z) &= \left[\frac{z}{z-1}\right]\{[A]\,[B]\,E(z)\,[\beta_1+\beta_2] + [C\Delta D(z)]\} \\
&= \frac{z}{z-1}AB(\beta_1+\beta_2)E(z) + \frac{z}{z-1}C\Delta D(z)
\end{aligned}
\tag{4.79}
$$

con

$$
\begin{aligned}
A &= \frac{(1-r)(K_1+K_2)z - K_1(1-r)}{(z-1)(z-r)+(1-r)(K_1+K_2)z - K_1(1-r)} \\
B &= \left(K_p + \frac{z}{z-1}K_c\right) \\
C &= \frac{HK_b\omega z}{(z-r)\left[(z-1)(z-r)+(1-r)(K_1+K_2)z-K_1(1-r)\right]}
\end{aligned}
$$

de donde podemos obtener la relación entre la señal de error y la diferencia entre las perturbaciones dinámicas de ambos bucles de control como

$$
\frac{E(z)}{\Delta D(z)} = \frac{Cz}{\left[1 - AB(\beta_1+\beta_2)\right]z - 1}
\tag{4.80}
$$

Así, la ecuación característica viene dada por el denominador de la ecuación anterior, es decir

$$
P(z) = \left[1 - AB(\beta_1+\beta_2)\right]z - 1 = a_5z^5 + a_4z^4 + a_3z^3 + a_2z^2 + a_1z + a_0
\tag{4.81}
$$

donde (suponiendo $\beta_1 = \beta_2 = 0.5$):

$$\begin{aligned}
a_5 &= 1 \\
a_4 &= K_1(1-r) + K_2(1-r) - 2(r+1) + K_p(K_1 r - K_1) + K_c(K_1 r - K_1) \\
a_3 &= K_p(2K_1 - r^2 K_1 - K_1 r + K_2 - r^2 K_2) + K_c(rK_2 - K_2 r^2 + K_1 - K_1 r^2) + \\
&+ K_1(r^2 + 2r - 3) + K_2(r^2 - 2) + r^2 + 7r + 3 \\
a_2 &= K_p\left[K_1(2r^2 + r - 1) + K_2(r^2 - r)\right] + K_c\left[K_1(r^2 - r)\right] + K_1(-3r^2 + r + 2) + \\
&+ K_2(1 - r^2) - 3r^2 - 6r - 1 \\
a_1 &= K_p\left[K_1(r - r^2)\right] + K_1(r^2 - 1) - r^2 - 2r \\
a_0 &= rK_1(1-r)
\end{aligned}$$

Podemos componer por grupos los distintos bloques que forman las igualdades anteriores de manera que podemos poner

$$\begin{aligned}
P(z) &= z^5 + \left[X_1(K_p + K_c) + X_2\right] z^4 + \left[X_3 K_p + X_4 K_c + X_5\right] z^3 + \\
&+ \left[X_6 K_p + X_7 K_c + X_8\right] z^2 + \left[X_9 K_p + X_{10}\right] z + X_9
\end{aligned} \tag{4.82}$$

donde

$$\begin{aligned}
X_1 &= K_1(r-1) \\
X_2 &= (K_1 + K_2)(1-r) \\
X_3 &= -r^2(K_1 + K_2) + K_1(2-r) + K_2 \\
X_4 &= -r^2(K_1 + K_2) + K_2 r + K_1 \\
X_5 &= r^2(K_1 + K_2 + 1) + r(2K_1 + 7) - 3K_1 - 2K_2 + 3 \\
X_6 &= r^2(2K_1 + K_2) + r(K_1 - K_2) + K_1 \\
X_7 &= K_1(r^2 - r) \\
X_8 &= -r^2(3K_1 + K_2 + 3) + r(K_1 - 6) + 2K_1 + K_2 - 1 \\
X_9 &= K_1(r - r^2) \\
X_{10} &= r^2(K_1 - 1) - 2r - K_1
\end{aligned}$$

Según el criterio de estabilidad de Jury, para que el sistema sea estable, deben cumplirse las siguientes condiciones

- $P(z)\mid_{z=1} > 0$

$$\begin{aligned} 0 &< a_5 + a_4 + a_3 + a_2 + a_1 + a_0 \\ 0 &< K_p(X_1 + X_3 + X_6 + X_9) + K_c(X_1 + X_4 + X_7) + (X_2 + X_8 + X_9 + X_{10} + 1) \end{aligned}$$

de donde obtenemos la ecuación de una recta frontera de las que definen la región de estabilidad del sistema

$$K_c < \left(\frac{1 + X_2 + X_5 + X_8 + X_9 + X_{10}}{X_1 + X_4 + X_7} + K_p\frac{X_1 + X_3 + X_6 + X_9}{X_1 + X_4 + X_7}\right) \tag{4.83}$$

- $P(z)\mid_{z=-1} < 0$

$$\begin{aligned} 0 &> a_1 - a_2 + a_3 - a_4 + a_5 - a_0 \\ 0 &> X_1K_p + X_1K_c + X_2 - X_3K_p - X_4K_c - X_5 + X_6K_p + X_7K_c + \\ &+ X_8 + X_9 - X_9K_p - X_{10} - 1 \end{aligned}$$

y de la misma manera que en el punto anterior, obtenemos la ecuación de otra recta frontera

$$K_c < \left(\frac{1 + X_5 - X_8 - X_9 + X_{10}}{X_1 - X_4 + X_7} + K_p\frac{X_3 - X_6 + X_9 - X_1}{X_1 - X_4 + X_7}\right) \tag{4.84}$$

- $|a_0| < a_5$

$$\begin{aligned} 1 &> |K_1(r - r^2)| \qquad 0 \leq r \leq 1 \\ |K_1| &< \frac{1}{r - r^2} \end{aligned}$$

de donde obtenemos restricciones en el periodo de muestreo T, puesto que el parámetro (r) depende de este.

$$HK_bK_{px}K_{pwm} < \frac{1}{r - r^2} \tag{4.85}$$

y por último

- $|b_4| > b_0$

  lo cual, desarrollando, nos da como resultado

$$|X_9^2 - 1| > |X_9(X_1 - 1)K_p + X_1 X_9 K_c + X_9(X_2 - 1)| \tag{4.86}$$

- $|c_3| > c_0$

  de lo cual, podemos obtener

$$|b_4^2 - b_0^2| > |(a_0 a_3 - a_2 a_5)(a_0^2 - a_5^2) - (a_0 a_4 - a_1 a_5)(a_0 a_1 - a_4 a_5)| \tag{4.87}$$

- $|d_2| > d_0$

  de donde

$$|c_3^2 - c_0^2| > |(b_4^2 - b_0^2)(b_2 b_4 - b_0 b_2) - (b_1 b_4 - b_0 b_3)(b_3 b_4 - b_0 b_1)| \tag{4.88}$$

De esta forma podemos obtener otras tantas ecuaciones de rectas frontera para los valores de los coeficientes del regulador, que garantizan la estabilidad frente a perturbaciones dinámicas, garantizando un error de posición nulo.

Para demostrar que con el nuevo esquema de control, el error de orientación en estado estacionario $E'_{ss}$ debido a perturbaciones continuas también es nulo, substituyendo en la ecuación (4.80) $\triangle D(z)$ por $\frac{z}{z-1}\triangle D_0$ tenemos que

$$E'(z) = \frac{C\frac{z^2}{z-1}\triangle D_0}{[1 - AB(\beta_1 + \beta_2)]\, z - 1} \tag{4.89}$$

y aplicando ahora el teorema del valor final, tenemos finalmente

$$E'_{ss} = \lim_{z \to 1}(z-1)E'(z) = 0 \tag{4.90}$$

Por último, podemos observar el comportamiento del esquema de control con referencias cruzadas frente a distintos valores para los coeficientes de los reguladores tanto de los bucles de realimentación para cada motor, como para el lazo de realimentación de referencia cruzada. En particular, en la figura 4.21 se presenta la evolución de la señal de error por referencia cruzada para valores de los coeficientes de los reguladores que satisfacen todas las ecuaciones que garantizan la estabilidad, mientras que en la figura 4.22 se muestra la evolución de la señal de error, para unos valores de los coeficientes de

los reguladores que no garantizan alguna o varias de las ecuaciones que definen la región de estabilidad.

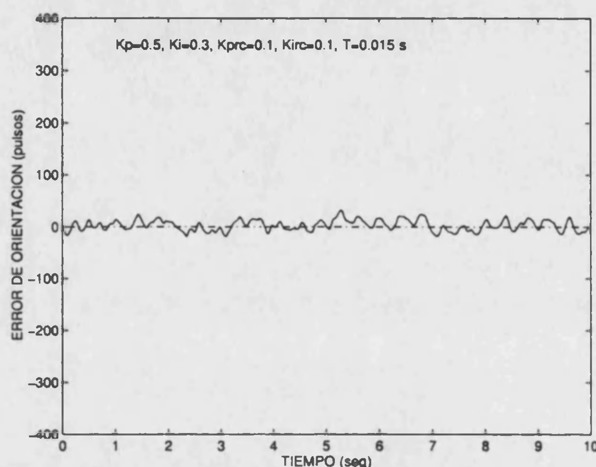



Figura 4.21: Evolución de la señal de error con un funcionamiento estable del esquema de control

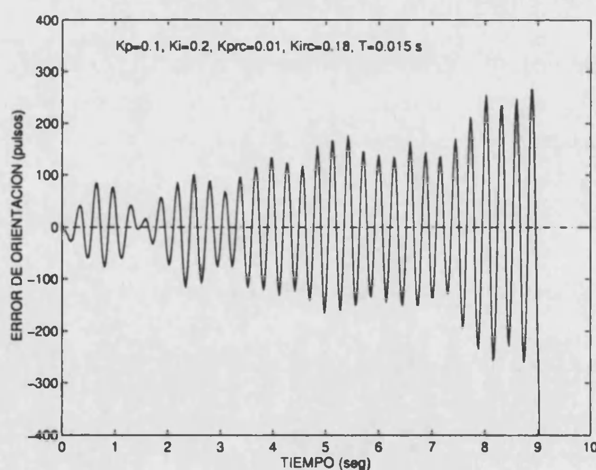



Figura 4.22: Evolución de la señal de error con un funcionamiento inestable del esquema de control

Como conclusión, se puede observar que el esquema de control propuesto representa una mejora respecto al esquema anterior por garantizar no sólo un error de orientación nulo frente a perturbaciones estáticas o dinámicas, sino por garantizar un error de posición nulo frente a entradas de tipo escalón de velocidad.

# Capítulo 5

# El pseudo-planificador. Una herramienta para la comparación de arquitecturas de robots móviles.

## 5.1 Arquitecturas de control como secuencias de procesos.

En esta sección se explicará cuáles son las ideas básicas que proponemos acerca de la secuenciación de procesos en un robot móvil. De acuerdo con uno de los propósitos primordiales que anima esta tesis, que es el de comprobar algunas ideas sobre arquitecturas para el control de robots móviles, era necesario ante todo diseñar un esquema suficientemente flexible en el que diversas (idealmente, cualquier) arquitectura de control pudiera ser probada sin variar sustancialmente, no ya el hardware, sino ni siquiera la configuración esencial del software. Mostraremos seguidamente cómo el esquema que proponemos, un secuenciador de procesos elementales de tipo ejecutor cíclico pero con repartos variables, puede satisfacer nuestros requerimientos.

Durante los últimos años se ha propuesto un considerable número de arquitecturas de control para aplicaciones robóticas, especialmente en robots móviles [Bro89], [MSH89], [HV97], [Nil94]; menos contribuciones aparecen en otros campos de la robótica como el ensamblaje [Mal91]. Estas arquitecturas son aparentemente completamente distintas, pero todas ellas comparten una característica común: están formadas por bloques elementales que actúan cuando son llamados (o les es permitido) por otros bloques, o cuando eventos externos, detectados por el sistema sensorial los activan.

Cuando cualquiera de estas arquitecturas se implementa en un sistema con un único

procesador, las diferencias entre ellas siempre pueden ser descritas en términos del orden en la activación de los procesos elementales, del tiempo de procesador asignado a cada uno y de las prioridades entre ellos. La necesidad de un sistema operativo para coordinar tanto la ejecución como la asignación de recursos al sistema y la atención a los eventos externos es universalmente conocida y algunos sistemas operativos han sido implementados con este fin, siendo TCA probablemente el más relevante [Sim94]. Este sistema, implementado por Simmons, se ha usado en varias aplicaciones, entre otras Xavier, el robot móvil de Carnegie Mellon. Está construido como una herramienta para combinar fácilmente comportamientos tanto deliberativos como reactivos. Su funcionamiento esencial consiste en disponer de un control central que recibe mensajes de los diferentes módulos y los envía a otros; algunos de estos módulos actúan físicamente, mientras que otros son visualizadores que informan cuando algunas condiciones, o bien relacionadas con el sistema sensorial o bien relacionadas con mensajes llegados al sistema de control central son satisfechas. Los módulos pueden ser activados o suprimidos, pero es tarea del sistema operativo asignar físicamente recursos para ellos.

TCA y otros son efectivamente operacionales, pero no comienzan analizando el problema de identificar los requerimientos que debe poseer el núcleo de un sistema que pueda trabajar sobre un robot móvil. Algunos de ellos son:

- El núcleo debe ser de tiempo real, es decir, deberá garantizar que cualquier petición será atendida en un tiempo acotado (y para algún tipo de petición, un tiempo pequeño). El procesamiento de las lecturas de los sensores de ultrasonidos para guiar el robot es el caso más evidente de su utilización, pero actividades de nivel superior como por ejemplo la generación de planes tampoco pueden ser retrasadas indefinidamente por falta de recursos.
- El robot debe poder recuperar cualquier actividad interrumpida por peticiones más urgentes justo en el punto en el que fue cortado. Esto requiere un mecanismo fiable para almacenar y recuperar contextos.
- Las diferentes tareas deben poder comunicar sus resultados o peticiones a otras tareas. Si se desea que el núcleo sea lo suficientemente general, no se debe poner restricciones sobre qué procesos pueden comunicarse con qué otros.
- Puesto que el mal funcionamiento del robot puede causarle serios daños no sólo a él, sino también al entorno, se debe proporcionar cierto grado de detección y tolerancia a fallos. Lo más simple es la implementación de un proceso *watch-dog* que periódicamente compruebe las condiciones críticas.
- El pseudo-planificador debe ser lo suficientemente reducido como para poder ser ejecutado sobre los computadores que suelen montarse sobre los robots móviles a veces no excesivamente potentes, aunque esto es cada vez más raro debido al incremento en la potencia de cálculo de los actuales microordenadores.
- El pseudo-planificador debe permitir una distribución variable de recursos, particu-

larmente el más valorado: el tiempo de procesador. Además, como plantearemos posteriormente, esa distribución se debe poder variar para cambiar de acuerdo a decisiones tomadas por el propio robot. Sólo de esta forma el cambio en las condiciones del entorno se puede tener en cuenta para una óptima (o al menos buena) asignación de recursos.

El pseudo-planificador propuesto cumple en buena medida todos los requerimientos presentados anteriormente y se propone como una herramienta para la realización de tests sobre las distintas arquitecturas existentes. El objetivo principal es mostrar cómo estas pueden ser implementadas por simples cambios de ciertos parámetros (en particular, el reparto entre procesos) lo cual caracteriza de hecho la arquitectura en concreto.

## 5.2 Descripción del mecanismo de pseudo-planificación.

Uno de los algoritmos usuales para la asignación de tiempo de CPU a diferentes procesos por parte del kernel es el de *round-robin*. Se basa en la definición de un intervalo de tiempo (quantum) el cual es el máximo tiempo permitido para la ejecución de cualquier proceso. Si un proceso determinado no ha acabado su ejecución en el tiempo permitido (un quantum), es eliminado por el núcleo y el siguiente proceso en espera toma el control de la CPU. Después de que todos los procesos en espera han consumido su quantum, el primero vuelve a tomar el control justo en el punto en el que fue interrumpido.

En el caso de los robots móviles, la mayoría de los procesos no llegan, se ejecutan y acaban, sino que por el contrario, están ejecutándose continuamente desde la conexión hasta la desconexión del sistema; son procesos especiales que intentan mantener al robot activo: avanzar, evitar obstáculos, etc. Además, estos procesos están sustentados por otros de más bajo nivel encargados de la lectura de los sensores, del filtrado de ruido, del control de los motores de tracción, etc. En este contexto, parece mejor pensar en un anillo de procesos que se ejecutan cíclicamente de manera que la política de *round-robin* se convierte símplemente en un mecanismo expulsor que asigna igual tiempo de procesador a cada uno de los procesos. Una definición más apropiada para el quantum de tiempo sería la siguiente: definimos un quantum como el tiempo necesario para ejecutar un ciclo completo de todos los procesos que pueden ejecutarse. En base a esto, podemos definir el reparto asignado a un proceso como la fracción del quantum (lo que equivale al tiempo de CPU efectivo) asignado a ese proceso. Podemos observar que este tipo de núcleo continúa manteniendo la característica de tiempo real, puesto que se puede considerar que cada uno de los N procesos se está ejecutando en un único procesador N veces más lento, de manera que el tiempo de respuesta a un determinado evento crecerá por un factor de N,

pero continuará estando acotado.

La política simple de round-robin cumple las condiciones propuestas anteriormente, excepto en lo referente a la asignación variable de recursos. Además, puede presentar problemas debido al excesivo tiempo de respuesta para algunos procesos, además de no ser conveniente para la ejecución de procesos intrínsecamente no cíclicos. Así, el pseudo-planificador propuesto utiliza una variante de este algoritmo que permite a todos o a algunos de los procesos el cambio de los repartos de forma dinámica durante su ejecución. Se basa en la existencia de un proceso supervisor el cual se activa periódicamente. Este proceso mantiene una tabla con el reparto asignado a cada proceso que puede ejecutarse, además de una cuenta de los pulsos de reloj transcurridos desde la última llamada, un registro del proceso que actualmente está en ejecución y un flag indicando si la tabla de repartos ha sido cambiada. El algoritmo es como sigue:

- Sea P el proceso actualmente en ejecución.
- Mediante un mecanismo de tiempo real se llama al proceso supervisor S, el cual:
  - Salva el contexto del proceso P ($P_c$).
  - Si P ha modificado la tabla de repartos (flag de repartos marcado) entonces :
    - Recalcular los repartos de los procesos.
    - Inicializar el flag de repartos a no marcado.
  - Calcular el tiempo de ejecución que le queda al proceso P ($T_{pr}$).
  - Si $T_{pr} \leq 0$ entonces:
    - Buscar el siguiente proceso Q cuyo reparto sea distinto de 0.
    - Restaurar el contexto de Q ($Q_c$).
  - sino
    - Restaurar el contexto de P ($P_c$).
  - Dormir al proceso supervisor.

De esta manera, las diferencias entre arquitecturas pueden ser formuladas en términos de orden de activación de procesos, prioridades y distribución de recursos. El orden de activación de los procesos puede ser alterado, puesto que éstos (todos, o al menos los permitidos) pueden acceder a la tabla general de repartos.

En lo que respecta a la distribución de recursos, se debe hacer un comentario interesante: el tiempo de CPU se asigna mediante el mecanismo de repartos; es lógico asignar a priori repartos mayores a los procesos considerados más importantes o que necesitan más tiempo. A pesar de esto, un algoritmo se puede programar como un nuevo proceso que cambie sus decisiones de acuerdo a la información obtenida durante su ejecución. Esto es importante en este contexto, puesto que uno de los puntos clave en el control de robots

móviles actual es la implementación de mecanismos para la focalización de la atención. Bajo este modelo, dichos mecanismos pueden ser simulados como uno o más procesos de tipo estímulo-respuesta íntimamente ligados al sistema sensorial, que incrementan el reparto de los procesos de acuerdo a la información suministrada por estos.

En lo que respecta al sistema de prioridades, éstas se pueden implementar también utilizando repartos. Si un proceso, por alguna razón de peso necesita todo el tiempo de CPU, siempre puede modificar la tabla de repartos asignándose a él mismo un reparto del 100% y un reparto del 0% al resto de procesos. Después de la finalización, puede o bien recalcular los repartos de todos ellos o volver a instaurar los valores que poseían anteriormente. Esto tiene un problema obvio: el proceso que dispone de todo el tiempo puede quedar bloqueado, con lo cual (y como mínimo) se destruiría la característica de tiempo real. Para evitar esto, el proceso supervisor implementa un mecanismo para suspender al proceso voraz en caso de que éste haya estado ejecutándose él sólo durante más de un número predeterminado de ciclos.

La comunicación entre procesos se hace actualmente a través de variables globales. El mecanismo de manejo de mensajes todavía no está implementado. La utilización de mensajes simplificaría la programación cuando el pseudo-planificador se utilizara para implementar una arquitectura de tipo jerárquico o vertical, pero su utilidad resultaría escasa en arquitecturas de tipo comportamental en las cuales el mundo se utiliza como su propio modelo, de manera que los mensajes se manifiestan como cambios en el mundo, lo cual deriva en diferentes lecturas de los sensores para otros procesos.

## 5.3 El pseudo-planificador y su implantación.

Veremos a continuación cómo las ideas básicas que propusimos en la sección anterior pueden implantarse sobre un robot móvil dotado de un sólo procesador. Ante todo, hemos de prestar especial atención a los requisitos que deberá cumplir el sistema operativo a bordo del robot de modo que las ideas antes expuestas se puedan realizar. En primer lugar, debería por supuesto permitir fácilmente la creación y destrucción de procesos, así como su activación o suspensión a voluntad. Además, debería poder acceder al hardware con relativa facilidad, al menos a los ports de usuario, y debería ser capaz de realizar la conmutación entre procesos de modo eficiente, simple, y con una granularidad controlable. El sistema operativo de uso público Linux cumple satisfactoriamente estos requisitos, según se explicará a continuación, y por ello se decidió escogerlo como base de la implantación. Otras alternativas comerciales, como QNX, fueron consideradas, pero la total accesibilidad al código fuente que ofrece Linux, así como la profundidad y extensión con que se encuentra documentado [Tro] fueron causa determinante de la elección.

El primer problema que se nos presenta es que, naturalmente, un mecanismo de secuenciación de procesos como el descrito en la sección anterior requiere una conmutación entre procesos (incluyendo, naturalmente, el cambio de contexto) suficientemente rápida y controlable. A ser posible, no deseabamos programar esta capacidad, sino aprovechar el propio secuenciador (*planificador*) del sistema operativo. El algoritmo de secuenciación programado en Linux que afecta a los procesos usuales no nos resultaba apropiado, puesto que no contempla la posibilidad de activar explícitamente alguno de ellos, sino que decide esto internamente en base a la prioridad del proceso, que es cambiada de acuerdo al tiempo que éste lleva esperando su turno. Afortunadamente, existe no obstante en Linux la posibilidad de modificar el tipo de un proceso: además del tipo usual, existen los procesos de tipo `ROUND_ROBIN`, y de tipo `SCHED_FIFO`. En cualquiera de estos dos casos, es el usuario el que fija la prioridad cuando crea el proceso, y el que debe cambiarla cuando lo estime oportuno, pues no hay algoritmo interno en el núcleo del sistema que lo haga. De entre los dos citados, son los `SCHED_FIFO` los que nos han servido para nuestro propósito. Los procesos declarados como del tipo `SCHED_FIFO` tienen una prioridad siempre mayor que los procesos usuales, y por tanto, si alguno de ellos está preparado para ejecutarse, y ningún otro proceso de tipo `SCHED_FIFO` con mayor prioridad lo está, éste se ejecuta. En el caso de existir dos procesos `SCHED_FIFO` con la misma prioridad, es el primero que se inició el que se ejecuta (de ahí el nombre de FIFO), y no será expulsado del procesador hasta que no haya terminado, devuelva voluntariamente el control definitiva o momentáneamente (pasando al estado de dormido), o bien otro proceso `SCHED_FIFO` con mayor prioridad esté listo. El hecho de que los procesos usuales, correspondientes a la actividad normal del sistema, no puedan correr presenta una importante ventaja: como quiera que cualquier proceso usual puede invocar a llamadas al sistema, que no pueden ser interrumpidas más que por otra llamada, y cuya duración en principio no podemos determinar, el hecho de dejarlos correr destruiría la característica de tiempo real. Como ejemplo, entre dos lecturas de la señal de los codificadores ópticos de las ruedas, el sistema podría decidir volcar alguna parte de la memoria RAM al disco duro (hacer *swap*) alterando así de modo sustancial el periodo de muestreo. No obstante, esto puede suponer, por supuesto, algún inconveniente: no dejar al núcleo del sistema que realize tareas de mantenimiento durante un tiempo prolongado puede ser fatal si realmente lo necesitaba. Por ello, el usuario deberá ser cuidadoso y no provocar un uso excesivo de la memoria, ni tampoco usar dispositivos externos que se controlen a través de drivers, como la red, o el disco duro. Dado el tipo de programas que necesitamos, esto no ha sido un problema para nosotros, pero en la sección de mejoras futuras se presentará alguna propuesta de solución.

La idea básica que va a permitir implantar el esquema de secuenciación que requerimos consiste en el uso exclusivo de procesos `SCHED_FIFO`, con la posibilidad de adoptar (en principio) sólo tres prioridades distintas, que llamaremos BAJA, MEDIA, y ALTA. Uno de los procesos creados será especial: lo llamaremos pseudo-planificador, porque conceptualmente corre inmediatamente por encima del secuenciador (*planificador*) del núcleo

del sistema, y ejecuta las funciones de decisión de la conmutación que éste no toma sobre procesos no usuales. No obstante, no ejecuta la propia conmutación, que sí es dejada al núcleo. A los procesos que el usuario escribe para realizar efectivamente las tareas requeridas los llamaremos procesos útiles.

El pseudo-planificador será el único proceso con prioridad ALTA. La mayor parte del tiempo estará dormido, con lo cual alguno de los procesos útiles, al que se habrá asignado la prioridad MEDIA, estará ejecutándose. Pero cuando el tiempo que en su momento se decidió asignar al proceso útil concluya, el pseudo-planificador se despertará, llevará la prioridad del proceso útil en curso a BAJA, y subirá a ALTA la del siguiente proceso útil cuyo turno haya llegado. Después, volverá a dormir. El algoritmo es:

El proceso padre:

- Registra todos los procesos útiles como procesos FIFO de prioridad BAJA. Todos ellos están preparados para bloquearse en su inicio, a la espera de una señal.
- Registra al pseudo-planificador como proceso FIFO de prioridad ALTA y le cede el control.

El pseudo-planificador:

- Inicializa proceso_actual al primer proceso.
- Manda a todos los procesos útiles la señal de desbloqueo. (No obstante, éstos aún no corren, puesto que su prioridad es menor que la del pseudo-planificador, ahora en ejecución.)
- Hace
  - Marca la prioridad de proceso_actual como BAJA.
  - Asigna a proceso_actual el siguiente proceso útil.
  - Marca la prioridad de proceso_actual como MEDIA.
  - Abandona el procesador (pasando al estado de dormido) durante el tiempo que deba asignarse a proceso_actual.
- Hasta la desconexión

No hay una conmutación explícita entre procesos; como dijimos, el propio núcleo del sistema la realizará cuando, al dormirse el pseudo-planificador, observe que cierto proceso (el llamado en el algoritmo proceso_actual) debe correr; y además, nótese que es el único que en este momento puede hacerlo. Por ello, es obvio que la granularidad del núcleo de Linux (intervalo de tiempo entre dos invocaciones sucesivas del auténtico

planificador) es la que marca igualmente la granularidad de nuestro esquema. En el núclo original de Linux (al menos, en la versión 2.0.32) fue fijada en 2 ms. Esto es excesivo para nuestros propósitos, especialmente por lo que respecta al periodo de muestreo de los codificadores y de activación de los sensores de ultrasonidos, y decidimos reducirla a unos 150 $\mu$s.[1] Con los ordenadores modernos, en particular el Pentium a 133 MHz que usamos, esto no supone ninguna carga excesiva, y de hecho, el núcleo usa para la conmutación entre procesos menos del 1 % del tiempo total del procesador. Otro pequeño problema afecta al hecho de que la llamada al sistema empleada por el pseudo-planificador para dormirse (*nanosleep)* debe despertarle con la suficiente precisión. Esto también requirió una pequeña modificación del código del núcleo de Linux. [2]

Hasta aquí hemos descrito el esquema básico. Ahora detallaremos las mejoras y refinamientos que nos conferirán una total flexibilidad. En primer lugar, no todos los procesos útiles que deseamos ejecutar son iguales. Algunos son totalmente imprescindibles, y periódicamente deben ejecutarse completos. Entre ellos están el algoritmo de control de los motores, incluyendo la lectura de los codificadores ópticos incrementales y el envío de la señal de potencia, que debe ejecutarse cada periodo de muestreo. Otros, como la activación de los sensores de ultrasonidos, pueden ser periódicos o no, a elección del usuario. Existen también procesos que pueden funcionar asícronamente, como la generación de mapas, y tareas similares de más alto nivel, a los que habría que dejar el máximo tiempo posible de proceso, pero sin que impidan nunca la ejecución a su debido tiempo de los procesos periódicos. Como indicamos en la sección anterior, este tiempo disponible vendrá controlado por el parámetro de reparto. Todos los procesos útiles que se usen deberán registrarse para ser conocidos por el pseudo-planificador, y este les dará control de modo secuencial, en el orden en que fueron registrados. De acuerdo al tiempo de procesador que les es asignado, dividiremos los procesos en:

- Procesos **sin reparto**: Al llegar su turno, el pseudo-planificador les cede el control. No son interrumpidos hasta que no concluyen. Obviamente, deben ser breves, y el usuario debe asegurarse de que no puedan quedar nunca bloqueados. Aquí se incluyen los procesos periódicos antes citados.
- Procesos **con reparto**: Deben escribirse como bucles infinitos. Cada vez que se les cede el control, corren durante el tiempo que el pseudo-planificador les permite, el cual es proporcional al valor de su parámetro de reparto en el momento en que son despertados. Al concluir su tiempo, el pseudo-planificador les expulsa del procesador; la siguiente vez que cualquiera de estos procesos vuelva a ser invocado, se reanudará en el punto exacto en que quedó interrumpido.

---

[1]Para hacer esto basta cambiar el valor de la constante HZ en el fichero `/usr/src/linux/include/asm-i386/params.h` y recompilar el núcleo.

[2]concretamente, el código de nanosleep en /usr/src/linux/kernel/sched.c

Como ya se comentó en la sección anterior, una de las claves de funcionamiento de nuestro esquema consiste en la posibilidad de variar el reparto dinámicamente. De hecho, la tabla de repartos es públicamente accesible, y cualquier proceso en cualquier momento puede variar tanto su propio reparto como el de cualquier otro. Sería conveniente establecer algún mecanismo que permitiera, si el usuario lo desea, una ordenación jerárquica de los procesos para condecer o denegar el permiso para los cambios de reparto. En la versión actual del software no se ha implantado, pero se deja como futura mejora. Lo único que en este momento existe es un indicador que permite señalar un proceso como de reparto no modificable.

Además de ello, se ha implantado aún una posibilidad más para los procesos con reparto: dijimos que debían funcionar como un bucle infinito, pero con objeto de poder sincronizarlos con otros procesos, podemos desear que cada vez que se les permite usar el procesador se ejecute, como máximo, una vuelta de su bucle. Esto es útil en procesos complejos, o cuya lógica dependa de las condiciones actuales, para los que es imposible predecir una duración. A éstos los llamaremos procesos de disparo (*one-shot processes*). En este tipo, si el proceso no ha concluído una vuelta de bucle cuando el pseudo-planificador le expulsa, se reanudará en su siguiente llamada por en punto en que se interrumpió. Sin embargo, si ha concluido una vuelta de bucle en su tiempo asignado, queda bloqueado esperando su expulsión. Para implantar esta última posibilidad fue necesaria la introducción de un proceso 'nulo' (*idle*) con prioridad estática comprendida entre la BAJA y la MEDIA que gastase el tiempo de procesador ejecutando un bucle infinito vacío, cuya única finalidad es impedir que algún proceso usual pudiera adquirir el control, llamar al sistema, y alterar el periodo de ciclo del pseudo-planificador mientras el *one-shot process* está bloqueado.

La programación concreta en Linux de todo esto ha sido posible gracias al uso de la excelente biblioteca de hilos (*threads*) incluida en las distribuciones [3] que permite la creación y destrucción de procesos ligeros que comparten las variables de su antecesor, lo que hace que la comunicación entre ellos por medio de variables globales sea muy simple. Pero además ofrece un mecanismo de señales y semáforos que hemos usado en los dos casos en los que es necesario bloquear procesos: el instante inicial de todos los procesos útiles, que esperan a que el pseudo-planificador les de permiso para empezar por primera vez, y el bloqueo que sufren los procesos de tipo *one-shot* al acabar una vuelta de su bucle.

En importante indicar que únicamente los procesos con reparto son implantados como *threads*. Los procesos sin reparto son simples procedimientos, que el pseudo-planificador llama cuando llega su turno. Uno de ellos comprueba que cierta condición sobre los sensores (en concreto, la activación de cierto sensor de contacto) no se haya producido. En caso afirmativo, devuelve un código de salida que interrumpe el ciclo del pseudo-

[3]Esta biblioteca fue programada por Xavier Leroy, email: Xavier.Leroy@inria.fr

planificador, realiza un cierre apropiado del hardware (corte de corriente, etc.), aniquila a los procesos útiles, y por fin concluye él mismo, devolviendo el control al núcleo usual de Linux.

La codificación concreta en C puede verse en el apéndice ??, documentada usando la herramienta CXRef. Sólo se comentarán aquí las porciones de código realmente nucleares: la estructura de datos usada para contener la información sobre un proceso, el bucle principal del pseudo-planificador y la función de cambio de repartos.

Estructura de datos para un proceso

```
/*+ See the meaning of these constants in the proc_info structure +*/
\\#define NO_SHARING     -1
\\#define MODIFIABLE      1
\\#define NOT_MODIFIABLE 0
\\#define MAX_SH_PROC    16

/*+ This structure contains the information about each process, except +*/
/*+ the pseudo-scheduler, that does not need it.                        +*/
typedef struct
 {
  pthread_t pth;              /*+ The identifier of the process as a thread +*/
  int nproc;                  /*+ The number of threads. +*/
  unsigned long allowed_time; /*+ For how long the process runs in each +*/
                              /*+ scheduler turn, in ns +*/
  float sharing[2];           /*+ The proportion of the total time that this  +*/
                              /*+ thread can spend. sharing[0] is the current +*/
                              /*+ one, whereas sharing[1] contains that which +*/
                              /*+ another thread wants for it in the next turn+*/
  int modifiable;             /*+ Flag to state if the sharing of the process +*/
                              /*+ process can be modified by others. Three    +*/
                              /*+ values are allowed:                         +*/
              /*+ NO_SHARING: this process must spend all the +*/
                              /*+                time it needs, and this cannot  +*/
    /*+            be altered +*/
           /*+ MODIFIABLE: Other threads can change this    +*/
                              /*+                process' sharing                +*/
           /*+ NOT_MODIFIABLE: Other threads can't do it.  +*/
 } proc_info;
```

El bucle principal del pseudo-planificador

```
void pseudo_sched(proc_info pr[MAX_PROC],int nproc,void *pass)
{
 int current;
 void *end_loop=(void *)0;
 struct timespec tsp;

 current=0;      /* Let's start in the first process (any other would do, too) */

 while (end_loop==(void *)0)
 {
```

```
  /* If the process that has to run now has sharing, let's sleep. */
  /* The process 'current', that has medium priority, will run */
  /* Otherwise, simply call the function. */

  if (pr[current].modifiable==NO_SHARING)
   end_loop=(fun[current])(pass);
  else
  {
   set_pr(pr[current].pth,MEDIUM_PR);
   nanosleep(&tsp,NULL);
   /* When we awake, the process (and everyone else) cannot run. */
   /* Then, change its priority to low, and state that it is not to be run. */
   set_pr(pr[current].pth,LOW_PR);
  }

  /* Determine which process comes next... */
  current=(current+1)%nproc;

  /* At the end of each turn, (before the next one begins) we could change */
  /* the sharings */

  if (current==0)
   touch_sharings(pr,nproc);

  ONE_SHOT_MAY_GO_ON;
 }

 /* Close_all, stored in the last position of the function array, */
 /* is called to stop robot and other cleaning tasks              */
 fun[MAX_PROC-1](pass);

 return;
}
```

Función para el cambio de los repartos

```
int touch_sharings(proc_info pr[MAX_PROC],int nproc)
{
 int i,j;
 float total_sha=0.0;

 i=0;
 while (i<nproc)
 {
  if (pr[i].modifiable!=NO_SHARING)
   total_sha+=pr[i].sharing[DESIRED_SH];
  i++;
 }

 if (total_sha>1.0)
 {
  /* We have an idle from which getting time... */
  if (fun[nproc-1]==my_idle)
  {
   total_sha-=pr[nproc-1].sharing[DESIRED_SH];
   /* If getting time from idle is enough... */
   if (total_sha<1.0)
    pr[nproc-1].sharing[DESIRED_SH]=pr[nproc-1].sharing[CURRENT_SH]=1.0-total_sha;
   /* if not, process sharings will have to be renormalized */
```

```
  else
  {
   for (i=0;i<nproc-1;i++)
   {
    pr[i].sharing[DESIRED_SH]/=total_sha;
    pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
   }
   /* and idle has no time for it */
   pr[nproc-1].sharing[DESIRED_SH]=pr[nproc-1].sharing[CURRENT_SH]=0.0;
  }
 }
 else
 {
  /* If we haven't idle: renormalize */
  for (i=0;i<nproc;i++)
  {
   pr[i].sharing[DESIRED_SH]/=total_sha;
   pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
  }
 }
}
else  /* we don't run out of time... */
 if (total_sha<1.0)  /* ...even may be we have too much... */
 {
  /* If we have an idle to which assign the remaining time, do it */
  if (fun[nproc-1]==my_idle)
   pr[nproc-1].sharing[DESIRED_SH]=pr[nproc-1].sharing[CURRENT_SH]=1.0-total_sha;
  /* but if we haven't idle,renormalize */
  else
   for (i=0;i<nproc;i++)
   {
    pr[i].sharing[DESIRED_SH]/=total_sha;
    pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
   }
 }

/* Then, buffer with the sharings to be sent is updated     */
/* and execution time for each process is appropriately set */

i=0;
j=1;
while (i<nproc)
{
 if (pr[i].modifiable==MODIFIABLE)
 {
  pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
  pr[i].allowed_time=pr[i].sharing[CURRENT_SH]*PS_SCHED_PERIOD;
  Sh_buff[j]=(unsigned char)((MAX_SH-2)*pr[i].sharing[CURRENT_SH]/total_sha);
  j++;
 }
 i++;
}
Sh_buff[j]=NO_MORE_SHARINGS;
return(0);
}
```

Por último y como ejemplo, en la figura 5.1 se puede observar un ejemplo del funcionamiento del pseudo-planificador en un intervalo de tiempo de una ejecución cualquiera. En este ejemplo se supone que los procesos P1, P2 y P3 son procesos sin reparto, es decir,

se ejecutan por completo e independientemente de su tiempo de ejecución al comienzo de cada ciclo (se supone que son procesos normalmente con un tiempo muy corto de ejecución), los procesos P4 y P5 son procesos con reparto y de tipo *one-shot*, es decir, tienen un tiempo asignado de ejecución; si en ese tiempo son capaces de ejecutarse una vez, lo harán y se quedarán esperando que acabe su tiempo asignado (un proceso especial se encarga de eso). En caso de no tener tiempo de ejecutarse una vez, al final de su tiempo concedido, son eliminados del procesador y la siguiente vez se restaurarán desde el punto en que se quedaron. Por último, los procesos P6 y P7 se supone que son procesos con reparto que se ejecutan como un bucle infinito durante todo el tiempo concedido. En el siguiente ciclo se restaurarán desde el punto en que se quedaron. Además, por cuestiones de simplificación, se supone que todos los procesos con reparto pueden modificar los repartos del resto, que a todos los procesos con reparto se les puede modificar el suyo y que los procesos encargados simplemente de consumir tiempo no se representan.

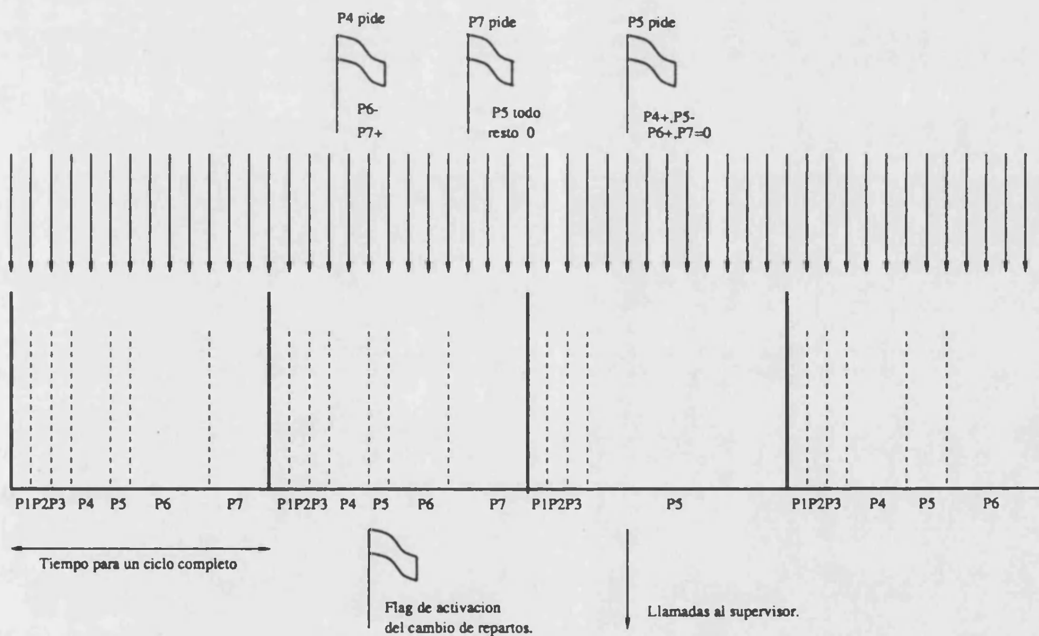
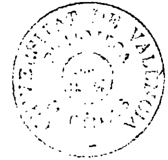



Figura 5.1: Posible evolución en los repartos de los procesos con el tiempo.



# Capítulo 6

# Implementación de diversas arquitecturas de control de robots móviles usando el pseudo-planificador.

En este capítulo se trata de mostrar cómo diversas arquitecturas de robots móviles pueden implementarse con nuestra herramienta de pseudoplanificación. En particular, se trata de llevar a la práctica el mecanismo mediante el cual una serie de tareas (procesos) programados para realizar un determinado fin o presentar un determinado comportamiento e introducidos como funciones en lenguaje C junto con sus prioridades y repartos asociados, pueden generar, dependiendo de la interrelación entre ellos en función del tiempo, una determinada arquitectura conocida. De esta misma manera, pensamos que es posible encontrar relaciones nuevas entre las tareas que pudieran mejorar el comportamiento del sistema al menos en determinados casos particulares (nueva arquitectura?).

Para comprobar el funcionamiento de nuestro pseudo-planificador, hemos elegido tres arquitecturas diferentes: un algoritmo de navegación simple basada en casos, la arquitectura de subsumción de Brooks y la arquitectura teleo-reactiva de Nilsson.

## 6.1 Navegación basada en casos

La navegación basada en casos es una aproximación eminentemente secuencial (arquitectura jerárquica) y válida normalmente sólo para entornos relativamente simples. Mediante esta aproximación, se intenta identificar una situación a partir de la interpretación de las

señales de los sensores y entonces se ejecuta la acción apropiada hasta su finalización. Por ejemplo, para realizar una determinada tarea, se pueden seguir un conjunto de reglas del estilo de: {**Si** Detectada pared a la izquierda **entonces** girar 15$^o$ a la derecha.}

En este caso, el procedimiento de interpretación de los sensores se implementa como un proceso (TEST), y cada acción elemental (AVANZAR, GIRAR, ...) como otros procesos $(A_1, A_2, ...A_n)$. El reparto inicial debe ser 100% para el proceso de interpretación sensorial y 0% para cada uno de los $A_i$ para $i = 1..n$. Al final del proceso de TEST, éste ha decidido qué acción se debe elegir para su subsiguiente ejecución, a la cual le asigna un reparto del 100% y del 0% al resto (incluido él). A continuación se ejecutará la acción asociada seleccionada, la cual, antes de acabar, volverá a asignar un reparto del 100% al proceso de TEST y del 0% al resto (de nuevo él incluido).

La acción seleccionada para implementar una arquitectura de tipo jerárquico como lo es la arquitectura basada en casos sobre nuestro robot RODNEY es la de avanzar a lo largo de un pasillo evitando chocar con las paredes rotando hacia el lado contrario al que se detecta una pared próxima con incrementos de 5$^o$ y cuando detecte una puerta abierta a su izquierda que la atraviese. El robot debe parar cuando detecte un obstáculo frontal a una distancia menor o igual a su distancia de seguridad (que es de 50 cm en la actualidad).

Para este ejemplo se han definido 4 procesos (TEST, ROTAR, AVANZAR y MOVER). El proceso de TEST (al cual debe haberse asignado inicialmente un reparto del 100%) se encarga de decidir en qué situación se encuentra el robot dentro de la misión programada. Las posibles situaciones programadas con sus acciones asociadas son:

- El sensor frontal detecta obstáculo a una distancia inferior a la de umbral de seguridad.
  - Parar el robot.
- El sensor 6 (lateral izquierdo) detecta obstáculo a una distancia inferior a la de umbral de seguridad.
  - Rotar 5$^o$ en sentido horario.
- El sensor 2 (lateral derecho) detecta obstáculo a una distancia inferior a la de umbral de seguridad.
  - Rotar 5$^o$ en sentido antihorario.
- El sensor 6 mide una distancia mayor que una predeterminada.
  - Detectada puerta abierta a la izquierda (*Detectada_puerta*=SI).
  - Avanzar una distancia de 30 cm.
- Detectada puerta abierta a la izquierda.
  - Activar Fase_1 (*Fase_1*=SI).
  - Rotar 90$^o$ en sentido antihorario.

- Fase_1 activa.
  - Avanzar una distancia de 100 cm.
- En cualquier otro caso.
  - Avanzar hacia adelante a velocidad constante.

Y las funciones que implementan dicho proceso son las siguientes:

```
void *TEST(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   Codigo_display=10;

   if ((lectura_sonars[0]<=LIMITE_INF) && (lectura_sonars[0]!=0))
     {
       Parar();
       Avanzando=NO;

       pr[8].sharing[DESIRED_SH]=1.0;      /* Test    */
       pr[9].sharing[DESIRED_SH]=0.0;      /* Rotar   */
       pr[10].sharing[DESIRED_SH]=0.0;      /* Avanzar */
       pr[11].sharing[DESIRED_SH]=0.0;     /* Mover */
     }

   else if ((lectura_sonars[NUM_SONARS-2]<=LIMITE_INF) &&
            (lectura_sonars[NUM_SONARS-2]!=0) && Detectada_puerta==NO)
     {
       Hay_que_rotar=SI;
       Avanzando=NO;
       Comando=GIRO_DER;
       outb(Comando,SENTIDO_GIRO);
       Angulo_a_rotar=5.0;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }

   else if ((lectura_sonars[2]<LIMITE_INF) &&
            (lectura_sonars[2]!=0) && Detectada_puerta==NO)
     {
       Hay_que_rotar=SI;
       Avanzando=NO;
       Comando=GIRO_IZQ;
       outb(Comando,SENTIDO_GIRO);
       Angulo_a_rotar=5.0;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
```

```
else if (Detectada_puerta)
  {
    Fase_1=SI;
    Detectada_puerta=NO;
    Hay_que_rotar=SI;
    Avanzando=NO;
    Comando=GIRO_IZQ;
    outb(Comando,SENTIDO_GIRO);
    Angulo_a_rotar=90.0;

    pr[8].sharing[DESIRED_SH]=0.0;
    pr[9].sharing[DESIRED_SH]=1.0;
    pr[10].sharing[DESIRED_SH]=0.0;
    pr[11].sharing[DESIRED_SH]=0.0;
  }

else if (Fase_1)
  {
    Fase_1=NO;
    Hay_que_mover=SI;
    Avanzando=NO;
    Comando=ADELANTE;
    outb(Comando,SENTIDO_GIRO);
    VELOC_IZQ=50;
    VELOC_DER=50;
    Distancia=1000;

    pr[8].sharing[DESIRED_SH]=0.0;
    pr[9].sharing[DESIRED_SH]=0.0;
    pr[10].sharing[DESIRED_SH]=0.0;
    pr[11].sharing[DESIRED_SH]=1.0;
  }

else if (lectura_sonars[NUM_SONARS-2]>(LIMITE_INF+UMBRAL2))
  {
    Hay_que_mover=SI;
    Detectada_puerta=SI;
    Avanzando=NO;
    Comando=ADELANTE;
    outb(Comando,SENTIDO_GIRO);
    VELOC_IZQ=50;
    VELOC_DER=50;
    Distancia=300;

    pr[8].sharing[DESIRED_SH]=0.0;
    pr[9].sharing[DESIRED_SH]=0.0;
    pr[10].sharing[DESIRED_SH]=0.0;
    pr[11].sharing[DESIRED_SH]=1.0;
  }
else
  {
    Comando=ADELANTE;
    outb(Comando,SENTIDO_GIRO);
    VELOC_IZQ=100;
    VELOC_DER=100;

    pr[8].sharing[DESIRED_SH]=0.0;
    pr[9].sharing[DESIRED_SH]=0.0;
    pr[10].sharing[DESIRED_SH]=1.0;
    pr[11].sharing[DESIRED_SH]=0.0;
  }
```

```
  ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}
```

El proceso ROTAR se encarga de realizar la rotación del robot un determinado ángulo y en el sentido seleccionado si la variable *Hay_que_rotar* está activa. En caso contrario se cede el control al proceso TEST. El código para este proceso es el siguiente:

```
void *ROTAR(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Hay_que_rotar)
     {
       Codigo_display=20;

       C_rot();

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
   else
     {
       pr[8].sharing[DESIRED_SH]=1.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}
```

El proceso AVANZAR es el encargado de guiar el robot en línea recta a la velocidad y sentido prefijados. Su código asociado es el siguiente:

```
void *AVANZAR(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   Codigo_display=30;

   Avanzando=SI;

   Comando=ADELANTE;
   outb(Comando,SENTIDO_GIRO);

   C_vel();
```

```
   pr[8].sharing[DESIRED_SH]=1.0;
   pr[9].sharing[DESIRED_SH]=0.0;
   pr[10].sharing[DESIRED_SH]=0.0;
   pr[11].sharing[DESIRED_SH]=0.0;

   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}
```

Por último, el proceso MOVER es el encargado de hacer avanzar el robot en el sentido y a la velocidad prefijados la distancia establecida, siempre y cuando la variable *Hay_que_mover* esté activa. En caso contrario se para el movimiento del robot y se cede el control al proceso TEST. Su código asociado es:

```
void *MOVER(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Hay_que_mover)
     {
       Codigo_display=40;

       Comando=ADELANTE;
       outb(Comando,SENTIDO_GIRO);

       C_pos_vel();

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=1.0;
     }
   else
     {
       Parar();

       pr[8].sharing[DESIRED_SH]=1.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}
```

Las gráficas obtenidas para los 4 procesos de reparto variable a lo largo de toda la maniobra se representan en las siguientes figuras en las que por motivos de claridad se han separado por intervalos temporales de 500 ciclos del pseudo-planificador. En ellas se puede observar que no existe solapamiento entre procesos a lo largo del tiempo puesto que

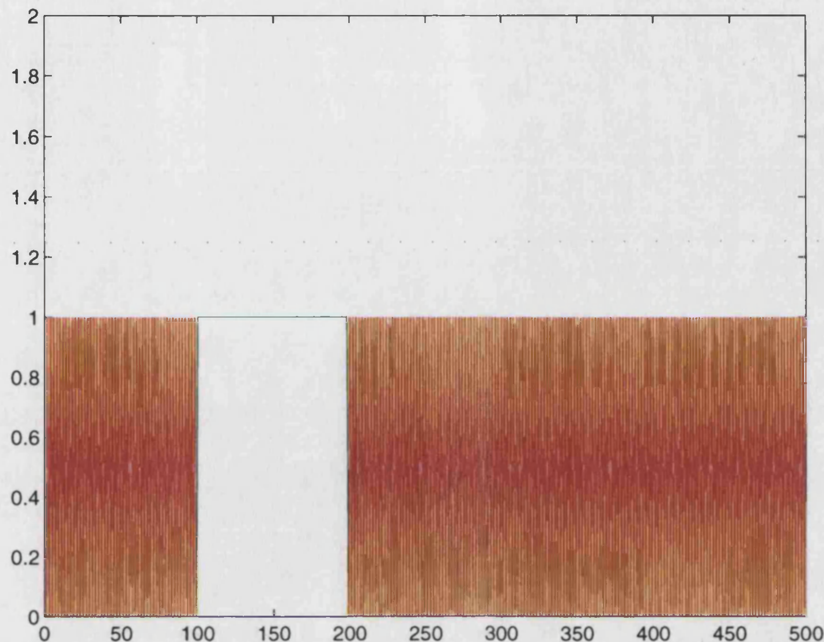



Figura 6.1: Evolución de los repartos de los procesos en funcion del tiempo. Fase 1

en todo momento uno de ellos y sólo uno, posee todo el tiempo de procesador. La relación entre colores en las gráficas y proceso asociado son: TEST:amarillo, ROTAR:verde, AVANZAR:rojo y MOVER:azul. Las fases de las gráficas se corresponden con cada uno de los intervalos de 500 ciclos desde el inicio hasta el fin de la maniobra.

En una segunda prueba realizando la misma maniobra, los resultados obtenidos en cuanto al porcentaje de tiempo de reparto respecto del tiempo total de un ciclo se puede ver en las siguientes gráficas, en las que en cada una de ellas se representa el reparto en cada vuelta del pseudo-planificador para cada uno de los procesos de reparto variable y donde la relación entre colores y procesos es la misma que en las gráficas para la maniobra anterior. Evidentemente, el solapamiento de todas ellas resultaría en una gráfica similar a las obtenidas anteriormente.

## 6.2 Arquitectura de Subsumción

Según la descripción original de esta arquitectura dada por Brooks en [Bro85] y posteriormente en [Bro89], en vez de resolver el problema basándose en una descomposición de la tarea en subtareas secuenciales para obtener la solución, se descompone el problema pro-

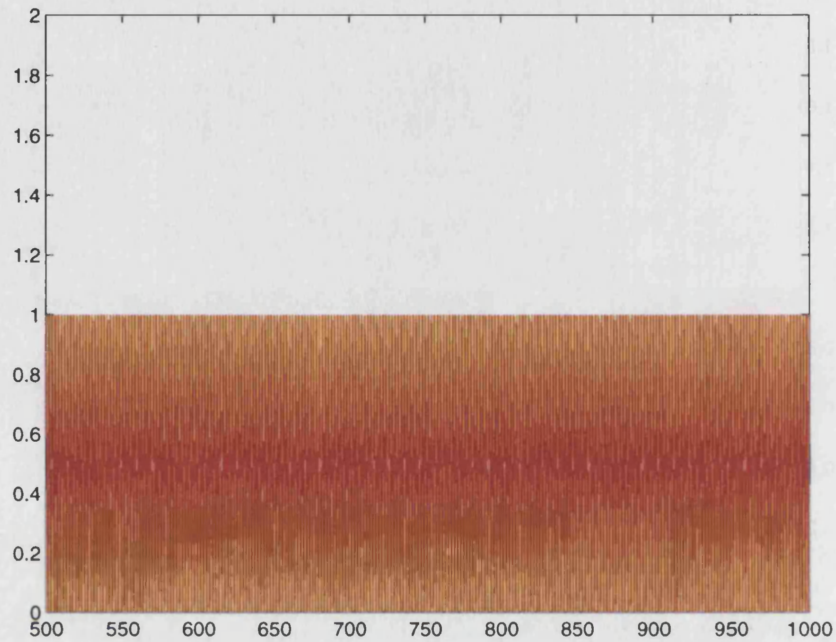



Figura 6.2: Evolución de los repartos de los procesos en funcion del tiempo. Fase 2

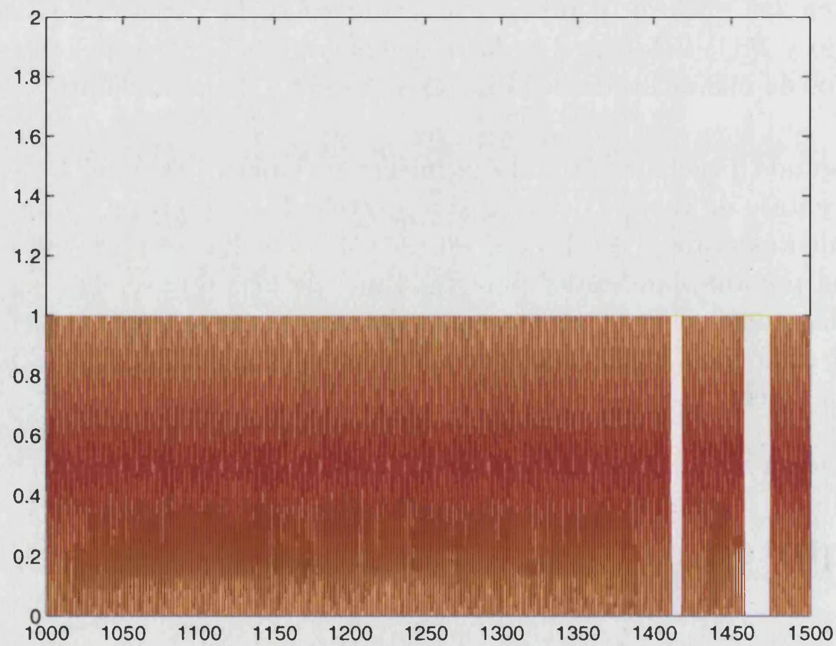



Figura 6.3: Evolución de los repartos de los procesos en funcion del tiempo. Fase 3

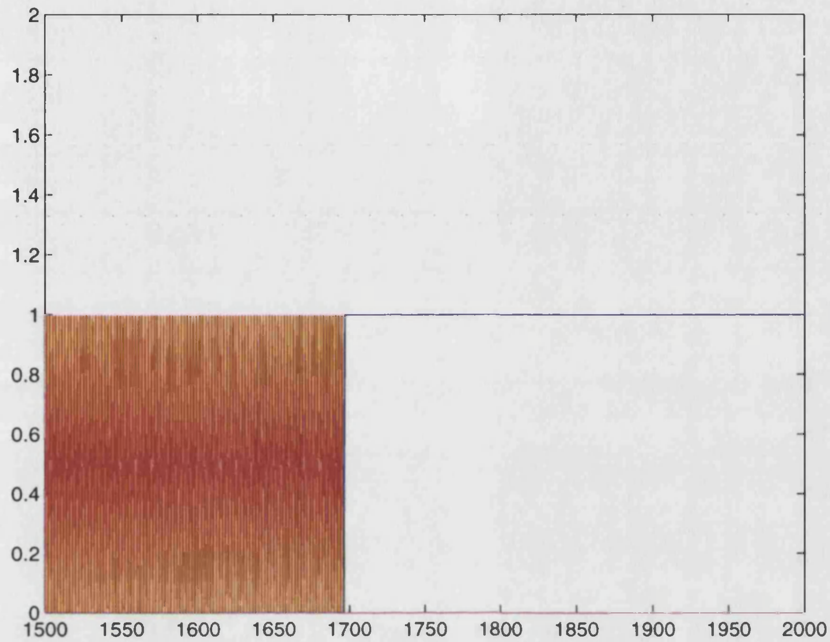



Figura 6.4: Evolución de los repartos de los procesos en funcion del tiempo. Fase 4

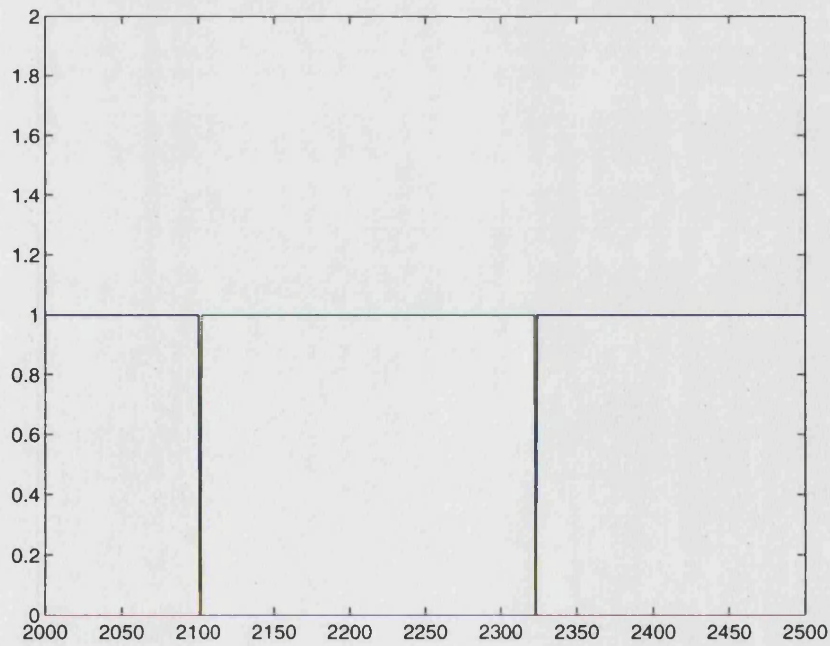



Figura 6.5: Evolución de los repartos de los procesos en funcion del tiempo. Fase 5

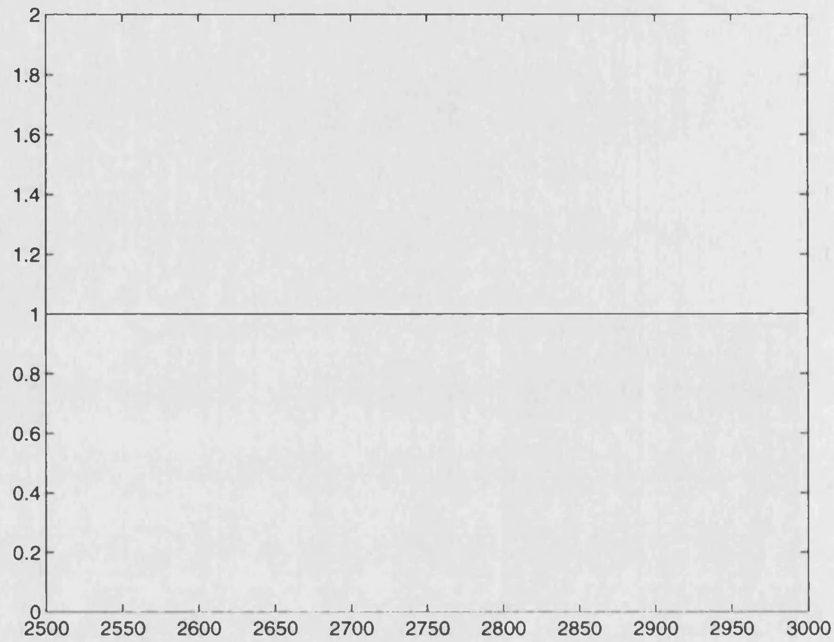



Figura 6.6: Evolución de los repartos de los procesos en funcion del tiempo. Fase 6

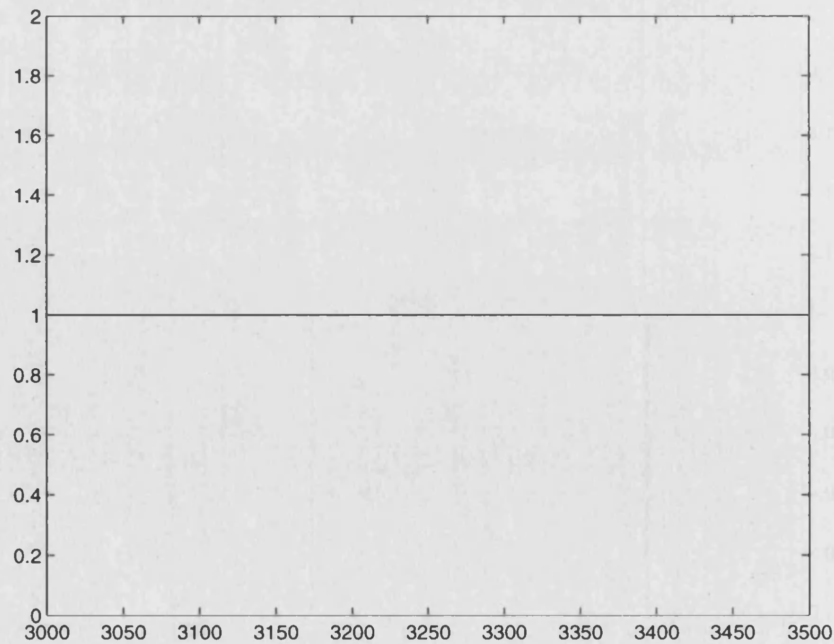



Figura 6.7: Evolución de los repartos de los procesos en funcion del tiempo. Fase 7

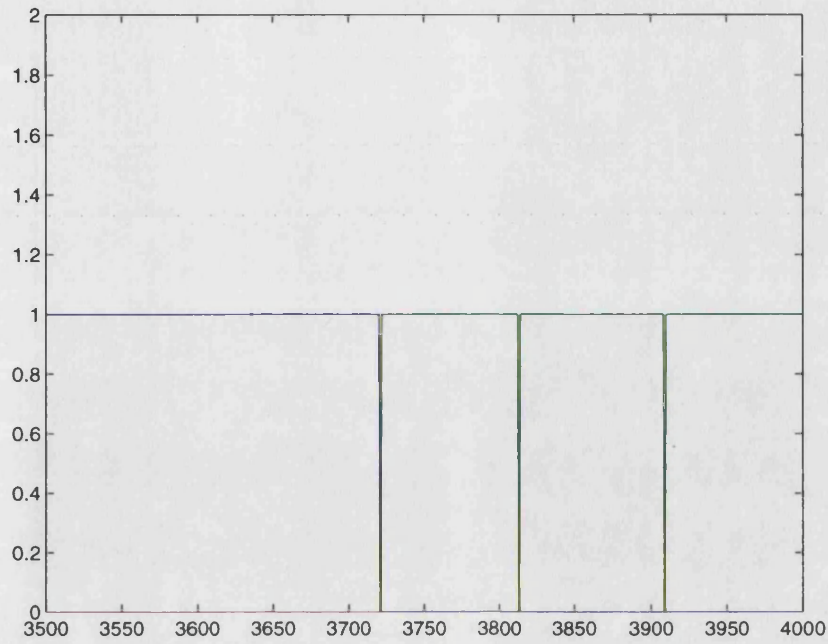



Figura 6.8: Evolución de los repartos de los procesos en funcion del tiempo. Fase 8

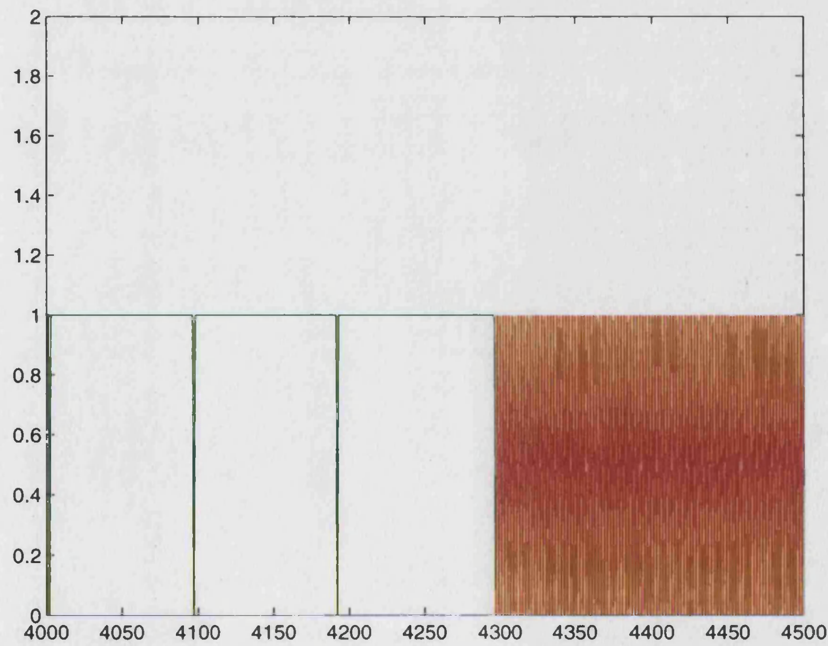



Figura 6.9: Evolución de los repartos de los procesos en funcion del tiempo. Fase 9

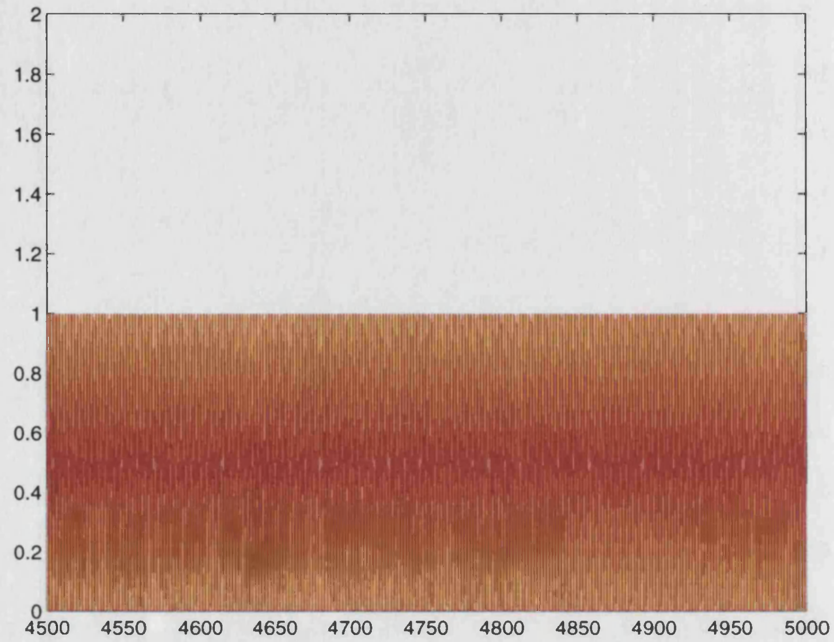



Figura 6.10: Evolución de los repartos de los procesos en funcion del tiempo. Fase 10

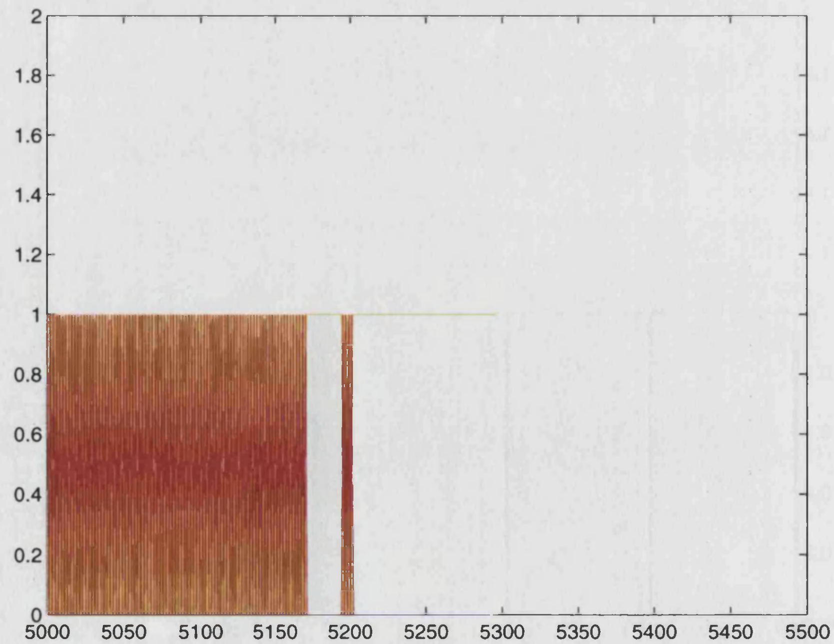



Figura 6.11: Evolución de los repartos de los procesos en funcion del tiempo. Fase 11

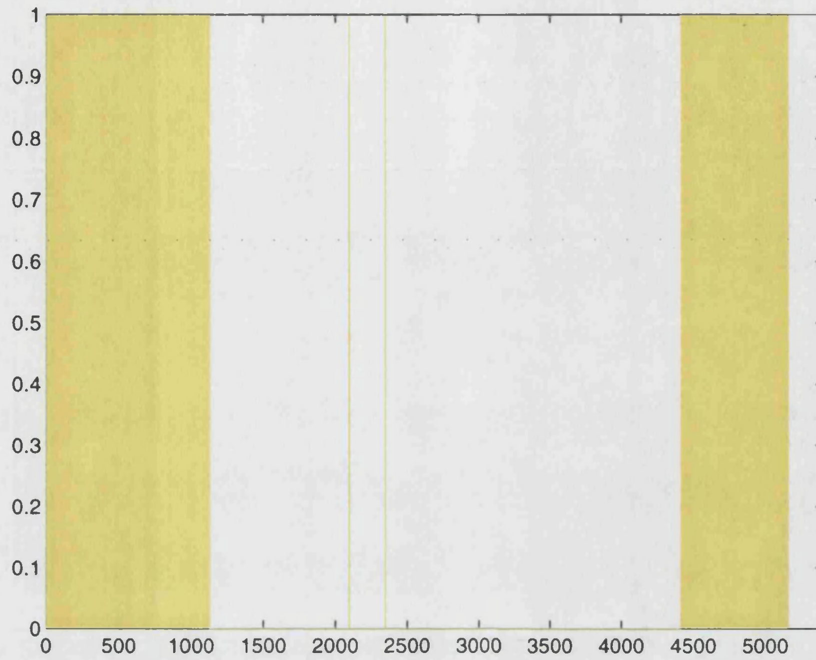



Figura 6.12: Evolución de los repartos para el proceso TEST a lo largo de toda la maniobra.

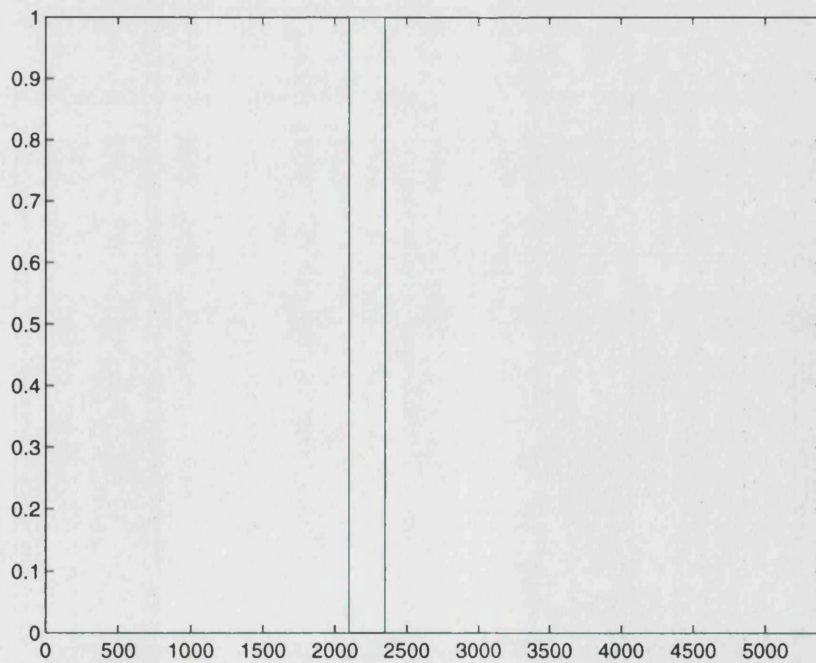



Figura 6.13: Evolución de los repartos para el proceso ROTAR a lo largo de toda la maniobra.

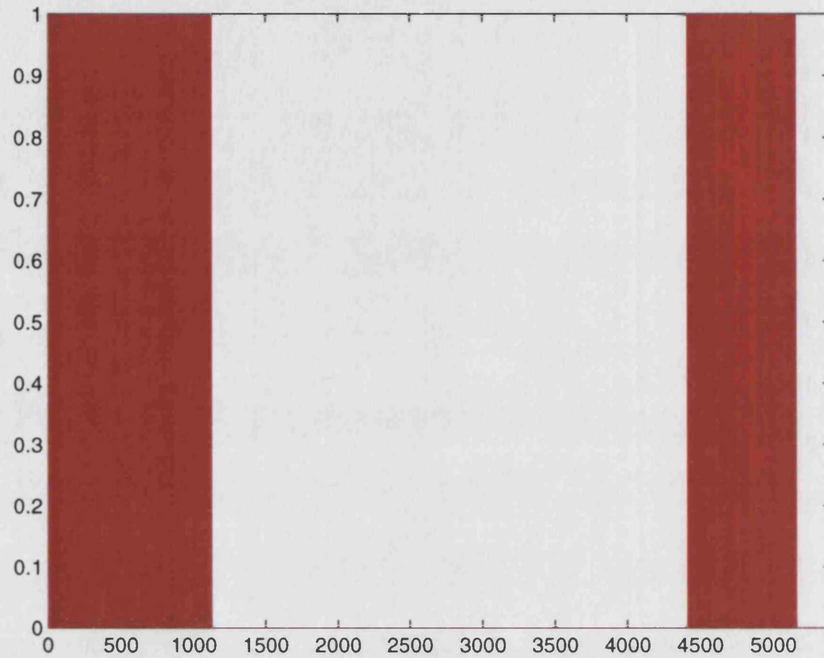



Figura 6.14: Evolución de los repartos para el proceso AVANZAR a lo largo de toda la maniobra.

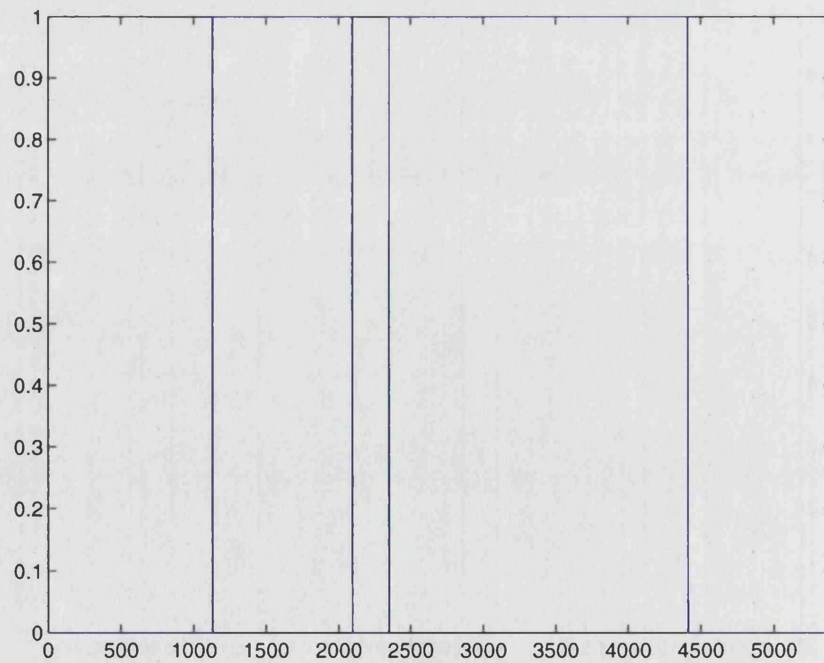



Figura 6.15: Evolución de los repartos para el proceso MOVER a lo largo de toda la maniobra.

curando la generación de las manifestaciones externas deseadas para un comportamiento total del robot satisfactorio en ese entorno.

Para este fin, se definen un número de niveles de competencia para el robot móvil. Un nivel de competencia es una especificación informal de un tipo determinado de comportamientos para dicho robot sobre todo su entorno de trabajo. Así, un nivel de competencia superior, implica una clase de comportamiento deseado más específico.

La idea clave de los niveles de competencia es que de esta manera, podemos construir capas del sistema de control, las cuales se corresponden con cada nivel de competencia, de manera que añadir un nuevo nivel, implica simplemente incluir una nueva capa sobre las ya existentes.

Brooks implementa cada bloque de los que forman un nivel de control como máquinas de estados finitos aumentadas (AFSM). Estas son unidades de proceso elementales que toman sus entradas o bien de los sensores directamente o bien de las salidas de otros módulos y que emiten sus salidas de tres posibles formas diferentes: como valores de entrada para otras AFSM's, como señales de inhibición que bloquean una salida determinada de una AFSM durante algún tiempo o como una entrada supresora que sustituye la señal de entrada recibida por una AFSM por la enviada a través de la entrada supresora durante un tiempo fijo. La topología de la red de interconexiones, así como el diseño de los niveles y bloques dentro de cada nivel se deciden por el programador.

En nuestro modelo, cada AFSM se implementa como un proceso. Los cables de comunicación se implementan mediante variables globales asociadas a variables booleanas para la activación/inhibición de mensajes. Estas variables booleanas se comprueban por el/los módulo/s implicado/s con el fin de decidir la salida que se debe presentar. Los repartos a los procesos se asignan al principio y no se modifican durante todo el tiempo de ejecución. El criterio, de acuerdo con Brooks es o bien asignar menos tiempo de procesador a los procesos que constituyen niveles superiores, puesto que estos son menos críticos o bien asignar el mismo reparto a todos e implementar periodos de activación/inhibición para los procesos que en un monento determinado deban estar bloqueados por la activación de un nivel inferior (urgencia mayor).

Como ejemplo de prueba, hemos construido un sistema de control con dos niveles de competencia. Su diagrama de bloques puede verse en la figura 6.16.

El nivel de competencia 0 es el nivel inferior del sistema de control y es el encargado de asegurar que el robot no entra en contacto con otros objetos de su entorno. Si algún otro cuerpo se aproxima al robot (y se detecta), éste se aleja en dirección contraria. Si en el transcurso de un movimiento el robot colisiona con un objeto, éste para automáticamente. Estas dos actitudes juntas serían en principio suficientes para que el robot no resultase

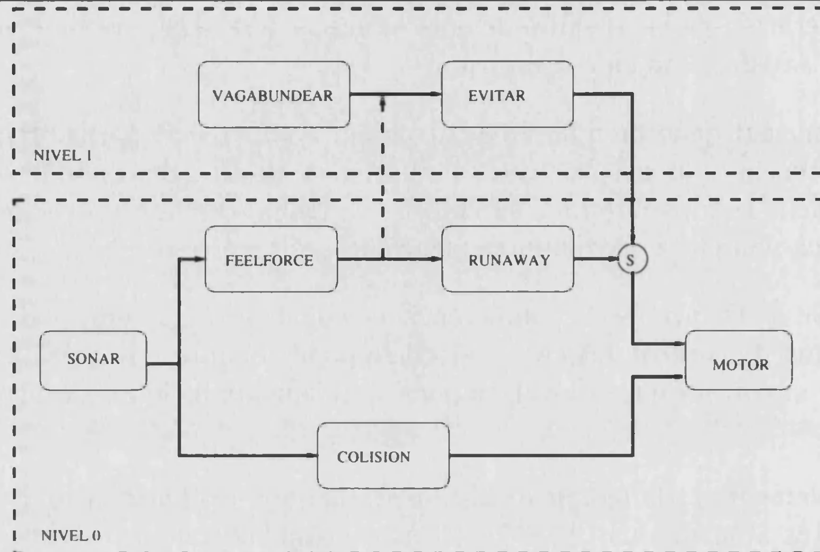



Figura 6.16: Diagrama de bloques y niveles para una arquitectura de subsumción básica.

dañado.

Los módulos que forman el nivel 0 de competencia son los siguientes:

- SONAR: es el módulo encargado de obtener un vector con los valores de las medidas de los sensores ultrasónicos (la actualización de cada sensor se realiza cada 15.2 ms.). En la actualidad se toma como medida definitiva una sóla lectura para cada sensor. En un futuro próximo se debe aumentar la fiabilidad de las medidas introduciendo un filtro. Este proceso se implementa como un proceso sin reparto (es decir de ejecución obligada en cada ciclo del pseudo-planificador y durante el tiempo necesario) debido a que se trata de un recurso vital para el funcionamiento del sistema completo.
- COLISION: es el módulo encargado de detectar el cierre de cualquier microinterruptor asociado normalmente con una colisión. En función de los microinterruptores cerrados, es capaz de conocer el ángulo por el que se ha colisionado. En la actualidad, la detección de una colisisión implica automáticamente una parada de emergencia, aunque podría tomarse cualquier otra decisión asociada a este evento. Este proceso se implementa de la misma manera y por la misma razón que el proceso anterior.
- FEELFORCE: este módulo se encarga de detectar objetos que caigan dentro del radio de seguridad del robot. Si algún sensor detecta este evento, se realiza la suma vectorial de los vectores de repulsión para cada uno de los sensores que lo cumplan (similar a un campo potencial artificial) dando como resultado el ángulo a girar para alejarse. El módulo de la fuerza de repulsión en la actualidad no es proporcional a la magnitud de la suma vectorial, sino que es fijo, alejando al robot unos 200 mm.

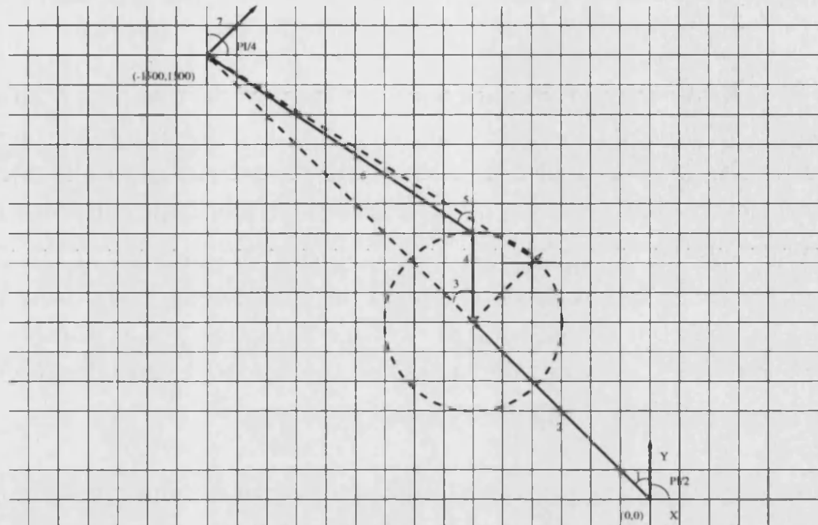



Figura 6.17: Maniobra a realizar como ejemplo de utilización del pseudo-planificador en la implementación de una arquitectura reactiva.

- RUNAWAY: es el módulo encargado de tomar la información de FEELFORCE y activar las funciones correspondientes de "rotar" y "mover" para dirigir el robot en el orden adecuado.
- MOTOR: este módulo está formado por el conjunto de funciones con capacidad de mover el robot (rotar, mover, avanzar,...). Es el interfaz entre los módulos del nivel 0 y el robot físico.

Como resultado de la implementación del nivel 0 de competencia, en su ejecución, el robot se comporta de la siguiente manera: inicialmente está parado. Si alguno o algunos de los sensores detectan objetos que caen dentro del radio de seguridad, el módulo FEELFORCE calcula el ángulo a girar y el módulo RUNAWAY activa en secuencia las funciones del módulo MOTOR para rotar el ángulo necesario y para avanzar una determinada distancia fija. Con la finalización de la ejecución de dichas funciones, el robot vuelve a permanecer parado hasta una nueva detección. Si se cierra cualquiera de los microinterruptores, el módulo COLISION genera una parada de emergencia.

El nivel de competencia 1 debe añadir alguna nueva capacidad sobre el nivel 0. En nuestro caso, el nivel 1 se encarga de llevar al robot desde una posición y orientación inicial hasta una posición y orientación final. En la figura 6.17 se presenta la maniobra a realizar.

La maniobra consiste en llevar al robot desde la posición inicial ($X = 0, Y = 0, ANGULO = PI/2$) hasta la posición final ($X = -1500, Y = 1500, ANGULO = PI/4$). Esta es la

misión del nivel de competencia 1 formada por los siguientes módulos:

- VAGABUNDEAR: es el módulo encargado de decidir la posición y orientación final deseada. Para la prueba, se trata de una posición y orientación fijas, pero este módulo podría ser el resultado de la salida de un planificador, de manera que el resultado para llevar al robot desde una posición inicial a una final fuera un conjunto de puntos por donde se debería pasar.
- AVOID: es el módulo encargado de llamar a las funciones del módulo MOTOR en el orden adecuado para llevar al robot al destino marcado por el módulo anterior. El código de este módulo subsume de alguna manera al del módulo RUNAWAY en el caso de que haya algún destino al que ir.

Como resultado en la ejecución del sistema de control cuando se ha incluido el nivel 1 de competencia, la secuencia de acciones es la siguiente (ver figura 6.17): el robot que parte de la posición y orientación $(X_i = 0, Y_i = 0, \theta_i = \pi/2)$ debe llegar a la posición y orientación $(X_f = -1500, Y_f = 1500, \theta_f = \pi/4)$. Los repartos iniciales para los cuatro procesos implicados en los dos niveles de competencia (FEELFORCE, RUNAWAY, VAGABUNDEAR y AVOID) son los mismos y de valor 0.2 (el pseudo-planificador generará un proceso IDLE que utilizará el tiempo restante hasta 1.0, o sea 0.2 también). La maniobra a realizar por el nivel 1 es:

- Rotar $\arctan(\frac{Y_f - Y_i}{X_f - X_i}) - \theta_i$ grados sobre el centro geométrico del robot.
- Avanzar una distancia de $\sqrt{(X_f - X_i)^2 + (Y_f - Y_i)^2}$.
- Rotar $\theta_f - \arctan(\frac{Y_f - Y_i}{X_f - X_i})$ grados sobre el centro geométrico del robot.

En el ejemplo, el robot comienza rotando $\pi/4$ (punto 1 de la gráfica). A continuación avanza en línea recta para llegar a las coordenadas del punto final (punto 2). En el punto (3), el sensor 6 ha detectado un obstáculo, de manera que los procesos del nivel 0 toman el control momentáneo del robot (activación de la variable SUPRESOR que funciona como tal) para rotar $-\pi/4$ (punto 3) y avanzar 200 mm. (punto 4). Con la finalización de la maniobra de huida (desactivación de la variable SUPRESOR), el nivel 1 vuelve a tomar el control, de manera que se recalcula el nuevo ángulo a rotar y la nueva distancia a recorrer para llegar desde la nueva posición y orientación a la deseada (puntos 5 y 6 y 7).

El trasvase de la arquitectura de niveles con sus módulos y su red de interconexiones para su ejecución en nuestro pseudo-planificador es tan simple como sustituir las líneas de código donde se activan las variables supresoras por las correspondientes que disponen repartos de 0.0 para los módulos de los niveles superiores al que toma la competencia y el reparto (siguiendo el criterio deseado) del tiempo de una vuelta del pseudo-planificador entre los módulos del nivel activo en ese momento, y sustituir las líneas de código donde

se desactivan las variables supresoras por las correspondientes que restauran los repartos establecidos al inicio de la ejecución. En nuestro ejemplo, el reparto inicial es de 0.2 para cada uno de los 4 procesos implicados. El módulo FEELFORCE tiene la capacidad de modificar estos repartos (simulación de la activación del supresor) para la inhibición de los módulos del nivel 1 y el módulo RUNAWAY tiene la capacidad de restaurar los repartos iniciales (simulación de la desactivación del supresor).

El código generado para implementar los dos niveles inferiores de la arquitectura reactiva de Brooks se presenta a continuación.

```
void *VAGABUNDEAR(void *pass)
{
  long xf=-1500,yf=1500;
  double angf=M_PI/4;

  BEGIN_BEH_MOD_P
   {
     X_final=xf;
     Y_final=yf;
     ANG_final=angf;

     if (Inicio_maniobra)
       {
         ang_wander=atan2((double)(Y_final-Y),(double)(X_final-X));
         Dist_wander=sqrt(((X_final-X)*(X_final-X))+((Y_final-Y)*(Y_final-Y)));

 if (ang_wander!=0.0)
   {
     f1=SI;
     f2=f3=f4=f5=f6=NO;
   }

         Inicio_maniobra=NO;
 primera_vez=SI;
       }

     ONE_SHOT_WAITS_HERE;
   }
  END_BEH_MOD_P
}

/* -------------------------------------------------------------- */

void *AVOID(void *pass)
{
  double angulo_int=0.0;

  BEGIN_BEH_MOD_P
   {
     if (!SUPRESOR)

       if (f1)
         {
           if (primera_vez)
     {
       angulo_int=ANGULO;
       primera_vez=NO;
```

```
  }
          Hay_que_rotar=SI;
    if (ang_wander>=angulo_int)
      {
        Codigo_display=11;
        Comando=GIRO_IZQ;
        outb(Comando,SENTIDO_GIRO);
        Angulo_a_rotar=ang_wander-angulo_int;
      }
    else
      {
        Codigo_display=12;
        Comando=GIRO_DER;
        outb(Comando,SENTIDO_GIRO);
        Angulo_a_rotar=angulo_int-ang_wander;
      }
     C_rot();
          }
        else if ((f2) && (Dist_wander!=0.0))
          {
            Hay_que_mover=SI;
    Codigo_display=13;
    VELOC_IZQ=100;
    VELOC_DER=100;
    Comando=ADELANTE;
    outb(Comando,SENTIDO_GIRO);
            Distancia=Dist_wander;
    C_pos_vel();
          }
        else if (f3)
          {
            Hay_que_rotar=SI;
    if (ANG_final>=ang_wander)
      {
        Codigo_display=14;
        Comando=GIRO_IZQ;
        outb(Comando,SENTIDO_GIRO);
                Angulo_a_rotar=ANG_final-ang_wander;
      }
    else
      {
        Codigo_display=15;
        Comando=GIRO_DER;
        outb(Comando,SENTIDO_GIRO);
        Angulo_a_rotar=ang_wander-ANG_final;
      }
    C_rot();
          }
        else
          {
            Fin_maniobra=SI;
    Codigo_display=16;
            Parar();
          }

        ONE_SHOT_WAITS_HERE;
    }
    END_BEH_MOD_P
  }
```

```
/* ---------------------------------------------------------------- */

/* Esta funci'on genera una fuerza repulsiva resultante de la suma de fuerzas
   de repulsi'on de los objetos que caen dentro del l'imite de seguridad e independiente
   de la distancia entre ellos y el robot. */

void *FEELFORCE(void *pass)
{
  int i,Detectado_obst,Detectado_obst_0;
  static int count=0;

  BEGIN_BEH_MOD_P
   {
     if ((!Hay_que_rotar) && (!SUPRESOR))

         if (count>(2*NUM_SONARS))
           {
     count=0;
     Fr=0.0;

     Detectado_obst_0=0;
     if ((lectura_sonars[0]<=LIMITE_INF)&&(lectura_sonars[0]!=0.0))
       {
                 Detectado_obst_0=1;
 SUPRESOR=SI;
       }

     Detectado_obst=0;
             for (i=1;i<NUM_SONARS;i++)
               if ((lectura_sonars[i]<=LIMITE_INF)&&(lectura_sonars[i]!=0.0))
                 {
           Fr+=(ANGULO_SENSOR*i);
           Detectado_obst=1;
   SUPRESOR=SI;
         }

             if (Detectado_obst)
               {
         Num_beeps=2;

                 if (Fr>(2*M_PI))
           Fr=((int)(Fr*180.0/M_PI)%360)*(M_PI/180.0);

                 f4=SI;
 Mover_distancia=ACABADA;
         f1=f2=f3=NO;

         if (Fr<M_PI)
           {
             Comando=GIRO_IZQ;
             outb(Comando,SENTIDO_GIRO);
             Angulo_a_rotar=M_PI-Fr;
           }
                 else
           {
             Comando=GIRO_DER;
             outb(Comando,SENTIDO_GIRO);
             Angulo_a_rotar=Fr-M_PI;
           }

                 if (Detectado_obst_0)
```

```
         Angulo_a_rotar=(Angulo_a_rotar+M_PI)/2.0;
             }
           else
             if (Detectado_obst_0)
               {
   Num_beeps=2;
         f4=SI;
   Mover_distancia=ACABADA;
   f1=f2=f3=NO;

         Comando=GIRO_IZQ;
         outb(Comando,SENTIDO_GIRO);
                 Angulo_a_rotar=M_PI;
       }
             else
       {
         f4=NO;
   Angulo_a_rotar=0.0;
 }
           }
   else
     count++;

     ONE_SHOT_WAITS_HERE;
    }
  END_BEH_MOD_P
}

/* ------------------------------------------------------------------------- */

void *RUNAWAY(void *pass)
{
  BEGIN_BEH_MOD_P
    {
      if (f4)
        {
  Hay_que_rotar=SI;
  Codigo_display=1;
  C_rot();
}
      else if (f5)
        {
  Hay_que_mover=SI;
  Codigo_display=2;
  VELOC_IZQ=100;
  VELOC_DER=100;
  Distancia=300;
  Comando=ADELANTE;
  outb(Comando,SENTIDO_GIRO);
  C_pos_vel();
}
      else if (f6)
        {
Codigo_display=3;
/*  Parar(); */
  SUPRESOR=NO;
        Inicio_maniobra=SI;
  Fin_maniobra=NO;
        }

     ONE_SHOT_WAITS_HERE;
```

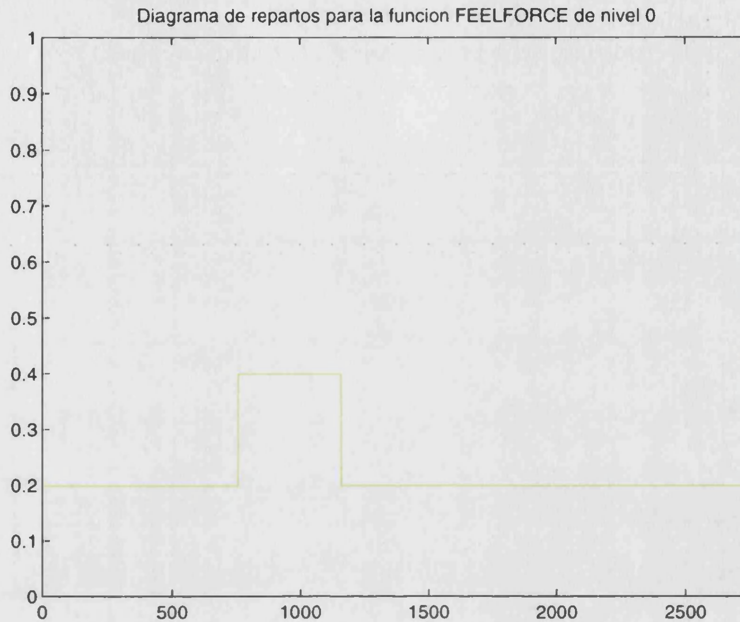



Figura 6.18: Evolución del reparto para el proceso FEELFORCE.

```
    }
  END_BEH_MOD_P
}

void init_architecture(int basics_loaded)
{
 int from_where;

 from_where=(basics_loaded ? NUM_BASIC_PROCESSES : 0);

 /* Funciones para ejemplo de arquitectura reactiva con 2 niveles
 de competencia */

 fun[from_where]=FEELFORCE;
 fun[from_where+1]=VAGABUNDEAR;
 fun[from_where+2]=AVOID;
 fun[from_where+3]=RUNAWAY;
}
```

Por último, en las gráficas desde la 6.18 hasta la 6.21 puede verse la evolución de los repartos a lo largo del tiempo que dura la maniobra que estamos describiendo para cada uno de los módulos.

Figura 6.19: Evolución del reparto para el proceso VAGABUNDEAR.

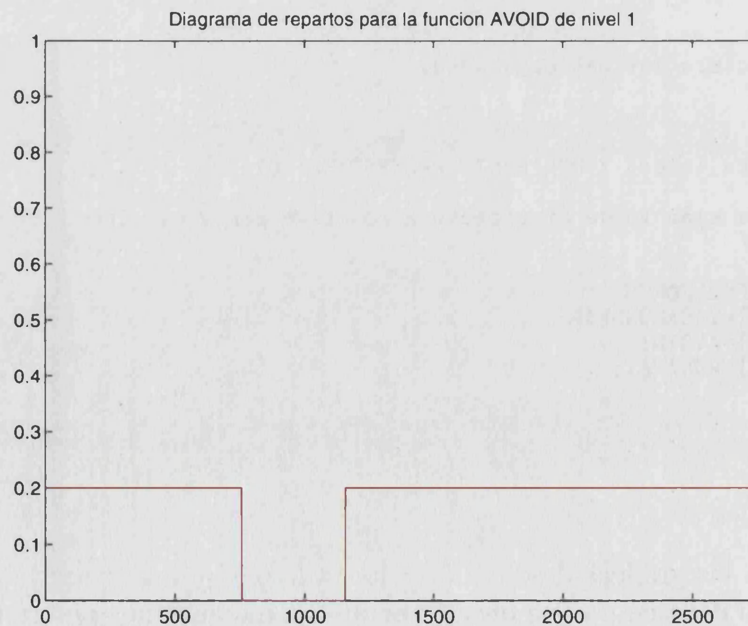



Figura 6.20: Evolución del reparto para el proceso AVOID.

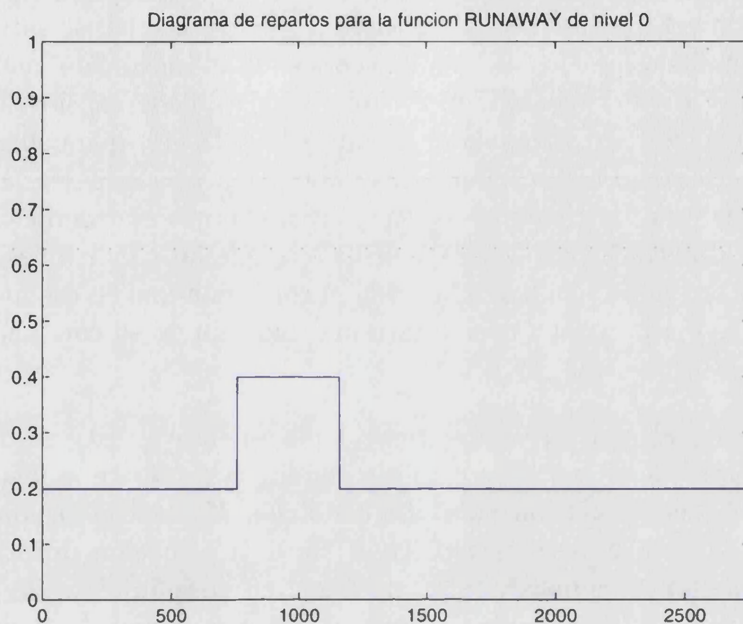



Figura 6.21: Evolución del reparto para el proceso RUNAWAY.

## 6.3 Arquitectura Teleo-Reactiva

En esta aproximación propuesta por Nilsson [Nil94], se identifica una secuencia teleo-reactiva como un programa de control que dirige al sistema hacia una meta (*teleo*) teniendo al mismo tiempo en cuenta las circunstancias de cambio en el entorno (*reactivo*). En su forma más simple, consiste en un conjunto ordenado de reglas de producción:

$$\begin{array}{ll} K_1 & \longrightarrow a_1 \\ K_2 & \longrightarrow a_2 \\ \cdots & \\ K_i & \longrightarrow a_i \\ \cdots & \\ K_m & \longrightarrow a_m \end{array}$$

donde las $K_i$ son condiciones (de las entradas de los sensores y del modelo del mundo) y las $a_i$ son acciones (sobre el mundo o para cambiar de alguna manera el modelo que se tiene de él). Una secuencia teleo-reactiva se interpreta de forma similar a la forma en que se interpretan algunos sistemas de producción. La lista de reglas se rastrea de arriba

hacia abajo para detectar la primera regla cuya parte condicional se satisface y entonces la acción correspondiente se ejecuta, suprimiéndose la ejecución de cualquier acción de rango inferior que estuviera efectuándose. Además, se puede considerar a las acciones teleo-reactivas más como durativas en el tiempo (AVANZAR) que como discretas. Una acción durativa permanece activa mientras su condición asociada sea la primera que se cumple dentro del conjunto de condicionales. Cuando la primera condición cierta cambia, la acción también lo hace. De esta manera se deduce que las condiciones deben evaluarse contínuamente; la acción asociada con la primera condición que en ese momento es cierta es siempre la que se ejecuta. Una acción termina sólo cuando su condición deja de ser la primera cierta en la lista.

Así, normalmente al escribir una secuencia teleo-reactiva, ésta se implementará de manera que la condición $K_1$ se tome como la condición meta, de manera que la correspondiente acción $a_1$ será la acción nula. La condición $K_2$ será la acción de menos peso de manera que cuando ésta se satisfaga (y $K_1$ no), la ejecución durativa de la acción asociada $a_2$ eventualmente guiará al sistema hacia el cumplimiento de la condición $K_1$ y así sucesivamente. Además y por supuesto, la acción asociada a una determinada condición puede ser una nueva secuencia teleo-reactiva.

Para comprobar el funcionamiento del pseudo-planificador en la implementación de una arquitectura teleo-reactiva, se planteó el problema del seguimiento de un objeto móvil por parte del robot. La descripción detallada del experimento es el siguiente:

La condición meta es que nuestro robot RODNEY alcance con su sensor ultrasónico frontal (sensor 0) la distancia mínima de seguridad. Al comienzo de la ejecución, el robot está parado. Si el sensor 0 (ver figura 6.22) detecta un objeto que está dentro de su radio de seguimiento (comprendido entre el umbral mínimo y un umbral máximo), se acercará a dicho objeto hasta que la distancia sea la mínima umbral de seguridad, en cuyo caso debe parar. Si el sensor 0 no detecta objetos dentro de su radio de seguimiento, pero el sensor 1 ó el sensor 2 sí lo hacen, el robot debe girar para orientarse en la dirección del objeto detectado. Por último, si no se cumple ninguna de las condiciones anteriores el robot debe parar. Con esta sencilla secuencia teleo-reactiva, el comportamiento del robot debe ser el de seguir cualquier objeto móvil detectado por sus sensores frontales.

Una posible secuencia teleo-reactiva para la resolución del experimento podría ser la siguiente:

$$\begin{aligned} NO_DETECTADO_OBJETO &\longrightarrow PARAR \\ DETECTADO_OBJETO_FRONTAL &\longrightarrow AVANZAR \\ DETECTADO_OBJETO_LATERAL &\longrightarrow ROTAR \end{aligned}$$

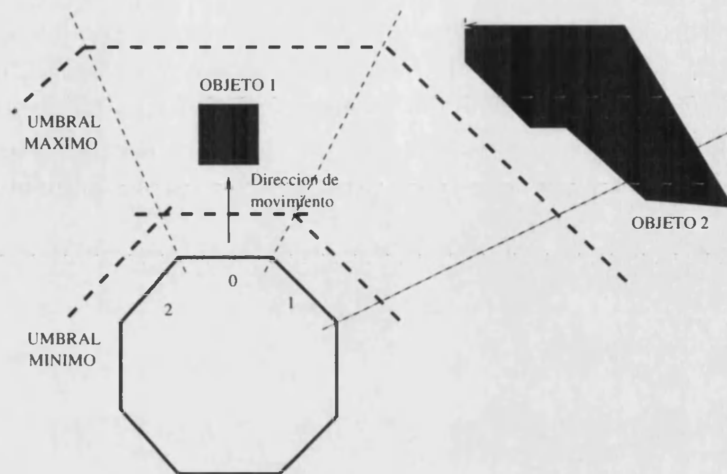



Figura 6.22: Disposición de objetos para el experimento de arquitectura teleo-reactiva.

Los procesos implementados para la prueba y su función son los siguientes:

- RECALCULAR: este proceso se encarga de recalcular el conjunto de condicionales de la secuencia teleo-reactiva completa. Dicho proceso debe verse más como la implementación de un circuito electrónico que como un sistema discreto. Se supone que contínuamente dispone del conjunto de condiciones actualizadas. En nuestro caso, dicho proceso debe implementarse como un proceso sin reparto para que en cada ciclo del pseudo-planificador sea ejecutado. Su código asociado es el siguiente:

```
void *RECALCULAR(void *pass)
{
  BEGIN_BEH_MOD_P
   {
      if ((lectura_sonars[0]>LIMITE_INF) && (lectura_sonars[0]<(2*LIMITE_INF)))
        Condicion1=SI;
      else
       {
         Condicion1=NO;
Avanzando=NO;
       }

      if ((lectura_sonars[1]>LIMITE_INF) && (lectura_sonars[1]<(2*LIMITE_INF)))
        Condicion2=SI;
      else
        Condicion2=NO;

      if ((lectura_sonars[NUM_SONARS-1]>LIMITE_INF) &&
                (lectura_sonars[NUM_SONARS-1]<(2*LIMITE_INF)))
        Condicion3=SI;
      else
        Condicion3=NO;
   }
 END_BEH_MOD_P
}
```

- TELEO: es el proceso encargado de asignar las condiciones con las acciones asociadas. Este proceso también se implementa como un proceso sin reparto, de manera que se asegura su ejecución en cada ciclo del pseudo-planificador. La acción asociada viene determinada en nuestro caso por repartos no nulos para los procesos implicados en la acción y repartos nulos para el resto de procesos. Su correspondiente código asociado es:

```
void *TELEO(void *pass)
{
 BEGIN_BEH_MOD_P
  {
   if (Hay_que_rotar)
    {
      pr[10].sharing[DESIRED_SH]=0.5;
      pr[11].sharing[DESIRED_SH]=0.0;
    }

   else if (!Hay_que_rotar && Condicion1)
    {
      Avanzando=SI;
      Comando=ADELANTE;
      outb(Comando,SENTIDO_GIRO);
      pr[10].sharing[DESIRED_SH]=0.0;
      pr[11].sharing[DESIRED_SH]=0.5;
    }

   else if (!Hay_que_rotar && !Condicion1 && Condicion2)
    {
      Hay_que_rotar=SI;
      Comando=GIRO_DER;
      outb(Comando,SENTIDO_GIRO);
      pr[10].sharing[DESIRED_SH]=0.5;
      pr[11].sharing[DESIRED_SH]=0.0;
    }

   else if (!Hay_que_rotar && !Condicion1 && !Condicion2 && Condicion3)
    {
      Hay_que_rotar=SI;
      Comando=GIRO_IZQ;
      outb(Comando,SENTIDO_GIRO);
      pr[10].sharing[DESIRED_SH]=0.5;
      pr[11].sharing[DESIRED_SH]=0.0;
    }

   else
    {
      Codigo_display=40;
      Parar();
    }
  }
 END_BEH_MOD_P
}
```

- ROTAR1: proceso encargado de realizar una rotación de $\pi/4$ en la dirección seleccionada. Su correspondiente código es:

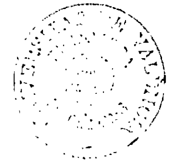
```
void *ROTAR1(void *pass)
{
```

```
 BEGIN_BEH_MOD_P
 {
   if (Hay_que_rotar)
     {
       Codigo_display=20;
       Angulo_a_rotar=M_PI/4;
       C_rot();
     }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}
```

- AVANZAR1: es el proceso encargado de mover el robot hacia adelante y su código viene definido por:

```
void *AVANZAR1(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Avanzando)
    {
     Codigo_display=30;
     VELOC_IZQ=50;
     VELOC_DER=50;
     C_vel();
    }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}
```

Como comentario final, cabe decir que aunque sencillos, estos ejemplos muestran bien cómo distintas concepciones, plasmadas como arquitecturas aparentemente muy diferentes, pueden sin embargo implementarse con muy poco esfuerzo y en un número reducido de líneas de código, una vez que todo aquello que es común ha sido dejado al cuidado del sistema y del pseudo-planificador. Creemos que el hecho de disponer de esta flexibilidad sin perder expresividad ni facilidad de acceso al hardware es una característica interesante del software diseñado.



# Capítulo 7

# Conclusiones y trabajo futuro

A raiz de los objetivos propuestos en el capítulo primero sobre las necesidades específicas a la hora de abordar la implementación el comportamiento y la comparación de diversas arquitecturas de robots móviles, podemos decir que:

- En lo que respecta al sistema mecánico, hemos sido capaces de diseñar, construir y poner en funcionamiento una plataforma base de muy bajo costo, la cual a pesar de no estar construída con materiales y procedimientos industriales, se comporta de manera muy satisfactoria y puede ser utilizada como plataforma de prueba de los tópicos usuales en el campo de la robótica móvil en la actualidad.
- En lo que respecta al sistema electrónico, se han diseñado y construído diversas tarjetas para gobernar tanto al sistema mecánico como al sensorial. Estas tarjetas, en este primer prototipo, están en algún caso cableadas y en otros se ha realizado el PCB. La ventaja principal que se tiene de esta manera es la completa accesibilidad a todos los recursos implementados, de manera que por ejemplo, se puede saber en todo momento y tanto por software como por hardware el estado del sistema odométrico, también en cualquier momento se puede enviar un código de estado al display así como recibirlo de los microinterruptores, además, los sensores ultrasónicos se pueden disparar en el instante en número y de la manera deseada según convenga...
- Por último, en las pruebas realizadas sobre el pseudo-planificador diseñado para la comparación de diversas arquitecturas, éste se ha manifestado como una herramienta eficaz para la implementación de cualquiera de ellas con un mínimo esfuerzo de programación y aunque las pruebas se han realizado sobre la misma plataforma hardware, la implementación del control de la misma y del pseudo-planificador están totalmente separados, de manera que éste podría sin gran dificultad ser portado a otro hardware siempre que fuera suficientemente abierto.

Así, podemos comprobar que nuestra idea de utilizar un ejecutor cíclico con un periodo

fijo pero con reparto variable, es un esquema muy conveniente para emular cualquiera de las arquitecturas utilizadas en la actualidad en el campo de la robótica móvil. Esto, también viene a ratificar nuestra idea de que cualquier arquitectura actual (e idealmente cualquiera que pudiera desarrollarse en un futuro), puede ser implementada mediante algún esquema de repartos entre los diversos bloques que la forman o incluso mediante diversos esquemas de repartos entre los distintos módulos (los cuales pueden poseer a su vez un esquema propio de repartos entre bloques).

Del ámplio abanico de posibilidades que puede brindar una rama de la ingeniería como es la robótica (en nuestro caso la móvil) en cuanto a líneas de investigación y desarrollo, varias son las direcciones en las que se tiene planteada continuación. Los objetivos inmediatos son los siguientes:

- En lo que respecta al apartado de sensorización y desarrollo de hardware específico para este fin, se tienen proyectados los siguientes trabajos:
  - La introducción de inclinómetros para la medida de la inclinación lateral y frontal.

    Si bien la inclinación frontal posee poca importancia en una configuración de robot como la que posee el nuestro, no ocurre lo mismo con la información referente a la inclinación lateral, pues esta inclinación puede estar indicando que el robot está atravesando un objeto dispuesto en la superficie sobre la que se mueve, hecho éste que debe ser tratado por el módulo de control para realizar las correcciones pertinentes.
  - La introducción de sensores de infrarrojos para la detección de obstáculos a distancias cortas.

    De esta manera, se puede cubrir el espacio que los sensores ultrasónicos de Polaroid dejan indetectable y que son las distancias inferiores a 40-50 cm.
  - La introducción de una cámara foveal, la cual posee un sensor retínico espacio-variante y el hardware asociado para poder realizar análisis de imágenes (por ejemplo el cálculo del flujo óptico para el análisis del tiempo al impacto) en tiempo real.
  - La puesta en marcha de los sensores ultrasónicos por variación de amplitud.

    La relación entre el consumo de corriente de la electrónica que ponía en funcionamiento este módulo y la información proporcionada por él (básicamente información binaria relativa a la detección de obstáculos a distancias inferiores a 50 cm y con errores sensibles) nos llevó a tomar la decisión de eliminarlo como elemento sensor. Lo que se pretende en un futuro próximo es rediseñar la electrónica para minimizarla tanto en consumo como en tamaño y volver a poner en funcionamiento el módulo para reforzar la información suministrada por los detectores de infrarrojos.
- En lo que respecta al esquema de control implementado, varios son los trabajos que

se propondrán, a saber:

- Si bien se posee una identificación aceptable de algunos de los módulos que constituyen el bucle de control, de otros, en particular del bloque motor-reductor se posee una función de transferencia aproximada, obtenida por procedimientos de identificación simples (la introducción de un escalón de tensión a la entrada y la identificación de la curva de respuesta pulsos/T para la obtención tanto de la ganancia estática como de la constante de tiempo). Se pretende realizar un proceso de identificación mediante algún algoritmo de estimación paramétrica (mínimos cuadrados, mínimos cuadrados extendidos,...) de manera que a partir de dichos valores, se pueda comprobar que las ecuaciones analíticas que garantizan la estabilidad del bucle de control desarrolladas en el capítulo 5 se cumplen en todos los casos.
- Desarrollo de un módulo denominado módulo de giro (módulo G) el cual se debe unir al esquema de control ya existente y que podría estar encargado de realizar giros en el vehículo incluso con comandos idénticos de velocidad en ambas ruedas directoras.

  La idea principal es que si en el esquema de control implementado en la actualidad se añade la salida del módulo de giro y se elimina/modifica el bloque de realimentación etiquetado con el valor $\beta$ según se puede observar en la figura 7.1, dependiendo tanto del signo como de la magnitud y/o de la derivada de la salida de dicho bloque, se puede forzar a que uno de los lazos de control aumente su velocidad respecto del otro. El resultado de esto es un camino curvo. De esta manera, se pueden obtener gráficas de la curvatura de la curva generada en función tanto de la variación de la salida del módulo G como del periodo de muestreo T, con lo cual, es posible que a partir de la ecuación de una trayectoria curva dada mediante cualquiera de los métodos más comunes de expresarlas (splines, $\beta$-splines, polinomios de interpolación, etc) pueda generarse dinámicamente y en función de T unos valores de salida para dicho módulo.

  La aplicación inmediata que puede tener dicho módulo es la corrección que normalmente es necesario introducir en ambos lazos de control como consecuencia de las diferencias entre los diámetros nominales y los reales en las ruedas directoras. Este hecho da como resultado un camino curvado incluso con un funcionamiento correcto del esquema de control. La única manera de corregirlo es o bien introduciendo pesos diferentes en la realimenteción por referencia cruzada a ambos bucles de control o bien mediante métodos de autolocalización externos al propio robot.
- En relación con uno de los trabajos proyectados en el capítulo de sensorización, posteriormente a la identificación paramétrica y a la verificación de las ecuaciones que garantizan la estabilidad, está el cálculo del lugar geométrico en el espacio de parámetros de los reguladores que garantizan la estabilidad del algoritmo de

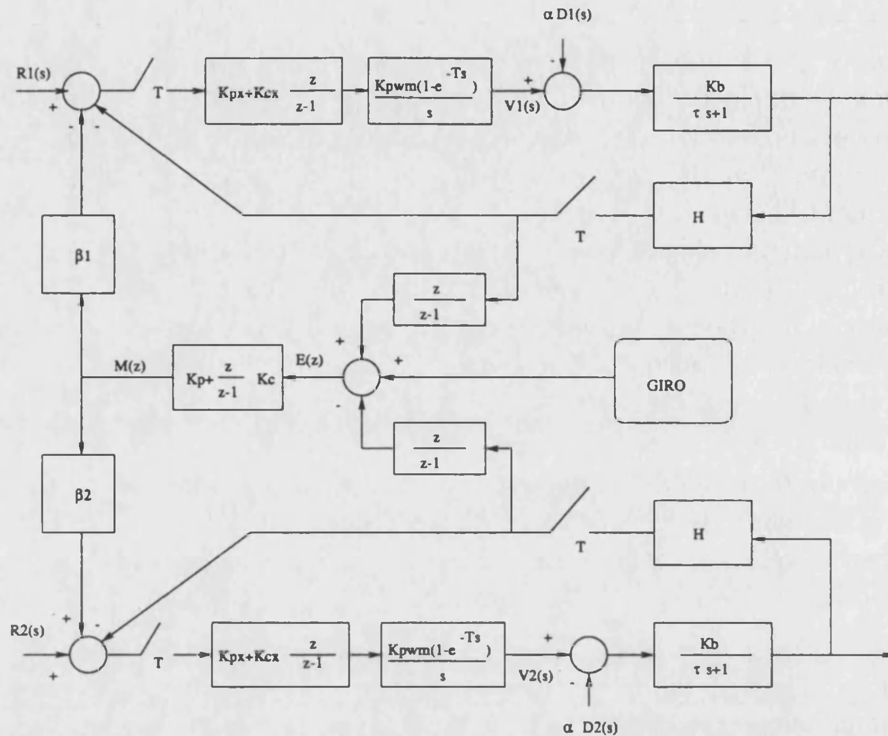



Figura 7.1: Diagrama de bloques del esquema de control propuesto incluyendo el módulo de giro

control.

 – Por último y también en relación con la introducción de un elemento hardware como son los inclinómetros, estaría la modificación del algoritmo de control para la corrección de la orientación en caso de inclinación lateral del robot como consecuencia de pasar por encima de un objeto.

   El hecho de que una de las ruedas tractoras pase por una superficie con una determinada altura respecto de la otra, hace que el camino recorrido por una sea diferente al recorrido por la otra. Como consecuencia, a la salida del obstáculo, la orientación será diferente a la que se llevaba antes de la entrada en el mismo. La idea es detectar la inclinación del vehículo e introducir en el esquema de control las correciones pertinentes para que este hecho no modifique (o afecte lo menos posible) a la trayectoria a seguir.

- Otro capítulo interesante a desarrollar es el de la fusión de la información procedente de los sensores. Así, paralelamente al desarrollo de nuevo hardware sensorial, se pretenden utilizar diversas ténicas de fusión para la obtención de información más rica y por lo tanto fiable del entorno.

- Las aplicaciones en tiempo real (como lo es la nuestra) son una clase especial de aplicaciones. Estas poseen un conjunto complejo de características que las distinguen de otros problemas software. Generalmente éstas deben complir restricciones más rigurosas. El funcionamiento correcto del sistema depende no sólo de los resultados de los cálculos, sino también del instante en que los resultados se produjeron. RTEMS, (Real-Time for Multiprocessors Systems) es un núcleo en tiempo real que proporciona esta facilidad para el desarrollo de aplicaciones empotradas (en principio militares). Se ha desarrollado a raiz de un estudio realizado en 1988 por el Centro de Ingeniería, Investigación y desarrollo del Grupo de Misiles de la Armada de EEUU. RTEMS incluye características tales como capacidades de multitarea, sistemas multiprocesadores (homogéneos y heterogéneos), control de eventos, sistema de prioridades, comunicación y sincronización entre tareas, localización de memoria dinámica y otros. Como ejemplo, se puede decir que objetos tales como tareas, colas, eventos, señales, semáforos y bloques de memoria se pueden diseñar como objetos globales y son accesibles por cualquier tarea con independencia del procesador en que resida tanto el objeto como la tarea en cuestión.

  Así pues, parece muy adecuado utilizar una herramienta como esta para la implementación de nuestro pseudo-planificador y comprobar su funcionamiento en un sistema multiprocesador (en particular en uno con 2 procesadores PENTIUM-II).

- Por último y como uno de los trabajos prioritarios en un futuro próximo, está la utilización del pseudo-planificador para el estudio de la evolución de arquitecturas. La idea principal es la utilización de algún tipo de esquema como por ejemplo el de algoritmos genéticos, para observar la evolución del conjunto de repartos a lo largo del tiempo en los distintos módulos que conforman el sistema completo, de manera que en función del tipo de misión a realizar por parte del robot y en función del tipo de restricciones impuestas, dicho sistema de repartos evolucionara de manera que se pudiera realizar la misión cumpliendo de manera óptima con los requisitos especificados. Es de esta manera como pensamos que pueden emerger nuevas ideas sobre las distintas relaciones organizativas en que los diferentes módulos que conforman el robot completo pueden asociarse.

# Apéndice A

# Código fuente

Este apéndice incluye el código completo y documentado de los programas más importantes realizados. En esta ocasión se ha decidido incluir el código fuente completo porque pensamos que se encuentra en un estado de desarrollo y documentación que puede hacerlo útil a otros programadores. El pseudo-planificador funciona correctamente con procesos arbitrarios, y los programas de control del hardware son, ciertamente, específicos, pero pueden servir de inspiración para cualquier sistema que use los ports de usuario como interfaces con el hardware. De cara precisamente a facilitar lo más posible este uso externo se ha decidido realizar y comentar todos los programas en inglés, excepto ciertos comentarios de uso muy específico en alguna arquitectura, así como generar tablas de referencia de uso de las variables, estructuras y macros. Esto último ha sido realizado de modo automático, también en inglés, por el programa de uso público cxref.

Los programas listados son los siguientes:

- `common.h`: constantes y macros usadas por el planficador, los procesos de bajo nivel, y las arquitecturas. Esencialmente, la estructura de proceso, la tabla global de procesos y las macros para comenzar y acabar un proceso.
- `ps_sched.h`: constantes, variables globales y macros usadas por el pseudo-planificador.
- `ps_sched.c`: la implementación de éste. Este programa es el que contiene a `main()`.
- `rodney.h`: constantes, variables globales y macros usadas por los procesos de control del hardware del robot, y por los procesos de bajo nivel encargados de la sensorización y acción.
- `rodney.c`: la implementación de las funciones de control de hardware y de sensorización y acción.
- `cases.c`: la arquitectura basada en casos descrita en el capítulo 6.
- `reactive1.c`: la arquitectura reactiva.
- `teleo_reactive.c`: la arquitecura teleo-reactiva. imagen Imaging-OFG.

- `ofg.c`: constantes, macros e implementación del manejo de la placa digital de imagen OFG.
- `monitor.c`: El programa de comunicación con el robot, encargado de recibir los datos de éste y mostrarlos, tanto en texto como mediante el interface gráfico sobre la imagen real de la cámara de a bordo usando la placa OFG.

Esencialmente, el planificador puede ser usado sin modificar el kernel de Linux, pero si se desea aumentar la granularidad es necesario:

- Sustituir la función nanosleep en las fuentes del kernel de Linux, fichero `/usr/src/linux/kernel/sched.c`, por la que se adjunta.
- Cambiar la constante HZ en las fuentes del kernel de Linux, fichero `/usr/src/linux/include/asm-i386/param.h`. Su valor por defecto es 100. Para conseguir una granularidad de $t$ $\mu s$. hay que usar un valor de HZ igual a $10^6/t$. En el kernel usual el scheduler se activa, pues, cada $10^4 \mu s = 10 ms$. En nuestro caso hemos usado $HZ = 6579$, con lo que el periodo de activación es de $152 \mu s$. El valor seleccionado para HZ deberá usarse también como valor para la constante HZ_NOW de `ps_sched.h`
- Recompilar el kernel de Linux, siguiendo las instruciones incluídas en todas las distribuciones.

Si no se desea recompilar el kernel, HZ_NOW en `ps_sched.h` debiera tomar el valor 100.

Sobre la escritura de procesos, éstos deben ser de la forma

```
void *PROCESO(void *pass)
{
 (declaracion de variables locales)

 BEGIN_BEH_MOD
 {
  <codigo C para el proceso>

  (Opcionalmente, si se trata de un proceso de una sola pasada:)
  ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD

}
```

Las macros BEGIN_BEH_MOD (comenzar módulo comportamental) y su correspondiente END_BEH_MOD están definidas en `common.h`, y deben verse como las marcas de inicio/fin de un bucle infinito, que se ejecuta tantas veces como sea posible en el tiempo actualmente asignado a ese proceso. Las respectivas versiones BEGIN_BEH_MOD_P y

END_BEH_MOD_P deberán usarse si el proceso debe acceder a los ports de E/S. En caso de incluir la llamada a ONE_SHOT_WAITS_HERE, debería hacerse justo antes de cerrar el módulo. En ese caso, el proceso queda interrumpido en ese punto hasta que su tiempo asignado expire, habiendo ejecutado como máximo una vuelta de bucle.

Sobre la adición de nuevas arquitecturas, se debe escribir uno o más ficheros `.h` y `.c` conteniendo todos los procesos que se consideren necesarios, cuyos nombres sean, p. ej., $P0$ hasta $PN$. En uno de los ficheros `.c` debe existir la función llamada `init_architecture` cuyo listado es:

```
void init_architecture(int basics_loaded)
{
 int from_where;

 from_where=(basics_loaded ? NUM_BASIC_PROCESSES : 0);

 fun[from_where]=P0;
 fun[from_where+1]=P1;
   .
   .
   .
 fun[from_where+N]=PN;
}
```

Solo los nombres de los procesos deben cambiarse. La variable global `fun`, declarada en `common.h`, es la tabla con los punteros a los procesos, y debe llamarse así. Además, los ficheros `.h` y `.c` que contienen la arquitectura deben incluírse en `ps_sched.c`, detrás de la inclusión de `rodney.c`. Hecho esto, se recompila todo con make usando el Makefile adjunto.

La invocación de `ps_sched` puede realizarse de dos maneras:

- Con un enlace simbólico al ejecutable de `ps_sched`, que debe llamarse `normal`, en cuyo caso se cargan los procesos sensoriales de bajo nivel sin reparto definidos en `rodney.c`, más los procesos de nuestra arquitectura, cada uno con el reparto y tipo asignados en la línea de comando
- O invocando directamente al ejecutable de `ps_sched`, en cuyo caso sólo los procesos explícitamente cargados en el array `fun` se ejecutan, cada uno con el tipo y reparto asignados en la línea de comandos.

# Program listings and references


1 File common.h . . . AppA/9
1.1 Type definitions . . . AppA/10
1.1.1 Typedef proc_info . . . AppA/10
1.2 Variables . . . AppA/10
1.2.1 Variable current_np . . . AppA/10
1.2.2 Variable Sh_buff . . . AppA/10
1.2.3 Variable fun . . . AppA/11
1.3 Actual file listing . . . AppA/11
2 File ps_sched.h . . . AppA/13
2.1 Variables . . . AppA/14
2.1.1 Variable current_np . . . AppA/14
2.1.2 Variable Sh_buff . . . AppA/14
2.1.3 Variable fun . . . AppA/15
2.1.4 Variable mut . . . AppA/15
2.1.5 Variable cond . . . AppA/15
2.1.6 Variable mshot . . . AppA/15
2.1.7 Variable cshot . . . AppA/15
2.1.8 Variable bcont_mut . . . AppA/15
2.1.9 Variable sch_t . . . AppA/15
2.2 Actual file listing . . . AppA/15
3 File ps_sched.c . . . AppA/16
3.1 Variables . . . AppA/17
3.1.1 Variable current_np . . . AppA/17
3.1.2 Variable Sh_buff . . . AppA/17
3.1.3 Variable fun . . . AppA/17
3.1.4 Variable mut . . . AppA/17
3.1.5 Variable cond . . . AppA/17
3.1.6 Variable mshot . . . AppA/17
3.1.7 Variable cshot . . . AppA/17
3.1.8 Variable bcont_mut . . . AppA/18
3.1.9 Variable sch_t . . . AppA/18
3.1.10 Variable Microswitches_code . . . AppA/18
3.1.11 Variable Microswitches_angle . . . AppA/18
3.1.12 Variable Display_code . . . AppA/18
3.1.13 Variable sonar_readings . . . AppA/18
3.1.14 Variable sonar_obstacle . . . AppA/18
3.1.15 Variable Son_buff . . . AppA/18
3.1.16 Variable Pos_or_buff . . . AppA/18
3.1.17 Variable Angle_to_rotate . . . AppA/18
3.1.18 Variable Rotation . . . AppA/19
3.1.19 Variable Must_rotate . . . AppA/19
3.1.20 Variable ang_wander . . . AppA/19
3.1.21 Variable Dist_wander . . . AppA/19
3.1.22 Variable X_final . . . AppA/19
3.1.23 Variable Y_final . . . AppA/19
3.1.24 Variable ANG_final . . . AppA/19
3.1.25 Variable f1 . . . AppA/19
3.1.26 Variable f2 . . . AppA/19
3.1.27 Variable f3 . . . AppA/19
3.1.28 Variable f4 . . . AppA/19
3.1.29 Variable f5 . . . AppA/20
3.1.30 Variable f6 . . . AppA/20
3.1.31 Variable Init_maneuver . . . AppA/20
3.1.32 Variable End_maneuver . . . AppA/20
3.1.33 Variable Fr . . . AppA/20
3.1.34 Variable SUPRESSOR . . . AppA/20
3.1.35 Variable first_time . . . AppA/20
3.1.36 Variable Mode . . . AppA/20
3.1.37 Variable Old_Mode . . . AppA/20
3.1.38 Variable Command . . . AppA/20
3.1.39 Variable LEFT_VEL . . . AppA/20
3.1.40 Variable RIGHT_VEL . . . AppA/21
3.1.41 Variable Distance . . . AppA/21
3.1.42 Variable Move_distance . . . AppA/21
3.1.43 Variable Must_move . . . AppA/21
3.1.44 Variable Must_go_ahead . . . AppA/21
3.1.45 Variable Moving_forward . . . AppA/21
3.1.46 Variable Door_detected . . . AppA/21
3.1.47 Variable Phase_1 . . . AppA/21
3.1.48 Variable G . . . AppA/21
3.1.49 Variable Total_left_pulses . . . AppA/21
3.1.50 Variable Total_right_pulses . . . AppA/21
3.1.51 Variable P_T_I . . . AppA/22
3.1.52 Variable P_T_D . . . AppA/22
3.1.53 Variable Count_left_pulses . . . AppA/22
3.1.54 Variable Count_right_pulses . . . AppA/22
3.1.55 Variable X . . . AppA/22
3.1.56 Variable Y . . . AppA/22
3.1.57 Variable ANGLE . . . AppA/22
3.1.58 Variable POSX . . . AppA/22
3.1.59 Variable POSY . . . AppA/22
3.1.60 Variable posx_new . . . AppA/22
3.1.61 Variable posy_new . . . AppA/22
3.1.62 Variable ind_centre . . . AppA/23
3.1.63 Variable Num_beeps . . . AppA/23
3.1.64 Variable Interval . . . AppA/23
3.1.65 Variable Whisper_freq . . . AppA/23
3.1.66 Variable Serial_mode . . . AppA/23
3.1.67 Variable Rec_buff . . . AppA/23
3.1.68 Variable Commands . . . AppA/23
3.1.69 Variable Tr_buff . . . AppA/23
3.1.70 Local Variables . . . AppA/23
3.2 Functions . . . AppA/24
3.2.1 Global Function ADVANCE() . . . AppA/24
3.2.2 Global Function COLISION_DETECTION() . . . AppA/24
3.2.3 Global Function C_pos() . . . AppA/24
3.2.4 Global Function C_pos_vel() . . . AppA/24
3.2.5 Global Function C_rot() . . . AppA/24
3.2.6 Global Function C_vel() . . . AppA/24
3.2.7 Global Function Close_all() . . . AppA/24
3.2.8 Global Function Consume() . . . AppA/24
3.2.9 Global Function Control() . . . AppA/24
3.2.10 Global Function Initialize() . . . AppA/24
3.2.11 Global Function InitializeMap() . . . AppA/24
3.2.12 Global Function InitializeSonars() . . . AppA/25

3.2.13 Global Function Initialize_all() . . . . . AppA/25
3.2.14 Global Function Latch() . . . . . AppA/25
3.2.15 Global Function MOVE() . . . . . AppA/25
3.2.16 Global Function ROTATE() . . . . . AppA/25
3.2.17 Global Function Read_Velocity() . . . . . AppA/25
3.2.18 Global Function StopRobot() . . . . . AppA/25
3.2.19 Global Function TEST() . . . . . AppA/25
3.2.20 Global Function TickSleep() . . . . . AppA/25
3.2.21 Global Function access_to_ports() . . . . . AppA/25
3.2.22 Global Function argument_length() . . . . . AppA/25
3.2.23 Global Function beep() . . . . . AppA/25
3.2.24 Global Function check_args() . . . . . AppA/26
3.2.25 Global Function end_tone() . . . . . AppA/26
3.2.26 Global Function get_pr() . . . . . AppA/26
3.2.27 Global Function get_sc_mode() . . . . . AppA/26
3.2.28 Global Function hard_latch() . . . . . AppA/26
3.2.29 Global Function init_architecture() . . . . . AppA/26
3.2.30 Global Function init_robot_hardware() . . . . . AppA/26
3.2.31 Global Function interval() . . . . . AppA/26
3.2.32 Global Function main() . . . . . AppA/26
3.2.33 Global Function map() . . . . . AppA/26
3.2.34 Global Function my_idle() . . . . . AppA/26
3.2.35 Global Function prepare_scheduler() . . . . . AppA/26
3.2.36 Global Function pseudo_sched() . . . . . AppA/26
3.2.37 Global Function ptrans() . . . . . AppA/27
3.2.38 Global Function read_byte_with_conf() . . . . . AppA/27
3.2.39 Global Function receive() . . . . . AppA/27
3.2.40 Global Function send() . . . . . AppA/27
3.2.41 Global Function serial() . . . . . AppA/27
3.2.42 Global Function set_attributes() . . . . . AppA/27
3.2.43 Global Function set_freq() . . . . . AppA/27
3.2.44 Global Function set_pr() . . . . . AppA/27
3.2.45 Global Function sonars() . . . . . AppA/27
3.2.46 Global Function sound() . . . . . AppA/27
3.2.47 Global Function start_tone() . . . . . AppA/27
3.2.48 Global Function touch_sharings() . . . . . AppA/27
3.2.49 Global Function trans_ult_and_sh() . . . . . AppA/28
3.2.50 Global Function usage() . . . . . AppA/28
3.2.51 Global Function visualize() . . . . . AppA/28
3.2.52 Global Function whisper() . . . . . AppA/28
3.2.53 Global Function write_two_bytes() . . . . . AppA/28
3.3 Actual file listing . . . . . AppA/28
4 File rodney.h . . . . . AppA/39
4.1 Type definitions . . . . . AppA/44
4.1.1 Type struct C . . . . . AppA/44
4.2 Variables . . . . . AppA/44
4.2.1 Variable current_np . . . . . AppA/44
4.2.2 Variable Sh_buff . . . . . AppA/44
4.2.3 Variable fun . . . . . AppA/45
4.2.4 Variable Microswitches_code . . . . . AppA/45
4.2.5 Variable Microswitches_angle . . . . . AppA/45
4.2.6 Variable Display_code . . . . . AppA/45
4.2.7 Variable sonar_readings . . . . . AppA/45
4.2.8 Variable sonar_obstacle . . . . . AppA/45
4.2.9 Variable Son_buff . . . . . AppA/45
4.2.10 Variable Pos_or_buff . . . . . AppA/45
4.2.11 Variable Angle_to_rotate . . . . . AppA/45
4.2.12 Variable Rotation . . . . . AppA/45





4.2.13 Variable Must_rotate . . . . . AppA/45
4.2.14 Variable ang_wander . . . . . AppA/45
4.2.15 Variable Dist_wander . . . . . AppA/45
4.2.16 Variable X_final . . . . . AppA/46
4.2.17 Variable Y_final . . . . . AppA/46
4.2.18 Variable ANG_final . . . . . AppA/46
4.2.19 Variable f1 . . . . . AppA/46
4.2.20 Variable f2 . . . . . AppA/46
4.2.21 Variable f3 . . . . . AppA/46
4.2.22 Variable f4 . . . . . AppA/46
4.2.23 Variable f5 . . . . . AppA/46
4.2.24 Variable f6 . . . . . AppA/46
4.2.25 Variable Init_maneuver . . . . . AppA/46
4.2.26 Variable End_maneuver . . . . . AppA/46
4.2.27 Variable Fr . . . . . AppA/46
4.2.28 Variable SUPRESSOR . . . . . AppA/46
4.2.29 Variable first_time . . . . . AppA/46
4.2.30 Variable Mode . . . . . AppA/46
4.2.31 Variable Old_Mode . . . . . AppA/47
4.2.32 Variable Command . . . . . AppA/47
4.2.33 Variable LEFT_VEL . . . . . AppA/47
4.2.34 Variable RIGHT_VEL . . . . . AppA/47
4.2.35 Variable Distance . . . . . AppA/47
4.2.36 Variable Move_distance . . . . . AppA/47
4.2.37 Variable Must_move . . . . . AppA/47
4.2.38 Variable Must_go_ahead . . . . . AppA/47
4.2.39 Variable Moving_forward . . . . . AppA/47
4.2.40 Variable Door_detected . . . . . AppA/47
4.2.41 Variable Phase_1 . . . . . AppA/47
4.2.42 Variable G . . . . . AppA/47
4.2.43 Variable Total_left_pulses . . . . . AppA/47
4.2.44 Variable Total_right_pulses . . . . . AppA/47
4.2.45 Variable P_T_I . . . . . AppA/48
4.2.46 Variable P_T_D . . . . . AppA/48
4.2.47 Variable Count_left_pulses . . . . . AppA/48
4.2.48 Variable Count_right_pulses . . . . . AppA/48
4.2.49 Variable X . . . . . AppA/48
4.2.50 Variable Y . . . . . AppA/48
4.2.51 Variable ANGLE . . . . . AppA/48
4.2.52 Variable POSX . . . . . AppA/48
4.2.53 Variable POSY . . . . . AppA/48
4.2.54 Variable posx_new . . . . . AppA/48
4.2.55 Variable posy_new . . . . . AppA/48
4.2.56 Variable ind_centre . . . . . AppA/48
4.2.57 Variable Num_beeps . . . . . AppA/48
4.2.58 Variable Interval . . . . . AppA/48
4.2.59 Variable Whisper_freq . . . . . AppA/49
4.2.60 Variable Serial_mode . . . . . AppA/49
4.2.61 Variable Rec_buff . . . . . AppA/49
4.2.62 Variable Commands . . . . . AppA/49
4.2.63 Variable Tr_buff . . . . . AppA/49
4.2.64 External Variables . . . . . AppA/49
4.2.65 Local Variables . . . . . AppA/49
4.3 Actual file listing . . . . . AppA/50
5 File rodney.c . . . . . AppA/57
5.1 Variables . . . . . AppA/57
5.1.1 Variable current_np . . . . . AppA/57
5.1.2 Variable Sh_buff . . . . . AppA/58

5.1.3 Variable fun . . . AppA/58
5.1.4 Variable Microswitches_code . . . AppA/58
5.1.5 Variable Microswitches_angle . . . AppA/58
5.1.6 Variable Display_code . . . AppA/58
5.1.7 Variable sonar_readings . . . AppA/58
5.1.8 Variable sonar_obstacle . . . AppA/58
5.1.9 Variable Son_buff . . . AppA/58
5.1.10 Variable Pos_or_buff . . . AppA/58
5.1.11 Variable Angle_to_rotate . . . AppA/58
5.1.12 Variable Rotation . . . AppA/59
5.1.13 Variable Must_rotate . . . AppA/59
5.1.14 Variable ang_wander . . . AppA/59
5.1.15 Variable Dist_wander . . . AppA/59
5.1.16 Variable X_final . . . AppA/59
5.1.17 Variable Y_final . . . AppA/59
5.1.18 Variable ANG_final . . . AppA/59
5.1.19 Variable f1 . . . AppA/59
5.1.20 Variable f2 . . . AppA/59
5.1.21 Variable f3 . . . AppA/59
5.1.22 Variable f4 . . . AppA/59
5.1.23 Variable f5 . . . AppA/60
5.1.24 Variable f6 . . . AppA/60
5.1.25 Variable Init_maneuver . . . AppA/60
5.1.26 Variable End_maneuver . . . AppA/60
5.1.27 Variable Fr . . . AppA/60
5.1.28 Variable SUPRESSOR . . . AppA/60
5.1.29 Variable first_time . . . AppA/60
5.1.30 Variable Mode . . . AppA/60
5.1.31 Variable Old_Mode . . . AppA/60
5.1.32 Variable Command . . . AppA/60
5.1.33 Variable LEFT_VEL . . . AppA/60
5.1.34 Variable RIGHT_VEL . . . AppA/61
5.1.35 Variable Distance . . . AppA/61
5.1.36 Variable Move_distance . . . AppA/61
5.1.37 Variable Must_move . . . AppA/61
5.1.38 Variable Must_go_ahead . . . AppA/61
5.1.39 Variable Moving_forward . . . AppA/61
5.1.40 Variable Door_detected . . . AppA/61
5.1.41 Variable Phase_1 . . . AppA/61
5.1.42 Variable G . . . AppA/61
5.1.43 Variable Total_left_pulses . . . AppA/61
5.1.44 Variable Total_right_pulses . . . AppA/61
5.1.45 Variable P_T_I . . . AppA/62
5.1.46 Variable P_T_D . . . AppA/62
5.1.47 Variable Count_left_pulses . . . AppA/62
5.1.48 Variable Count_right_pulses . . . AppA/62
5.1.49 Variable X . . . AppA/62
5.1.50 Variable Y . . . AppA/62
5.1.51 Variable ANGLE . . . AppA/62
5.1.52 Variable POSX . . . AppA/62
5.1.53 Variable POSY . . . AppA/62
5.1.54 Variable posx_new . . . AppA/62
5.1.55 Variable posy_new . . . AppA/62
5.1.56 Variable ind_centre . . . AppA/63
5.1.57 Variable Num_beeps . . . AppA/63
5.1.58 Variable Interval . . . AppA/63
5.1.59 Variable Whisper_freq . . . AppA/63
5.1.60 Variable Serial_mode . . . AppA/63



5.1.61 Variable Rec_buff . . . AppA/63
5.1.62 Variable Commands . . . AppA/63
5.1.63 Variable Tr_buff . . . AppA/63
5.1.64 Local Variables . . . AppA/63
5.2 Functions . . . AppA/63
5.2.1 Global Function COLISION_DETECTION() . . . AppA/63
5.2.2 Global Function C_pos() . . . AppA/64
5.2.3 Global Function C_pos_vel() . . . AppA/64
5.2.4 Global Function C_rot() . . . AppA/64
5.2.5 Global Function C_vel() . . . AppA/64
5.2.6 Global Function Close_all() . . . AppA/64
5.2.7 Global Function Consume() . . . AppA/64
5.2.8 Global Function Control() . . . AppA/64
5.2.9 Global Function Initialize() . . . AppA/64
5.2.10 Global Function InitializeMap() . . . AppA/64
5.2.11 Global Function InitializeSonars() . . . AppA/64
5.2.12 Global Function Initialize_all() . . . AppA/64
5.2.13 Global Function Latch() . . . AppA/64
5.2.14 Global Function Read_Velocity() . . . AppA/64
5.2.15 Global Function StopRobot() . . . AppA/64
5.2.16 Global Function access_to_ports() . . . AppA/64
5.2.17 Global Function argument_length() . . . AppA/64
5.2.18 Global Function beep() . . . AppA/65
5.2.19 Global Function end_tone() . . . AppA/65
5.2.20 Global Function hard_latch() . . . AppA/65
5.2.21 Global Function init_robot_hardware() . . . AppA/65
5.2.22 Global Function interval() . . . AppA/65
5.2.23 Global Function map() . . . AppA/65
5.2.24 Global Function ptrans() . . . AppA/65
5.2.25 Global Function read_byte_with_conf() . . . AppA/65
5.2.26 Global Function receive() . . . AppA/65
5.2.27 Global Function send() . . . AppA/65
5.2.28 Global Function serial() . . . AppA/65
5.2.29 Global Function set_freq() . . . AppA/65
5.2.30 Global Function sonars() . . . AppA/65
5.2.31 Global Function sound() . . . AppA/65
5.2.32 Global Function start_tone() . . . AppA/65
5.2.33 Global Function trans_ult_and_sh() . . . AppA/65
5.2.34 Global Function visualize() . . . AppA/66
5.2.35 Global Function whisper() . . . AppA/66
5.2.36 Global Function write_two_bytes() . . . AppA/66
5.3 Actual file listing . . . AppA/66
6 File cases.c . . . AppA/95
6.1 Variables . . . AppA/96
6.1.1 Variable current_np . . . AppA/96
6.1.2 Variable Sh_buff . . . AppA/96
6.1.3 Variable fun . . . AppA/96
6.2 Functions . . . AppA/96
6.2.1 Global Function ADVANCE() . . . AppA/96
6.2.2 Global Function MOVE() . . . AppA/96
6.2.3 Global Function ROTATE() . . . AppA/96
6.2.4 Global Function TEST() . . . AppA/96
6.2.5 Global Function init_architecture() . . . AppA/96
6.3 Actual file listing . . . AppA/97
7 File reactive1.c . . . AppA/100
7.1 Variables . . . AppA/101
7.1.1 Variable current_np . . . AppA/101
7.1.2 Variable Sh_buff . . . AppA/101

7.1.3 Variable fun . . . AppA/101
7.2 Functions . . . AppA/101
7.2.1 Global Function AVOID() . . . AppA/101
7.2.2 Global Function FEELFORCE() . . . AppA/101
7.2.3 Global Function RUNAWAY() . . . AppA/101
7.2.4 Global Function WANDER() . . . AppA/101
7.2.5 Global Function init_architecture() . . . AppA/101
7.3 Actual file listing . . . AppA/101
8 File teleo_reactive.c . . . AppA/106
8.1 Variables . . . AppA/106
8.1.1 Variable current_np . . . AppA/106
8.1.2 Variable Sh_buff . . . AppA/106
8.1.3 Variable fun . . . AppA/106
8.1.4 Variable Condition1 . . . AppA/107
8.1.5 Variable Condition2 . . . AppA/107
8.1.6 Variable Condition3 . . . AppA/107
8.2 Functions . . . AppA/107
8.2.1 Global Function ADVANCE1() . . . AppA/107
8.2.2 Global Function RECALCULATE() . . . AppA/107
8.2.3 Global Function ROTATE1() . . . AppA/107
8.2.4 Global Function TELEO() . . . AppA/107
8.2.5 Global Function init_architecture() . . . AppA/107
8.3 Actual file listing . . . AppA/107
9 File ofg.c . . . AppA/109
9.1 Functions . . . AppA/111
9.1.1 Global Function box_ov() . . . AppA/111
9.1.2 Global Function closed_pol() . . . AppA/111
9.1.3 Global Function dot() . . . AppA/111
9.1.4 Global Function dot_ov() . . . AppA/111
9.1.5 Global Function draw_arena() . . . AppA/111
9.1.6 Global Function draw_little_robot() . . . AppA/111
9.1.7 Global Function draw_robot() . . . AppA/111
9.1.8 Global Function draw_sh_boxes() . . . AppA/111
9.1.9 Global Function grab() . . . AppA/111
9.1.10 Global Function hbar_ov() . . . AppA/111
9.1.11 Global Function init_ofg() . . . AppA/111
9.1.12 Global Function init_ofg_ov() . . . AppA/111
9.1.13 Global Function lee_pix() . . . AppA/111
9.1.14 Global Function line() . . . AppA/111
9.1.15 Global Function snap() . . . AppA/111
9.2 Actual file listing . . . AppA/112
10 File monitor.c . . . AppA/120
10.1 Type definitions . . . AppA/122
10.1.1 Typedef com_st . . . AppA/122
10.2 Variables . . . AppA/122
10.2.1 Variable deb . . . AppA/122
10.2.2 Variable quit . . . AppA/122
10.2.3 Variable show . . . AppA/122
10.2.4 Variable serial_locked . . . AppA/122
10.2.5 Variable rec . . . AppA/122
10.2.6 Variable trans . . . AppA/122
10.2.7 Variable command_names . . . AppA/123
10.2.8 Variable in_pos_or . . . AppA/123
10.2.9 Variable last_sending_time . . . AppA/123
10.3 Functions . . . AppA/123
10.3.1 Global Function access_to_ports() . . . AppA/123
10.3.2 Global Function analyze() . . . AppA/123
10.3.3 Global Function box_ov() . . . AppA/123





10.3.4 Global Function closed_pol() . . . AppA/123
10.3.5 Global Function dot() . . . AppA/123
10.3.6 Global Function dot_ov() . . . AppA/123
10.3.7 Global Function draw_arena() . . . AppA/123
10.3.8 Global Function draw_little_robot() . . . AppA/123
10.3.9 Global Function draw_robot() . . . AppA/123
10.3.10 Global Function draw_sh_boxes() . . . AppA/123
10.3.11 Global Function fill_bar() . . . AppA/124
10.3.12 Global Function fill_command_names() . . . AppA/124
10.3.13 Global Function get_command_code() . . . AppA/124
10.3.14 Global Function grab() . . . AppA/124
10.3.15 Global Function has_to_be_sent() . . . AppA/124
10.3.16 Global Function hbar_ov() . . . AppA/124
10.3.17 Global Function help_on_command() . . . AppA/124
10.3.18 Global Function init_ofg() . . . AppA/124
10.3.19 Global Function init_ofg_ov() . . . AppA/124
10.3.20 Global Function is_in() . . . AppA/124
10.3.21 Global Function lee_pix() . . . AppA/124
10.3.22 Global Function line() . . . AppA/124
10.3.23 Global Function main() . . . AppA/124
10.3.24 Global Function parse() . . . AppA/124
10.3.25 Global Function process_incoming_byte() . . . AppA/125
10.3.26 Global Function random_update() . . . AppA/125
10.3.27 Global Function read_byte() . . . AppA/125
10.3.28 Global Function read_two_bytes() . . . AppA/125
10.3.29 Global Function rec_error() . . . AppA/125
10.3.30 Global Function rec_loop() . . . AppA/125
10.3.31 Global Function send_com() . . . AppA/125
10.3.32 Global Function send_command() . . . AppA/125
10.3.33 Global Function set_show() . . . AppA/125
10.3.34 Global Function show_help() . . . AppA/125
10.3.35 Global Function show_pos_or() . . . AppA/125
10.3.36 Global Function snap() . . . AppA/125
10.3.37 Global Function update_pos_or_rep() . . . AppA/125
10.3.38 Global Function update_sharing_rep() . . . AppA/125
10.3.39 Global Function update_ult_rep() . . . AppA/125
10.3.40 Global Function wait_until_ready() . . . AppA/126
10.3.41 Global Function write_byte_and_wait_conf() . . . AppA/126
10.3.42 Global Function write_packet() . . . AppA/126
10.4 Actual file listing . . . AppA/126

# 1 File common.h

Types

## Included Files

```
#include </usr/include/stdio.h>
#include </usr/include/stdlib.h>
#include </usr/include/math.h>
#include </usr/include/errno.h>
#include </usr/include/sys/perm.h>
#include </usr/include/asm/io.h>
#include </usr/include/sys/time.h>
#include </usr/include/sys/resource.h>
#include </usr/include/signal.h>
#include </usr/include/sched.h>
#include </usr/include/pthread.h>
```

## Preprocessor definitions

```
#define COMMON
#define _REENTRANT
#define __REENTRANT
#define YES 1
#define NO 0
#define MAX_PROC 16
```

Macro for friendly use of the former function. It HAS to be used with a long long argument

```
#define GET_CPUCTR( x )
```

See the meaning of these constants in the proc_info structure

```
#define NO_SHARING 1
#define MODIFIABLE 1
#define NOT_MODIFIABLE 0
#define MAX_SH_PROC 16
#define NUM_BASIC_PROCESSES 8
#define NAME_NORMAL "normal"
```

Constants to improve readibility

```
#define CURRENT_SH 0
```



```
#define DESIRED_SH 1
```

This constant must be known by the host (monitor) program

```
#define SHARING_TRANSMIT_CODE 253
#define NO_MORE_SHARINGS 101
#define MAX_SH 102
```

SONS_START_CODE must be invoked by all sons, inmediately after their local variable declaration section. This will be done, in turn, by BEGIN_BEH_MOD. This macro stops teh process until the scheduler allows it to start. There are two variants of this: the normal one, and the PORT, which is called by the processes that have to ask for permissions to gain access to the I/O ports.

```
#define SONS_START_CODE
#define BEGIN_BEH_MOD
#define BEGIN_BEH_MOD_P
#define END_BEH_MOD
#define END_BEH_MOD_P END_BEH_MOD
```

ONE_SHOT_WAITS_HERE is called by processes (mostly, of unbounded time) that have to be executed exactly one per sched cycle

```
#define ONE_SHOT_WAITS_HERE
```

SONS_CAN_START is called by the scheduler thread, only once, to say all sons they can proceed, since all their brothers have been registered and are ready.

```
#define SONS_CAN_START
```

ONE_SHOT_MAY_GO_ON is called by the pseudo_scheduler after any unbounded time processes has been executed. This will allow it (or any other of this kind) to run again when its turn arrives, in the next cycle.

```
#define ONE_SHOT_MAY_GO_ON
```

## 1.1 Type definitions

### 1.1.1 Typedef proc_info

This structure contains the information about each process, except the pseudo-scheduler, that does not need it. NOT_MODIFIABLE: Other threads can't do it.

```
typedef struct {...} proc_info
struct
  {
    pthread_t pth;                      The identifier of the process as a thread
    int nproc;                          The number of threads.
    unsigned long allowed_time;         For how long the process runs in each
    float sharing[2];                   The proportion of the total time that this
    int modifiable;                     Flag to state if the sharing of the process
  }
```

## 1.2 Variables

### 1.2.1 Variable current_np

The current number of registered processes

```
int current_np
```

### 1.2.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

```
unsigned char Sh_buff[16+2]
```

### 1.2.3 Variable fun

```
void* (*fun[16])(void*)
```

## 1.3 Actual file listing

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>
#include <sys/perm.h>
#include <asm/io.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <signal.h>
#include <time.h>
#include <sched.h>
#include <pthread.h>

#define COMMON

#define _REENTRANT
#define __REENTRANT

#define YES 1
#define NO  0

#define MAX_PROC 16  /*+ The maximum number of son threads that can exist +*/

/*+ Function to read tick count register to obtain accurate time measures
Works only on Pentiums +*/
void get_cpuctr(unsigned long *,unsigned long *);

/*+ Macro for friendly use of the former function.
 It HAS to be used with a long long argument    +*/
#define GET_CPUCTR(x) get_cpuctr(((unsigned long *)(&x)),((unsigned long *)(&x))+1)

/*+ See the meaning of these constants in the proc_info structure +*/
#define NO_SHARING    -1
#define MODIFIABLE     1
#define NOT_MODIFIABLE 0
#define MAX_SH_PROC   16

#define NUM_BASIC_PROCESSES 8
#define NAME_NORMAL "normal"

/*+ Constants to improve readibility +*/

#define CURRENT_SH 0
#define DESIRED_SH 1

/*+ This constant must be known by the host (monitor) program +*/
#define SHARING_TRANSMIT_CODE 253
#define NO_MORE_SHARINGS      101
#define MAX_SH                102

/*+ This structure contains the information about each process, except
 the pseudo-scheduler, that does not need it.                        +*/
```



```
typedef struct
 {
  pthread_t pth;               /*+ The identifier of the process as a thread +*/
  int nproc;                   /*+ The number of threads. +*/
  unsigned long allowed_time;      /*+ For how long the process runs in each +*/
                                   /*+ scheduler turn, in ns +*/
  float sharing[2];          /*+ The proportion of the total time that this  +*/
                             /*+ thread can spend. sharing[0] is the current +*/
                             /*+ one, whereas sharing[1] contains that which +*/
                             /*+ another thread wants for it in the next turn+*/
  int modifiable;            /*+ Flag to state if the sharing of the process +*/
                             /*+ process can be modified by others. Three    +*/
                             /*+ values are allowed:                         +*/
               /*+ NO_SHARING: this process must spend all the +*/
                             /*+              time it needs, and this cannot  +*/
    /*+             be altered +*/
           /*+ MODIFIABLE: Other threads can change this   +*/
                             /*+               process' sharing              +*/
           /*+ NOT_MODIFIABLE: Other threads can't do it.  +*/
 } proc_info;

/*+ The current number of registered processes +*/
int current_np;

/*+ Transmission buffer for sharings must be known by ps_sched, which
  updates it every scheduler cycle, and by the robot process that send it
  through the serial port.                                              +*/
unsigned char Sh_buff[MAX_PROC+2];

/*+ Macros with code that involves the use of global variables.          +*/

/*+ SONS_START_CODE must be invoked by all sons, inmediately after their
local variable declaration section. This will be done, in turn, by
BEGIN_BEH_MOD. This macro stops teh process until the scheduler allows
it to start. There are two variants of this: the normal one, and the
PORT, which is called by the processes that have to ask for permissions
to gain access to the I/O ports.
+*/

#define SONS_START_CODE \
 static int first_time=1,indep_thread_process,stop=0; \
 proc_info *pr; \
 pr=(proc_info *)pass; \
 indep_thread_process=(sch_t!=pthread_self()); \
 if ((indep_thread_process) && (first_time)) \
 { \
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,NULL); \
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL); \
  pthread_mutex_lock(&mut); \
  pthread_cond_wait(&cond,&mut); \
  pthread_mutex_unlock(&mut); \
 } \

#define BEGIN_BEH_MOD \
 SONS_START_CODE \
 if (first_time) \
  first_time=0; \
 do
```

```
#define BEGIN_BEH_MOD_P \
 SONS_START_CODE \
 if (first_time) \
 { \
  access_to_ports(); \
  first_time=0; \
 } \
 do

#define END_BEH_MOD \
 while ((indep_thread_process) && (!stop)); \
 if (!indep_thread_process) \
  return(( stop ? (void *)1 : (void *)0 )); \
 else \
  pthread_exit(( stop ? (void *)1 : (void *)0 ));

#define END_BEH_MOD_P END_BEH_MOD

/*+ ONE_SHOT_WAITS_HERE is called by processes (mostly, of unbounded time)
that have to be executed exactly one per sched cycle                +*/
#define ONE_SHOT_WAITS_HERE \
 pthread_mutex_lock(&mshot); \
 pthread_cond_wait(&cshot,&mshot); \
 pthread_mutex_unlock(&mshot)

/*+ SONS_CAN_START is called by the scheduler thread, only once, to say all
 sons they can proceed, since all their brothers have been registered
 and are ready.   +*/
#define SONS_CAN_START \
 pthread_mutex_lock(&mut); \
 pthread_cond_broadcast(&cond); \
 pthread_mutex_unlock(&mut)

/*+ ONE_SHOT_MAY_GO_ON is called by the pseudo_scheduler after any unbounded
 time processes has been executed. This will allow it (or any other of
 this kind) to run again when its turn arrives, in the next cycle.        +*/
#define ONE_SHOT_MAY_GO_ON \
 pthread_mutex_lock(&mshot); \
 pthread_cond_signal(&cshot); \
 pthread_mutex_unlock(&mshot)

/*+ This function has to be written in the application program, but called
 by main in the scheduler. It is the only one known by both modules.    +*/
int Initialize_all(int init_basics);

/*+ This global variable is the array of functions containing the addresses
 of the actual code that each thread must execute when allowed to do so
 by the pseudo-scheduler.                                               +*/
void *(*fun[MAX_PROC])(void *);
```

# 2 File ps_sched.h

Types



## Included Files

```
#include "common.h"                                        (Section 1)
   #include </usr/include/stdio.h>
   #include </usr/include/stdlib.h>
   #include </usr/include/math.h>
   #include </usr/include/errno.h>
   #include </usr/include/sys/perm.h>
   #include </usr/include/asm/io.h>
   #include </usr/include/sys/time.h>
   #include </usr/include/sys/resource.h>
   #include </usr/include/signal.h>
   #include </usr/include/sched.h>
   #include </usr/include/pthread.h>
```

## Preprocessor definitions

`#define PS_SCHED`

`#define STOP_TO_SEE_SHARINGS YES`

`#define MAX_LOOPS 10000`

`#define HZ_NOW 6579`

`#define SCHED_PERIOD`

`#define PS_SCHED_PERIOD`

15.2 miliseconds aprox. now

`#define STACK_SIZE`

Some constants and macros to improve readibility

`#define PARENT 0`

`#define SON 1`

`#define SPECIAL_IDLE 2`

`#define HIGH_PR 18`

`#define MEDIUM_PR 9`

`#define LOW_PR 3`

`#define NO_ONE 1`

`#define POL_NAME( p )`

## 2.1 Variables

### 2.1.1 Variable current_np

The current number of registered processes

`int current_np`

Inc. from: common.h *(Section 1.2.1)*

### 2.1.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

`unsigned char Sh_buff[16+2]`

Inc. from: common.h *(Section 1.2.2)*

### 2.1.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h *(Section 1.2.3)*

### 2.1.4 Variable mut

Mutual exclusion flags and conditions that are used by the scheduler and by the son threads.

`pthread_mutex_t mut`

### 2.1.5 Variable cond

`pthread_cond_t cond`

### 2.1.6 Variable mshot

`pthread_mutex_t mshot`

### 2.1.7 Variable cshot

`pthread_cond_t cshot`

### 2.1.8 Variable bcont_mut

`pthread_mutex_t bcont_mut`

### 2.1.9 Variable sch_t

The thread identifier of the scheduler

`pthread_t sch_t`

## 2.2 Actual file listing

```
#define PS_SCHED

#ifndef COMMON
#include "common.h"
#else
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/perm.h>
#include <asm/io.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <signal.h>
#include <time.h>
#include <sched.h>
#include <pthread.h>   /*+ Needed by the thread library +*/
#endif

#define STOP_TO_SEE_SHARINGS YES

/* To be elliminated in the future. From here... */
#define MAX_LOOPS 10000
#define HZ_NOW            6579  /* It should coincide with the value the */
                                /* constant HZ in /usr/include/asm/param.h */
                                /* had when the running kernel was compiled. */
#define SCHED_PERIOD       ((int)(1000000000.0/HZ_NOW)) /* in nanoseconds. */
```



```
                                                   /* To be found from the kernel */
  /* 152 microseconds aprox. now */
/* ... to here. */

#define PS_SCHED_PERIOD   100*SCHED_PERIOD /*+ Time for a total pseudo-scheduler loop +*/
                                           /*+ 15.2 miliseconds aprox. now +*/

#define STACK_SIZE 16*1024  /*+ The stack size of EACH son process. +*/

/*+ Some constants and macros to improve readibility +*/

#define PARENT       0
#define SON          1
#define SPECIAL_IDLE 2

#define HIGH_PR    18
#define MEDIUM_PR  9
#define LOW_PR     3

#define NO_ONE -1

#define POL_NAME(p) (p==SCHED_OTHER ? "SCHED_OTHER" : \
                    (p==SCHED_FIFO ? "SCHED_FIFO" : \
                    (p==SCHED_RR ? "SCHED_RR" : "UNKNOWN")))

/*+ Mutual exclusion flags and conditions that are used by the scheduler
 and by the son threads.                                              +*/
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mshot=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cshot=PTHREAD_COND_INITIALIZER;
pthread_mutex_t bcont_mut=PTHREAD_MUTEX_INITIALIZER;
/*+ The thread identifier of the scheduler                            +*/
pthread_t sch_t;

/*+ Prototypes of the functions in this module +*/
void set_pr(pthread_t,int);
int get_pr(pthread_t);
char *get_sc_mode(pthread_t);
void *my_idle(void *);
int TickSleep(const int *,int *);
void usage(char *);
int check_args(int,char *[],proc_info[MAX_PROC]);
int touch_sharings(proc_info[MAX_PROC],int);
int set_attributes(pthread_attr_t *,int);
void pseudo_sched(proc_info[MAX_PROC],int,void *);
void *prepare_sched(void *);
```

# 3 File ps_sched.c

Types

## Included Files

`#include </usr/include/stdio.h>`

`#include </usr/include/stdlib.h>`

`#include </usr/include/errno.h>`

```
#include </usr/include/sys/perm.h>
#include </usr/include/asm/io.h>
#include </usr/include/sys/time.h>
#include </usr/include/sys/resource.h>
#include </usr/include/signal.h>
#include </usr/include/sched.h>
#include </usr/include/pthread.h>
#include "common.h"                    (Section 1)
   #include </usr/include/math.h>
#include "ps_sched.h"                  (Section 2)
#include "rodney.c"                    (Section 5)
   #include "rodney.h"                 (Section 4)
#include "cases.c"                     (Section 6)
```

## 3.1 Variables

### 3.1.1 Variable current_np

The current number of registered processes

`int current_np`

Inc. from: common.h *(Section 1.2.1)*

### 3.1.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

`unsigned char Sh_buff[16+2]`

Inc. from: common.h *(Section 1.2.2)*

### 3.1.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h *(Section 1.2.3)*

### 3.1.4 Variable mut

Mutual exclusion flags and conditions that are used by the scheduler and by the son threads.

`pthread_mutex_t mut`

Inc. from: ps_sched.h *(Section 2.1.4)*

### 3.1.5 Variable cond

`pthread_cond_t cond`

Inc. from: ps_sched.h *(Section 2.1.5)*

### 3.1.6 Variable mshot

`pthread_mutex_t mshot`

Inc. from: ps_sched.h *(Section 2.1.6)*

### 3.1.7 Variable cshot

`pthread_cond_t cshot`

Inc. from: ps_sched.h *(Section 2.1.7)*



### 3.1.8 Variable bcont_mut

`pthread_mutex_t bcont_mut`

Inc. from: ps_sched.h *(Section 2.1.8)*

### 3.1.9 Variable sch_t

The thread identifier of the scheduler

`pthread_t sch_t`

Inc. from: ps_sched.h *(Section 2.1.9)*

### 3.1.10 Variable Microswitches_code

Microswitch control variable

`unsigned char Microswitches_code`

Inc. from: rodney.h *(Section 4.2.4)*

### 3.1.11 Variable Microswitches_angle

`float Microswitches_angle`

Inc. from: rodney.h *(Section 4.2.5)*

### 3.1.12 Variable Display_code

Display control variable

`unsigned char Display_code`

Inc. from: rodney.h *(Section 4.2.6)*

### 3.1.13 Variable sonar_readings

Raw sonar readings

`float sonar_readings[8]`

Inc. from: rodney.h *(Section 4.2.7)*

### 3.1.14 Variable sonar_obstacle

Sonar obstacle detection

`unsigned char sonar_obstacle[8]`

Inc. from: rodney.h *(Section 4.2.8)*

### 3.1.15 Variable Son_buff

The sonar transmission buffer

`unsigned char Son_buff[8+1]`

Inc. from: rodney.h *(Section 4.2.9)*

### 3.1.16 Variable Pos_or_buff

The pos-or transmission buffer

`float Pos_or_buff[4]`

Inc. from: rodney.h *(Section 4.2.10)*

### 3.1.17 Variable Angle_to_rotate

Angle to rotate (in radians)

`double Angle_to_rotate`

Inc. from: rodney.h *(Section 4.2.11)*

### 3.1.18 Variable Rotation

`int Rotation`

Inc. from: rodney.h *(Section 4.2.12)*

### 3.1.19 Variable Must_rotate

`int Must_rotate`

Inc. from: rodney.h *(Section 4.2.13)*

### 3.1.20 Variable ang_wander

Variables for the procedure to go to a final location

`double ang_wander`

Inc. from: rodney.h *(Section 4.2.14)*

### 3.1.21 Variable Dist_wander

`float Dist_wander`

Inc. from: rodney.h *(Section 4.2.15)*

### 3.1.22 Variable X_final

`long X_final`

Inc. from: rodney.h *(Section 4.2.16)*

### 3.1.23 Variable Y_final

`long Y_final`

Inc. from: rodney.h *(Section 4.2.17)*

### 3.1.24 Variable ANG_final

`double ANG_final`

Inc. from: rodney.h *(Section 4.2.18)*

### 3.1.25 Variable f1

`int f1`

Inc. from: rodney.h *(Section 4.2.19)*

### 3.1.26 Variable f2

`int f2`

Inc. from: rodney.h *(Section 4.2.20)*

### 3.1.27 Variable f3

`int f3`

Inc. from: rodney.h *(Section 4.2.21)*

### 3.1.28 Variable f4

`int f4`

Inc. from: rodney.h *(Section 4.2.22)*



### 3.1.29 Variable f5

`int f5`

Inc. from: rodney.h *(Section 4.2.23)*

### 3.1.30 Variable f6

`int f6`

Inc. from: rodney.h *(Section 4.2.24)*

### 3.1.31 Variable Init_maneuver

`int Init_maneuver`

Inc. from: rodney.h *(Section 4.2.25)*

### 3.1.32 Variable End_maneuver

`int End_maneuver`

Inc. from: rodney.h *(Section 4.2.26)*

### 3.1.33 Variable Fr

`float Fr`

Inc. from: rodney.h *(Section 4.2.27)*

### 3.1.34 Variable SUPRESSOR

`int SUPRESSOR`

Inc. from: rodney.h *(Section 4.2.28)*

### 3.1.35 Variable first_time

`int first_time`

Inc. from: rodney.h *(Section 4.2.29)*

### 3.1.36 Variable Mode

Control variables Desired control mode

`int Mode`

Inc. from: rodney.h *(Section 4.2.30)*

### 3.1.37 Variable Old_Mode

`int Old_Mode`

Inc. from: rodney.h *(Section 4.2.31)*

### 3.1.38 Variable Command

`int Command`

Inc. from: rodney.h *(Section 4.2.32)*

### 3.1.39 Variable LEFT_VEL

Desired velocities in pulses/T

`unsigned int LEFT_VEL`

Inc. from: rodney.h *(Section 4.2.33)*

### 3.1.40 Variable RIGHT_VEL

`unsigned int RIGHT_VEL`

Inc. from: rodney.h *(Section 4.2.34)*

### 3.1.41 Variable Distance

Desired distance (in mm.)

`long Distance`

Inc. from: rodney.h *(Section 4.2.35)*

### 3.1.42 Variable Move_distance

`int Move_distance`

Inc. from: rodney.h *(Section 4.2.36)*

### 3.1.43 Variable Must_move

`int Must_move`

Inc. from: rodney.h *(Section 4.2.37)*

### 3.1.44 Variable Must_go_ahead

`int Must_go_ahead`

Inc. from: rodney.h *(Section 4.2.38)*

### 3.1.45 Variable Moving_forward

`int Moving_forward`

Inc. from: rodney.h *(Section 4.2.39)*

### 3.1.46 Variable Door_detected

`int Door_detected`

Inc. from: rodney.h *(Section 4.2.40)*

### 3.1.47 Variable Phase_1

`int Phase_1`

Inc. from: rodney.h *(Section 4.2.41)*

### 3.1.48 Variable G

Value used by the rotation module

`int G`

Inc. from: rodney.h *(Section 4.2.42)*

### 3.1.49 Variable Total_left_pulses

Pulses counted up to now

`long Total_left_pulses`

Inc. from: rodney.h *(Section 4.2.43)*

### 3.1.50 Variable Total_right_pulses

`long Total_right_pulses`

Inc. from: rodney.h *(Section 4.2.44)*



### 3.1.51 Variable P_T_I

`long P_T_I`

Inc. from: rodney.h *(Section 4.2.45)*

### 3.1.52 Variable P_T_D

`long P_T_D`

Inc. from: rodney.h *(Section 4.2.46)*

### 3.1.53 Variable Count_left_pulses

Pulses desired for doing a certain movement

`long Count_left_pulses`

Inc. from: rodney.h *(Section 4.2.47)*

### 3.1.54 Variable Count_right_pulses

`long Count_right_pulses`

Inc. from: rodney.h *(Section 4.2.48)*

### 3.1.55 Variable X

Position/orientation of the robot over the workarea (mm. and radians, respectively)

`double X`

Inc. from: rodney.h *(Section 4.2.49)*

### 3.1.56 Variable Y

`double Y`

Inc. from: rodney.h *(Section 4.2.50)*

### 3.1.57 Variable ANGLE

`double ANGLE`

Inc. from: rodney.h *(Section 4.2.51)*

### 3.1.58 Variable POSX

Robot position on the grid cell map

`int POSX`

Inc. from: rodney.h *(Section 4.2.52)*

### 3.1.59 Variable POSY

`int POSY`

Inc. from: rodney.h *(Section 4.2.53)*

### 3.1.60 Variable posx_new

`int posx_new`

Inc. from: rodney.h *(Section 4.2.54)*

### 3.1.61 Variable posy_new

`int posy_new`

Inc. from: rodney.h *(Section 4.2.55)*

#### 3.1.62 Variable ind_centre

`int ind_centre`

Inc. from: rodney.h *(Section 4.2.56)*

#### 3.1.63 Variable Num_beeps

Sound control variables. Any process can change them. Number of beeps we want to emit.

`int Num_beeps`

Inc. from: rodney.h *(Section 4.2.57)*

#### 3.1.64 Variable Interval

Entonation interval; 0 to 7 for ascending, -1 to -7 for descending

`int Interval`

Inc. from: rodney.h *(Section 4.2.58)*

#### 3.1.65 Variable Whisper_freq

Frequency for the whisper

`unsigned int Whisper_freq`

Inc. from: rodney.h *(Section 4.2.59)*

#### 3.1.66 Variable Serial_mode

Serial transmission control variable Possible values are READING, RECEIVING and SENDING

`int Serial_mode`

Inc. from: rodney.h *(Section 4.2.60)*

#### 3.1.67 Variable Rec_buff

The three serial buffers: received binary packet, received commands and bytes to transmit.

`unsigned char Rec_buff[254+1]`

Inc. from: rodney.h *(Section 4.2.61)*

#### 3.1.68 Variable Commands

The three serial buffers: received binary packet, received commands and bytes to transmit.

`unsigned char Commands[254+1]`

Inc. from: rodney.h *(Section 4.2.62)*

#### 3.1.69 Variable Tr_buff

The three serial buffers: received binary packet, received commands and bytes to transmit.

`unsigned char Tr_buff[254+1]`

Inc. from: rodney.h *(Section 4.2.63)*

#### 3.1.70 Local Variables

**cell**
The cell struct

`static struct C cell[300][300/4]`

Inc. from: rodney.h *(Section 4.2.65)*



### 3.2 Functions

#### 3.2.1 Global Function ADVANCE()

`void* ADVANCE ( void* pass )`

Inc. from: cases.c *(Section 6.2.1)*

#### 3.2.2 Global Function COLISION_DETECTION()

`void* COLISION_DETECTION ( void* pass )`

Inc. from: rodney.c *(Section 5.2.1)*

#### 3.2.3 Global Function C_pos()

`void C_pos ( void )`

Inc. from: rodney.c *(Section 5.2.2)*

#### 3.2.4 Global Function C_pos_vel()

`void C_pos_vel ( void )`

Inc. from: rodney.c *(Section 5.2.3)*

#### 3.2.5 Global Function C_rot()

`void C_rot ( void )`

Inc. from: rodney.c *(Section 5.2.4)*

#### 3.2.6 Global Function C_vel()

`void C_vel ( void )`

Inc. from: rodney.c *(Section 5.2.5)*

#### 3.2.7 Global Function Close_all()

`void* Close_all ( void* pass )`

Inc. from: rodney.c *(Section 5.2.6)*

#### 3.2.8 Global Function Consume()

`int Consume ( unsigned char commands[254+1] )`

Inc. from: rodney.c *(Section 5.2.7)*

#### 3.2.9 Global Function Control()

`void Control ( void )`

Inc. from: rodney.c *(Section 5.2.8)*

#### 3.2.10 Global Function Initialize()

`int Initialize ( int encoder )`

Inc. from: rodney.c *(Section 5.2.9)*

#### 3.2.11 Global Function InitializeMap()

`void InitializeMap ( void )`

Inc. from: rodney.c *(Section 5.2.10)*

### 3.2.12 Global Function InitializeSonars()

`void InitializeSonars ( void )`

Inc. from: rodney.c *(Section 5.2.11)*

### 3.2.13 Global Function Initialize_all()

`int Initialize_all ( int init_basic_processes )`

Inc. from: rodney.c *(Section 5.2.12)*

### 3.2.14 Global Function Latch()

`void Latch ( int encoder )`

Inc. from: rodney.c *(Section 5.2.13)*

### 3.2.15 Global Function MOVE()

`void* MOVE ( void* pass )`

Inc. from: cases.c *(Section 6.2.2)*

### 3.2.16 Global Function ROTATE()

`void* ROTATE ( void* pass )`

Inc. from: cases.c *(Section 6.2.3)*

### 3.2.17 Global Function Read_Velocity()

`long Read_Velocity ( int encoder )`

Inc. from: rodney.c *(Section 5.2.14)*

### 3.2.18 Global Function StopRobot()

`void StopRobot ( void )`

Inc. from: rodney.c *(Section 5.2.15)*

### 3.2.19 Global Function TEST()

`void* TEST ( void* pass )`

Inc. from: cases.c *(Section 6.2.4)*

### 3.2.20 Global Function TickSleep()

`int TickSleep ( const int* req, int* rem )`

### 3.2.21 Global Function access_to_ports()

`int access_to_ports ( void )`

Inc. from: rodney.c *(Section 5.2.16)*

### 3.2.22 Global Function argument_length()

`int argument_length ( unsigned char command_code )`

Inc. from: rodney.c *(Section 5.2.17)*

### 3.2.23 Global Function beep()

`void beep ( void )`

Inc. from: rodney.c *(Section 5.2.18)*



### 3.2.24 Global Function check_args()

`int check_args ( int argc, char* argv[], proc_info pr[16] )`

### 3.2.25 Global Function end_tone()

`void end_tone ( void )`

Inc. from: rodney.c *(Section 5.2.19)*

### 3.2.26 Global Function get_pr()

`int get_pr ( pthread_t pt )`

### 3.2.27 Global Function get_sc_mode()

`char* get_sc_mode ( pthread_t pt )`

### 3.2.28 Global Function hard_latch()

`void hard_latch ( void )`

Inc. from: rodney.c *(Section 5.2.20)*

### 3.2.29 Global Function init_architecture()

`void init_architecture ( int basics_loaded )`

Inc. from: cases.c *(Section 6.2.5)*

### 3.2.30 Global Function init_robot_hardware()

`void init_robot_hardware ( void )`

Inc. from: rodney.c *(Section 5.2.21)*

### 3.2.31 Global Function interval()

`void interval ( void )`

Inc. from: rodney.c *(Section 5.2.22)*

### 3.2.32 Global Function main()

`int main ( int argc, char* argv[] )`

### 3.2.33 Global Function map()

`void* map ( void* pass )`

Inc. from: rodney.c *(Section 5.2.23)*

### 3.2.34 Global Function my_idle()

and avoid the accomplishing of the deadline. See comments in prepare_scheduler

`void* my_idle ( void* pass )`

### 3.2.35 Global Function prepare_scheduler()

`void* prepare_scheduler ( void* pass )`

### 3.2.36 Global Function pseudo_sched()

The real core of the algorithm.

`void pseudo_sched ( proc_info pr[16], int nproc, void* pass )`

### 3.2.37 Global Function ptrans()

```
void* ptrans ( void* pass )
```
Inc. from: rodney.c *(Section 5.2.24)*

### 3.2.38 Global Function read_byte_with_conf()

```
int read_byte_with_conf ( unsigned char* pc )
```
Inc. from: rodney.c *(Section 5.2.25)*

### 3.2.39 Global Function receive()

```
int receive ( void )
```
Inc. from: rodney.c *(Section 5.2.26)*

### 3.2.40 Global Function send()

```
int send ( void )
```
Inc. from: rodney.c *(Section 5.2.27)*

### 3.2.41 Global Function serial()

```
void* serial ( void* pass )
```
Inc. from: rodney.c *(Section 5.2.28)*

### 3.2.42 Global Function set_attributes()

```
int set_attributes ( pthread_attr_t* pt, int who )
```

### 3.2.43 Global Function set_freq()

```
void set_freq ( char note )
```
Inc. from: rodney.c *(Section 5.2.29)*

### 3.2.44 Global Function set_pr()

```
void set_pr ( pthread_t pt, int prio )
```

### 3.2.45 Global Function sonars()

```
void* sonars ( void* pass )
```
Inc. from: rodney.c *(Section 5.2.30)*

### 3.2.46 Global Function sound()

```
void* sound ( void* pass )
```
Inc. from: rodney.c *(Section 5.2.31)*

### 3.2.47 Global Function start_tone()

```
void start_tone ( void )
```
Inc. from: rodney.c *(Section 5.2.32)*

### 3.2.48 Global Function touch_sharings()

```
int touch_sharings ( proc_info pr[16], int nproc )
```



### 3.2.49 Global Function trans_ult_and_sh()

```
void* trans_ult_and_sh ( void* pass )
```
Inc. from: rodney.c *(Section 5.2.33)*

### 3.2.50 Global Function usage()

```
void usage ( char* pr_name )
```

### 3.2.51 Global Function visualize()

```
void* visualize ( void* pass )
```
Inc. from: rodney.c *(Section 5.2.34)*

### 3.2.52 Global Function whisper()

```
void whisper ( void )
```
Inc. from: rodney.c *(Section 5.2.35)*

### 3.2.53 Global Function write_two_bytes()

```
int write_two_bytes ( unsigned char c, unsigned char c2 )
```
Inc. from: rodney.c *(Section 5.2.36)*

## 3.3 Actual file listing

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/perm.h>
#include <asm/io.h>

#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <signal.h>
#include <time.h>

#include <sched.h>

#include <pthread.h>   /*+ Needed by the thread library +*/

#ifndef COMMON
#include "common.h"
#endif

#include "ps_sched.h"

#include "rodney.c"

/* Here, the file with the functions for the users' defined architecture */
/* must be included. When writing these file, remember that the function  */
/* for initializing the function array must be called 'init_architecture' */

#include "cases.c"
/* #include "reactive1.c" */
/* #include "teleo_reactive.c" */

void set_pr(pthread_t pt,int prio)
```

```
{
 int ret;
 struct sched_param scp;

 scp.sched_priority=prio;
 if ((ret=pthread_setschedparam(pt,SCHED_FIFO,&scp)))
 {
  switch(ret)
  {
   case ESRCH: perror("set_pr: Invalid target thread");
               fprintf(stderr,"Thread with identif. %ld",pt);
               exit(1);
               break;
   case EINVAL: perror("set_pr: Sched. policy is invalid, or priority value is invalid for that poli
                exit(1);
                break;
   case EPERM: perror("set_pr: permissions problem");
               exit(1);
               break;
   case EFAULT: perror("set_pr: parameters out of memory scope");
                exit(1);
                break;
   default: perror("set_pr: Unknown error");
            fprintf(stderr,"errno is %d\n",errno);
            exit(1);
            break;
  }
 }
}

/* ------------------------------------------------------------------- */

int get_pr(pthread_t pt)
{
 int ret,policy;
 struct sched_param scp;

 ret=pthread_getschedparam(pt,&policy,&scp);
 if (ret)
 {
  switch(ret)
  {
   case ESRCH: perror("get_pr: Invalid or terminated target thread");
               fprintf(stderr,"pthread_getschedparam returned %d\n",ret);
               exit(1);
               break;
   case EFAULT: perror("get_pr: policy or parameter are outside the thread's memory scope");
                fprintf(stderr,"pthread_getschedparam returned %d\n",ret);
                exit(1);
                break;
   default: perror("get_pr: Unknown error");
            fprintf(stderr,"pthread_getschedparam returned %d\n",ret);
            fprintf(stderr,"errno is %d\n",errno);
            exit(1);
            break;
  }
 }
 printf("\nNo error in pthread_getschedparam, and it has returned (%s,%d) for thread %ld",
```



```
   POL_NAME(policy),ret,pthread_self());
 return(ret);
}

char *get_sc_mode(pthread_t pt)
{
 static char mode[20];
 int ret,policy;
 struct sched_param scp;

 ret=pthread_getschedparam(pt,&policy,&scp);
 if (ret)
 {
  switch(ret)
  {
   case ESRCH: perror("get_sc_mode: invalid or terminated target thread");
               exit(1);
               break;
   case EFAULT: perror("get_sc_mode: policy or parameter are outside the thread's memory scope")
                fprintf(stderr,"pthread_getschedparam returned %d\n",ret);
                exit(1);
                break;
   default: perror("get_pr: Unknown error");
            fprintf(stderr,"pthread_getschedparam returned %d\n",ret);
            fprintf(stderr,"errno is %d\n",errno);
            exit(1);
            break;
  }
 }
 else
  strcpy(mode,POL_NAME(policy));
 return(mode);
}

/* ------------------------------------------------------------------------ */
/*+ This does nothing (is an infinite loop) but it is needed for two things: +*/
/*+ Wasting time when total sharing os less than 1, AND                      +*/
/*+ Keeping the CPU busy while a one shot process has finished before its +*/
/*+ assiged time. Otherwise, system processes (like swapping) could enter +*/
/*+ and avoid the accomplishing of the deadline. See comments in prepare_scheduler +*/

void *my_idle(void *pass)
{
 BEGIN_BEH_MOD
 {
 }
 END_BEH_MOD
}

/* ------------------------------------------------------------------ */

int TickSleep(const int *req,int *rem)
{
  struct timespec nreq,nrem;
  int cod;

  nreq.tv_sec=*req/HZ_NOW;
  nreq.tv_nsec=(int)((1000000000.0/HZ_NOW)*(*req%HZ_NOW));
```

```
  if ((cod=nanosleep(&nreq,&nrem))!=0)
    if (errno==EINTR && rem!=NULL)
      *rem=nrem.tv_sec*HZ_NOW+(int)(nrem.tv_nsec*(HZ_NOW/1000000000.0));
  return cod;
}

/* ------------------------------------------------------------------------ */

void usage(char *pr_name)
{
 fprintf(stderr,"Usage: %s <sharing> <modif> ....\n",pr_name);
 fprintf(stderr,"  where sharing is a float in ]0..1[ and modif is S, Y or N\n");
 fprintf(stderr,"  Sum of sharings of Y/N processes must be 1.00\n");
 fprintf(stderr,"  Sharing of 'S' processes must be given, but it is ignored\n");
 fprintf(stderr," Maximum number of processes is %d\n",MAX_PROC);
}

/* ------------------------------------------------------------------------ */

int check_args(int argc,char *argv[],proc_info pr[MAX_PROC])
{
 int i,j,k;
 char ch,tmp_str[50];
 float tot_sh=0.0;
 long total_ns=0;

 if ((argc>2*MAX_PROC+1) || (!(argc%2)) || (argc<3))
 {
  usage(argv[0]);
  exit(0);
 }

 if (!strcmp(argv[0],NAME_NORMAL))
 {
  printf("%d basic processes will be registered as no-sharing processes\n",
         NUM_BASIC_PROCESSES);
  for (i=0;i<NUM_BASIC_PROCESSES;i++)
  {
   pr[i].modifiable=NO_SHARING;
   pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH]=0.0;
  }
  j=NUM_BASIC_PROCESSES;
 }
 else
  j=0;

 for (i=1,tot_sh=0.0;i<argc;i+=2)
 {
  strcpy(tmp_str,argv[i]);
  pr[j].sharing[CURRENT_SH]=pr[j].sharing[DESIRED_SH]=(float)atof(tmp_str);
  switch(argv[i+1][0])
  {
   case 'Y': pr[j].modifiable=MODIFIABLE;
             tot_sh+=pr[j].sharing[CURRENT_SH];
             break;
   case 'N': pr[j].modifiable=NOT_MODIFIABLE;
             tot_sh+=pr[j].sharing[CURRENT_SH];
```



```
             break;
   case 'S': pr[j].modifiable=NO_SHARING;
             pr[j].sharing[CURRENT_SH]=pr[j].sharing[DESIRED_SH]=0.0;
             break;
   default: perror("Invalid type for thread. Must be 'Y', 'N' or 'S'");
            usage(argv[0]);
            exit(0);
            break;
  }
  j++;
 }

 if (tot_sh>1.00)
 {
  fprintf(stderr,"Error: total sharing is bigger than 1\n");
  exit(0);
 }
 else
  if (tot_sh<1.00)
  {
   pr[j].sharing[CURRENT_SH]=pr[j].sharing[DESIRED_SH]=1.0-tot_sh;

   /* The sharing of the idle process is, of course, modifiable. It is, */
   /* indeed, the first place to get time when anyone else needs it.    */
   pr[j].modifiable=MODIFIABLE;
   pr[j].allowed_time=pr[j].sharing[CURRENT_SH]*PS_SCHED_PERIOD;

   /* AND don't forget to register the idle process */

   fun[j]=my_idle;
   printf("Warning: remaining sharing (%4.2f) has been assigned to the idle process, pos. %d\n",1
   j++;
  }

 for (k=0;k<j;k++)
 {
  pr[k].allowed_time=pr[k].sharing[CURRENT_SH]*PS_SCHED_PERIOD;
  total_ns+=pr[k].allowed_time;
 }

 for (i=0;i<j;i++)
  pr[i].nproc=j;

 for (i=0;i<j;i++)
  printf(" Process %d: sharing %4.2f, allowed_time %ld ns, modif %d\n",
  i,pr[i].sharing[CURRENT_SH],pr[i].allowed_time,pr[i].modifiable);

 printf("Total duration: %ld ns\n",total_ns);

 if (STOP_TO_SEE_SHARINGS==YES)
 {
  printf("Everything right? Can we start? (y/n) ");
  fflush(stdin);
  ch=getchar();
  fflush(stdin);
  if ((ch!='y') && (ch!='Y'))
   exit(0);
 }
```

```
 return(j);
}

/* ------------------------------------------------------------------ */

int touch_sharings(proc_info pr[MAX_PROC],int nproc)
{
 int i,j;
 float total_sha=0.0;

 /* Sharing modifications, when needed, should be done here... */
 i=0;
 while (i<nproc)
 {
  if (pr[i].modifiable!=NO_SHARING)
   total_sha+=pr[i].sharing[DESIRED_SH];
  i++;
 }

 if (total_sha>1.0)
 {
  /* We have an idle from which getting time... */
  if (fun[nproc-1]==my_idle)
  {
   total_sha-=pr[nproc-1].sharing[DESIRED_SH];
   /* If getting time from idle is enough... */
   if (total_sha<1.0)
    pr[nproc-1].sharing[DESIRED_SH]=pr[nproc-1].sharing[CURRENT_SH]=1.0-total_sha;
   /* if not, process sharings will have to be renormalized */
   else
   {
    for (i=0;i<nproc-1;i++)
    {
     pr[i].sharing[DESIRED_SH]/=total_sha;
     pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
    }
    /* and idle has no time for it */
    pr[nproc-1].sharing[DESIRED_SH]=pr[nproc-1].sharing[CURRENT_SH]=0.0;
   }
  }
  else
  {
   /* If we haven't idle: renormalize */
   for (i=0;i<nproc;i++)
   {
    pr[i].sharing[DESIRED_SH]/=total_sha;
    pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
   }
  }
 }
 else  /* we don't run out of time... */
  if (total_sha<1.0)  /* ...even may be we have too much... */
  {
   /* If we have an idle to which assign the remaining time, do it */
   if (fun[nproc-1]==my_idle)
    pr[nproc-1].sharing[DESIRED_SH]=pr[nproc-1].sharing[CURRENT_SH]=1.0-total_sha;
   /* but if we haven't idle,renormalize */
```



```
   else
    for (i=0;i<nproc;i++)
    {
     pr[i].sharing[DESIRED_SH]/=total_sha;
     pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
    }
  }

 /* Then, buffer with the sharings to be sent is updated     */
 /* and execution time for each process is appropriately set */

 i=0;
 j=1;
 while (i<nproc)
 {
  if (pr[i].modifiable==MODIFIABLE)
  {
   pr[i].sharing[CURRENT_SH]=pr[i].sharing[DESIRED_SH];
   pr[i].allowed_time=pr[i].sharing[CURRENT_SH]*PS_SCHED_PERIOD;
   Sh_buff[j]=(unsigned char)((MAX_SH-2)*pr[i].sharing[CURRENT_SH]/total_sha);
   j++;
  }
  i++;
 }
 Sh_buff[j]=NO_MORE_SHARINGS;
 return(0);
}

/* ------------------------------------------------------------------ */

/*+ The real core of the algorithm. +*/

void pseudo_sched(proc_info pr[MAX_PROC],int nproc,void *pass)
{
 int current,sh_error=0;
 void *end_loop=(void *)0;
 struct timespec tsp;

 tsp.tv_sec=0;
 current=0;      /* Let's start in the first process (any other would do, too) */

 while ((end_loop==(void *)0) && (!sh_error))
 {
  /* If the process that has to run now has sharing, let's sleep. */
  /* The process 'current', that has medium priority, will run */
  /* Otherwise, simply call the function. */

  if (pr[current].modifiable==NO_SHARING)
   end_loop=(fun[current])(pass);
  else
   if (pr[current].allowed_time!=0)
   {
    set_pr(pr[current].pth,MEDIUM_PR);
    tsp.tv_nsec=pr[current].allowed_time;
    nanosleep(&tsp,NULL);
    /* When we awake, the process (and everyone else) cannot run. */
    /* Then, change its priority to low, and state that it is not to be run. */
    set_pr(pr[current].pth,LOW_PR);
```

```
   }

  /* Determine which process comes next... */
  current=(current+1)%nproc;

  /* At the end of each turn, (before the next one begins) we could change */
  /* the sharings */

  if (current==0)
   sh_error=touch_sharings(pr,nproc);

  ONE_SHOT_MAY_GO_ON;
 }

 /* Close_all, stored in the last position of the function array, */
 /* is called to stop robot and other cleaning tasks             */
 fun[MAX_PROC-1](pass);

 return;
}

/* ----------------------------------------------------------------------- */

void *prepare_scheduler(void *pass)
{
 int nproc,ret,i,j;
 proc_info *pr;
 pthread_attr_t sons_att;
 pthread_t idle_t;

 pr=(proc_info *)pass;
 nproc=pr[0].nproc;

 /* Setting attributes for the sons */
 if (set_attributes(&sons_att,SON))
  pthread_exit((void *)NULL);

 for (i=0;i<nproc;i++)
 {
  /* Only those sons which have sharing must be created as threads... */
  if (pr[i].modifiable!=NO_SHARING)
  {
   if ((ret=pthread_create(&pr[i].pth,&sons_att,fun[i],pass)))
   {
    if (ret==EAGAIN)
     perror("pthread_create (sons) ");
    else
     perror("pthread_create (sons), unknown error");
    fflush(stdout);
    /* In the case of an arror, apart from signaling it, we must kill */
    /* the threads created up to now before exit */
    for (j=0;j<i;j++)
     if (pr[j].modifiable!=NO_SHARING)
      pthread_cancel(pr[j].pth);
    pthread_exit((void *)0);
   }
   else
   {
```



```
    printf("\nThread %d created with identifier %ld",i,pr[i].pth);
    fflush(stdout);
   }
  }
  else
  {
   pr[i].pth=-1;
   printf("\nFunction %d will not be a thread",i);
   fflush(stdout);
  }
 }

 /* Now, children can go on to do its task.. */
 /* but really they won't do, since this thread (the parent) has */
 /* a higher priority... */

 /* Let's allow them to start, but before, some things to do: */

 /* First, let's register the idle with a priority under the low, but behind  */
 /* the middle. In that way, when a medium-priority level process voluntarily */
 /* relinquish the processor by calling pthread_cond_wait, our idle will run, */
 /* and will not allow normal processes (i.e., disk chaching or so) to run,   */
 /* which would alter the real-time operation.                                */
 /* NOTICE: this is ANOTHER instance of my_idle. The first instance has been  */
 /* registered as a normal process, just to spent the time that remains in    */
 /* usual situations (i.e., when not all processor time is used by the user   */
 /* This is not the case here: this instance will only run in the special     */
 /* circumstance described before                                             */
 if ((LOW_PR==MEDIUM_PR) || (LOW_PR==MEDIUM_PR+1))
 {
  fprintf(stderr,"Error: low and middle priorities cannot be so close.");
  for (i=0;i<nproc;i++)
   pthread_cancel(pr[i].pth);

  pthread_exit((void *)0);
 }

 set_attributes(&sons_att,SPECIAL_IDLE);

 if ((ret=pthread_create(&idle_t,&sons_att,my_idle,pass)))
 {
  if (ret==EAGAIN)
   perror("pthread_create (my_idle) ");
  else
   perror("pthread_create (my_idle), unknown error");
  fflush(stdout);
  /* In the case of an arror, apart from signaling it, we must kill */
  /* the threads created up to now before exit */
  for (i=0;i<nproc;i++)
  {
   if (pr[i].modifiable!=NO_SHARING)
    pthread_cancel(pr[i].pth);
  }
  pthread_exit((void *)0);
 }

 /* A long time to ensure all son processes are already awaiting parent's signal... */
 sleep(2);
```

```
 /* And, finally,... */
 SONS_CAN_START;

 pseudo_sched(pr,nproc,pass);

 /* The parent kill the sons... */
 for (i=0;i<nproc;i++)
 {
  if (pr[i].modifiable!=NO_SHARING)
   pthread_cancel(pr[i].pth);
 }

 /* ...even the smallest one! */
 pthread_cancel(idle_t);

 pthread_exit((void *)0);
}

/* ------------------------------------------------------------------ */

int set_attributes(pthread_attr_t *pt,int who)
{
 struct sched_param scp;

 if (pthread_attr_init(pt))
 {
  perror("pthread_attr_init");
  return(1);
 }
 if (pthread_attr_setdetachstate(pt,PTHREAD_CREATE_JOINABLE))
 {
  perror("pthread_attr_setdetachstate");
  return(1);
 }
 if (pthread_attr_setschedpolicy(pt,SCHED_FIFO))
 {
  perror("pthread_attr_setschedpolicy");
  return(1);
 }
 switch (who)
 {
  case PARENT: scp.sched_priority=HIGH_PR;
               break;
  case SON: scp.sched_priority=LOW_PR;
            break;
  case SPECIAL_IDLE: scp.sched_priority=(LOW_PR+MEDIUM_PR)/2;
                     break;
  default: scp.sched_priority=LOW_PR;
           break;
 }
 if (pthread_attr_setschedparam(pt,&scp))
 {
  perror("pthread_attr_setschedparam");
  return(1);
 }
 if (pthread_attr_setinheritsched(pt,PTHREAD_EXPLICIT_SCHED))
 {
```



```
  perror("pthread_attr_setinheritsched");
  return(1);
 }
 return(0);
}

/* ------------------------------------------------------------------ */

int main(int argc,char *argv[])
{
 proc_info pr[MAX_PROC];
 pthread_attr_t parent_att;
 int nproc,nreg_proc,ret,init_basics;

 if (!strcmp(argv[0],NAME_NORMAL))
  init_basics=1;
 else
  init_basics=0;

 nreg_proc=Initialize_all(init_basics);

 /* After cheking arguments, nproc will contain the number of son processes */
 /* to execute, NOT including the pseudo-scheduler */
 current_np=nproc=check_args(argc,argv,pr);

 printf("\n%d processes (NOT including the pseudo-scheduler) are to be created. Main is %d",nproc
 fflush(stdout);

 /* If my_idle was registered by check_args, we have one process more, but */
 /* the user didn't know that, and has not to be aware of it.             */

 if (nreg_proc==nproc-1)
 {
  if (fun[nproc-1]==my_idle)
   nreg_proc++;
  else
  {
   fprintf(stderr,"\nmy_idle should appear at the end of the array process! (pos %d)",nproc-1);
   exit(0);
  }
 }

 if (nreg_proc!=nproc)
 {
  fprintf(stderr,"\nInconsistency registering processes.");
  fprintf(stderr,"\nInitialize_all says you want %d processes",nreg_proc);
  fprintf(stderr,"\nCommand line parameters say you want %d",nproc);
  fprintf(stderr,"\nCheck your code and/or command line par.\n");
  exit(0);
 }

 /* Setting attributes for the parent. */
 if (set_attributes(&parent_att,PARENT))
  exit(1);

 /* And creating the parent's thread.  */
 if ((ret=pthread_create(&sch_t,&parent_att,prepare_scheduler,(void *)pr)))
 {
```

```
  if (ret==EAGAIN)
  {
   perror("pthread_create: not enough resources, or too many threads");
   exit(1);
  }
  else
  {
   perror("pthread_create: unknown error");
   exit(1);
  }
 }

 /* We will not come back here, to the poor and sched-other main process, */
 /* until the scheduler function does not finish.                          */

 printf("\nWaiting for scheduler (ident %ld) to finish...",sch_t);
 fflush(stdout);
 pthread_join(sch_t,(void **)&ret);
 (fun[MAX_PROC-1])((void *)pr);

 printf("\nScheduler finished.\n BYE !! \n");

 exit(0);
}
```

## 4 File rodney.h

Types

### Included Files

`#include "common.h"` *(Section 1)*

```
#include </usr/include/stdio.h>
#include </usr/include/stdlib.h>
#include </usr/include/math.h>
#include </usr/include/errno.h>
#include </usr/include/sys/perm.h>
#include </usr/include/asm/io.h>
#include </usr/include/sys/time.h>
#include </usr/include/sys/resource.h>
#include </usr/include/signal.h>
#include </usr/include/sched.h>
#include </usr/include/pthread.h>
```

### Preprocessor definitions

`#define RODNEY`

A few constants related with the sharing's transmission

`#define SHARING_DATA_NOT_READY 0`

`#define SHARING_DATA_READY 255`

`#define TIME_TO_TRANSMIT_SHARINGS 33`

Constants to identify commands. They must be consistent with their values in the monitor program.

`#define MAX_NUM_COM 100`

`#define DEB 0`

`#define HELP 1`



`#define QUIT 2`

`#define STOP 3`

`#define MOVE_V 4`

`#define MOVE_D 5`

`#define ROT 6`

`#define ROT_W 7`

`#define ULT_SHOT 8`

`#define SOUND 9`

Base address of microswitches and display controller card

`#define MICROSWITCHES_BASE 0x0230`

Base address of sonar controller card

`#define SONARS_BASE_ADD 0x310`

Number of currently installed sonars

`#define NUM_SONARS 8`

Limits for sonar obstacle detection

`#define INF_DET_LIMIT 50`

`#define SUP_DET_LIMIT 250`

`#define THRESHOLD 10`

`#define THRESHOLD1 20`

`#define THRESHOLD2 100`

Conversion factor from counts to cm.

`#define SONARS_COEFF 0.5435`

Codes for sonar obstacle detection

`#define NOT_SCANNED 0`

`#define NO_OBSTACLE 1`

`#define OBSTACLE 2`

`#define OBSTACLE_VERY_CLOSE 3`

Mark to signal that there are (or not) sonar data ready to be transmitted

`#define SONAR_DATA_READY 255`

`#define SONAR_DATA_NOT_READY 0`

How often (in scheduler cycles) we must transmit the sonar data

`#define TIME_TO_TRANSMIT_SONARS 33`

Mark to signal that there are (or not) position and orientation data ready to be transmitted

`#define POS_OR_DATA_READY 1.0`

`#define POS_OR_DATA_NOT_READY 0.0`

How often (in scheduler cycles) we must transmit the sonar data

`#define TIME_TO_TRANSMIT_POS_OR 33`

The code to be put in the transmission buffer to indicate that NUM_SONARS bytes of sonar readings come next. Host program must know about this code

`#define SONAR_TRANSMIT_CODE 254`

Code to indicate that a sonar is not detecting any obstacle. Must be known by the host (monitor) program.

```
#define OBSTACLE_FAR_AWAY 128
```

Mark to signal that there are (or not) received commands to be interpreted and executed

```
#define COMMAND_READY 1
#define COMMAND_NOT_READY 0
```

Base address for motors and turning direction registers

```
#define LEFT_MOTOR 0x301
#define RIGHT_MOTOR 0x300
#define TURNING_DIRECTION 0x302
```

Codes assigned to possible robot motions

```
#define FORWARDS 0x00
#define BACKWARDS 0x03
#define LEFT_TURN 0x02
#define RIGTH_TURN 0x01
```

Constants for the rotation variable

```
#define ROT_FINISHED 1
#define NO_ROT_FINISHED 0
#define YES 1
#define NO 0
#define MAX_ROT_LOOPS 50
```

Codes assigned to control mode

```
#define C_VEL 0
#define C_POS 1
#define C_POS_VEL 2
#define C_ROT 3
#define STOP_M 4
```

Base address for odometric system and codes assigned to each encoder

```
#define ENCODER_BASE_ADD 0x360
#define LEFT_ENCODER 1
#define RIGHT_ENCODER 0
```

Important addresses for the odometric controller chip

```
#define LEFT_ENC_DATA
#define LEFT_ENC_CON
#define RIGHT_ENC_DATA
#define RIGHT_ENC_CON
#define LOAD
```

Important codes for programming the odometric system

```
#define MASTER_RESET 0x20
#define INPUT_SETUP 0x68
#define CUAD_X1 0xC1
#define CUAD_X2 0xC2
#define CUAD_X4 0xC3
```



```
#define ADDR_RESET 0x01
#define LATCH_CONT 0x02
#define RESET_CONT 0x04
#define PRESET_CONT 0x08
#define BINAR_CONT 0x80
```

Codes for the power to be sent to the motors

```
#define MAX_POW 0xFB
#define MIN_POW 0x00
#define BASE_POW 0x45
```

Minimum and maximum allowed velocity, distance and rotation angle values

```
#define MIN_VEL 0
#define MAX_VEL 200
#define MIN_DIST 0
#define MAX_DIST 10000
#define MIN_ANG 0.0
#define MAX_ANG
```

Constants for I/O port access permissions

```
#define DISPLACEMENT1 4
#define DISPLACEMENT2 3
#define DISPLACEMENT3 3
#define DISPLACEMENT4 8
```

Mechanical constants of our robot

```
#define r 252.5
#define d 100.0
#define N 500
#define R 43
```

Number of pulses before reaching the destination in which stop signal must be sent

```
#define THRESHOLD_POS 150
```

These constants must be known by the host (monitor) program

```
#define POS_OR_TRANSMIT_CODE 252
#define LENGTH_POS_OR 12
```

Radial distance from robot's centre to each sensor (mm.)

```
#define DIST_CENTRE_SENS 280
```

Aperture angle of sensitivity, according to the radiation diagram (+-10 degrees or +-0.348 rad)

```
#define APERTURE_ANG 0.174
```

Sepparation angle between sensors, according to the physical placement (radians)

```
#define ANGLE_SENSOR 0.78539
```

Number of grid cells in which the workspace is divided

```
#define CELL_NUMBER 300
```

Increment for raster in obstacle detection between +-APERTURE_ANGLE

```
#define INC 0.05
```

Real physical size of one cell of the grid (mm.)

```
#define GRID_SIZE_X 300
#define GRID_SIZE_Y 300
```

Hardware addresses for the loudspeaker port and muter

```
#define BEEP 0x42
#define MUTER 0x61
```

Duration (in jiffies) of one beep

```
#define BEEP_DUR 3
```

Constant to signal that no interval has to be emitted

```
#define NO_INTERVAL 10
```

Constants for serial port management Adresses of serial ports in most machines...

```
#define COM1 0x3F8
#define COM2 0x2F8
```

The serial port we are using

```
#define SERIAL COM2
```

Generic values...

```
#define MAX_BAUD 115200L
#define PAL_5 0x00
#define PAL_6 0x01
#define PAL_7 0x02
#define PAL_8 0x03
#define STOP_1 0x00
#define STOP_2 0x04
#define NO_PARITY 0x00
#define EVEN_PAR 0x18
#define ODD_PAR 0x08
```

...and those we choose to use

```
#define SERIAL_VEL 9600
#define PAL_LENGTH PAL_8
#define STOP_BITS STOP_1
#define PARITY NO_PARITY
```

These must be known by the host (monitor) program

```
#define CONF_CODE 255
#define START_BIN_PACK 251
#define MAX_ARG_COM_BYTES 64
#define DIV_CODE 129
#define NOT_A_CODE 249
```

Transmision/reception modes

```
#define READING 1
#define SENDING 2
#define RECEIVING 3
```



Lengths of all buffers

```
#define BUF_LEN 254
```

Number of trials to read/write a character

```
#define NUM_TRIALS 5
```

Serial transmision error codes

```
#define OK 0
#define TOO_LONG_LEN_REC 1
#define REC_BUFF_NOT_EMPTY 2
#define BIN_BUFF_EXCEEDED 3
#define COM_BUFF_EXCEEDED 4
#define NOT_READY_TO_READ 5
#define NOT_READY_TO_WRITE 6
```

Other transmision codes to signal states

```
#define STILL_RECEIVING 100
#define RECEPTION_FINISHED 150
#define STILL_TRANSMITTING 200
#define TRANSMISION_FINISHED 250
```

Macros to know if the serial port has something to be read and if it is ready to write in it.

```
#define SOMETHING_TO_READ
#define READY_TO_WRITE
```

## 4.1 Type definitions

### 4.1.1 Type struct C

The cell struct

```
struct C
  {
    unsigned int x1:2;
    unsigned int x2:2;
    unsigned int x3:2;
    unsigned int x4:2;
  }
```

## 4.2 Variables

### 4.2.1 Variable current_np

The current number of registered processes

```
int current_np
```

Inc. from: common.h *(Section 1.2.1)*

### 4.2.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

```
unsigned char Sh_buff[16+2]
```

Inc. from: common.h *(Section 1.2.2)*

#### 4.2.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h (Section 1.2.3)

#### 4.2.4 Variable Microswitches_code

Microswitch control variable

`unsigned char Microswitches_code`

#### 4.2.5 Variable Microswitches_angle

`float Microswitches_angle`

#### 4.2.6 Variable Display_code

Display control variable

`unsigned char Display_code`

#### 4.2.7 Variable sonar_readings

Raw sonar readings

`float sonar_readings[8]`

#### 4.2.8 Variable sonar_obstacle

Sonar obstacle detection

`unsigned char sonar_obstacle[8]`

#### 4.2.9 Variable Son_buff

The sonar transmission buffer

`unsigned char Son_buff[8+1]`

#### 4.2.10 Variable Pos_or_buff

The pos-or transmission buffer

`float Pos_or_buff[4]`

#### 4.2.11 Variable Angle_to_rotate

Angle to rotate (in radians)

`double Angle_to_rotate`

#### 4.2.12 Variable Rotation

`int Rotation`

#### 4.2.13 Variable Must_rotate

`int Must_rotate`

#### 4.2.14 Variable ang_wander

Variables for the procedure to go to a final location

`double ang_wander`

#### 4.2.15 Variable Dist_wander

`float Dist_wander`



#### 4.2.16 Variable X_final

`long X_final`

#### 4.2.17 Variable Y_final

`long Y_final`

#### 4.2.18 Variable ANG_final

`double ANG_final`

#### 4.2.19 Variable f1

`int f1`

#### 4.2.20 Variable f2

`int f2`

#### 4.2.21 Variable f3

`int f3`

#### 4.2.22 Variable f4

`int f4`

#### 4.2.23 Variable f5

`int f5`

#### 4.2.24 Variable f6

`int f6`

#### 4.2.25 Variable Init_maneuver

`int Init_maneuver`

#### 4.2.26 Variable End_maneuver

`int End_maneuver`

#### 4.2.27 Variable Fr

`float Fr`

#### 4.2.28 Variable SUPRESSOR

`int SUPRESSOR`

#### 4.2.29 Variable first_time

`int first_time`

#### 4.2.30 Variable Mode

Control variables Desired control mode

`int Mode`

#### 4.2.31 Variable Old_Mode

`int Old_Mode`

#### 4.2.32 Variable Command

`int Command`

#### 4.2.33 Variable LEFT_VEL

Desired velocities in pulses/T

`unsigned int LEFT_VEL`

#### 4.2.34 Variable RIGHT_VEL

`unsigned int RIGHT_VEL`

#### 4.2.35 Variable Distance

Desired distance (in mm.)

`long Distance`

#### 4.2.36 Variable Move_distance

`int Move_distance`

#### 4.2.37 Variable Must_move

`int Must_move`

#### 4.2.38 Variable Must_go_ahead

`int Must_go_ahead`

#### 4.2.39 Variable Moving_forward

`int Moving_forward`

#### 4.2.40 Variable Door_detected

`int Door_detected`

#### 4.2.41 Variable Phase_1

`int Phase_1`

#### 4.2.42 Variable G

Value used by the rotation module

`int G`

#### 4.2.43 Variable Total_left_pulses

Pulses counted up to now

`long Total_left_pulses`

#### 4.2.44 Variable Total_right_pulses

`long Total_right_pulses`



#### 4.2.45 Variable P_T_I

`long P_T_I`

#### 4.2.46 Variable P_T_D

`long P_T_D`

#### 4.2.47 Variable Count_left_pulses

Pulses desired for doing a certain movement

`long Count_left_pulses`

#### 4.2.48 Variable Count_right_pulses

`long Count_right_pulses`

#### 4.2.49 Variable X

Position/orientation of the robot over the workarea (mm. and radians, respectively)

`double X`

#### 4.2.50 Variable Y

`double Y`

#### 4.2.51 Variable ANGLE

`double ANGLE`

#### 4.2.52 Variable POSX

Robot position on the grid cell map

`int POSX`

#### 4.2.53 Variable POSY

`int POSY`

#### 4.2.54 Variable posx_new

`int posx_new`

#### 4.2.55 Variable posy_new

`int posy_new`

#### 4.2.56 Variable ind_centre

`int ind_centre`

#### 4.2.57 Variable Num_beeps

Sound control variables. Any process can change them. Number of beeps we want to emit.

`int Num_beeps`

#### 4.2.58 Variable Interval

Entonation interval; 0 to 7 for ascending, -1 to -7 for descending

`int Interval`

#### 4.2.59 Variable Whisper_freq

Frequency for the whisper

```
unsigned int Whisper_freq
```

#### 4.2.60 Variable Serial_mode

Serial transmission control variable Possible values are READING, RECEIVING and SENDING

```
int Serial_mode
```

#### 4.2.61 Variable Rec_buff

The three serial buffers: received binary packet, received commands and bytes to transmit.

```
unsigned char Rec_buff[254+1]
```

#### 4.2.62 Variable Commands

The three serial buffers: received binary packet, received commands and bytes to transmit.

```
unsigned char Commands[254+1]
```

#### 4.2.63 Variable Tr_buff

The three serial buffers: received binary packet, received commands and bytes to transmit.

```
unsigned char Tr_buff[254+1]
```

#### 4.2.64 External Variables

**mut**
Variables defined in the ps_sched module, that must be known here. Mutual exclusion flags and conditions that are used by the scheduler and by the son threads, and also the thread identifier of the scheduler

```
extern pthread_mutex_t mut
```

**nmut**
Variables defined in the ps_sched module, that must be known here. Mutual exclusion flags and conditions that are used by the scheduler and by the son threads, and also the thread identifier of the scheduler

```
extern pthread_mutex_t nmut
```

**mshot**
Variables defined in the ps_sched module, that must be known here. Mutual exclusion flags and conditions that are used by the scheduler and by the son threads, and also the thread identifier of the scheduler

```
extern pthread_mutex_t mshot
```

**bcont_mut**
Variables defined in the ps_sched module, that must be known here. Mutual exclusion flags and conditions that are used by the scheduler and by the son threads, and also the thread identifier of the scheduler

```
extern pthread_mutex_t bcont_mut
```

**cond**

```
extern pthread_cond_t cond
```

**cshot**

```
extern pthread_cond_t cshot
```

**sch_t**

```
extern pthread_t sch_t
```

#### 4.2.65 Local Variables

**cell**

```
static struct C cell[300][300/4]
```



### 4.3 Actual file listing

```
#ifndef COMMON
#include "common.h"
#endif

#define RODNEY

/*+ A few constants related with the sharing's transmission +*/
#define SHARING_DATA_NOT_READY     0
#define SHARING_DATA_READY       255
#define TIME_TO_TRANSMIT_SHARINGS 33

/* SHARING_TRANSMIT_CODE is not defined here, as it should be, but in common.h,
 matching the definition of the monitor program, since ps_sched, which fills
 the sharing array, needs it
*/
/* #define SHARING_TRANSMIT_CODE    253 */

/*+ Constants to identify commands. They must be consistent with their
 values in the monitor program.                                     +*/
#define MAX_NUM_COM   100
#define DEB             0
#define HELP            1
#define QUIT            2
#define STOP            3
#define MOVE_V          4
#define MOVE_D          5
#define ROT             6
#define ROT_W           7
#define ULT_SHOT        8
#define SOUND           9

/*+ Constants and global variables related with the hardware of the robot +*/

/****************************** MICROSWITCHES ******************************/

/*+ Base address of microswitches and display controller card +*/
#define MICROSWITCHES_BASE 0x0230

/*+ Microswitch control variable +*/
unsigned char Microswitches_code=0;
float Microswitches_angle=0.0;

/******************************** DISPLAY **********************************/

/*+ Display control variable +*/
unsigned char Display_code=0;

/******************************** SONARS ***********************************/

/*+ Base address of sonar controller card +*/
#define SONARS_BASE_ADD 0x310

/*+ Number of currently installed sonars +*/
#define NUM_SONARS 8

/*+ Limits for sonar obstacle detection +*/
#define INF_DET_LIMIT 50
```

```
#define SUP_DET_LIMIT 250
#define THRESHOLD     10
#define THRESHOLD1    20
#define THRESHOLD2    100

/** Conversion factor from counts to cm. **/
#define SONARS_COEFF  0.5435

/** Codes for sonar obstacle detection **/
#define NOT_SCANNED         0
#define NO_OBSTACLE         1
#define OBSTACLE            2
#define OBSTACLE_VERY_CLOSE 3

/** Mark to signal that there are (or not) sonar data ready to be transmitted **/
#define SONAR_DATA_READY   255
#define SONAR_DATA_NOT_READY 0
/** How often (in scheduler cycles) we must transmit the sonar data **/
#define TIME_TO_TRANSMIT_SONARS 33  /** This is now roughly half second **/

/** Mark to signal that there are (or not) position and orientation data ready
 to be transmitted **/
#define POS_OR_DATA_READY     1.0
#define POS_OR_DATA_NOT_READY 0.0
/** How often (in scheduler cycles) we must transmit the sonar data **/
#define TIME_TO_TRANSMIT_POS_OR 33  /** This is now roughly half second **/

/** The code to be put in the transmission buffer to indicate that NUM_SONARS
 bytes of sonar readings come next. Host program must know about this code **/
#define SONAR_TRANSMIT_CODE  254
/** Code to indicate that a sonar is not detecting any obstacle. Must be known
 by the host (monitor) program. **/
#define OBSTACLE_FAR_AWAY     128

/** Mark to signal that there are (or not) received commands to be
 interpreted and executed                                         **/
#define COMMAND_READY     1
#define COMMAND_NOT_READY 0

/** Raw sonar readings **/
float sonar_readings[NUM_SONARS];

/** Sonar obstacle detection **/
unsigned char sonar_obstacle[NUM_SONARS];

/** The sonar transmission buffer **/
unsigned char Son_buff[NUM_SONARS+1]={SONAR_DATA_NOT_READY,0,0,0,0,0,0,0,0};

/** The pos-or transmission buffer **/
float Pos_or_buff[4]={POS_OR_DATA_NOT_READY,0.0,0.0,0.0};

/*********************** MOTORS AND ENCODERS ******************************/

/** Base address for motors and turning direction registers **/
#define LEFT_MOTOR             0x301
#define RIGHT_MOTOR            0x300
#define TURNING_DIRECTION      0x302
```



```
/** Codes assigned to possible robot motions **/
#define FORWARDS          0x00
#define BACKWARDS         0x03
#define LEFT_TURN         0x02
#define RIGTH_TURN        0x01

/** Constants for the rotation variable **/
#define ROT_FINISHED    1
#define NO_ROT_FINISHED 0
#define YES             1
#define NO              0
#define MAX_ROT_LOOPS  50

/** Codes assigned to control mode **/
#define C_VEL     0
#define C_POS     1
#define C_POS_VEL 2
#define C_ROT     3
#define STOP_M    4

/** Base address for odometric system and codes assigned to each encoder **/
#define ENCODER_BASE_ADD    0x360
#define LEFT_ENCODER            1
#define RIGHT_ENCODER           0

/** Important addresses for the odometric controller chip **/
#define LEFT_ENC_DATA      ENCODER_BASE_ADD+0
#define LEFT_ENC_CON       ENCODER_BASE_ADD+1
#define RIGHT_ENC_DATA     ENCODER_BASE_ADD+4
#define RIGHT_ENC_CON      ENCODER_BASE_ADD+5
#define LOAD               ENCODER_BASE_ADD+8

/** Important codes for programming the odometric system **/
#define MASTER_RESET     0x20
#define INPUT_SETUP      0x68
#define CUAD_X1          0xC1
#define CUAD_X2          0xC2
#define CUAD_X4          0xC3
#define ADDR_RESET       0x01
#define LATCH_CONT       0x02
#define RESET_CONT       0x04
#define PRESET_CONT      0x08
#define BINAR_CONT       0x80

/** Codes for the power to be sent to the motors **/
#define MAX_POW          0xFB
#define MIN_POW          0x00
#define BASE_POW         0x45

/** Minimum and maximum allowed velocity, distance and rotation angle values **/
#define MIN_VEL      0
#define MAX_VEL    200  /** Pulses/T **/
#define MIN_DIST     0
#define MAX_DIST  10000  /** mm. **/
#define MIN_ANG     0.0
#define MAX_ANG  2*M_PI  /** Radians **/

/** Constants for I/O port access permissions **/
```

```
#define DISPLACEMENT1 4
#define DISPLACEMENT2 3
#define DISPLACEMENT3 3
#define DISPLACEMENT4 8

/*+ Mechanical constants of our robot +*/
#define r 252.5 /*+ Half-distance between director wheels in mm. +*/
#define d 100.0 /*+ Diameter of director wheels  in mm.          +*/
#define N 500   /*+ Encoder resolution (counts/turn)             +*/
#define R 43    /*+ Gear ratio                                   +*/

/*+ Angle to rotate (in radians) +*/
double Angle_to_rotate=0.0;
int Rotation=ROT_FINISHED;
int Must_rotate=NO;

/*+ Variables for the procedure to go to a final location +*/
double ang_wander=0.0;
float Dist_wander=0.0;
long X_final,Y_final;
double ANG_final;
int f1=NO,f2=NO,f3=NO,f4=NO,f5=NO,f6=NO;
int Init_maneuver=YES,End_maneuver=NO;
float Fr=0.0;
int SUPRESSOR=NO,first_time=YES;

/*+ Control variables
 Desired control mode +*/
int Mode=C_VEL;
int Old_Mode=C_VEL;
int Command=FORWARDS;

/*+ Number of pulses before reaching the destination in which stop signal must be sent +*/
#define THRESHOLD_POS 150

/*+ Desired velocities in pulses/T +*/
unsigned int LEFT_VEL=75;
unsigned int RIGHT_VEL=75;

/*+ Desired distance (in mm.) +*/
long Distance=0L;
int Move_distance=ROT_FINISHED;
int Must_move=NO;

int Must_go_ahead;
int Moving_forward=NO;

int Door_detected=NO;
int Phase_1=NO;

/*+ Value used by the rotation module +*/
int G=0;

/*+ Pulses counted up to now +*/
long Total_left_pulses=0L;
long Total_right_pulses=0L;
long P_T_I=0L;
long P_T_D=0L;
```



```
/*+ Pulses desired for doing a certain movement +*/
long Count_left_pulses=0L;
long Count_right_pulses=0L;

/****************************** MAP ****************************************/

/*+ These constants must be known by the host (monitor) program +*/
#define POS_OR_TRANSMIT_CODE   252
#define LENGTH_POS_OR           12  /*+ Three floats have to be transmitted +*/

/*+ Position/orientation of the robot over the workarea (mm. and radians, respectively) +*/
double X=0.0;
double Y=0.0;
double ANGLE=M_PI/2.0;

/*+ Radial distance from robot's centre to each sensor (mm.) +*/
#define DIST_CENTRE_SENS 280

/*+ Aperture angle of sensitivity, according to the radiation diagram (+-10 degrees or +-0.348 r
#define APERTURE_ANG 0.174

/*+ Sepparation angle between sensors, according to the physical placement (radians) +*/
#define ANGLE_SENSOR 0.78539

/*+ Number of grid cells in which the workspace is divided +*/
#define CELL_NUMBER 300

/*+ Increment for raster in obstacle detection between +-APERTURE_ANGLE +*/
#define INC 0.05

/*+ Real physical size of one cell of the grid (mm.) +*/
#define GRID_SIZE_X 300
#define GRID_SIZE_Y 300

/*+ Robot position on the grid cell map +*/
int POSX=(int)(CELL_NUMBER/2);
int POSY=(int)(CELL_NUMBER/2);
int posx_new=(int)(CELL_NUMBER/2);
int posy_new=(int)(CELL_NUMBER/2);
int ind_centre;

/*+ The cell struct +*/
static struct C
{
  unsigned int x1:2;
  unsigned int x2:2;
  unsigned int x3:2;
  unsigned int x4:2;
}cell[CELL_NUMBER][CELL_NUMBER/4];


/****************************** LOUDSPEAKER ********************************/

/*+ Hardware addresses for the loudspeaker port and muter +*/
#define BEEP  0x42
#define MUTER 0x61
```

```
/** Duration (in jiffies) of one beep **/
#define BEEP_DUR 3
/** Constant to signal that no interval has to be emitted **/
#define NO_INTERVAL -10

/** Sound control variables. Any process can change them.
 Number of beeps we want to emit. **/
int Num_beeps=0;
/** Entonation interval; 0 to 7 for ascending, -1 to -7 for descending **/
int Interval=NO_INTERVAL;
/** Frequency for the whisper **/
unsigned int Whisper_freq=0;

/*********************************** SERIAL PORT ******************************/

/** Constants for serial port management
 Adresses of serial ports in most machines... **/
#define COM1 0x3F8
#define COM2 0x2F8
/** The serial port we are using                 **/
#define SERIAL COM2
/** Constants to set up serial port parameters   **/
/** Generic values...                            **/
#define MAX_BAUD 115200L

#define PAL_5     0x00
#define PAL_6     0x01
#define PAL_7     0x02
#define PAL_8     0x03

#define STOP_1    0x00
#define STOP_2    0x04

#define NO_PARITY 0x00
#define EVEN_PAR  0x18
#define ODD_PAR   0x08

/** ...and those we choose to use                    **/
#define SERIAL_VEL 9600
#define PAL_LENGTH PAL_8
#define STOP_BITS  STOP_1
#define PARITY     NO_PARITY

/** These must be known by the host (monitor) program **/
#define CONF_CODE       255
#define START_BIN_PACK  251
#define MAX_ARG_COM_BYTES      64
#define DIV_CODE              129
#define NOT_A_CODE            249

/** Transmision/reception modes                 **/
#define READING    1
#define SENDING    2
#define RECEIVING 3

/** Lengths of all buffers                      **/
#define BUF_LEN 254
/** Number of trials to read/write a character **/
```



```
#define NUM_TRIALS 5

/** Serial transmision error codes                                 **/
#define OK                   0
#define TOO_LONG_LEN_REC     1
#define REC_BUFF_NOT_EMPTY 2
#define BIN_BUFF_EXCEEDED  3
#define COM_BUFF_EXCEEDED  4
#define NOT_READY_TO_READ  5
#define NOT_READY_TO_WRITE 6
/** Other transmision codes to signal states **/
#define STILL_RECEIVING        100
#define RECEPTION_FINISHED     150
#define STILL_TRANSMITTING     200
#define TRANSMISION_FINISHED 250

/** Macros to know if the serial port has something to be read and if it is
 ready to write in it.                                                     **/
#define SOMETHING_TO_READ (inb(SERIAL+5) & 0x01)
#define READY_TO_WRITE    (inb(SERIAL+5) & 0x20)

/** Serial transmission control variable **/
int Serial_mode=READING;   /** Possible values are READING, RECEIVING and SENDING   **/
/** The three serial buffers: received binary packet, received commands
 and bytes to transmit.                                                 **/
unsigned char Rec_buff[BUF_LEN+1],Commands[BUF_LEN+1],Tr_buff[BUF_LEN+1];

/****************************** SCHEDULER VARIABLES *************************/

/** Variables defined in the ps_sched module, that must be known here.
 Mutual exclusion flags and conditions that are used by the scheduler
 and by the son threads, and also the thread identifier of the scheduler **/
extern pthread_mutex_t mut,nmut,mshot,bcont_mut;
extern pthread_cond_t cond,cshot;
extern pthread_t sch_t;

/** Prototypes of functions in this module. All of them have to do with
 the control of our robot, and are therefore specific to it.
 First, utillery functions... **/
void init_architecture(int basics_loaded);
void init_robot_hardware(void);
int access_to_ports(void);
void C_rot(void);
void C_pos(void);
void C_vel(void);
void C_pos_vel(void);
void Control(void);
void Point_to(unsigned short,unsigned int,unsigned);
void CalculateMapPosition(float,float,unsigned);
void InitializeMap(void);
void beep(void);
void interval(void);
void whisper(void);
int read_byte(unsigned char *);
int write_byte(unsigned char);
int receive(void);
int send(void);
void StopRobot(void);
```

```
void InitialiseSonars(void);

/*+ No-sharing low-level functions for low-level tasks, to be called periodically +*/
void *sonars(void *);
void *COLISION_DETECTION(void *);
void *visualize(void *);
void *map(void *);
void *sound(void *);
void *serial(void *);
void *ptrans(void *);
void *trans_ult_and_sh(void *);
void *TEST(void *);
void *ROTATE(void *);
void *ADVANCE(void *);
void *MOVE(void *);
void *FEELFORCE(void *);
void *RUNAWAY(void *);
void *WANDER(void *);
void *AVOID(void *);
void *RECALCULATE(void *);
void *TELEO(void *);
void *ROTATE1(void *);
void *ADVANCE1(void *);

/*+ serial could have sharing, if a higher bandwith is required... +*/
void *serial(void *);
```

# 5 File rodney.c

Types

## Included Files

```
#include "common.h"                                   (Section 1)
   #include </usr/include/stdio.h>
   #include </usr/include/stdlib.h>
   #include </usr/include/math.h>
   #include </usr/include/errno.h>
   #include </usr/include/sys/perm.h>
   #include </usr/include/asm/io.h>
   #include </usr/include/sys/time.h>
   #include </usr/include/sys/resource.h>
   #include </usr/include/signal.h>
   #include </usr/include/sched.h>
   #include </usr/include/pthread.h>
#include "rodney.h"                                   (Section 4)
```

## 5.1 Variables

### 5.1.1 Variable current_np

The current number of registered processes

`int current_np`

Inc. from: common.h *(Section 1.2.1)*



### 5.1.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

`unsigned char Sh_buff[16*2]`

Inc. from: common.h *(Section 1.2.2)*

### 5.1.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h *(Section 1.2.3)*

### 5.1.4 Variable Microswitches_code

Microswitch control variable

`unsigned char Microswitches_code`

Inc. from: rodney.h *(Section 4.2.4)*

### 5.1.5 Variable Microswitches_angle

`float Microswitches_angle`

Inc. from: rodney.h *(Section 4.2.5)*

### 5.1.6 Variable Display_code

Display control variable

`unsigned char Display_code`

Inc. from: rodney.h *(Section 4.2.6)*

### 5.1.7 Variable sonar_readings

Raw sonar readings

`float sonar_readings[8]`

Inc. from: rodney.h *(Section 4.2.7)*

### 5.1.8 Variable sonar_obstacle

Sonar obstacle detection

`unsigned char sonar_obstacle[8]`

Inc. from: rodney.h *(Section 4.2.8)*

### 5.1.9 Variable Son_buff

The sonar transmission buffer

`unsigned char Son_buff[8+1]`

Inc. from: rodney.h *(Section 4.2.9)*

### 5.1.10 Variable Pos_or_buff

The pos-or transmission buffer

`float Pos_or_buff[4]`

Inc. from: rodney.h *(Section 4.2.10)*

### 5.1.11 Variable Angle_to_rotate

Angle to rotate (in radians)

`double Angle_to_rotate`

Inc. from: rodney.h *(Section 4.2.11)*

#### 5.1.12 Variable Rotation

`int Rotation`

Inc. from: rodney.h *(Section 4.2.12)*

#### 5.1.13 Variable Must_rotate

`int Must_rotate`

Inc. from: rodney.h *(Section 4.2.13)*

#### 5.1.14 Variable ang_wander

Variables for the procedure to go to a final location

`double ang_wander`

Inc. from: rodney.h *(Section 4.2.14)*

#### 5.1.15 Variable Dist_wander

`float Dist_wander`

Inc. from: rodney.h *(Section 4.2.15)*

#### 5.1.16 Variable X_final

`long X_final`

Inc. from: rodney.h *(Section 4.2.16)*

#### 5.1.17 Variable Y_final

`long Y_final`

Inc. from: rodney.h *(Section 4.2.17)*

#### 5.1.18 Variable ANG_final

`double ANG_final`

Inc. from: rodney.h *(Section 4.2.18)*

#### 5.1.19 Variable f1

`int f1`

Inc. from: rodney.h *(Section 4.2.19)*

#### 5.1.20 Variable f2

`int f2`

Inc. from: rodney.h *(Section 4.2.20)*

#### 5.1.21 Variable f3

`int f3`

Inc. from: rodney.h *(Section 4.2.21)*

#### 5.1.22 Variable f4

`int f4`

Inc. from: rodney.h *(Section 4.2.22)*



#### 5.1.23 Variable f5

`int f5`

Inc. from: rodney.h *(Section 4.2.23)*

#### 5.1.24 Variable f6

`int f6`

Inc. from: rodney.h *(Section 4.2.24)*

#### 5.1.25 Variable Init_maneuver

`int Init_maneuver`

Inc. from: rodney.h *(Section 4.2.25)*

#### 5.1.26 Variable End_maneuver

`int End_maneuver`

Inc. from: rodney.h *(Section 4.2.26)*

#### 5.1.27 Variable Fr

`float Fr`

Inc. from: rodney.h *(Section 4.2.27)*

#### 5.1.28 Variable SUPRESSOR

`int SUPRESSOR`

Inc. from: rodney.h *(Section 4.2.28)*

#### 5.1.29 Variable first_time

`int first_time`

Inc. from: rodney.h *(Section 4.2.29)*

#### 5.1.30 Variable Mode

Control variables Desired control mode

`int Mode`

Inc. from: rodney.h *(Section 4.2.30)*

#### 5.1.31 Variable Old_Mode

`int Old_Mode`

Inc. from: rodney.h *(Section 4.2.31)*

#### 5.1.32 Variable Command

`int Command`

Inc. from: rodney.h *(Section 4.2.32)*

#### 5.1.33 Variable LEFT_VEL

Desired velocities in pulses/T

`unsigned int LEFT_VEL`

Inc. from: rodney.h *(Section 4.2.33)*

### 5.1.34 Variable RIGHT_VEL

```
unsigned int RIGHT_VEL
```

Inc. from: rodney.h *(Section 4.2.34)*

### 5.1.35 Variable Distance

Desired distance (in mm.)

```
long Distance
```

Inc. from: rodney.h *(Section 4.2.35)*

### 5.1.36 Variable Move_distance

```
int Move_distance
```

Inc. from: rodney.h *(Section 4.2.36)*

### 5.1.37 Variable Must_move

```
int Must_move
```

Inc. from: rodney.h *(Section 4.2.37)*

### 5.1.38 Variable Must_go_ahead

```
int Must_go_ahead
```

Inc. from: rodney.h *(Section 4.2.38)*

### 5.1.39 Variable Moving_forward

```
int Moving_forward
```

Inc. from: rodney.h *(Section 4.2.39)*

### 5.1.40 Variable Door_detected

```
int Door_detected
```

Inc. from: rodney.h *(Section 4.2.40)*

### 5.1.41 Variable Phase_1

```
int Phase_1
```

Inc. from: rodney.h *(Section 4.2.41)*

### 5.1.42 Variable G

Value used by the rotation module

```
int G
```

Inc. from: rodney.h *(Section 4.2.42)*

### 5.1.43 Variable Total_left_pulses

Pulses counted up to now

```
long Total_left_pulses
```

Inc. from: rodney.h *(Section 4.2.43)*

### 5.1.44 Variable Total_right_pulses

```
long Total_right_pulses
```

Inc. from: rodney.h *(Section 4.2.44)*



### 5.1.45 Variable P_T_I

```
long P_T_I
```

Inc. from: rodney.h *(Section 4.2.45)*

### 5.1.46 Variable P_T_D

```
long P_T_D
```

Inc. from: rodney.h *(Section 4.2.46)*

### 5.1.47 Variable Count_left_pulses

Pulses desired for doing a certain movement

```
long Count_left_pulses
```

Inc. from: rodney.h *(Section 4.2.47)*

### 5.1.48 Variable Count_right_pulses

```
long Count_right_pulses
```

Inc. from: rodney.h *(Section 4.2.48)*

### 5.1.49 Variable X

Position/orientation of the robot over the workarea (mm. and radians, respectively)

```
double X
```

Inc. from: rodney.h *(Section 4.2.49)*

### 5.1.50 Variable Y

```
double Y
```

Inc. from: rodney.h *(Section 4.2.50)*

### 5.1.51 Variable ANGLE

```
double ANGLE
```

Inc. from: rodney.h *(Section 4.2.51)*

### 5.1.52 Variable POSX

Robot position on the grid cell map

```
int POSX
```

Inc. from: rodney.h *(Section 4.2.52)*

### 5.1.53 Variable POSY

```
int POSY
```

Inc. from: rodney.h *(Section 4.2.53)*

### 5.1.54 Variable posx_new

```
int posx_new
```

Inc. from: rodney.h *(Section 4.2.54)*

### 5.1.55 Variable posy_new

```
int posy_new
```

Inc. from: rodney.h *(Section 4.2.55)*

#### 5.1.56 Variable ind_centre

`int ind_centre`

Inc. from: rodney.h *(Section 4.2.56)*

#### 5.1.57 Variable Num_beeps

Sound control variables. Any process can change them. Number of beeps we want to emit.

`int Num_beeps`

Inc. from: rodney.h *(Section 4.2.57)*

#### 5.1.58 Variable Interval

Entonation interval; 0 to 7 for ascending, -1 to -7 for descending

`int Interval`

Inc. from: rodney.h *(Section 4.2.58)*

#### 5.1.59 Variable Whisper_freq

Frequency for the whisper

`unsigned int Whisper_freq`

Inc. from: rodney.h *(Section 4.2.59)*

#### 5.1.60 Variable Serial_mode

Serial transmission control variable Possible values are READING, RECEIVING and SENDING

`int Serial_mode`

Inc. from: rodney.h *(Section 4.2.60)*

#### 5.1.61 Variable Rec_buff

The three serial buffers: received binary packet, received commands and bytes to transmit.

`unsigned char Rec_buff[254+1]`

Inc. from: rodney.h *(Section 4.2.61)*

#### 5.1.62 Variable Commands

The three serial buffers: received binary packet, received commands and bytes to transmit.

`unsigned char Commands[254+1]`

Inc. from: rodney.h *(Section 4.2.62)*

#### 5.1.63 Variable Tr_buff

The three serial buffers: received binary packet, received commands and bytes to transmit.

`unsigned char Tr_buff[254+1]`

Inc. from: rodney.h *(Section 4.2.63)*

#### 5.1.64 Local Variables

**cell**
The cell struct

`static struct C cell[300][300/4]`

Inc. from: rodney.h *(Section 4.2.65)*

### 5.2 Functions

#### 5.2.1 Global Function COLISION_DETECTION()

`void* COLISION_DETECTION ( void* pass )`



#### 5.2.2 Global Function C_pos()

`void C_pos ( void )`

#### 5.2.3 Global Function C_pos_vel()

`void C_pos_vel ( void )`

#### 5.2.4 Global Function C_rot()

`void C_rot ( void )`

#### 5.2.5 Global Function C_vel()

`void C_vel ( void )`

#### 5.2.6 Global Function Close_all()

`void* Close_all ( void* pass )`

#### 5.2.7 Global Function Consume()

`int Consume ( unsigned char commands[254+1] )`

#### 5.2.8 Global Function Control()

`void Control ( void )`

#### 5.2.9 Global Function Initialize()

`int Initialize ( int encoder )`

#### 5.2.10 Global Function InitializeMap()

`void InitializeMap ( void )`

#### 5.2.11 Global Function InitializeSonars()

`void InitializeSonars ( void )`

#### 5.2.12 Global Function Initialize_all()

`int Initialize_all ( int init_basic_processes )`

#### 5.2.13 Global Function Latch()

`void Latch ( int encoder )`

#### 5.2.14 Global Function Read_Velocity()

`long Read_Velocity ( int encoder )`

#### 5.2.15 Global Function StopRobot()

`void StopRobot ( void )`

#### 5.2.16 Global Function access_to_ports()

`int access_to_ports ( void )`

#### 5.2.17 Global Function argument_length()

`int argument_length ( unsigned char command_code )`

### 5.2.18 Global Function beep()

```
void beep ( void )
```

### 5.2.19 Global Function end_tone()

```
void end_tone ( void )
```

### 5.2.20 Global Function hard_latch()

```
void hard_latch ( void )
```

### 5.2.21 Global Function init_robot_hardware()

```
void init_robot_hardware ( void )
```

### 5.2.22 Global Function interval()

```
void interval ( void )
```

### 5.2.23 Global Function map()

```
void* map ( void* pass )
```

### 5.2.24 Global Function ptrans()

```
void* ptrans ( void* pass )
```

### 5.2.25 Global Function read_byte_with_conf()

```
int read_byte_with_conf ( unsigned char* pc )
```

### 5.2.26 Global Function receive()

```
int receive ( void )
```

### 5.2.27 Global Function send()

```
int send ( void )
```

### 5.2.28 Global Function serial()

```
void* serial ( void* pass )
```

### 5.2.29 Global Function set_freq()

```
void set_freq ( char note )
```

### 5.2.30 Global Function sonars()

```
void* sonars ( void* pass )
```

### 5.2.31 Global Function sound()

```
void* sound ( void* pass )
```

### 5.2.32 Global Function start_tone()

```
void start_tone ( void )
```

### 5.2.33 Global Function trans_ult_and_sh()

```
void* trans_ult_and_sh ( void* pass )
```



### 5.2.34 Global Function visualize()

```
void* visualize ( void* pass )
```

### 5.2.35 Global Function whisper()

```
void whisper ( void )
```

### 5.2.36 Global Function write_two_bytes()

```
int write_two_bytes ( unsigned char c, unsigned char c2 )
```

## 5.3 Actual file listing

```
#ifndef COMMON
#include "common.h"
#endif

#ifndef RODNEY
#include "rodney.h"
#endif

/* ------------------------------------------------------------------ */

int access_to_ports(void)
{
 if (ioperm(MICROSWITCHES_BASE,DISPLACEMENT1,1))
 {
  perror("ioperm: Error asking for I/O port access to microswitches");
  exit(0);
 }
 if (ioperm(SONARS_BASE_ADD,DISPLACEMENT2,1))
 {
  perror("ioperm: Error asking for I/O port access to sonars");
  exit(0);
 }
 if (ioperm(0x300,DISPLACEMENT3,1))
 {
  perror("ioperm: Error asking for I/O port access to 0x300");
  exit(0);
 }
 if (ioperm(ENCODER_BASE_ADD,DISPLACEMENT4,1))
 {
  perror("ioperm: Error asking for I/O port access to encoders");
  exit(0);
 }
 if (ioperm(BEEP,2,1))
 {
  perror("ioperm: Error asking for I/O port access to loudspeaker");
  exit(0);
 }
 if (ioperm(MUTER,1,1))
 {
  perror("ioperm: Error asking for I/O port access to muter");
  exit(0);
 }
 if (ioperm(SERIAL,8,1))
 {
  perror("ioperm: Error asking for I/O port access to serial port");
  exit(0);
```

```
 }
 return(1);
}

/* --------------------------------------------------------------- */

void *COLISION_DETECTION(void *pass)
{
 BEGIN_BEH_MOD_P
 {
  Microswitches_code=inb(MICROSWITCHES_BASE+3);

  switch(Microswitches_code)
   {
     case 0xEF: Microswitches_angle=22.5;
                stop=1;
break;
     case 0xDF: Microswitches_angle=67.5;
                stop=1;
break;
     case 0xBF: Microswitches_angle=112.5;
                stop=1;
break;
     case 0x7F: Microswitches_angle=157.5;
                stop=1;
break;
     case 0xF7: Microswitches_angle=202.5;
                stop=1;
break;
     case 0xFB: Microswitches_angle=247.5;
                stop=1;
break;
     case 0xFD: Microswitches_angle=292.5;
                stop=1;
                break;
     case 0xFF: Microswitches_angle=0.0;
                stop=0;
break;
     default:   stop=1;
                break;
   }
 }
 END_BEH_MOD_P
}

/* --------------------------------------------------------------- */

void *visualize(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Display_code<100)
    outb((unsigned char)(16*(Display_code/10)+(Display_code%10)),MICROSWITCHES_BASE+1);
   else
    outb((unsigned char)Display_code,MICROSWITCHES_BASE+1);
 }
 END_BEH_MOD_P
}
```



```
/* --------------------------------------------------------------- */

void InitializeSonars(void)
{
 unsigned char lsb,msb,certitude,sensor=1;
 float x;
 int index,i=0,Finished;
 struct timespec nreq,nrem;

 nreq.tv_sec=0;
 nreq.tv_nsec=80000000L;

 for (index=0;index<5;index++)
  {
    i=0;
    sensor=1;
    Finished=0;

    while(!Finished)
      {
        outb(sensor,SONARS_BASE_ADD);
        nanosleep(&nreq,&nrem);

        certitude=inb(SONARS_BASE_ADD+2);

        lsb=inb(SONARS_BASE_ADD);
        msb=inb(SONARS_BASE_ADD+1);

        x=(float)(((msb<<8)|lsb)*SONARS_COEFF*certitude);

        if (x==0)
          sonar_obstacle[i]=NO_OBSTACLE;
        else if ((x>INF_DET_LIMIT) && (x<=SUP_DET_LIMIT))
          sonar_obstacle[i]=OBSTACLE;
        else
          sonar_obstacle[i]=OBSTACLE_VERY_CLOSE;

        sonar_readings[i]=x;

        sensor=sensor<<1;
i++;

        if (sensor==0)
          Finished=1;
      }
  }
}

/* ------------------------------------------------------------------- */

void *sonars(void *pass)
{
 unsigned char lsb,msb,certitude;
 float x;
 static unsigned char Sonar_to_fire=1,Sonar_to_read=NUM_SONARS-1;

 BEGIN_BEH_MOD_P
```

```
 {
    certitude=inb(SONARS_BASE_ADD+2);

    lsb=inb(SONARS_BASE_ADD);
    msb=inb(SONARS_BASE_ADD+1);

    x=(float)(((msb<<8)|lsb)*SONARS_COEFF*certitude);

    if ((x>=INF_DET_LIMIT) && (x<=SUP_DET_LIMIT))
      sonar_obstacle[Sonar_to_read]=OBSTACLE;
    else if ((x>=INF_DET_LIMIT) && (x<=INF_DET_LIMIT+THRESHOLD))
      sonar_obstacle[Sonar_to_read]=OBSTACLE_VERY_CLOSE;
    else
      sonar_obstacle[Sonar_to_read]=NO_OBSTACLE;

    sonar_readings[Sonar_to_read]=x;

    outb(Sonar_to_fire,SONARS_BASE_ADD);

    Sonar_to_fire<<=1;

    if (Sonar_to_fire==0)
      Sonar_to_fire=1;

    Sonar_to_read++;

    if (Sonar_to_read>(NUM_SONARS-1))
      Sonar_to_read=0;
 }
 END_BEH_MOD_P
}

/* ---------------------------------------------------------------- */

void InitializeMap(void)
{
  register unsigned short i,j;

  for (i=0;i<CELL_NUMBER;i++)
   for (j=0;j<CELL_NUMBER/4;j++)
    {
      cell[i][j].x1=NOT_SCANNED;
      cell[i][j].x2=NOT_SCANNED;
      cell[i][j].x3=NOT_SCANNED;
      cell[i][j].x4=NOT_SCANNED;
    }

  return;
}

/* ---------------------------------------------------------------- */

int Initialize(int);

int Initialize(int encoder)
{
  switch (encoder)
  {
```



```
    case 0:  outb(MASTER_RESET,LEFT_ENC_CON);
     outb(INPUT_SETUP,LEFT_ENC_CON);
     outb(CUAD_X1,LEFT_ENC_CON);
     outb(RESET_CONT,LEFT_ENC_CON);
     break;

    case 1:  outb(MASTER_RESET,RIGHT_ENC_CON);
     outb(INPUT_SETUP,RIGHT_ENC_CON);
     outb(CUAD_X1,RIGHT_ENC_CON);
     outb(RESET_CONT,RIGHT_ENC_CON);
     break;

    default: printf("\nError when programming the odometry system");
             exit(1);
  }

  return(0);
}

/* ---------------------------------------------------------------- */

/* Function to transfer the value of the pulse counter circuit */
/* to the latch register for software reading                  */

void Latch(int);

void Latch(int encoder)
{
  switch (encoder)
  {
    case 0:  outb(LATCH_CONT,LEFT_ENC_CON);
             break;

    case 1:  outb(LATCH_CONT,RIGHT_ENC_CON);
             break;

    default: break;
  }
}

/* ---------------------------------------------------------------- */

/* Function to transfer the value of the pulse counter circuit */
/* to the latch register for hardware reading                  */

void hard_latch(void);

void hard_latch(void)
{
  outb(0,LOAD);
}

/* ---------------------------------------------------------------- */

/* Function to read encoder angular velocity */

long Read_Velocity(int);
```

```
long Read_Velocity(int encoder)
{
  long velocity;

  switch (encoder)
  {
    case 0:  outb(ADDR_RESET,LEFT_ENC_CON);
      velocity=(long)inb(LEFT_ENC_DATA);
      velocity|=((long)inb(LEFT_ENC_DATA)<<8);
      velocity|=((long)inb(LEFT_ENC_DATA)<<16);
      outb(RESET_CONT,LEFT_ENC_CON);
      return((velocity<<8)/256);

    case 1:  outb(ADDR_RESET,RIGHT_ENC_CON);
      velocity=(long)inb(RIGHT_ENC_DATA);
      velocity|=((long)inb(RIGHT_ENC_DATA)<<8);
      velocity|=((long)inb(RIGHT_ENC_DATA)<<16);
              outb(RESET_CONT,RIGHT_ENC_CON);
      return((velocity<<8)/256);

    default: return(0L);
  }
}

/* ------------------------------------------------------------------------- */

void *map(void *pass)
{
  double pulses_per_mm;
  double delta_left=0.0,delta_right=0.0;
  double delta_c=0.0,delta_angle=0.0;
  long dif_pulses_left=0L,dif_pulses_right=0L;
  static long pulses_total_left_for=0L,pulses_total_right_for=0L;

  BEGIN_BEH_MOD_P
  {
   pulses_per_mm=(N*R)/(M_PI*d);

   dif_pulses_left=P_T_I-pulses_total_left_for;
   dif_pulses_right=P_T_D-pulses_total_right_for;

   pulses_total_left_for=P_T_I;
   pulses_total_right_for=P_T_D;

   delta_left=(double)dif_pulses_left/pulses_per_mm;
   delta_right=(double)dif_pulses_right/pulses_per_mm;
   delta_c=((delta_left+delta_right)/2);

   delta_angle=((delta_right-delta_left)/(2*r));

   ANGLE+=delta_angle;

   if (ANGLE>M_PI)
     ANGLE-=(2*M_PI);
   if (ANGLE<-M_PI)
     ANGLE+=(2*M_PI);
```



```
   X=X+delta_c*cos(ANGLE);
   Y=Y+delta_c*sin(ANGLE);

  }
 END_BEH_MOD_P
}

/* ------------------------------------------------------------------------- */

void Control(void)
{
  outb(Command,TURNING_DIRECTION);

  switch(Mode)
  {
     case C_VEL:     Display_code=10;
                     C_vel();
     break;
     case C_POS:     Display_code=20;
                     C_pos();
     break;
     case C_POS_VEL: Display_code=30;
                     C_pos_vel();
     break;
     case C_ROT:     Display_code=40;
                     C_rot();
     break;
     case STOP_M:    Display_code=50;
     StopRobot();
     break;
     default:        break;
  }

  return;
}

/* -------------------------------------------------------------------- */

void C_rot(void)
{
 long Veloc_left=0L,Veloc_right=0L;
 float K_p=0.003,K_d=0.001;
 int Delta_Error=100;
 long Error_left=(long)Delta_Error+1,Error_right=(long)Delta_Error+1;
 unsigned char Power_left=MIN_POW+1,Power_right=MIN_POW+1;
 int turn_dir=Command;

 static long Error_left_for=0L,Error_right_for=0L;
 static long Pos_left=0L,Pos_right=0L;
 static int index=0;

 if (Rotation==ROT_FINISHED)
   {
     Rotation=NO_ROT_FINISHED;
     Error_left_for=0L;
     Error_right_for=0L;
     Pos_left=0L;
     Pos_right=0L;
```

```
    index=0;
  }

Count_left_pulses=(long)((Angle_to_rotate*r*N*R)/(M_PI*d));
Count_right_pulses=(long)((Angle_to_rotate*r*N*R)/(M_PI*d));

if ((Angle_to_rotate>=0.0)&&(Angle_to_rotate<=0.01))
  {
    Rotation=ROT_FINISHED;
    Must_rotate=NO;
    f1=NO;
    if (f3)
      f2=NO;
    else
      f2=YES;
    f3=NO;

    if (f4)
      {
f4=NO;
        f5=YES;
      }
  }

else

switch(Command)
 {
   case RIGTH_TURN:

     if ((labs(Error_left)>Delta_Error) || (labs(Error_right)>Delta_Error))
      {
        Latch(LEFT_ENCODER);
        Veloc_left=Read_Velocity(LEFT_ENCODER);
        Pos_left+=Veloc_left;
P_T_I+=Veloc_left;

        Latch(RIGHT_ENCODER);
        Veloc_right=Read_Velocity(RIGHT_ENCODER);
        Pos_right+=Veloc_right;
P_T_D+=Veloc_right;

        Error_left_for=Error_left;
        Error_right_for=Error_right;

        Error_left=Count_left_pulses-labs(Pos_left);
        Error_right=Count_right_pulses-labs(Pos_right);

        if (Error_left>Delta_Error)
         {
   turn_dir &= 0x02;
   outb(turn_dir,TURNING_DIRECTION);
   Power_left=(unsigned char)(BASE_POW+K_p*(float)Error_left+
K_d*(float)(Error_left_for-Error_left));
         }
        else if (Error_left<(-Delta_Error))
         {
   turn_dir |= 0x01;
```



```
   outb(turn_dir,TURNING_DIRECTION);
           Power_left=(unsigned char)(BASE_POW-K_p*(float)Error_left-
K_d*(float)(Error_left_for-Error_left));
         }

        if (Error_right>Delta_Error)
         {
   turn_dir |= 0x02;
   outb(turn_dir,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW+K_p*(float)Error_right+
             K_d*(float)(Error_right_for-Error_right));
         }
        else if (Error_right<(-Delta_Error))
         {
   turn_dir &= 0x01;
   outb(turn_dir,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW-K_p*(float)Error_right-
             K_d*(float)(Error_right_for-Error_right));
         }

        if ((Error_left<=Delta_Error) && (Error_left>=(-Delta_Error)) &&
   (Error_right<=Delta_Error) && (Error_right>=(-Delta_Error)))
         {
   Power_left=MIN_POW;
   Power_right=MIN_POW;
   index++;
  }

if (index>MAX_ROT_LOOPS)
  {
    Rotation=ROT_FINISHED;
            Must_rotate=NO;
    f1=NO;
    if (f3)
      f2=NO;
    else
      f2=YES;
    f3=NO;

    if (f4)
      {
        f4=NO;
f5=YES;
      }
    }

        outb(Power_left,LEFT_MOTOR);
        outb(Power_right,RIGHT_MOTOR);
      }
     break;

   case LEFT_TURN:

     if ((labs(Error_left)>Delta_Error) || (labs(Error_right)>Delta_Error))
      {
        Latch(LEFT_ENCODER);
        Veloc_left=Read_Velocity(LEFT_ENCODER);
        Pos_left+=Veloc_left;
```

```
P_T_I+=Veloc_left;

        Latch(RIGHT_ENCODER);
        Veloc_right=Read_Velocity(RIGHT_ENCODER);
        Pos_right+=Veloc_right;
P_T_D+=Veloc_right;

        Error_left_for=Error_left;
        Error_right_for=Error_right;

        Error_left=Count_left_pulses-labs(Pos_left);
        Error_right=Count_right_pulses-labs(Pos_right);

        if (Error_left>Delta_Error)
         {
   turn_dir |= 0x01;
   outb(turn_dir,TURNING_DIRECTION);
   Power_left=(unsigned char)(BASE_POW+K_p*(float)Error_left+
K_d*(float)(Error_left_for-Error_left));
         }

if (Error_left<(-Delta_Error))
         {
   turn_dir &= 0x02;
   outb(turn_dir,TURNING_DIRECTION);
           Power_left=(unsigned char)(BASE_POW-K_p*(float)Error_left-
K_d*(float)(Error_left_for-Error_left));
         }

        if (Error_right>Delta_Error)
         {
   turn_dir &= 0x01;
   outb(turn_dir,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW+K_p*(float)Error_right+
               K_d*(float)(Error_right_for-Error_right));
         }

if (Error_right<(-Delta_Error))
         {
   turn_dir |= 0x02;
   outb(turn_dir,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW-K_p*(float)Error_right-
               K_d*(float)(Error_right_for-Error_right));
         }

        if ((Error_left<=Delta_Error) && (Error_left>=(-Delta_Error)) &&
   (Error_right<=Delta_Error) && (Error_right>=(-Delta_Error)))
         {
   Power_left=MIN_POW;
   Power_right=MIN_POW;
   index++;
 }

if (index>MAX_ROT_LOOPS)
  {
    Rotation=ROT_FINISHED;
            Must_rotate=NO;
    f1=NO;
```



```
     if (f3)
       f2=NO;
     else
       f2=YES;
     f3=NO;

     if (f4)
       {
         f4=NO;
 f5=YES;
       }
   }

         outb(Power_left,LEFT_MOTOR);
         outb(Power_right,RIGHT_MOTOR);
       }
      break;

     default: break;
  }
 return;
}

/* -------------------------------------------------------------- */

void C_pos(void)
{
 long Veloc_left=0L,Veloc_right=0L;
 static long Pos_left=0L,Pos_right=0L;
 float K_p=0.003,K_d=0.002;
 int Delta_Error=100;   /* Allowed error (pulses). 1 pulse is 0.014 mm. */
 long Error_left=(long)Delta_Error+1,Error_right=(long)Delta_Error+1;
 static long Error_left_for=0L,Error_right_for=0L;
 unsigned char Power_left=0x00,Power_right=0x00;

 Count_left_pulses=(long)((Distance*N*R)/(M_PI*d));
 Count_right_pulses=(long)((Distance*N*R)/(M_PI*d));

 switch(Command)
  {
    case FORWARDS:
      if ((labs(Error_left)>Delta_Error) || (labs(Error_right)>Delta_Error))
       {
         Latch(LEFT_ENCODER);
         Veloc_left=Read_Velocity(LEFT_ENCODER);
         Pos_left+=(Veloc_left);
 Total_left_pulses+=Pos_left;
 P_T_I+=Veloc_left;

         Latch(RIGHT_ENCODER);
         Veloc_right=Read_Velocity(RIGHT_ENCODER);
         Pos_right+=(Veloc_right);
 Total_right_pulses+=Pos_right;
 P_T_D+=Veloc_right;

         Error_left_for=Error_left;
         Error_right_for=Error_right;
```

```
        Error_left=Count_left_pulses-labs(Pos_left);
        Error_right=Count_right_pulses-labs(Pos_right);

        if (Error_left>Delta_Error)
         {
   Command&=0x02;
   outb(Command,TURNING_DIRECTION);
   Power_left=(unsigned char)(BASE_POW+K_p*(float)Error_left+
 K_d*(float)(Error_left_for-Error_left));
         }
        else if (Error_left<(-Delta_Error))
         {
   Command|=0x01;
   outb(Command,TURNING_DIRECTION);
           Power_left=(unsigned char)(BASE_POW-K_p*(float)Error_left-
 K_d*(float)(Error_left_for-Error_left));
         }

        if (Error_right>Delta_Error)
         {
   Command&=0x01;
   outb(Command,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW+K_p*(float)Error_right+
               K_d*(float)(Error_right_for-Error_right));
         }
        else if (Error_right<(-Delta_Error))
         {
   Command|=0x02;
   outb(Command,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW-K_p*(float)Error_right-
               K_d*(float)(Error_right_for-Error_right));
         }

        if ((Error_left<=Delta_Error) && (Error_left>=(-Delta_Error)) &&
   (Error_right<=Delta_Error) && (Error_right>=(-Delta_Error)))
         {
   Power_left=MIN_POW;
   Power_right=MIN_POW;
         }

        outb(Power_left,LEFT_MOTOR);
        outb(Power_right,RIGHT_MOTOR);
      }
     break;

    case BACKWARDS:
     if ((labs(Error_left)>Delta_Error) || (labs(Error_right)>Delta_Error))
      {
        Latch(LEFT_ENCODER);
        Veloc_left=Read_Velocity(LEFT_ENCODER);
        Pos_left+=(Veloc_left);
 Total_left_pulses+=Pos_left;
 P_T_I+=Veloc_left;

        Latch(RIGHT_ENCODER);
        Veloc_right=Read_Velocity(RIGHT_ENCODER);
        Pos_right+=(Veloc_right);
 Total_right_pulses+=Pos_right;
```



```
 P_T_D+=Veloc_right;

        Error_left_for=Error_left;
        Error_right_for=Error_right;

        Error_left=Count_left_pulses-labs(Pos_left);
        Error_right=Count_right_pulses-labs(Pos_right);

        if (Error_left>Delta_Error)
         {
   Command|=0x01;
   outb(Command,TURNING_DIRECTION);
   Power_left=(unsigned char)(BASE_POW+K_p*(float)Error_left+
 K_d*(float)(Error_left_for-Error_left));
         }
        else if (Error_left<(-Delta_Error))
         {
   Command&=0x02;
   outb(Command,TURNING_DIRECTION);
           Power_left=(unsigned char)(BASE_POW-K_p*(float)Error_left-
 K_d*(float)(Error_left_for-Error_left));
         }

        if (Error_right>Delta_Error)
         {
   Command|=0x02;
   outb(Command,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW+K_p*(float)Error_right+
               K_d*(float)(Error_right_for-Error_right));
         }
        else if (Error_right<(-Delta_Error))
         {
   Command&=0x01;
   outb(Command,TURNING_DIRECTION);
   Power_right=(unsigned char)(BASE_POW-K_p*(float)Error_right-
               K_d*(float)(Error_right_for-Error_right));
         }

        if ((Error_left<=Delta_Error) && (Error_left>=(-Delta_Error)) &&
   (Error_right<=Delta_Error) && (Error_right>=(-Delta_Error)))
         {
   Power_left=MIN_POW;
   Power_right=MIN_POW;
         }

        outb(Power_left,LEFT_MOTOR);
        outb(Power_right,RIGHT_MOTOR);
      }
     break;

    default: break;
  }
 return;
}

/* -------------------------------------------------------------- */

void C_vel(void)
```

```
{
 long Veloc_left=0L,Veloc_right=0L,M=0L;
 long Total_pulse_diff=0L;
 float K_p1=0.01,K_i1=0.1,K_p2=0.5,K_i2=0.01;
 long Error_left=0L,Error_right=0L,Error_dif_i=0L,Error_dif_d=0L,Dif_pulses_i=0L;
 long Error_dif_left=0L,Error_dif_right=0L;

 static long Error_left_for=0L,Error_right_for=0L;
 static unsigned int Power_left=MIN_POW,Power_right=MIN_POW;

 if (Moving_forward==NO)
   {
    Moving_forward=YES;
    Error_left_for=0L;
    Error_right_for=0L;
    Power_left=MIN_POW;
    Power_right=MIN_POW;
   }

 Latch(LEFT_ENCODER);
 Veloc_left=Read_Velocity(LEFT_ENCODER);

 Latch(RIGHT_ENCODER);
 Veloc_right=Read_Velocity(RIGHT_ENCODER);

 Total_left_pulses+=Veloc_left;
 Total_right_pulses+=Veloc_right;

 P_T_I+=Veloc_left;
 P_T_D+=Veloc_right;

 Total_pulse_diff=Total_left_pulses-Total_right_pulses;
 Dif_pulses_i=Veloc_left-Veloc_right+G;

 M=K_p2*Dif_pulses_i+K_i2*Total_pulse_diff;

 Error_dif_i=Error_dif_d=M;

 Error_left=(long)LEFT_VEL-Veloc_left-Error_dif_i;
 Error_right=(long)RIGHT_VEL-Veloc_right+Error_dif_d;

 Error_dif_left=Error_left-Error_left_for;
 Error_dif_right=Error_right-Error_right_for;

 Power_left+=(unsigned int)(Error_left*K_i1+Error_dif_left*K_p1);
 Power_right+=(unsigned int)(Error_right*K_i1+Error_dif_right*K_p1);

 if (Power_left>MAX_POW) Power_left=0x45;
 if (Power_right>MAX_POW) Power_right=0x45;

 Error_left_for=Error_left;
 Error_right_for=Error_right;

 outb((unsigned char)Power_left,LEFT_MOTOR);
 outb((unsigned char)Power_right,RIGHT_MOTOR);

 return;
}
```



```
/* --------------------------------------------------------------- */

void StopRobot(void)
{
  outb(0,LEFT_MOTOR);
  outb(0,RIGHT_MOTOR);
  LEFT_VEL=0;
  RIGHT_VEL=0;

  return;
}

/* --------------------------------------------------------------- */

void C_pos_vel(void)
{
 long Veloc_left=0L,Veloc_right=0L,M=0L;
 long Total_pulse_diff=0L;
 float K_p1=0.01,K_i1=0.1,K_p2=0.5,K_i2=0.01;
 long Error_left=0L,Error_right=0L,Error_dif_i=0L,Error_dif_d=0L,Dif_pulses_i=0L;
 long Error_dif_left=0L,Error_dif_right=0L;

 static long Error_left_for=0L,Error_right_for=0L;
 static unsigned int Power_left=MIN_POW,Power_right=MIN_POW;

 if (Move_distance==ROT_FINISHED)
   {
    Move_distance=NO_ROT_FINISHED;
    Error_left_for=0L;
    Error_right_for=0L;
    Power_left=MIN_POW;
    Power_right=MIN_POW;
    Count_left_pulses=P_T_I+(long)((Distance*N*R)/(M_PI*d));
    Count_right_pulses=P_T_D+(long)((Distance*N*R)/(M_PI*d));
   }

 if ((P_T_I<(Count_left_pulses-THRESHOLD_POS)) &&
      (P_T_D<(Count_right_pulses-THRESHOLD_POS)))
    {
     Latch(LEFT_ENCODER);
     Veloc_left=Read_Velocity(LEFT_ENCODER);
     Total_left_pulses+=Veloc_left;
     P_T_I+=Veloc_left;

     Latch(RIGHT_ENCODER);
     Veloc_right=Read_Velocity(RIGHT_ENCODER);
     Total_right_pulses+=Veloc_right;
     P_T_D+=Veloc_right;

     Total_pulse_diff=Total_left_pulses-Total_right_pulses+G;
     Dif_pulses_i=Veloc_left-Veloc_right+G;

     M=K_p2*Dif_pulses_i+K_i2*Total_pulse_diff;

     Error_dif_i=Error_dif_d=M;

     Error_left=(long)LEFT_VEL-Veloc_left-Error_dif_i;
```

```
      Error_right=(long)RIGHT_VEL-Veloc_right+Error_dif_d;

      Error_dif_left=Error_left-Error_left_for;
      Error_dif_right=Error_right-Error_right_for;

      Power_left+=(unsigned int)(Error_left*K_i1+Error_dif_left*K_p1);
      Power_right+=(unsigned int)(Error_right*K_i1+Error_dif_right*K_p1);

      if (Power_left>MAX_POW) Power_left=0x40;
      if (Power_right>MAX_POW) Power_right=0x40;

      Error_left_for=Error_left;
      Error_right_for=Error_right;

      outb((unsigned char)Power_left,LEFT_MOTOR);
      outb((unsigned char)Power_right,RIGHT_MOTOR);
     }
 else
     {
      Move_distance=ROT_FINISHED;
      Must_move=NO;
      f2=NO;
      f3=YES;
      if (f5)
        {
   f5=NO;
   f6=YES;
 }
     }

 return;
}

/* ------------------------------------------------------------------ */

void init_robot_hardware(void)
{
 int i;

 for (i=0;i<NUM_SONARS;i++)
  sonar_readings[i]=0.0;

 if (access_to_ports())
  printf("\nAccess to ports granted.");
 else
  exit(0);

 outb(0,SONARS_BASE_ADD);

 Initialize(LEFT_ENCODER);
 Initialize(RIGHT_ENCODER);

 InitializeMap();

 outb(FORWARDS,TURNING_DIRECTION);
}

/* ------------------------------------------------------------------------ */
```



```
void set_freq(char note)
{
 static unsigned char notes[9][2]={{8,232},{7,240},{7,18},{6,152},
                                   {5,242},{5,76},{4,184},{4,116},{2,58}};
 unsigned char low,high;

 if (
     ((note>='c') && (note<='g')) ||
     ((note>='A') && (note<='C'))
    )
 {
  if (note>='c')
  {
   low=notes[note-'c'][0];
   high=notes[note-'c'][1];
  }
  else
  {
   low=notes[note-'A'+5][0];
   high=notes[note-'A'+5][1];
  }
 }
 else
 {
  low=notes[8][0];
  high=notes[8][1];
 }
 outb(0xB6,BEEP+1);
 outb(high,BEEP);
 outb(low,BEEP);
}

/* ------------------------------------------------------------------------ */

void start_tone(void)
{
 unsigned char ret;

 ret=inb(MUTER);
 ret|=0x3;
 outb(ret,MUTER);
}

/* ------------------------------------------------------------------------ */

void end_tone(void)
{
 unsigned char ret;

 ret=inb(MUTER);
 ret&=~0x3;
 outb(ret,MUTER);
}

/* ------------------------------------------------------------------------ */

void beep(void)
```

```
{
 static int init_freq=1,count=0;

 if ((init_freq) && (Num_beeps>0))
 {
  set_freq('h');
  init_freq=0;
 }
 if (Num_beeps>0)
 {
  if (count==0)
   start_tone();
  if (count==BEEP_DUR)
   end_tone();
  if (count==3*BEEP_DUR)
  {
   count=0;
   Num_beeps--;
  }
  else
   count++;
 }
 if (Num_beeps==0)
  init_freq=1;
}

/* ------------------------------------------------------------------------ */

void interval(void)
{
 static int init_freq=1,count=0;
 static char symbols[9]={'c','d','e','f','g','A','B','C','h'};

 if (Interval==NO_INTERVAL)
  return;

 if (init_freq==1)
 {
  if ((Interval>7) || (Interval<-7))
   set_freq('h');
  else
   set_freq((Interval>0) ? symbols[0] : symbols[-Interval]);
 }

 if (init_freq==2)
 {
  if ((Interval>7) || (Interval<-7))
   set_freq('h');
  else
   set_freq((Interval>0) ? symbols[Interval] : symbols[0]);
 }

 if ((count==0) || (count==2*BEEP_DUR))
 {
  start_tone();
 }

 if (count==BEEP_DUR)
```



```
 {
  end_tone();
  init_freq++;
 }

 count++;

 if (count==3*BEEP_DUR)
 {
  end_tone();
  init_freq=1;
  count=0;
  Interval=NO_INTERVAL;
 }
}

/* ------------------------------------------------------------------------ */

void whisper(void)
{
 unsigned char low,high;
 unsigned int freq;
 static unsigned int old_whisper_freq=0;
 static unsigned char playing=0;

 if (Whisper_freq!=old_whisper_freq)
 {
  if (Whisper_freq)
  {
   freq=(unsigned int)(1193180.0/(float)Whisper_freq+0.5);
   low=(unsigned char)(freq%256);
   high=(unsigned char)(freq/256);
   outb(0xB6,BEEP+1);
   outb(low,BEEP);
   outb(high,BEEP);
   if (!playing)
   {
    start_tone();
    playing=1;
   }
  }
  else
  {
   end_tone();
   playing=0;
  }
  old_whisper_freq=Whisper_freq;
 }
 return;
}

/* ------------------------------------------------------------------------ */

void *sound(void *pass)
{
 BEGIN_BEH_MOD_P
 {
  if (Num_beeps>0)
```

```
  beep();
 else
  if (Interval!=NO_INTERVAL)
   interval();
  else
   if (Whisper_freq)
    whisper();
 }
 END_BEH_MOD_P
}

/* ------------------------------------------------------------------------ */

int read_byte_with_conf(unsigned char *pc)
{
 int try;

 /* It tries for NUM_TRIAL times to see if a byte is ready to be read */
 try=0;
 while ((!SOMETHING_TO_READ) && (try<NUM_TRIALS))
  try++;
 if (try<NUM_TRIALS)
  *pc=inb(SERIAL);
 else
  return(NOT_READY_TO_READ);

 for (try=0;try<7000;try++);

 /* After the byte has been read, it tries to write the confirmation code */
 /* and returns if this operation has been successful or not             */

 /* It checks the register to know if we can write for NUM_TRIAL times */
 try=0;
 while ((!(inb(SERIAL+5) & 0x20)) && (try<NUM_TRIALS))
  try++;
 if (try<NUM_TRIALS)
 {
  outb(CONF_CODE,SERIAL);
  return(OK);
 }
 else
  return(NOT_READY_TO_WRITE);
}

/* ------------------------------------------------------------------------ */

int write_two_bytes(unsigned char c,unsigned char c2)
{
 long try;
 unsigned char serial_code;

 /* It checks the register to know if we can write for NUM_TRIAL times */
 try=0;
 serial_code=inb(SERIAL+5);
 while ((!(serial_code & 0x20)) && (try<NUM_TRIALS))
 {
  try++;
  serial_code=inb(SERIAL+5);
```



```
 }
 if (try<NUM_TRIALS)
  outb(c,SERIAL);
 else
  return(NOT_READY_TO_WRITE);

 /* A dirty trick to allow serial port to be ready again. */
 /* Doing nothing for a little bit                        */
 /* To be improved... */
 for (try=0;try<6000;try++);

 try=0;
 while ((!(serial_code & 0x20)) && (try<NUM_TRIALS))
 {
  try++;
  serial_code=inb(SERIAL+5);
 }
 if (try<NUM_TRIALS)
 {
  outb(c2,SERIAL);
  return(OK);
 }
 else
  return(NOT_READY_TO_WRITE);
}

/* ------------------------------------------------------------------------ */

int argument_length(unsigned char command_code)
{
 /* Constants for command codes are defined at rodney.h */
 switch (command_code)
 {
  /* quit */
  case QUIT: return(sizeof(int));
  /* stop */
  case STOP: return(0);
  /* move */
  case MOVE_V: return(sizeof(int));
  /* moved */
  case MOVE_D: return(2*sizeof(int));
  /* rot */
  case ROT: return(sizeof(int));
  /* rotw */
  case ROT_W: return(2*sizeof(int));
  /* sonars */
  case ULT_SHOT: return(sizeof(int));
  /* sound */
  case SOUND: return(sizeof(int));
  default: return(0);
 }
}

/* ------------------------------------------------------------------------ */

int receive(void)
{
 unsigned char c;
```

```
int ret;
static int receiving_length=0,receiving_bin_packet=0;
static int receiving_command_arguments=0,length=0;

ret=read_byte_with_conf(&c);
if (ret!=OK)
 return(ret);

/* If it is the character that indicates that a binary packet is to arrive... */
if (c==START_BIN_PACK)
 /* Check that the receiving buffer has empty place */
 if ((int)Rec_buff[0]>BUF_LEN-1)
  return(REC_BUFF_NOT_EMPTY);
 /* If it has, the next byte will be received in next loop will be the */
 /* length (in bytes) of the binary packet. */
 else
 {
  receiving_length=1;
  return(STILL_RECEIVING);
 }

/* We are about to receive the length of the binary packet. */
if (receiving_length)
{
 /* Check that it is not too long... */
 if ((int)c>BUF_LEN)
 {
  length=0;
  receiving_length=0;
  receiving_bin_packet=0;
  return(TOO_LONG_LEN_REC);
 }
 /* If not, take note of it in the first position of the reception buffer */
 Rec_buff[0]=c;
 length=0;
 receiving_length=0;
 receiving_bin_packet=1;
 return(STILL_RECEIVING);
}

/* Here, we are receiving the actual bytes. We know how many. */
if (receiving_bin_packet)
{
 length++;
 Rec_buff[length]=c;
 /* When we have receieved the last one, we will be ready again. */
 if (length==(int)Rec_buff[0])
 {
  receiving_bin_packet=0;
  length=0;
  /* Bytes are now in the buffer. SOMEONE ELSE WILL HAVE TO CONSUME THEM. */
  return(RECEPTION_FINISHED);
 }
 else
  return(STILL_RECEIVING);
}

if (receiving_command_arguments)
```



```
 {
  /* Arguments are stored after the command. */
  Commands[Commands[1]+2]=c;
  Commands[1]++;
  receiving_command_arguments--;
  if (receiving_command_arguments)
   return(STILL_RECEIVING);
  else
  {
   Commands[0]=COMMAND_READY;
   return(RECEPTION_FINISHED);
  }
 }

 /* If the received byte was not the binary packet start mark, neither it */
 /* is part of a binary packet, neither is part of a command argument,    */
 /* then it is a command. If there is empty room in the Commands buffer,  */
 /* put the byte in it.                                                    */
 /* Structure of the command buffer is:                                    */
 /*  Commands[0] indicates if there are commands ready to be consumed.    */
 /*  Commands[1] indicates the length of the buffer.                      */
 /*  Commands[2 to BUFF_LEN] is the buffer itself.                        */
 if (Commands[1]<BUF_LEN)
 {
  Commands[0]=COMMAND_NOT_READY;
  Commands[(int)Commands[1]+2]=c;
  Commands[1]++;
  /* We check the length of the argument list of this command. */
  /* It might be zero, some commands have no arguments.        */
  if ((receiving_command_arguments=argument_length(c))!=0)
   return(STILL_RECEIVING);
  else
  {
   Commands[0]=COMMAND_READY;
   return(receiving_command_arguments ? STILL_RECEIVING : RECEPTION_FINISHED);
  }
 }
 /* SOMEONE ELSE WILL HAVE TO CONSUME THE COMMANDS FROM THE COMMAND BUFFER */
 /* If not, it will be sooner or later full, and in this case commands are */
 /* simply ignored.                                                         */
 else
  return(RECEPTION_FINISHED);

}

/* ---------------------------------------------------------------------- */

int send(void)
{
 unsigned char c,c2;
 int ret;
 static int trans_pos=1;

 if (trans_pos<=(int)Tr_buff[0])
 {
  c=Tr_buff[trans_pos];
  trans_pos++;
```

```
  c2=(trans_pos>Tr_buff[0]) ? NOT_A_CODE : Tr_buff[trans_pos];
  trans_pos++;

  ret=write_two_bytes(c,c2);

  if (ret!=OK)
   return(ret);
 }

 if (trans_pos>Tr_buff[0])
 {
  trans_pos=1;
  Tr_buff[0]=0;
  return(TRANSMISION_FINISHED);
 }
 else
  return(STILL_TRANSMITTING);
}

/* ------------------------------------------------------------------------ */

void *serial(void *pass)
{
 static int init=1;
 unsigned divider;
 unsigned char pal;
 int ret;

 BEGIN_BEH_MOD_P
 {
  if (init)
  {
   /* Only the first time we come into here serial port is initialized */
   divider=(unsigned)(MAX_BAUD/SERIAL_VEL);
   outb(0x80,SERIAL+3);
   outb(divider%256,SERIAL);
   outb(divider/256,SERIAL+1);
   pal=PAL_LENGTH | STOP_BITS | PARITY;
   outb(pal,SERIAL+3);
   outb(0,SERIAL+1);
   for (ret=0;ret<BUF_LEN;ret++)
    Rec_buff[ret]=Tr_buff[ret]=Commands[ret]=0;
   init=0;
  }

  switch (Serial_mode)
  {
   /* Listen to what comes form the user is the most important task. */
   /* We will not emit new information until we are sure that we don't */
   /* have nothing else to read. */
   case READING: if (SOMETHING_TO_READ)
                 {
  ret=receive();
  switch(ret)
  {
   case STILL_RECEIVING:
          Serial_mode=RECEIVING;
          break;
```



```
   case RECEPTION_FINISHED:
          Serial_mode=((int)Tr_buff[0]) ? SENDING : READING;
          break;
   default: /* receive is returning an error... */
          break;
  }
 }
                 else
                  Serial_mode=((int)Tr_buff[0]) ? SENDING : READING;
                 break;
   case RECEIVING:
                 if (SOMETHING_TO_READ)
                 {
  ret=receive();
  switch(ret)
  {
   case STILL_RECEIVING:
          Serial_mode=RECEIVING;
          break;
   case RECEPTION_FINISHED:
          Serial_mode=((int)Tr_buff[0]) ? SENDING : READING;
          break;
   default: /* receive is returning an error... */
          break;
  }
 }
                 break;
   case SENDING: if ((int)Tr_buff[0])
                 {
  ret=send();
  switch(ret)
  {
   case STILL_TRANSMITTING:
          Serial_mode=SENDING;
          break;
   case TRANSMISION_FINISHED:
          Serial_mode=READING;
          break;
   default: /* send is rturning an error... */
          break;
                  }
 }
 else
                  Serial_mode=READING;
                 break;
   default: break;
  }
 }
 END_BEH_MOD_P
}

/* ------------------------------------------------------------------------ */

int Consume(unsigned char commands[BUF_LEN+1])
{
 int i,code,arg[8];

 i=2;
```

```
 while (i<commands[1]+2)
 {
  /* Let's look at this command's code... */
  code=(int)commands[i];
  i++;
  switch (code)
  {
   case QUIT: return(0); /* Argument to quit is ignored, unless needed in the future */
   case STOP: arg[0]=(int)(commands+i);
              i+=argument_length(STOP)+1;
              Display_code=(int)STOP;
              break;
   case MOVE_V: arg[0]=(int)(commands+i);
                i+=argument_length(MOVE_V)+1;
                Display_code=(int)MOVE_V;
                break;
   case MOVE_D: arg[0]=(int)(commands+i);
                arg[1]=(int)(commands+i+sizeof(int));
                i+=argument_length(MOVE_D)+1;
                Display_code=(int)MOVE_D;
                break;
   case ROT: arg[0]=(int)(commands+i);
             i+=argument_length(ROT)+1;
             Display_code=(int)ROT;
             break;
   case ROT_W: arg[0]=(int)(commands+i);
               arg[1]=(int)(commands+i+sizeof(int));
               i+=argument_length(ROT_W)+1;
               Display_code=(int)ROT_W;
               break;
   case ULT_SHOT: arg[0]=(int)(commands+i);
                  i+=argument_length(ULT_SHOT)+1;
                  Display_code=(int)ULT_SHOT;
                  break;
   case SOUND: arg[0]=(int)(commands+i);
               i+=argument_length(SOUND)+1;
               Display_code=(int)SOUND;
               break;
   default: Display_code=55;
            break;
  }
 }
 /* Once the buffer of commands has been consumed, it is emptyed */
 commands[0]=COMMAND_NOT_READY;
 commands[1]=(unsigned char)0;

 return(1);
}

/* -------------------------------------------------------------------------- */

void *ptrans(void *pass)
{
 int i,len;
 unsigned char *p_to_pos;

 BEGIN_BEH_MOD_P
 {
```



```
 if ((Rec_buff[0]!=(unsigned char)0) && (Tr_buff[0]+Rec_buff[0]<BUF_LEN))
 {
  for (i=1;i<(int)Rec_buff[0]+1;i++)
   Tr_buff[i]=Rec_buff[i];
  Tr_buff[i]=10;
  Tr_buff[0]=Rec_buff[0]+1;
  Rec_buff[0]=(unsigned char)0;
 }
 else
  if (Tr_buff[0]+Rec_buff[0]>=BUF_LEN)
   Rec_buff[0]=(unsigned char)0;

 if ((Commands[0]==COMMAND_READY) && (Tr_buff[0]+Commands[1]<BUF_LEN))
 {
  i=2;
  while (i<(int)Commands[1]+2)
  {
   Tr_buff[Tr_buff[0]+1]=Commands[i]+DIV_CODE;
   len=argument_length(Commands[i]);
   Tr_buff[Tr_buff[0]+2]=(unsigned char)len;
   Tr_buff[0]+=2;
   i+=(len+1);
  }
  if (Consume(Commands)==0)
   stop=1;
 }
 else
  if (Tr_buff[0]+Commands[1]>=BUF_LEN)
  {
   Commands[0]=COMMAND_NOT_READY;
   Commands[1]=(unsigned char)0;
  }

 /* Sonar readings are put into the transmission buffer */

 if ((Son_buff[0]==SONAR_DATA_READY) && (Tr_buff[0]+NUM_SONARS+1<BUF_LEN))
 {
  Tr_buff[Tr_buff[0]+1]=SONAR_TRANSMIT_CODE;
  Tr_buff[0]++;
  for (i=0;i<NUM_SONARS;i++)
  {
   Tr_buff[Tr_buff[0]+1]=Son_buff[i+1];
   Tr_buff[0]++;
  }
  Son_buff[0]=SONAR_DATA_NOT_READY;
 }
 else
  if (Tr_buff[0]+NUM_SONARS+1>=BUF_LEN)
   Son_buff[0]=SONAR_DATA_NOT_READY;

 /* Current sharings are put into the buffer, too */
 if ((Sh_buff[0]==SHARING_DATA_READY) && (Tr_buff[0]+MAX_SH_PROC+1<BUF_LEN))
 {
  Tr_buff[Tr_buff[0]+1]=SHARING_TRANSMIT_CODE;
  Tr_buff[0]++;
  i=1;
  do
```

```
    {
     Tr_buff[Tr_buff[0]+1]=Sh_buff[i];
     Tr_buff[0]++;
     i++;
    }
   while ((Sh_buff[i-1]!=NO_MORE_SHARINGS) && (i<MAX_SH_PROC));
   Sh_buff[0]=SHARING_DATA_NOT_READY;
  }
  else
   if (Tr_buff[0]+MAX_SH_PROC+1>=BUF_LEN)
    Sh_buff[0]=SHARING_DATA_NOT_READY;

  /* and finally, position and orientation */
  if ((Pos_or_buff[0]==POS_OR_DATA_READY) && (Tr_buff[0]+LENGTH_POS_OR+1<BUF_LEN))
  {
   Tr_buff[Tr_buff[0]+1]=POS_OR_TRANSMIT_CODE;
   Tr_buff[0]++;
   p_to_pos=(unsigned char*)(&Pos_or_buff[1]);
   for (i=0;i<LENGTH_POS_OR;i++)
    {
     Tr_buff[Tr_buff[0]+1]=*p_to_pos;
     Tr_buff[0]++;
     p_to_pos++;
    }
   Pos_or_buff[0]=POS_OR_DATA_NOT_READY;
  }
  else
   if (Tr_buff[0]+LENGTH_POS_OR>=BUF_LEN)
    Pos_or_buff[0]=POS_OR_DATA_NOT_READY;
 }
 END_BEH_MOD_P
}

/* ------------------------------------------------------------------------ */

void *trans_ult_and_sh(void *pass)
{
 static int count_u=0,count_s=0,count_pos_or=0;
 int i;

 BEGIN_BEH_MOD_P
 {
  if (count_u==TIME_TO_TRANSMIT_SONARS)
  {
   Son_buff[0]=SONAR_DATA_READY;
   for (i=0;i<NUM_SONARS;i++)
    if (sonar_obstacle[i]==NO_OBSTACLE)
     Son_buff[i+1]=OBSTACLE_FAR_AWAY;
    else
    {
     if (sonar_readings[i]>SUP_DET_LIMIT)
      sonar_readings[i]=SUP_DET_LIMIT;
     if (sonar_readings[i]<INF_DET_LIMIT)
      sonar_readings[i]=INF_DET_LIMIT;
     Son_buff[i+1]=(unsigned char)(OBSTACLE_FAR_AWAY*(sonar_readings[i]-INF_DET_LIMIT)/(SUP_DET_LIMI
    }
   count_u=0;
  }
```



```
  else
   count_u++;

  /* Filling the Sh_buff array is done by the touch_sharing function */
  /* in ps_sched.c. It appeared a more appropriate way, since this   */
  /* has to do with the scheduler, and not with the hardware.        */

  /* If it is time to transmit sharings... */
  if (count_s==TIME_TO_TRANSMIT_SHARINGS)
  {
   /* AND there are process of modifiable sharing whose data have to be transmitted... */
   if (Sh_buff[1]!=NO_MORE_SHARINGS)
    Sh_buff[0]=SHARING_DATA_READY;
   /* Anyway, start the count again... */
   count_s=0;
  }
  else
   count_s++;

  if (count_pos_or==TIME_TO_TRANSMIT_POS_OR)
  {
   Pos_or_buff[0]=POS_OR_DATA_READY;
   Pos_or_buff[1]=X;
   Pos_or_buff[2]=Y;
   Pos_or_buff[3]=Angle_to_rotate;
   count_pos_or=0;
  }
  else
   count_pos_or++;

 }
 END_BEH_MOD_P
}

/* ------------------------------------------------------------------------ */

void *Close_all(void *pass)
{
  /* Cut the current to the motors */
  outb(0,LEFT_MOTOR);
  outb(0,RIGHT_MOTOR);
  /* Shut speaker up */
  end_tone();

  return(0);
}

/* ------------------------------------------------------------------------ */

int Initialize_all(int init_basic_processes)
{
 int i;

 /* Two tasks are to be done here: initalizations of the system hardware, */
 /* and registering of the functions that must be called by the pseudo-   */
 /* scheduler. */

 init_robot_hardware();
```

```
InitializeSonars();

Son_buff[0]=SONAR_DATA_NOT_READY;
Sh_buff[0]=SHARING_DATA_NOT_READY;

/* Registering the functions consists simply in filling a function array */
/* with them. The calling order will be as set in the array. The type of */
/* each function (no sharing, fixed sharing or modifiable sharing) will  */
/* be set from the information in the command line parameter to main     */

/* What should a user do? */
/* FIll the functions' array with NULL... */
for (i=0;i<MAX_PROC;i++)
 fun[i]=NULL;

/* Fill the array with the names of his/her functions... */

if (init_basic_processes)
{
 /* These are the basic processes. Unless otherwise stated, they */
 /* will be registered by default as no sharing processes. */

 fun[0]=COLISION_DETECTION;
 fun[1]=visualize;
 fun[2]=sonars;
 fun[3]=map;
 fun[4]=sound;
 fun[5]=serial;
 fun[6]=ptrans;
 fun[7]=trans_ult_and_sh;
}

/* The function that the user has passed will contain the initializations  */
/* of the array of functions to those needed for his/her architecture, and */
/* so is here called.                                                       */

init_architecture(init_basic_processes);

/* In the last position, set up the cleaning-up function */
fun[MAX_PROC-1]=Close_all;

/* and return the number of functions he/she has registered. Just for */
/* cheking with the information passed to the command line.           */

i=0;
while (fun[i]!=NULL)
 i++;

printf("\nInitialize all has finished, returning %d",i);
return(i);
}
```

# 6 File cases.c

Types



## Included Files

```
#include "common.h"                                  (Section 1)
  #include </usr/include/stdio.h>
  #include </usr/include/stdlib.h>
  #include </usr/include/math.h>
  #include </usr/include/errno.h>
  #include </usr/include/sys/perm.h>
  #include </usr/include/asm/io.h>
  #include </usr/include/sys/time.h>
  #include </usr/include/sys/resource.h>
  #include </usr/include/signal.h>
  #include </usr/include/sched.h>
  #include </usr/include/pthread.h>
```

## 6.1 Variables

### 6.1.1 Variable current_np

The current number of registered processes

`int current_np`

Inc. from: common.h *(Section 1.2.1)*

### 6.1.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

`unsigned char Sh_buff[16+2]`

Inc. from: common.h *(Section 1.2.2)*

### 6.1.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h *(Section 1.2.3)*

## 6.2 Functions

### 6.2.1 Global Function ADVANCE()

`void* ADVANCE ( void* pass )`

### 6.2.2 Global Function MOVE()

`void* MOVE ( void* pass )`

### 6.2.3 Global Function ROTATE()

`void* ROTATE ( void* pass )`

### 6.2.4 Global Function TEST()

`void* TEST ( void* pass )`

### 6.2.5 Global Function init_architecture()

`void init_architecture ( int basics_loaded )`

## 6.3 Actual file listing

```
#ifndef COMMON
#include "common.h"
#endif

void *TEST(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   Display_code=10;

   if ((sonar_readings[0]<=INF_DET_LIMIT) && (sonar_readings[0]!=0))
     {
       StopRobot();
       Moving_forward=NO;

       pr[8].sharing[DESIRED_SH]=1.0;    /* Test    */
       pr[9].sharing[DESIRED_SH]=0.0;    /* Rotar   */
       pr[10].sharing[DESIRED_SH]=0.0;    /* Avanzar */
       pr[11].sharing[DESIRED_SH]=0.0;   /* Mover */
     }

   else if ((sonar_readings[NUM_SONARS-2]<=INF_DET_LIMIT) && (sonar_readings[0]!=0) &&
             Door_detected==NO)
     {
       Must_rotate=YES;
       Moving_forward=NO;
       Command=RIGTH_TURN;
       outb(Command,TURNING_DIRECTION);
       Angle_to_rotate=5.0*M_PI/180.0;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }

   else if ((sonar_readings[2]<INF_DET_LIMIT) && (sonar_readings[0]!=0) &&
             Door_detected==NO)
     {
       Must_rotate=YES;
       Moving_forward=NO;
       Command=LEFT_TURN;
       outb(Command,TURNING_DIRECTION);
       Angle_to_rotate=5.0*M_PI/180.0;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }

   else if (Door_detected)
     {
       Phase_1=YES;
       Door_detected=NO;
       Must_rotate=YES;
       Moving_forward=NO;
```



```
       Command=LEFT_TURN;
       outb(Command,TURNING_DIRECTION);
       Angle_to_rotate=M_PI/2;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }

   else if (Phase_1)
     {
       Phase_1=NO;
       Must_move=YES;
       Moving_forward=NO;
       Command=FORWARDS;
       outb(Command,TURNING_DIRECTION);
       LEFT_VEL=50;
       RIGHT_VEL=50;
       Distance=1500;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=1.0;
     }

   else if (sonar_readings[NUM_SONARS-2]>(INF_DET_LIMIT+THRESHOLD2))
     {
       Must_move=YES;
       Door_detected=YES;
       Moving_forward=NO;
       Command=FORWARDS;
       outb(Command,TURNING_DIRECTION);
       LEFT_VEL=50;
       RIGHT_VEL=50;
       Distance=700;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=1.0;
     }
   else
     {
       Command=FORWARDS;
       outb(Command,TURNING_DIRECTION);
       LEFT_VEL=100;
       RIGHT_VEL=100;

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=1.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
 ONE_SHOT_WAITS_HERE;
 }
END_BEH_MOD_P;
```

```
}

/* ----------------------------------------------------------------- */

void *ROTATE(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Must_rotate)
     {
       Display_code=20;

       C_rot();

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=1.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
   else
     {
       pr[8].sharing[DESIRED_SH]=1.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}

/* ----------------------------------------------------------------- */

void *ADVANCE(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   Display_code=30;

   Moving_forward=YES;

   Command=FORWARDS;
   outb(Command,TURNING_DIRECTION);

   C_vel();

   pr[8].sharing[DESIRED_SH]=1.0;
   pr[9].sharing[DESIRED_SH]=0.0;
   pr[10].sharing[DESIRED_SH]=0.0;
   pr[11].sharing[DESIRED_SH]=0.0;

   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}

/* ----------------------------------------------------------------- */
```



```
void *MOVE(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Must_move)
     {
       Display_code=40;

       Command=FORWARDS;
       outb(Command,TURNING_DIRECTION);

       C_pos_vel();

       pr[8].sharing[DESIRED_SH]=0.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=1.0;
     }
   else
     {
       StopRobot();

       pr[8].sharing[DESIRED_SH]=1.0;
       pr[9].sharing[DESIRED_SH]=0.0;
       pr[10].sharing[DESIRED_SH]=0.0;
       pr[11].sharing[DESIRED_SH]=0.0;
     }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}

void init_architecture(int basics_loaded)
{
 int from_where;

 from_where=(basics_loaded ? NUM_BASIC_PROCESSES : 0);
 /* Funciones para ejemplo de arquitectura basada en casos */
 fun[from_where]=TEST;
 fun[from_where+1]=ROTATE;
 fun[from_where+2]=ADVANCE;
 fun[from_where+3]=MOVE;
}
```

## 7 File reactive1.c

Types

### Included Files

```
#include "common.h"                                    (Section 1)
  #include </usr/include/stdio.h>
  #include </usr/include/stdlib.h>
  #include </usr/include/math.h>
```

```
#include </usr/include/errno.h>
#include </usr/include/sys/perm.h>
#include </usr/include/asm/io.h>
#include </usr/include/sys/time.h>
#include </usr/include/sys/resource.h>
#include </usr/include/signal.h>
#include </usr/include/sched.h>
#include </usr/include/pthread.h>
```

## 7.1 Variables

### 7.1.1 Variable current_np

The current number of registered processes

`int current_np`

Inc. from: common.h *(Section 1.2.1)*

### 7.1.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

`unsigned char Sh_buff[16+2]`

Inc. from: common.h *(Section 1.2.2)*

### 7.1.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h *(Section 1.2.3)*

## 7.2 Functions

### 7.2.1 Global Function AVOID()

`void* AVOID ( void* pass )`

### 7.2.2 Global Function FEELFORCE()

`void* FEELFORCE ( void* pass )`

### 7.2.3 Global Function RUNAWAY()

`void* RUNAWAY ( void* pass )`

### 7.2.4 Global Function WANDER()

`void* WANDER ( void* pass )`

### 7.2.5 Global Function init_architecture()

`void init_architecture ( int basics_loaded )`

## 7.3 Actual file listing

```
#ifndef COMMON
#include "common.h"
#endif

void *WANDER(void *pass)
{
  long xf=-1500,yf=1500;
  double angf=M_PI/4;
```



```
  BEGIN_BEH_MOD_P
   {
     X_final=xf;
     Y_final=yf;
     ANG_final=angf;

     if (Init_maneuver)
       {
         ang_wander=atan2((double)(Y_final-Y),(double)(X_final-X));
         Dist_wander=sqrt(((X_final-X)*(X_final-X))+((Y_final-Y)*(Y_final-Y)));

 if (ang_wander!=0.0)
   {
     f1=YES;
     f2=f3=f4=f5=f6=NO;
   }

         Init_maneuver=NO;
 first_time=YES;
       }

     ONE_SHOT_WAITS_HERE;
   }
  END_BEH_MOD_P
}

/* ---------------------------------------------------------------- */

void *AVOID(void *pass)
{
  double angulo_int=0.0;

  BEGIN_BEH_MOD_P
   {
     if (pr[9].sharing[CURRENT_SH]!=0.0 &&
             pr[10].sharing[CURRENT_SH]!=0.0)

       if (f1)
         {
           if (first_time)
     {
       angulo_int=ANGLE;
       first_time=NO;
     }

          Must_rotate=YES;
    if (ang_wander>=angulo_int)
      {
        Display_code=11;
        Command=LEFT_TURN;
        outb(Command,TURNING_DIRECTION);
        Angle_to_rotate=ang_wander-angulo_int;
      }
    else
      {
        Display_code=12;
        Command=RIGTH_TURN;
```

```
        outb(Command,TURNING_DIRECTION);
        Angle_to_rotate=angulo_int-ang_wander;
      }
    C_rot();
        }
      else if ((f2) && (Dist_wander!=0.0))
        {
          Must_move=YES;
   Display_code=13;
   LEFT_VEL=100;
   RIGHT_VEL=100;
   Command=FORWARDS;
   outb(Command,TURNING_DIRECTION);
          Distance=Dist_wander;
   C_pos_vel();
        }
      else if (f3)
        {
          Must_rotate=YES;
   if (ANG_final>=ang_wander)
     {
      Display_code=14;
      Command=LEFT_TURN;
      outb(Command,TURNING_DIRECTION);
          Angle_to_rotate=ANG_final-ang_wander;
     }
   else
     {
      Display_code=15;
      Command=RIGTH_TURN;
      outb(Command,TURNING_DIRECTION);
      Angle_to_rotate=ang_wander-ANG_final;
     }
   C_rot();
        }
        else
        {
          End_maneuver=YES;
   Display_code=16;
          StopRobot();
        }

      ONE_SHOT_WAITS_HERE;
    }
   END_BEH_MOD_P
}

/* ------------------------------------------------------------------ */

/* Esta funci'on genera una fuerza repulsiva resultante de la suma de fuerzas
   de repulsi'on de los objetos que caen dentro del l'imite de seguridad e independiente
   de la distanci entre ellos y el robot. */

void *FEELFORCE(void *pass)
{
  int i,Detectado_obst,Detectado_obst_0;
  static int count=0;
```



```
  BEGIN_BEH_MOD_P
   {
    if (!Must_rotate)

        if (count>(2*NUM_SONARS))
          {
    count=0;
    Fr=0.0;

    Detectado_obst_0=0;
    if ((sonar_readings[0]<=INF_DET_LIMIT)&&(sonar_readings[0]!=0.0))
      {
            Detectado_obst_0=1;
pr[8].sharing[DESIRED_SH]=0.4;
            pr[9].sharing[DESIRED_SH]=0.0;
            pr[10].sharing[DESIRED_SH]=0.0;
            pr[11].sharing[DESIRED_SH]=0.4;
      }

    Detectado_obst=0;
            for (i=1;i<NUM_SONARS;i++)
              if ((sonar_readings[i]<=INF_DET_LIMIT)&&(sonar_readings[i]!=0.0))
                {
          Fr+=(ANGLE_SENSOR*i);
          Detectado_obst=1;
  pr[8].sharing[DESIRED_SH]=0.4;
  pr[9].sharing[DESIRED_SH]=0.0;
              pr[10].sharing[DESIRED_SH]=0.0;
              pr[11].sharing[DESIRED_SH]=0.4;
        }

            if (Detectado_obst)
              {
        Num_beeps=2;

                if (Fr>(2*M_PI))
          Fr=((int)(Fr*180.0/M_PI)%360)*(M_PI/180.0);

                f4=YES;
Move_distance=ROT_FINISHED;
        f1=f2=f3=NO;

        if (Fr<M_PI)
          {
            Command=LEFT_TURN;
            outb(Command,TURNING_DIRECTION);
            Angle_to_rotate=M_PI-Fr;
          }
                else
          {
            Command=RIGTH_TURN;
            outb(Command,TURNING_DIRECTION);
            Angle_to_rotate=Fr-M_PI;
          }

                if (Detectado_obst_0)
          Angle_to_rotate=(Angle_to_rotate+M_PI)/2.0;
              }
```

```
            else
              if (Detectado_obst_0)
                {
  Num_beeps=2;
          f4=YES;
  Move_distance=ROT_FINISHED;
  f1=f2=f3=NO;

          Command=LEFT_TURN;
          outb(Command,TURNING_DIRECTION);
                  Angle_to_rotate=M_PI;
        }
              else
        {
          f4=NO;
  Angle_to_rotate=0.0;
 }
           }
  else
    count++;

   ONE_SHOT_WAITS_HERE;
  }
 END_BEH_MOD_P
}

/* ---------------------------------------------------------------------- */

void *RUNAWAY(void *pass)
{
 BEGIN_BEH_MOD_P
   {
     if (f4)
       {
 Must_rotate=YES;
 Display_code=1;
 C_rot();
}
     else if (f5)
       {
 Must_move=YES;
 Display_code=2;
 LEFT_VEL=100;
 RIGHT_VEL=100;
 Distance=300;
 Command=FORWARDS;
 outb(Command,TURNING_DIRECTION);
 C_pos_vel();
}
     else if (f6)
       {
 Display_code=3;
 pr[8].sharing[DESIRED_SH]=0.2;
         pr[9].sharing[DESIRED_SH]=0.2;
         pr[10].sharing[DESIRED_SH]=0.2;
         pr[11].sharing[DESIRED_SH]=0.2;
         Init_maneuver=YES;
 End_maneuver=NO;
```



```
        }

    ONE_SHOT_WAITS_HERE;
   }
  END_BEH_MOD_P
}

void init_architecture(int basics_loaded)
{
 int from_where;

 from_where=(basics_loaded ? NUM_BASIC_PROCESSES : 0);

 /* Funciones para ejemplo de arquitectura reactiva con 2 niveles de competencia */
 fun[from_where]=FEELFORCE;
 fun[from_where+1]=WANDER;
 fun[from_where+2]=AVOID;
 fun[from_where+3]=RUNAWAY;
}
```

# 8 File teleo_reactive.c

Types

## Included Files

```
#include "common.h"                                   (Section 1)
   #include </usr/include/stdio.h>
   #include </usr/include/stdlib.h>
   #include </usr/include/math.h>
   #include </usr/include/errno.h>
   #include </usr/include/sys/perm.h>
   #include </usr/include/asm/io.h>
   #include </usr/include/sys/time.h>
   #include </usr/include/sys/resource.h>
   #include </usr/include/signal.h>
   #include </usr/include/sched.h>
   #include </usr/include/pthread.h>
```

## 8.1 Variables

### 8.1.1 Variable current_np

The current number of registered processes

`int current_np`

Inc. from: common.h *(Section 1.2.1)*

### 8.1.2 Variable Sh_buff

Transmission buffer for sharings must be known by ps_sched, which updates it every scheduler cycle, and by the robot process that send it through the serial port.

`unsigned char Sh_buff[16+2]`

Inc. from: common.h *(Section 1.2.2)*

### 8.1.3 Variable fun

`void* (*fun[16])(void*)`

Inc. from: common.h *(Section 1.2.3)*

#### 8.1.4 Variable Condition1

```
int Condition1
```

#### 8.1.5 Variable Condition2

```
int Condition2
```

#### 8.1.6 Variable Condition3

```
int Condition3
```

## 8.2 Functions

### 8.2.1 Global Function ADVANCE1()

```
void* ADVANCE1 ( void* pass )
```

### 8.2.2 Global Function RECALCULATE()

```
void* RECALCULATE ( void* pass )
```

### 8.2.3 Global Function ROTATE1()

```
void* ROTATE1 ( void* pass )
```

### 8.2.4 Global Function TELEO()

```
void* TELEO ( void* pass )
```

### 8.2.5 Global Function init_architecture()

```
void init_architecture ( int basics_loaded )
```

## 8.3 Actual file listing

```
#ifndef COMMON
#include "common.h"
#endif

/* Global variables to communicate the processes in this module */
int Condition1,Condition2,Condition3;

void *RECALCULATE(void *pass)
{
  BEGIN_BEH_MOD_P
   {
     if ((sonar_readings[0]>INF_DET_LIMIT) && (sonar_readings[0]<(2*INF_DET_LIMIT)))
       Condition1=YES;
     else
      {
        Condition1=NO;
Moving_forward=NO;
      }

     if ((sonar_readings[1]>INF_DET_LIMIT) && (sonar_readings[1]<(2*INF_DET_LIMIT)))
       Condition2=YES;
     else
       Condition2=NO;

     if ((sonar_readings[NUM_SONARS-1]>INF_DET_LIMIT) && (sonar_readings[NUM_SONARS-1]<(2*INF_DET_LI
       Condition3=YES;
```



```
      else
        Condition3=NO;
   }
 END_BEH_MOD_P
}

/* ---------------------------------------------------------------------------------------- */

void *TELEO(void *pass)
{
 BEGIN_BEH_MOD_P
  {
   if (Must_rotate)
    {
      pr[10].sharing[DESIRED_SH]=0.5;
      pr[11].sharing[DESIRED_SH]=0.0;
    }

   else if (!Must_rotate && Condition1)
    {
      Moving_forward=YES;
      Command=FORWARDS;
      outb(Command,TURNING_DIRECTION);
      pr[10].sharing[DESIRED_SH]=0.0;
      pr[11].sharing[DESIRED_SH]=0.5;
    }

   else if (!Must_rotate && !Condition1 && Condition2)
    {
      Must_rotate=YES;
      Command=RIGTH_TURN;
      outb(Command,TURNING_DIRECTION);
      pr[10].sharing[DESIRED_SH]=0.5;
      pr[11].sharing[DESIRED_SH]=0.0;
    }

   else if (!Must_rotate && !Condition1 && !Condition2 && Condition3)
    {
      Must_rotate=YES;
      Command=LEFT_TURN;
      outb(Command,TURNING_DIRECTION);
      pr[10].sharing[DESIRED_SH]=0.5;
      pr[11].sharing[DESIRED_SH]=0.0;
    }

   else
    {
      Display_code=40;
      StopRobot();
    }
  }
 END_BEH_MOD_P
}

/* ------------------------------------------------------------------ */

void *ROTATE1(void *pass)
{
```

```
 BEGIN_BEH_MOD_P
 {
   if (Must_rotate)
     {
       Display_code=20;
       Angle_to_rotate=M_PI/4;
       C_rot();
     }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}

/* ------------------------------------------------------------------ */

void *ADVANCE1(void *pass)
{
 BEGIN_BEH_MOD_P
 {
   if (Moving_forward)
    {
     Display_code=30;
     LEFT_VEL=50;
     RIGHT_VEL=50;
     C_vel();
    }
   ONE_SHOT_WAITS_HERE;
 }
 END_BEH_MOD_P;
}

/* ------------------------------------------------------------------ */

void init_architecture(int basics_loaded)
{
 int from_where;

 from_where=(basics_loaded ? NUM_BASIC_PROCESSES : 0);

 /* Functions to implement and example of teleo-reactive architecture */

 fun[from_where]=RECALCULATE;
 fun[from_where+1]=TELEO;
 fun[from_where+2]=ROTATE1;
 fun[from_where+3]=ADVANCE1;
}
```

## 9 File ofg.c

### Preprocessor definitions

```
#define OFG_PORT 0x300
#define P OFG_PORT
#define CONTROL P
#define O_LUT_CON
#define O_LUT_ADD
```



```
#define ACQ
#define POIN_CON
#define PAN
#define SCROLL
#define O_LUT_DATA
#define VBCON
#define I_LUT_ADD
#define I_LUT_DATA
#define ADC_COMM
#define PBCON
#define HMASK
#define VMASK
#define PBUF
#define X
#define Y
#define DATA
#define OLCOLR0 0x01
#define OLCOLR1 0x02
#define LUTMD 0x04
#define NI30HZ 0x08
#define OBANK0 0x10
#define OBANK1 0x20
#define OBANK2 0x40
#define OBANK3 0x80
#define ERASE 0
#define RED 1
#define GREEN 2
#define BLUE 4
#define WHITE 8
#define IN_OV 1
#define DOUBLE 2
#define outportb( port, value )
#define inportb( port )
#define outportw( port, value )
#define inportw( port )
#define CX 408
#define CY 115
#define MAX_SIDES 12
#define OR 360
#define OC 24
#define KX 0.06
#define KY 0.072
#define RSH
#define RLO
```

## 9.1 Functions

### 9.1.1 Global Function box_ov()

```
void box_ov ( int x, int y, int w, int h, unsigned char col )
```

### 9.1.2 Global Function closed_pol()

```
void closed_pol ( int np, float vert[12][2], unsigned char col )
```

### 9.1.3 Global Function dot()

```
void dot ( int x, int y, unsigned char col )
```

### 9.1.4 Global Function dot_ov()

```
void dot_ov ( int x, int y, unsigned char col )
```

### 9.1.5 Global Function draw_arena()

```
void draw_arena ( void )
```

### 9.1.6 Global Function draw_little_robot()

```
void draw_little_robot ( float pos_or[3], unsigned char col )
```

### 9.1.7 Global Function draw_robot()

```
void draw_robot ( void )
```

### 9.1.8 Global Function draw_sh_boxes()

```
void draw_sh_boxes ( void )
```

### 9.1.9 Global Function grab()

```
void grab ( void )
```

### 9.1.10 Global Function hbar_ov()

```
void hbar_ov ( int x, int y, int len, unsigned char col )
```

### 9.1.11 Global Function init_ofg()

```
void init_ofg ( void )
```

### 9.1.12 Global Function init_ofg_ov()

```
void init_ofg_ov ( void )
```

### 9.1.13 Global Function lee_pix()

```
unsigned char lee_pix ( int x, int y )
```

### 9.1.14 Global Function line()

```
void line ( int x1, int y1, int x2, int y2, unsigned char color, int overlay )
```

### 9.1.15 Global Function snap()

```
void snap ( void )
```



## 9.2 Actual file listing

```
/* Constantes */
#define OFG_PORT 0x300
#define P           OFG_PORT


#define CONTROL     P
#define O_LUT_CON   P+0x01
#define O_LUT_ADD   P+0x02
#define ACQ         P+0x04
#define POIN_CON    P+0x05
#define PAN         P+0x06
#define SCROLL      P+0x07
#define O_LUT_DATA  P+0x08
#define VBCON       P+0x0A
#define I_LUT_ADD   P+0x0C
#define I_LUT_DATA  P+0x0D
#define ADC_COMM    P+0x0E
#define PBCON       P+0x10
#define HMASK       P+0x12
#define VMASK       P+0x14
#define PBUF        P+0x16
#define X           P+0x18
#define Y           P+0x1A
#define DATA        P+0x1C

#define OLCOLR0     0x01
#define OLCOLR1     0x02
#define LUTMD       0x04
#define NI30HZ      0x08
#define OBANK0      0x10
#define OBANK1      0x20
#define OBANK2      0x40
#define OBANK3      0x80

#define ERASE 0
#define RED   1
#define GREEN 2
#define BLUE  4
#define WHITE 8

#define IN_OV  1
#define DOUBLE 2

#define outportb(port,value) outb(value,port)
#define inportb(port) inb(port)
#define outportw(port,value) outw(value,port)
#define inportw(port) inw(port)

/* Variables globales */
/*
unsigned char *letra;
*/

void init_ofg(void)
{
 int col,tabla,n,x,y;
```

```
ioperm(P,0x20,1);

/* Initialize to ramp all output LUTs */
for (col=0; col<3; col++)
 for (tabla=0; tabla<16; tabla++)
 {
  outportb(P+1,(tabla<<4)+col);
  for (n=0;n<256;n++)
  {
   outportb(P+2,n);
   outportb(P+8,n);
  }
 }

/* Initialize the input LUT */
outportb(P+0xC,0);
for (n=0;n<256;n++)
 outportb(P+0xD,n);

/* Registro de control */
outportb(P+0,0x0C);
/* Registro PAN */
outportb(P+0x06,0);
/* Registro SCROLL */
outportb(P+0x07,0);
/* Registro ADC */
outportb(P+0x0C,0);
outportb(P+0x0E,0);
/* Registro HOST MASK */
outportw(P+0x12,0);
/* Registro VIDEO MASK */
outportw(P+0x14,0);
/* Registro VIDEO BUS CONTROL */
outportb(P+0x0A,0x3A);
/* Referencia negativa */
outportb(P+0x0C,0x1);
outportb(P+0x0E,0);
/* Referencia positiva */
outportb(P+0x0C,0x2);
outportb(P+0x0E,0xFF); /* 0x9A */
/* Registro ACQUISITION CONTROL */
outportb(P+4,0x38);
/* Registro PIXEL BUFFER CONTROL -> Set Z-mode y otras cosas */
outportw(P+0x10,0x4040);
/* Inicializacion de variables globales */
/*
 letra = (unsigned char*) 0xF000FA6EL;
*/
/* Limpia toda la memoria (overlay incluido,creo) */
outportw(PBCON,0x4040);
for (x=0;x<512;x++)
 for (y=0;y<512;y++)
 {
  outportw(P+0x18,y);
  outportw(P+0x1A,x);
  outportw(P+0x1C,0x0000);
 }
}
```



```
void grab(void)
{
 outportw(VMASK,0x0F00);
 outportb(P+4,0xF2);
}

void snap(void)
{
 outportb(P+4,0xB2);
 /*
 while ( inportb(P+4) != 0x04 ) ;
 */
}

unsigned char lee_pix(int x, int y)
{
 outportw(P+0x18,x);
 outportw(P+0x1A,y);
 return( inportw(P+0x1C) );
}

void dot(int x,int y,unsigned char col)
{
 outportw(P+0x18,x);
 outportw(P+0x1A,y);
 outportw(P+0x1C,col);
}

void dot_ov(int x,int y,unsigned char col)
{
 outportw(X,x);
 outportw(Y,y);
 outportw(DATA,(unsigned int)(col<<8));
}

void box_ov(int x,int y,int w,int h,unsigned char col)
{
 int i,j;

 if (x+w>511)
  w=511-x;
 if (y+h>511)
  h=511-y;

 for (i=x;i<x+w;i++)
 {
  dot_ov(i,y,col);
  dot_ov(i,y+h-1,col);
 }
 for (j=y+1;j<y+h-1;j++)
 {
  dot_ov(x,j,col);
  dot_ov(x+w-1,j,col);
 }
}

void line(int x1,int y1,int x2,int y2,unsigned char color,int overlay)
```

```
{
 int dx,dy,dxd,dyd,dxr,dyr;
 int incx,incy,e,er,ed;
 int x,y,i;

 dx=1;
 dy=1;
 incx=x2-x1;
 incy=y2-y1;
 if (incy<0)
 {
  dy=-dy;
  incy=-incy;
 }
 if (incx<0)
 {
  dx=-dx;
  incx=-incx;
 }
 dxd=dx;
 dyd=dy;
 if (incx>=incy)
  dy=0;
 else
 {
  i=incx;
  incx=incy;
  incy=i;
  dx=0;
 }
 dyr=dy;
 dxr=dx;
 /* Inicializaciones */
 er=2*incy;
 ed=2*incy-2*incx;
 e=2*incy-incx;
 x=x1;
 y=y1;
 /* Dibujo de la recta */
 for (i=0;i<incx;i++)
 {
  if (overlay)
   dot_ov(x,y,color);
  else
   dot(x,y,color);
  if (e>=0)
  {
   x+=dxd;
   y+=dyd;
   e+=ed;
  }
  else
  {
   x+=dxr;
   y+=dyr;
   e+=er;
  }
 }
```



```
}

void hbar_ov(int x,int y,int len,unsigned char col)
{
 unsigned int xu,yu;
 static unsigned int ucol[16]=
 {
  0x00FF,0x0EFF,0x0DFF,0x0FFF,
  0x0BFF,0x0FFF,0x0FFF,0x0FFF,
  0x07FF,0x0FFF,0x0FFF,0x0FFF,
  0x0FFF,0x0FFF,0x0FFF,0x0FFF
 };

 xu=(unsigned int)x;
 yu=(unsigned int)y;

 outportw(HMASK,ucol[col]);
 outportw(PBCON,0x5010);
 outportw(X,x);
 outportw(Y,y);
 outportw(DATA,(col==ERASE) ? 0x0000 : 0xFF00);
 if (len==DOUBLE)
 {
  outportw(X,x+8);
  outportw(Y,y);
  outportw(DATA,(col==ERASE) ? 0x0000 : 0x0F00);
 }
}

void init_ofg_ov(void)
{
 int col,table,n;

 /* Initialize output LUTS 1,2,4 and 8 with R,G,B and white */

 /* Red in table 1 */
 table=1;
 outportb(P+1,(table<<4)+0);
 for (n=0;n<256;n++)
 {
  outportb(P+2,n);
  outportb(P+8,255);
 }
 /* Green in table 2 */
 table=2;
 outportb(P+1,(table<<4)+1);
 for (n=0;n<256;n++)
 {
  outportb(P+2,n);
  outportb(P+8,255);
 }
 /* Blue in table 4 */
 table=4;
 outportb(P+1,(table<<4)+2);
 for (n=0;n<256;n++)
 {
  outportb(P+2,n);
  outportb(P+8,255);
```

```
 }
 /* White in table 8 */
 table=8;
 for (col=0;col<3;col++)
 {
  outportb(P+1,(table<<4)+col);
  for (n=0;n<256;n++)
  {
   outportb(P+2,n);
   outportb(P+8,255);
  }
 }
 /* Protect the eight lower bit planes, unprotect the four higher (overlay) planes */
 outportw(HMASK,0x00FF);
 /* Set the pixel tranfer to Z-mode (ITMODE=OTMODE=0) */
 outportw(PBCON,0x4040);
}

void draw_sh_boxes(void)
{
 int max_np,i;

 max_np=MAX_SH_PROC;
 /* The big white box containing all the sharings bars */
 box_ov(24,384,24*max_np+6,120,WHITE);
 /* The boxes for the sharings bars */
 for (i=0;i<max_np;i++)
  box_ov(24*i+28,387,21,102,BLUE);
}

#define CX 408
#define CY 115

void draw_robot(void)
{
 int cx,cy;

 /* The center of the octogon */
 cx=CX;
 cy=CY;

 /* The 8 sides... */
 line(cx-25,cy-60,cx+25,cy-60,WHITE,IN_OV);
 line(cx+25,cy-60,cx+60,cy-25,WHITE,IN_OV);
 line(cx+60,cy-25,cx+60,cy+25,WHITE,IN_OV);
 line(cx+60,cy+25,cx+25,cy+60,WHITE,IN_OV);

 line(cx+25,cy+60,cx-25,cy+60,WHITE,IN_OV);
 line(cx-25,cy+60,cx-60,cy+25,WHITE,IN_OV);
 line(cx-60,cy+25,cx-60,cy-25,WHITE,IN_OV);
 line(cx-60,cy-25,cx-25,cy-60,WHITE,IN_OV);

 /* The central arrow */
 line(cx,cy-30,cx,cy+30,WHITE,IN_OV);
 line(cx-3,cy-27,cx,cy-30,WHITE,IN_OV);
 line(cx+3,cy-27,cx,cy-30,WHITE,IN_OV);

 /* The camera... */
```



```
 box_ov(cx-4,cy-48,8,6,WHITE);
 box_ov(cx-2,cy-51,4,3,WHITE);

 /* The lines emating from the camera... */
 line(cx,cy-51,cx-30,cy-81,WHITE,IN_OV);
 line(cx,cy-51,cx+30,cy-81,WHITE,IN_OV);

 /* The boxes for the wheels... */
 box_ov(cx-50,cy-10,4,20,WHITE);
 box_ov(cx+46,cy-10,4,20,WHITE);

 /* and the boxes for the ultrasonic sensors */
 box_ov(cx-11,cy-107,20,40,BLUE);
 box_ov(cx+45,cy- 88,20,40,BLUE);
 box_ov(cx+69,cy- 21,20,40,BLUE);
 box_ov(cx+45,cy+ 46,20,40,BLUE);

 box_ov(cx-11,cy+ 65,20,40,BLUE);
 box_ov(cx-67,cy+ 46,20,40,BLUE);
 box_ov(cx-91,cy- 21,20,40,BLUE);
 box_ov(cx-67,cy- 88,20,40,BLUE);

}

#define MAX_SIDES 12
#define OR 360
#define OC  24
#define KX  0.06
#define KY  0.072

#define RSH 230.0/2.0
#define RLO 550.0/2.0

void closed_pol(int np,float vert[MAX_SIDES][2],unsigned char col)
{
 int i;

 for (i=0;i<np-1;i++)
  line((int)(OC+KX*vert[i][0]+0.5),(int)(OR-KY*vert[i][1]+0.5),
       (int)(OC+KX*vert[i+1][0]+0.5),(int)(OR-KY*vert[i+1][1]+0.5),col,IN_OV);
 line((int)(OC+KX*vert[np-1][0]+0.5),(int)(OR-KY*vert[np-1][1]+0.5),
      (int)(OC+KX*vert[0][0]+0.5),(int)(OR-KY*vert[0][1]+0.5),col,IN_OV);
}

void draw_little_robot(float pos_or[3],unsigned char col)
{
 int i;

 float vrob[MAX_SIDES][2]={
   {-RSH,RLO},{RSH,RLO},{RLO,RSH},{RLO,-RSH},
   {RSH,-RLO},{-RSH,-RLO},{-RLO,-RSH},{-RLO,RSH},
   {0.0,-RSH},{0.0,RSH},{-40.0,RSH-70.0},{40.0,RSH-70.0}},rob_now[MAX_SIDES][2];
 double initx,inity,init_or;

 initx=(double)pos_or[0];
 inity=(double)pos_or[1];
 init_or=(double)pos_or[2];
```

```
 for (i=0;i<12;i++)
 {
  rob_now[i][0]=cos(init_or)*vrob[i][0]-sin(init_or)*vrob[i][1]+(float)initx;
  rob_now[i][1]=sin(init_or)*vrob[i][0]+cos(init_or)*vrob[i][1]+(float)inity;
 }

 outportw(HMASK,0x00FF);
 outportw(PBCON,0x4040);

 closed_pol(8,rob_now,col);
 line((int)(OC+KX*rob_now[8][0]+0.5),(int)(OR-KY*rob_now[8][1]+0.5),
      (int)(OC+KX*rob_now[9][0]+0.5),(int)(OR-KY*rob_now[9][1]+0.5),
      col,IN_OV);
 line((int)(OC+KX*rob_now[9][0]+0.5),(int)(OR-KY*rob_now[9][1]+0.5),
      (int)(OC+KX*rob_now[10][0]+0.5),(int)(OR-KY*rob_now[10][1]+0.5),
      col,IN_OV);
 line((int)(OC+KX*rob_now[9][0]+0.5),(int)(OR-KY*rob_now[9][1]+0.5),
      (int)(OC+KX*rob_now[11][0]+0.5),(int)(OR-KY*rob_now[11][1]+0.5),
      col,IN_OV);

}

void draw_arena(void)
{
 float vertices[MAX_SIDES][2]={
  {0.0,0.0},{0.0,4860.0},{820.0,4860.0},{820.0,4170.0},{1460.0,4170.0},
  {1460.0,4570.0},{4640.0,4570.0},{4640.0,4250.0},{4740.0,4250.0},{4740.0,0.0},
  {0.0,0.0},{0.0,0.0}};

 closed_pol(10,vertices,WHITE);
}

/*
void esc_car(int x1, int y1, char let, int tam)
{
int x, y, xx, yy;

for (y=0; y<8; y++)
{
for (x=0; x<8; x++)
{
if ( letra[8*let+y] & (0x80>>x) )
{
for (xx=0; xx<tam; xx++)
for (yy=0; yy<tam; yy++)
dot(x1+tam*x+xx,y1+tam*y+yy,0xFF);
}
else if ( tam<3 )
{
for (xx=0; xx<tam; xx++)
for (yy=0; yy<tam; yy++)
dot(x1+tam*x+xx,y1+tam*y+yy,0);
}
}
}
}

void esc(int x1, int y1, char *cad, int tam)
```



```
{
int n;

for (n=0; cad[n]!=0; n++)
esc_car(x1+tam*8*n,y1,cad[n],tam);
}
 */
```

# 10 File monitor.c

## Included Files

```
#include </usr/include/stdio.h>
#include </usr/include/stdlib.h>
#include </usr/include/math.h>
#include </usr/include/string.h>
#include </usr/include/errno.h>
#include </usr/include/curses.h>
#include </usr/include/sys/perm.h>
#include </usr/include/asm/io.h>
#include </usr/include/pthread.h>
#include "ofg.c"
```

*(Section 9)*

## Preprocessor definitions

```
#define _REENTRANT
#define __REENTRANT
#define MAX_ARG_LENGTH 255
```

Constants defined in the robot, too, with the same name and value

```
#define MAX_SH_PROC 16
#define NUM_SONARS 8
#define LENGTH_POS_OR 12
#define MAX_ARG_COM_BYTES 64
#define DIV_CODE 129
#define CONF_CODE 255
#define SONAR_TRANSMIT_CODE 254
#define SHARING_TRANSMIT_CODE 253
#define POS_OR_TRANSMIT_CODE 252
#define START_BIN_PACKET 251
#define CORRECT_BYTE 250
#define NOT_A_CODE 249
#define MAX_SH 102
#define NO_MORE_SHARINGS 101
#define OBSTACLE_FAR_AWAY 128
#define SHOW_SONARS 0x01
```

```
#define SHOW_SHARINGS 0x02
#define SHOW_POS_OR 0x04
#define SHOW_BIN_PACKET 0x08
```

Adresses of serial ports in most machines...

```
#define COM1 0x3F8
#define COM2 0x2F8
```

The serial port we are using

```
#define SERIAL COM2
```

Generic values...

```
#define MAX_BAUD 115200L
#define PAL_5 0x00
#define PAL_6 0x01
#define PAL_7 0x02
#define PAL_8 0x03
#define STOP_1 0x00
#define STOP_2 0x04
#define NO_PARITY 0x00
#define EVEN_PAR 0x18
#define ODD_PAR 0x08
```

and those we choose to use

```
#define SERIAL_VEL 9600
#define PAL_LENGTH PAL_8
#define STOP_BITS STOP_1
#define PARITY NO_PARITY
```

Lengths of all buffers

```
#define BUF_LEN 254
```

Number of trials to read/write a character

```
#define NUM_TRIALS 10
```

Error codes

```
#define OUR_OK 0
#define TOO_LONG_LEN_REC 1
#define REC_BUFF_NOT_EMPTY 2
#define BIN_BUFF_EXCEEDED 3
#define COM_BUFF_EXCEEDED 4
#define NOT_READY_TO_READ 5
#define NOT_READY_TO_WRITE 6
#define CONF_ERROR 7
```

Constants for the parser

```
#define BIN_START_CHAR '@'
#define BIN_END_CHAR '@'
#define COMMAND_CHAR '+'
```



```
#define PROMPT "RODNEY> "
#define PARSE_ERROR 8
#define MAX_NUM_COM 100
#define DEB 0
#define HELP 1
#define QUIT 2
#define STOP 3
#define MOVE_V 4
#define MOVE_D 5
#define ROT 6
#define ROT_W 7
#define ULT_SHOT 8
#define SOUND 9
#define GET_CPUCTR( x )
#define DELAY 266000
```

## 10.1 Type definitions

### 10.1.1 Typedef com_st

```
typedef struct {...} com_st
struct
  {
    unsigned char command;
    unsigned char length;
    unsigned char arguments[64];
  }
```

## 10.2 Variables

### 10.2.1 Variable deb

```
int deb
```

### 10.2.2 Variable quit

```
int quit
```

### 10.2.3 Variable show

```
int show
```

### 10.2.4 Variable serial_locked

```
pthread_mutex_t serial_locked
```

### 10.2.5 Variable rec

```
WINDOW* rec
```

### 10.2.6 Variable trans

```
WINDOW* trans
```

#### 10.2.7 Variable command_names

```
char command_names[100][9]
```

#### 10.2.8 Variable in_pos_or

```
float in_pos_or[3]
```

#### 10.2.9 Variable last_sending_time

```
long long last_sending_time
```

### 10.3 Functions

#### 10.3.1 Global Function access_to_ports()

```
int access_to_ports ( void )
```

#### 10.3.2 Global Function analyze()

```
int analyze ( char* st, com_st* com )
```

#### 10.3.3 Global Function box_ov()

```
void box_ov ( int x, int y, int w, int h, unsigned char col )
```

Inc. from: ofg.c *(Section 9.1.1)*

#### 10.3.4 Global Function closed_pol()

```
void closed_pol ( int np, float vert[12][2], unsigned char col )
```

Inc. from: ofg.c *(Section 9.1.2)*

#### 10.3.5 Global Function dot()

```
void dot ( int x, int y, unsigned char col )
```

Inc. from: ofg.c *(Section 9.1.3)*

#### 10.3.6 Global Function dot_ov()

```
void dot_ov ( int x, int y, unsigned char col )
```

Inc. from: ofg.c *(Section 9.1.4)*

#### 10.3.7 Global Function draw_arena()

```
void draw_arena ( void )
```

Inc. from: ofg.c *(Section 9.1.5)*

#### 10.3.8 Global Function draw_little_robot()

```
void draw_little_robot ( float pos_or[3], unsigned char col )
```

Inc. from: ofg.c *(Section 9.1.6)*

#### 10.3.9 Global Function draw_robot()

```
void draw_robot ( void )
```

Inc. from: ofg.c *(Section 9.1.7)*

#### 10.3.10 Global Function draw_sh_boxes()

```
void draw_sh_boxes ( void )
```

Inc. from: ofg.c *(Section 9.1.8)*



#### 10.3.11 Global Function fill_bar()

```
void fill_bar ( int x, int y1, int y2, unsigned char col )
```

#### 10.3.12 Global Function fill_command_names()

```
void fill_command_names ( char cn[100][9] )
```

#### 10.3.13 Global Function get_command_code()

```
int get_command_code ( char tokens[10][20], int place )
```

#### 10.3.14 Global Function grab()

```
void grab ( void )
```

Inc. from: ofg.c *(Section 9.1.9)*

#### 10.3.15 Global Function has_to_be_sent()

```
int has_to_be_sent ( int command )
```

#### 10.3.16 Global Function hbar_ov()

```
void hbar_ov ( int x, int y, int len, unsigned char col )
```

Inc. from: ofg.c *(Section 9.1.10)*

#### 10.3.17 Global Function help_on_command()

```
void help_on_command ( WINDOW* w, int code )
```

#### 10.3.18 Global Function init_ofg()

```
void init_ofg ( void )
```

Inc. from: ofg.c *(Section 9.1.11)*

#### 10.3.19 Global Function init_ofg_ov()

```
void init_ofg_ov ( void )
```

Inc. from: ofg.c *(Section 9.1.12)*

#### 10.3.20 Global Function is_in()

```
int is_in ( char c, char* t )
```

#### 10.3.21 Global Function lee_pix()

```
unsigned char lee_pix ( int x, int y )
```

Inc. from: ofg.c *(Section 9.1.13)*

#### 10.3.22 Global Function line()

```
void line ( int x1, int y1, int x2, int y2, unsigned char color, int overlay )
```

Inc. from: ofg.c *(Section 9.1.14)*

#### 10.3.23 Global Function main()

```
int main ( int argc, char* argv[] )
```

#### 10.3.24 Global Function parse()

```
int parse ( char* buff, int init, com_st* com )
```

### 10.3.25 Global Function process_incoming_byte()

```
void process_incoming_byte ( unsigned char b2, unsigned char son_read[8], unsigned char
curr_sh[16], unsigned char curr_pos_or[12] )
```

### 10.3.26 Global Function random_update()

```
void random_update ( unsigned char sh[16] )
```

### 10.3.27 Global Function read_byte()

```
int read_byte ( unsigned char* pc )
```

### 10.3.28 Global Function read_two_bytes()

```
int read_two_bytes ( unsigned char* pc, unsigned char* pc2 )
```

### 10.3.29 Global Function rec_error()

```
void rec_error ( char message[100], unsigned char byte )
```

### 10.3.30 Global Function rec_loop()

```
void* rec_loop ( void* pass )
```

### 10.3.31 Global Function send_com()

```
int send_com ( com_st curr )
```

### 10.3.32 Global Function send_command()

```
void send_command ( unsigned char buff[254] )
```

### 10.3.33 Global Function set_show()

```
int set_show ( char* tok, int* pshow, char* message )
```

### 10.3.34 Global Function show_help()

```
void show_help ( WINDOW* w, int code )
```

### 10.3.35 Global Function show_pos_or()

```
void show_pos_or ( unsigned char pos_or[12] )
```

### 10.3.36 Global Function snap()

```
void snap ( void )
```

Inc. from: ofg.c *(Section 9.1.15)*

### 10.3.37 Global Function update_pos_or_rep()

```
void update_pos_or_rep ( unsigned char cp[12], unsigned char ocp[12] )
```

### 10.3.38 Global Function update_sharing_rep()

```
void update_sharing_rep ( unsigned char read[16], unsigned char old_read[16] )
```

### 10.3.39 Global Function update_ult_rep()

```
void update_ult_rep ( unsigned char read[8], unsigned char old_read[8] )
```



### 10.3.40 Global Function wait_until_ready()

```
void wait_until_ready ( void )
```

### 10.3.41 Global Function write_byte_and_wait_conf()

```
int write_byte_and_wait_conf ( unsigned char c )
```

### 10.3.42 Global Function write_packet()

```
void write_packet ( unsigned char buff[254] )
```

## 10.4 Actual file listing

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <errno.h>
#include <curses.h>
#include <sys/perm.h>
#include <asm/io.h>
#include <sys/types.h>

#include <pthread.h>

#define _REENTRANT
#define __REENTRANT

#define MAX_ARG_LENGTH 255

/*+ Constants defined in the robot, too, with the same name and value +*/
#define MAX_SH_PROC              16
#define NUM_SONARS                8
#define LENGTH_POS_OR            12
#define MAX_ARG_COM_BYTES        64

#define DIV_CODE                129
#define CONF_CODE               255
#define SONAR_TRANSMIT_CODE     254
#define SHARING_TRANSMIT_CODE   253
#define POS_OR_TRANSMIT_CODE    252
#define START_BIN_PACKET        251
#define CORRECT_BYTE            250
#define NOT_A_CODE              249

#define MAX_SH                  102
#define NO_MORE_SHARINGS        101
#define OBSTACLE_FAR_AWAY       128

#include "ofg.c"

typedef struct
{
 unsigned char command;
 unsigned char length;
 unsigned char arguments[MAX_ARG_COM_BYTES];
} com_st;

int deb=0;
```

```
int quit=0;

#define SHOW_SONARS      0x01
#define SHOW_SHARINGS    0x02
#define SHOW_POS_OR      0x04
#define SHOW_BIN_PACKET 0x08
/* Show must be a bit-or of some or all of the avobe constants */
int show = SHOW_POS_OR;

pthread_mutex_t serial_locked=PTHREAD_MUTEX_INITIALIZER;
WINDOW *rec,*trans;

/** Constants for serial port management        **/
/** Adresses of serial ports in most machines... **/
#define COM1 0x3F8
#define COM2 0x2F8
/** The serial port we are using                **/
#define SERIAL COM2
/** Constants to set up serial port parameters   **/
/** Generic values...                            **/
#define MAX_BAUD 115200L

#define PAL_5     0x00
#define PAL_6     0x01
#define PAL_7     0x02
#define PAL_8     0x03

#define STOP_1    0x00
#define STOP_2    0x04

#define NO_PARITY 0x00
#define EVEN_PAR  0x18
#define ODD_PAR   0x08
/** and those we choose to use                   **/
#define SERIAL_VEL 9600
#define PAL_LENGTH PAL_8
#define STOP_BITS  STOP_1
#define PARITY     NO_PARITY

/** Lengths of all buffers                 **/
#define BUF_LEN 254
/** Number of trials to read/write a character **/
#define NUM_TRIALS 10
/** Error codes                            **/
#define OUR_OK              0
#define TOO_LONG_LEN_REC    1
#define REC_BUFF_NOT_EMPTY 2
#define BIN_BUFF_EXCEEDED  3
#define COM_BUFF_EXCEEDED  4
#define NOT_READY_TO_READ  5
#define NOT_READY_TO_WRITE 6
#define CONF_ERROR          7

/** Constants for the parser **/
#define BIN_START_CHAR '@'
#define BIN_END_CHAR   '@'
#define COMMAND_CHAR   '+'
```



```
#define PROMPT "RODNEY> "
#define PARSE_ERROR    8

#define MAX_NUM_COM   100
#define DEB           0
#define HELP          1
#define QUIT          2
#define STOP          3
#define MOVE_V        4
#define MOVE_D        5
#define ROT           6
#define ROT_W         7
#define ULT_SHOT      8
#define SOUND         9

char command_names[MAX_NUM_COM][9];
float in_pos_or[3];

long long last_sending_time;

/* ------------------------------------------------------------------------ */

void get_cpuctr(unsigned long *pxl,unsigned long *pxh);
#define GET_CPUCTR(x) get_cpuctr((unsigned long *)(&x),((unsigned long *)(&x))+1)
#define DELAY 266000

void wait_until_ready()
{
 long long now;
 int i;

 do
 {
  GET_CPUCTR(now);
  for (i=0;i<100;i++);
 }
 while ((now-last_sending_time)<DELAY);

 return;
}


/* ------------------------------------------------------------------------ */

void fill_command_names(char cn[MAX_NUM_COM][9])
{
 int i;

 for (i=0;i<MAX_NUM_COM;i++)
  cn[i][0]='\0';
 strcpy(cn[DEB],"deb");
 strcpy(cn[HELP],"help");
 strcpy(cn[QUIT],"quit");
 strcpy(cn[STOP],"stop");
 strcpy(cn[MOVE_V],"move");
 strcpy(cn[MOVE_D],"moved");
 strcpy(cn[ROT],"rot");
 strcpy(cn[ROT_W],"rotw");
```

```
 strcpy(cn[ULT_SHOT],"sonars");
 strcpy(cn[SOUND],"sound");
}

/* ------------------------------------------------------------------------- */

void help_on_command(WINDOW *w,int code)
{
 switch (code)
 {
  case DEB: wprintw(w,"+deb (no arguments): sets debugging to OFF\n");
            wprintw(w,"+deb and a string of up to four character of 'shpb'\n");
            wprintw(w,"     sets debugging to ON, showing requested information\n");
            wrefresh(w);
            break;
  case HELP: wprintw(w,"+help (no arguments): give help on all commands\n");
             wprintw(w,"+help <command_name>: give help on one command\n");
             wrefresh(w);
             break;
  case QUIT: wprintw(w,"+quit: quits from this program. Robot goes on\n");
             wprintw(w,"+quit all: send a command to stop robot and quits from this program\n");
             wrefresh(w);
             break;
  case STOP: wprintw(w,"+stop (no arguments): stops robot\n");
             wrefresh(w);
             break;
  case MOVE_V: wprintw(w,"+move <velocity>: move the robot facing its front with certain velocity\n"
               wrefresh(w);
               break;
  case MOVE_D: wprintw(w,"+moved <vel> <dist>: robot advances a given distance at a given velocity\n
               wrefresh(w);
               break;
  case ROT: wprintw(w,"+rot <angle>: rotates the robot a given angle (in degrees)\n");
            wrefresh(w);
            break;
  case ROT_W: wprintw(w,"+rotw <angle> <w>: rotates the robot a given angle (in degrees)\n");
              wprintw(w,"                   at a given angular velocity (in dg/sec)\n");
              wrefresh(w);
              break;
  case ULT_SHOT: wprintw(w,"+sonars <1/0> ... <1/0>: turn the firing of each sonar being 1-->ON and
                 wprintw(w,"                         as much numbers as sonars must be given\n");
                 wrefresh(w);
                 break;
  case SOUND: wprintw(w,"+sound <1/0>: (no arguments): sets robot sound state to ON (1) or OFF (0)\n
              wrefresh(w);
              break;
  default: wprintw(w,"Help on command %d (%s) not implemented\n",code,command_names[code]);
           wrefresh(w);
           break;
 }
}

/* ------------------------------------------------------------------------- */

void show_help(WINDOW *w,int code)
{
 int i;
```



```
 if (code==-1)
 {
  i=0;
  while (command_names[i][0]!='\0')
  {
   help_on_command(w,i);
   i++;
  }
 }
 else
  help_on_command(w,code);
}

/* ------------------------------------------------------------------------- */

int access_to_ports(void)
{
 if (ioperm(SERIAL,8,1))
 {
  perror("ioperm: Asking for I/O port access to serial port");
  exit(0);
 }
 return(1);
}

/* ---------------------------------------------------------------------- */

int read_byte(unsigned char *pc)
{
 int try;

 try=0;
 while ((!(inb(SERIAL+5) & 0x01)) && (try<NUM_TRIALS))
  try++;
 if (try<NUM_TRIALS)
 {
  *pc=inb(SERIAL);
  return(OUR_OK);
 }
 else
 {
  fprintf(stderr,"Error reading byte.\n");
  return(NOT_READY_TO_READ);
 }
}

/* ------------------------------------------------------------------------ */

int read_two_bytes(unsigned char *pc,unsigned char *pc2)
{
 int try;

 try=0;
 while ((!(inb(SERIAL+5) & 0x01)) && (try<NUM_TRIALS))
  try++;
 if (try<NUM_TRIALS)
  *pc=inb(SERIAL);
 else
```

```
 {
  wprintw(rec,"Error reading byte.\n");
  return(NOT_READY_TO_READ);
 }

 try=0;
 while ((!(inb(SERIAL+5) & 0x01)) && (try<NUM_TRIALS))
  try++;
 if (try<NUM_TRIALS)
 {
  *pc2=inb(SERIAL);
  return(OUR_OK);
 }
 else
 {
  *pc2=NOT_A_CODE;
  return(OUR_OK);
 }

}

/* ------------------------------------------------------------------------ */

int write_byte_and_wait_conf(unsigned char c)
{
 int try;
 unsigned char b;

 wait_until_ready();

 try=0;
 while ((!(inb(SERIAL+5) & 0x20)) && (try<NUM_TRIALS))
  try++;
 if (try<NUM_TRIALS)
 {
  outb(c,SERIAL);
 }
 else
 {
  wprintw(rec,"Error writing byte.\n");
  GET_CPUCTR(last_sending_time);
  return(NOT_READY_TO_WRITE);
 }

 while (!(inb(SERIAL+5) & 0x01));
 b=inb(SERIAL);

 if (b!=CONF_CODE)
 {
  wprintw(trans,"Error: read byte %x instead of the confirmation code.\n",b);
  GET_CPUCTR(last_sending_time);
  return(CONF_ERROR);
 }

 GET_CPUCTR(last_sending_time);

 return(OUR_OK);
}
```



```
/* ------------------------------------------------------------------------ */

void write_packet(unsigned char buff[BUF_LEN])
{
 int i;
 if ((write_byte_and_wait_conf(START_BIN_PACKET)==OUR_OK) &&
     (write_byte_and_wait_conf((unsigned char)(strlen(buff)-2))==OUR_OK))
 {
  for (i=1;i<strlen(buff)-1;i++)
   if (write_byte_and_wait_conf(buff[i])!=OUR_OK)
    break;
 }
 return;
}

/* ------------------------------------------------------------------------ */

int get_command_code(char tokens[10][20],int place)
{
 int i,code,one_out;

 /* The first token must be the command name (not preceeded by the COMMAND_CHAR) */
 /* We look for it in the command name list */
 i=0;
 one_out= place ? 0 : 1;
 code=DIV_CODE;
 while ((command_names[i][0]!='\0') && (code==DIV_CODE))
 {
  if (!strcmp(command_names[i],tokens[place]+one_out))
   code=i;
  else
   i++;
 }
 return(code);

}

/* ------------------------------------------------------------------------ */

int is_in(char c,char *t)
{
 int i,len;

 i=0;
 len=strlen(t);
 while ((t[i]!=c) && (i<len))
  i++;

 return(i<len);
}

/* ------------------------------------------------------------------------ */

int set_show(char *tok,int *pshow,char *message)
{
 int count;
```

```
 if (strlen(tok)>4)
 {
  strcpy(message,"Error: wrong argument to deb command");
  return(0);
 }
 *pshow=0;
 count=0;
 message[0]='\0';
 if ( is_in('s',tok) || is_in('S',tok) )
 {
  *pshow |= SHOW_SONARS;
  strcat(message," sonars");
  count++;
 }
 if ( is_in('h',tok) || is_in('H',tok) )
 {
  *pshow |= SHOW_SHARINGS;
  strcat(message," sharings");
  count++;
 }
 if ( is_in('p',tok) || is_in('P',tok) )
 {
  *pshow |= SHOW_POS_OR;
  strcat(message," pos./orient.");
  count++;
 }
 if ( is_in('b',tok) || is_in('B',tok) )
 {
  *pshow |= SHOW_BIN_PACKET;
  strcat(message," binary_packets");
  count++;
 }
 if (count!=strlen(tok))
  wprintw(trans,"\nWarning: %d character(s) of the argument could not be interpreted",strlen(tok)-co
 wprintw(trans,"\n");
 return(1);

}

/* ---------------------------------------------------------------------- */

int analyze(char *st,com_st *com)
{
 char sep[3]={' ','\0'},tokens[10][20],cop_st[255],*nt,message[30];
 int i,num_tokens,integer_args[10],son_to_send,help_code;
 float float_args[10];
 unsigned char code;

 /* The token list for this command is cleaned */
 for (i=0;i<10;i++)
 {
  tokens[i][0]='\0';
  integer_args[i]=0;
  float_args[i]=0.0;
 }
 /* Argument is copied, since strtok might alter it */
 strcpy(cop_st,st);
```



```
 /* The first character must be the COMMAND_CHAR */
 if (cop_st[0]!=COMMAND_CHAR)
 {
  if (deb)
  {
   wprintw(trans,"%s does not start by %c\n",cop_st,COMMAND_CHAR);
   wrefresh(trans);
  }
  return(PARSE_ERROR);
 }

 /* Now, divide the command string into tokens. */
 strcpy(tokens[0],strtok(cop_st,sep));
 i=1;
 while ((nt=strtok(NULL,sep))!=NULL)
 {
  strcpy(tokens[i],nt);
  i++;
 }
 num_tokens=i;

 code=get_command_code(tokens,0);

 if (code==DIV_CODE)
 {
  if (deb)
  {
   wprintw(trans,"command '%s' is not in the list of commands\n",tokens[0]+1);
   wrefresh(trans);
  }
  return(PARSE_ERROR);
 }

 /* If we are here, the command with code 'code' has been found. */
 /* Each command needs sepparate analysis.                        */
 if (deb)
 {
  wprintw(trans,"Performing analysis of command number %d\n",code);
  wrefresh(trans);
 }
 switch (code)
 {
  /* deb, quit and stop (0,1 and 2) have no arguments */
  /* Also, deb, help, and one of the quit modalities have only internal */
  /* effect and are not transmitted */
  case DEB: com->command=code;
            com->length=0;
            if (num_tokens==1)
            {
             if (deb==0)
     wprintw(trans,"debugging mode was already OFF\n");
    else
    {
     deb=0;
     wprintw(trans,"debugging mode turned to OFF\n");
    }
   }
            else
```

```
          if (num_tokens!=2)
           return(PARSE_ERROR);
          else
          {
           deb=set_show(tokens[1],&show,message);
           wprintw(trans,"debugging mode turned to %s, showing %s\n",deb ? "ON" : "OFF",message);
          }
  wrefresh(trans);
         break;
case HELP: com->command=code;
          if (num_tokens==1)
          {
           com->length=0;
           show_help(trans,-1);
   }
          else
          {
           if (num_tokens!=2)
     return(PARSE_ERROR);
    help_code=get_command_code(tokens,1);
    if (help_code==DIV_CODE)
    {
     wprintw(trans,"I can give help on that command, it does not exists");
     wrefresh(trans);
     return(OUR_OK);
    }
    else
     show_help(trans,help_code);
   }
          break;
/* quit (argument: 1 integer) */
case QUIT: com->command=code;
          com->length=0;
          if (num_tokens==2)
          {
    errno=0;
           integer_args[0]=(int)strtol(tokens[1],NULL,10);
           if (errno)
            return(PARSE_ERROR);
           com->length=sizeof(int);
           memcpy(com->arguments,&integer_args[0],(size_t)com->length);
           quit=2;
   }
          else
           quit=1;
          wprintw(trans,"quit has %d tokens, and quit is %d\n",num_tokens,quit);
          break;
/* stop (no arguments) */
case STOP: com->command=code;
          com->length=0;
          break;
/* move (argument: 1 integer) */
case MOVE_V: com->command=code;
            if (num_tokens!=2)
             return(PARSE_ERROR);
            errno=0;
            integer_args[0]=(int)strtol(tokens[1],NULL,10);
            if (errno)
```



```
             return(PARSE_ERROR);
            com->length=sizeof(int);
            memcpy(com->arguments,&integer_args[0],(size_t)com->length);
            break;
/* moved (argument: 2 integers) */
case MOVE_D: com->command=code;
            if (num_tokens!=3)
             return(PARSE_ERROR);
            errno=0;
            integer_args[0]=(int)strtol(tokens[1],NULL,10);
            if (errno)
             return(PARSE_ERROR);
            integer_args[1]=(int)strtol(tokens[2],NULL,10);
            if (errno)
             return(PARSE_ERROR);
            com->length=2*sizeof(int);
            memcpy(com->arguments,&integer_args[0],(size_t)com->length);
            break;
/* rot (argument: 1 integer) */
case ROT: com->command=code;
         if (num_tokens!=2)
          return(PARSE_ERROR);
         errno=0;
         integer_args[0]=(int)strtol(tokens[1],NULL,10);
         if (errno)
          return(PARSE_ERROR);
         com->length=sizeof(int);
         memcpy(com->arguments,&integer_args[0],(size_t)com->length);
         break;
/* rotd (argument: 2 integers) */
case ROT_W: com->command=code;
           if (num_tokens!=3)
            return(PARSE_ERROR);
           errno=0;
           integer_args[0]=(int)strtol(tokens[1],NULL,10);
           if (errno)
            return(PARSE_ERROR);
           integer_args[1]=(int)strtol(tokens[2],NULL,10);
           if (errno)
            return(PARSE_ERROR);
           com->length=2*sizeof(int);
           memcpy(com->arguments,&integer_args[0],(size_t)com->length);
           break;
/* sonars (argument: NUM_SONARS integers (0/1), that will be compressed to 1 integer) */
case ULT_SHOT: com->command=code;
              if (num_tokens!=NUM_SONARS+1)
               return(PARSE_ERROR);
              son_to_send=0;
              for (i=0;i<NUM_SONARS;i++)
              {
               errno=0;
integer_args[i]=(int)strtol(tokens[i+1],NULL,10);
if (errno)
 return(PARSE_ERROR);
if ((integer_args[i]!=0) && (integer_args[i]!=1))
 return(PARSE_ERROR);
if (integer_args[i]==1)
 son_to_send |= 1<<i;
```

```
 }
                 com->length=1;
                 memcpy(com->arguments,&integer_args[0],(size_t)com->length);
                 break;
  /* sound (argument: 1 integer meaning 0-->OFF and 1-->ON) */
  case SOUND: com->command=code;
              if (num_tokens!=2)
               return(PARSE_ERROR);
              errno=0;
              integer_args[0]=(int)strtol(tokens[1],NULL,10);
              if (errno)
               return(PARSE_ERROR);
              com->length=sizeof(int);
              memcpy(com->arguments,&integer_args[0],(size_t)com->length);
              break;
  default: com->command=MAX_NUM_COM;
           com->length=0;
           break;
 }

 return(OUR_OK);
}

/* ------------------------------------------------------------------------ */

int parse(char *buff,int init,com_st *com)
{
 int i=0,j=0;
 char strc[255];

 while (buff[init+i]==' ')
  i++;
 while ((buff[init+i]!=';') && (buff[init+i]!='\0'))
 {
  strc[j]=buff[init+i];
  i++;
  j++;
 }
 strc[j]='\0';

 if (i==0)
  return(0);

 if (analyze(strc,com)==OUR_OK)
  return((buff[init+i]=='\0') ? init+i : init+i+1);
 else
 {
  wprintw(trans,"Parse error in command '%s'\nSubsequent commands (if any) are ignored\n",strc);
  return(0);
 }

}

/* -------------------------------------------------------------------------- */

int send_com(com_st curr)
{
 int i;
```



```
 if (write_byte_and_wait_conf(curr.command)!=OUR_OK)
 {
  if (deb)
  {
   wprintw(trans,"The command code could not be sent.");
   wrefresh(trans);
  }
  return(1);
 }
 if (deb)
 {
  wprintw(trans,"Sent command code for command %d\n",curr.command);
  wrefresh(trans);
 }
 i=0;
 while (i<curr.length)
 {
  if (write_byte_and_wait_conf(curr.arguments[i])!=OUR_OK)
   return(1);
  i++;
 }
 if (deb)
 {
  wprintw(trans,"Sent %d bytes of arguments\n",curr.length);
  wrefresh(trans);
 }
 return(OUR_OK);
}

/* -------------------------------------------------------------------------- */

int has_to_be_sent(int command)
{
 switch (command)
 {
  case DEB: return(0);
  case HELP: return(0);
  case QUIT: if (quit==1)
              return(0);
             else
              return(1);
  default: return(1);
 }
}

/* -------------------------------------------------------------------------- */

void send_command(unsigned char buff[BUF_LEN])
{
 int i=0;
 com_st current_command;

 while ((i=parse(buff,i,&current_command)))
 {
  if (has_to_be_sent(current_command.command))
  {
   if (send_com(current_command)!=OUR_OK)
```

```
    wprintw(trans,"Error sending command %s with %d bytes of arguments\n",
    command_names[current_command.command],
    current_command.length);
  }
 }
}

/* ---------------------------------------------------------------------------- */
void fill_bar(int x,int y1,int y2,unsigned char col)
{
 int i,ny1,ny2;

 if (y1>y2)
 {
  ny1=y2;
  ny2=y1;
 }
 else
 {
  ny1=y1;
  ny2=y2;
 }

 for (i=ny1;i<=ny2;i++)
  hbar_ov(x,i,DOUBLE,col);
}

/* ------------------------------------------------------------------------ */

void update_ult_rep(unsigned char read[NUM_SONARS],unsigned char old_read[NUM_SONARS])
{
 static int pos[NUM_SONARS][2]=
       {
{CX-8,CY-106},{CX+48,CY-87},{CX+72,CY-20},{CX+48,CY+47},
        {CX-8,CY+ 66},{CX-64,CY+47},{CX-88,CY-20},{CX-64,CY-87}
       },empty=0;
 int i,l,l1;

 for (i=0;i<NUM_SONARS;i++)
 {
  if (read[i]==OBSTACLE_FAR_AWAY)
   empty++;
  else
   empty=0;
  if (empty!=1)
  {
   l=((int)read[i]*37)/OBSTACLE_FAR_AWAY;
   l1=((int)old_read[i]*37)/OBSTACLE_FAR_AWAY;

   if (l<0) l=0;
   if (l>37) l=37;
   if (l1<0) l1=0;
   if (l1>37) l1=37;

   if (l>l1)
    fill_bar(pos[i][0],pos[i][1]+l,pos[i][1]+l1,ERASE);

   if (l<l1)
```



```
    fill_bar(pos[i][0],pos[i][1]+l1,pos[i][1]+l,RED);

   if (l==37)
   {
    hbar_ov(pos[i][0],pos[i][1]+37,DOUBLE,ERASE);
    hbar_ov(pos[i][0],pos[i][1]+37,DOUBLE,GREEN);
   }

   old_read[i]=read[i];
  }
 }

}

/* ---------------------------------------------------------------------------- */

void update_sharing_rep(unsigned char read[MAX_SH_PROC],unsigned char old_read[MAX_SH_PROC])
{
 int i,l,l1,j;

 i=0;
 while ((read[i]!=NO_MORE_SHARINGS) && (i<MAX_SH_PROC))
 {
  l=(int)read[i]*98/MAX_SH;
  l1=(int)old_read[i]*98/MAX_SH;

  if (l>l1)
   for (j=l1+1;j<=l;j++)
    hbar_ov(24*i+32,486-j,DOUBLE,RED);

  if (l<l1)
   for (j=l1;j>l;j--)
    hbar_ov(24*i+32,486-j,DOUBLE,ERASE);

  if (l==0)
   hbar_ov(24*i+32,486,DOUBLE,GREEN);

  old_read[i]=read[i];
  i++;
 }

}

/* ---------------------------------------------------------------------------- */

void update_pos_or_rep(unsigned char cp[LENGTH_POS_OR],
                       unsigned char ocp[LENGTH_POS_OR])
{
 static int first_time=1;
 int i;
 float pos_or[3];

 if (first_time)
 {
  for (i=0;i<LENGTH_POS_OR/sizeof(float);i++)
   *(float *)(cp+(i*sizeof(float)))=*(float *)(ocp+(i*sizeof(float)))=0.0;
  first_time=0;
  return;
```

```
 }

 for (i=0;i<LENGTH_POS_OR/sizeof(float);i++)
  pos_or[i]=*(float *)(ocp+i*sizeof(float))+in_pos_or[i];

 draw_little_robot(pos_or,ERASE);

 for (i=0;i<LENGTH_POS_OR/sizeof(float);i++)
  pos_or[i]=*(float *)(cp+i*sizeof(float))+in_pos_or[i];

 draw_little_robot(pos_or,WHITE);

 for (i=0;i<LENGTH_POS_OR;i++)
  ocp[i]=cp[i];

 return;
}

/* ------------------------------------------------------------------------- */

void random_update(unsigned char sh[MAX_SH_PROC])
{
 int i;

 for (i=0;i<MAX_SH_PROC;i++)
  sh[i]=(unsigned char)(255.0*rand()/(RAND_MAX+1.0));

}

/* ------------------------------------------------------------------------- */

void rec_error(char message[100],unsigned char byte)
{
 wprintw(rec,"Error while %s. Received byte with value %3d\n",message,(int)byte);
 wrefresh(rec);
}

/* ------------------------------------------------------------------------- */

void show_pos_or(unsigned char pos_or[LENGTH_POS_OR])
{
 int i;
 float *fp;

 for (i=0;i<LENGTH_POS_OR/sizeof(float);i++)
 {
  fp=(float *)(pos_or+(i*sizeof(float)));
  if (i<2)
   wprintw(rec,"%3f ",*fp+in_pos_or[i]);
  else
   wprintw(rec,"%3f ",(180.0/M_PI)*(*fp+in_pos_or[i]));
 }

 wprintw(rec,"\n");
 wrefresh(rec);
}

/* ------------------------------------------------------------------------- */
```



```
void process_incoming_byte(unsigned char b2,
   unsigned char son_read[NUM_SONARS],
   unsigned char curr_sh[MAX_SH_PROC],
                            unsigned char curr_pos_or[LENGTH_POS_OR])
{
 static int receiving_command_conf=0,receiving_sonar_readings=0;
 static int receiving_sharings=0,receiving_position_orient=0;
 static int receiving_binary_packet=0;

 int i,cx,cy;

 if (receiving_sonar_readings)
 {
  if (b2>=DIV_CODE)
  {
   rec_error("receiving sonars",b2);
   receiving_sonar_readings=0;
  }
  else
  {
   son_read[receiving_sonar_readings-1]=b2;
   receiving_sonar_readings++;
   if (receiving_sonar_readings==NUM_SONARS+1)
   {
    receiving_sonar_readings=0;
    if ((deb) && (show & SHOW_SONARS))
    {
     for (i=0;i<NUM_SONARS;i++)
      wprintw(rec,"%3d ",son_read[i]);
     wprintw(rec,"\n");
     wrefresh(rec);

    }
   }
  }
  return;
 }

 if (receiving_sharings)
 {
  if (b2>=MAX_SH)
  {
   rec_error("receiving sharings",b2);
   receiving_sharings=0;
  }
  else
  {
   curr_sh[receiving_sharings-1]=b2;
   /*
   wprintw(rec,"(%d,%3d) ",receiving_sharings-1,(int)b2);
   */
   /*
   if (receiving_sharings==MAX_SH_PROC+1)
   */
   if ((b2==NO_MORE_SHARINGS) || (receiving_sharings>MAX_SH_PROC))
   {
    if ((deb) && (show & SHOW_SHARINGS))
```

```
  {
   for (i=0;i<receiving_sharings;i++)
    wprintw(rec,"%3d ",curr_sh[i]);
   wprintw(rec,"\n");
   wrefresh(rec);
  }
  if (receiving_sharings>MAX_SH_PROC)
   curr_sh[receiving_sharings-1]=NO_MORE_SHARINGS;
  receiving_sharings=0;
 }
 else
  receiving_sharings++;
 }
 return;
}

if (receiving_position_orient)
{
 curr_pos_or[receiving_position_orient-1]=b2;
 receiving_position_orient++;
 if (receiving_position_orient==LENGTH_POS_OR+1)
 {
  receiving_position_orient=0;
  if ((deb) && (show & SHOW_POS_OR))
   show_pos_or(curr_pos_or);
 }
 return;
}

if (receiving_command_conf)
{
 if (b2>MAX_ARG_COM_BYTES)
  rec_error("receiving number of bytes of command arguments",b2);
 else
 {
  wprintw(rec,"together with its %d bytes of arguments.\n",(int)b2);
  wrefresh(rec);
 }
 receiving_command_conf=0;
 return;
}

/* If we have arrived here, we are in 'normal' mode */
switch (b2)
{
 case SONAR_TRANSMIT_CODE: if ((deb) && (show & SHOW_SONARS))
                           {
                            wprintw(rec,"Sonars: ");
   wrefresh(rec);
  }
                           receiving_sonar_readings=1;
                           break;
 case SHARING_TRANSMIT_CODE: if ((deb) && (show & SHOW_SHARINGS))
                             {
                              wprintw(rec,"Sharings: ");
    wrefresh(rec);
   }
                              receiving_sharings=1;
```



```
                              break;
 case POS_OR_TRANSMIT_CODE: if ((deb) && (show & SHOW_POS_OR))
                            {
                             wprintw(rec,"Position/orientation: ");
  wrefresh(rec);
 }
                            receiving_position_orient=1;
                            break;
 case START_BIN_PACKET: if ((deb) && (show & SHOW_BIN_PACKET))
                        {
                         wprintw(rec,"Binary packet: ");
 wrefresh(rec);
}
                        receiving_binary_packet=1;
                        break;
 case NOT_A_CODE: break;
 default:
 {
  getyx(trans,cy,cx);
  if ((b2>=DIV_CODE) && (b2<DIV_CODE+MAX_NUM_COM) && (command_names[(int)b2-DIV_CODE][0]!='\0'))
  {
   wprintw(rec,"Command %s received, ",command_names[(int)b2-DIV_CODE]);
   wrefresh(rec);
   receiving_command_conf=1;
  }
  else
   rec_error("awaiting next known code",b2);
  wmove(trans,cy,cx);
 }
 break;
 }
}

/* ------------------------------------------------------------------------ */

void *rec_loop(void *pass)
{
 unsigned char b2,b2_2;
 int i;

 unsigned char son_read[NUM_SONARS],last_son_read[NUM_SONARS];
 unsigned char curr_sh[MAX_SH_PROC],last_sh[MAX_SH_PROC];
 unsigned char curr_pos_or[LENGTH_POS_OR],last_pos_or[LENGTH_POS_OR];

 for (i=0;i<NUM_SONARS;i++)
 {
  son_read[i]=OBSTACLE_FAR_AWAY;
  last_son_read[i]=OBSTACLE_FAR_AWAY;
 }
 for (i=0;i<MAX_SH_PROC;i++)
 {
  curr_sh[i]=0;
  last_sh[i]=0;
 }

 access_to_ports();

 ioperm(P,0x20,1);
```

```
 wrefresh(rec);
 while (!quit)
 {
  pthread_mutex_lock(&serial_locked);
  if (inb(SERIAL+5) & 0x01)
   /* if (read_byte(&b2)) */
   if (read_two_bytes(&b2,&b2_2))
   {
    pthread_mutex_unlock(&serial_locked);
    wprintw(rec,"\nREADING ERROR\n");
   }
   else
   {
    pthread_mutex_unlock(&serial_locked);

    process_incoming_byte(b2,son_read,curr_sh,curr_pos_or);
    if (b2_2!=NOT_A_CODE)
     process_incoming_byte(b2_2,son_read,curr_sh,curr_pos_or);

    update_ult_rep(son_read,last_son_read);
    update_sharing_rep(curr_sh,last_sh);
    update_pos_or_rep(curr_pos_or,last_pos_or);
   }
  else
   pthread_mutex_unlock(&serial_locked);
 }
 return((void *)0);
}

/* ------------------------------------------------------------------------ */

int main(int argc, char *argv[])
{
 unsigned divider;
 unsigned char pal;
 char buff[BUF_LEN];
 pthread_t pth;
 WINDOW *all;
 int line_div;
 int i;

 if (argc!=4)
 {
  fprintf(stderr,"Usage: %s <init_x> <init_y> <init_orient>\n",argv[0]);
  exit(1);
 }

 for (i=1;i<argc;i++)
 {
  errno=0;
  in_pos_or[i-1]=strtod(argv[i],NULL);
  if (errno)
  {
   fprintf(stderr,"Error: argument %d must be a floating point number, not %s\n",
                 i,argv[i]);
   exit(1);
  }
```



```
 }

 fill_command_names(command_names);

 access_to_ports();

 init_ofg();
 init_ofg_ov();

 draw_sh_boxes();
 draw_robot();
 draw_arena();

 draw_little_robot(in_pos_or,WHITE);

 grab();

 /*
 for (i=0;i<10;i++)
 {
  draw_little_robot(in_pos_or,WHITE);
  sleep(1);
  draw_little_robot(in_pos_or,ERASE);
  in_pos_or[1]+=200.0;
  in_pos_or[2]+=0.1;
 }
 */

 divider=(unsigned)(MAX_BAUD/SERIAL_VEL);
 outb(0x80,SERIAL+3);
 outb(divider%256,SERIAL);
 outb(divider/256,SERIAL+1);
 pal=PAL_LENGTH | STOP_BITS | PARITY;
 outb(pal,SERIAL+3);
 outb(0,SERIAL+1);

 buff[0]='\0';

 all=initscr();
 cbreak();
 echo();

 /* Upper window use three quarters of the screen */
 line_div=3*LINES/4;
 /* Horizontal lines are drawn to divide both parts... */
 move(0,0);
 hline(ACS_HLINE,COLS);
 move(0,18);
 wprintw(all,"| Command interpreter for transmission |");
 move(line_div,0);
 hline(ACS_HLINE,COLS);
 move(line_div,28);
 wprintw(all,"| Reception window |");
 wrefresh(all);

 /* And both windows are created */
 trans=newwin(line_div-1,COLS,1,0);
 rec=newwin(LINES-line_div-1,COLS,line_div+1,0);
```

```
 /* Both of them can do automatic scroll, and the scrolling region is */
 /* for both cases the whole window. */
 scrollok(trans,TRUE);
 scrollok(rec,TRUE);
 wsetscrreg(trans,0,line_div-1);
 wsetscrreg(rec,0,LINES-line_div-1);

 /* Create the thread for the reception loop... */
 pthread_create(&pth,NULL,rec_loop,(void *)rec);
 /* and be sure it will be listening before the transmision may start. */
 sleep(1);

 werase(trans);
 wprintw(trans,"\nWrite strings in the command prompt.\n");
 wprintw(trans,"Strings are composed by commands and other information.\n");
 wprintw(trans,"Commands have the form %c<reserved_word><list_of_parameters>.\n",
COMMAND_CHAR);
 wprintw(trans,"Binary packets have the form %c<bytes>%c.\n",
BIN_START_CHAR,BIN_END_CHAR);
 wprintw(trans,"Any other format is ignored. If in doubt, use +help\n");
 wprintw(trans,"Program ends with +quit\n\n");
 wprintw(trans,"Starting with robot at (%f,%f), orient (%f)\n",
               in_pos_or[0],in_pos_or[1],in_pos_or[2]);
 wrefresh(trans);
 while (!quit)
 {
  wprintw(trans,"%s",PROMPT);
  wrefresh(trans);
  flushinp();
  wgetnstr(trans,buff,BUF_LEN);
  if (((buff[0]!=BIN_START_CHAR) || (buff[strlen(buff)-1]!=BIN_END_CHAR)) &&
       (buff[0]!=COMMAND_CHAR)
     )
  {
   wprintw(trans,
  "Error: you must surround your message by %c, or start your command with %c\n",
        BIN_START_CHAR,COMMAND_CHAR);
   wprintw(trans,"Instead, you typed %s\n",buff);
   wrefresh(trans);
  }
  else
  {
   pthread_mutex_lock(&serial_locked);
   if (buff[0]==COMMAND_CHAR)
    send_command(buff);
   else
    write_packet(buff);
   pthread_mutex_unlock(&serial_locked);
  }
 }

 delwin(trans);
 delwin(rec);
 clear();
 refresh();
 system("stty sane");
 snap();
 return(0);
```



```
}
```

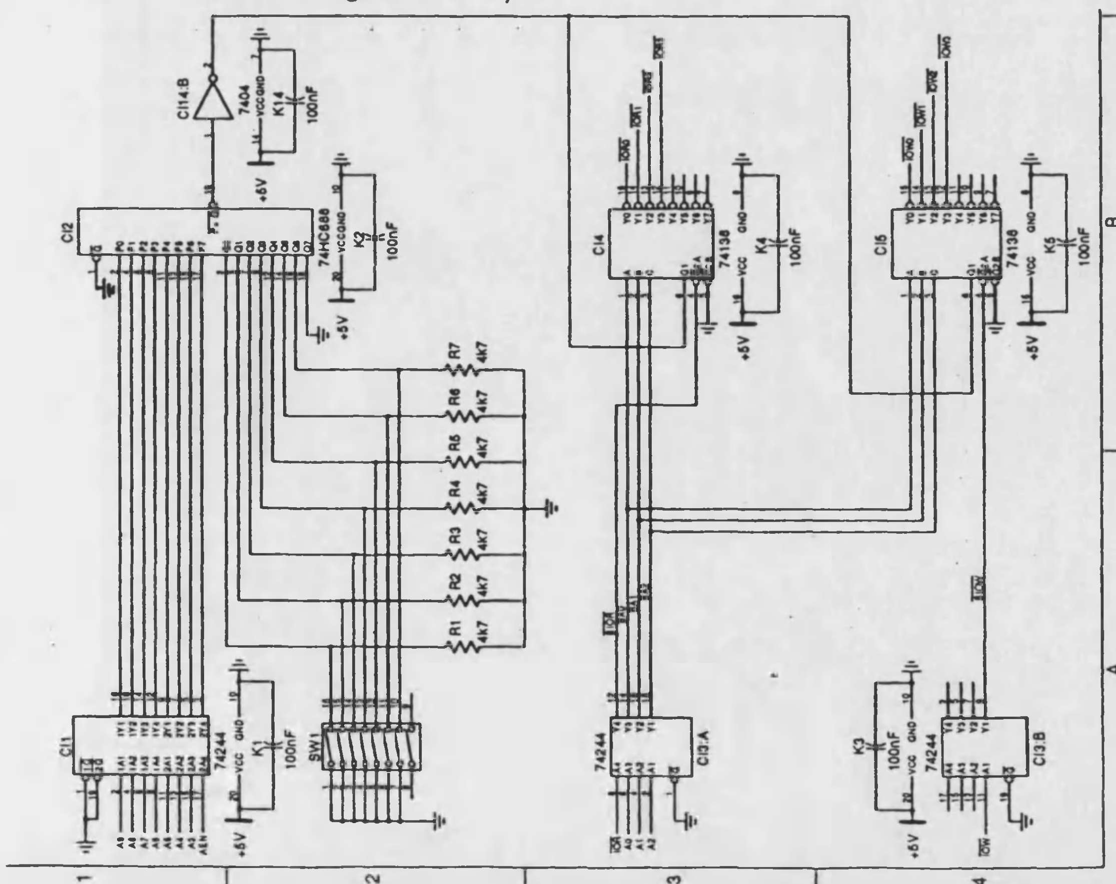
# Apéndice B

# Esquemáticos

En este apartado se presentan los esquemáticos de las tarjetas diseñadas e implementadas sobre nuestro robot Rodney. Se trata de tarjetas a ser insertadas en el bus ISA de la placa base del robot. Todas ellas utilizan el espacio libre para el usuario de direcciones de E/S.

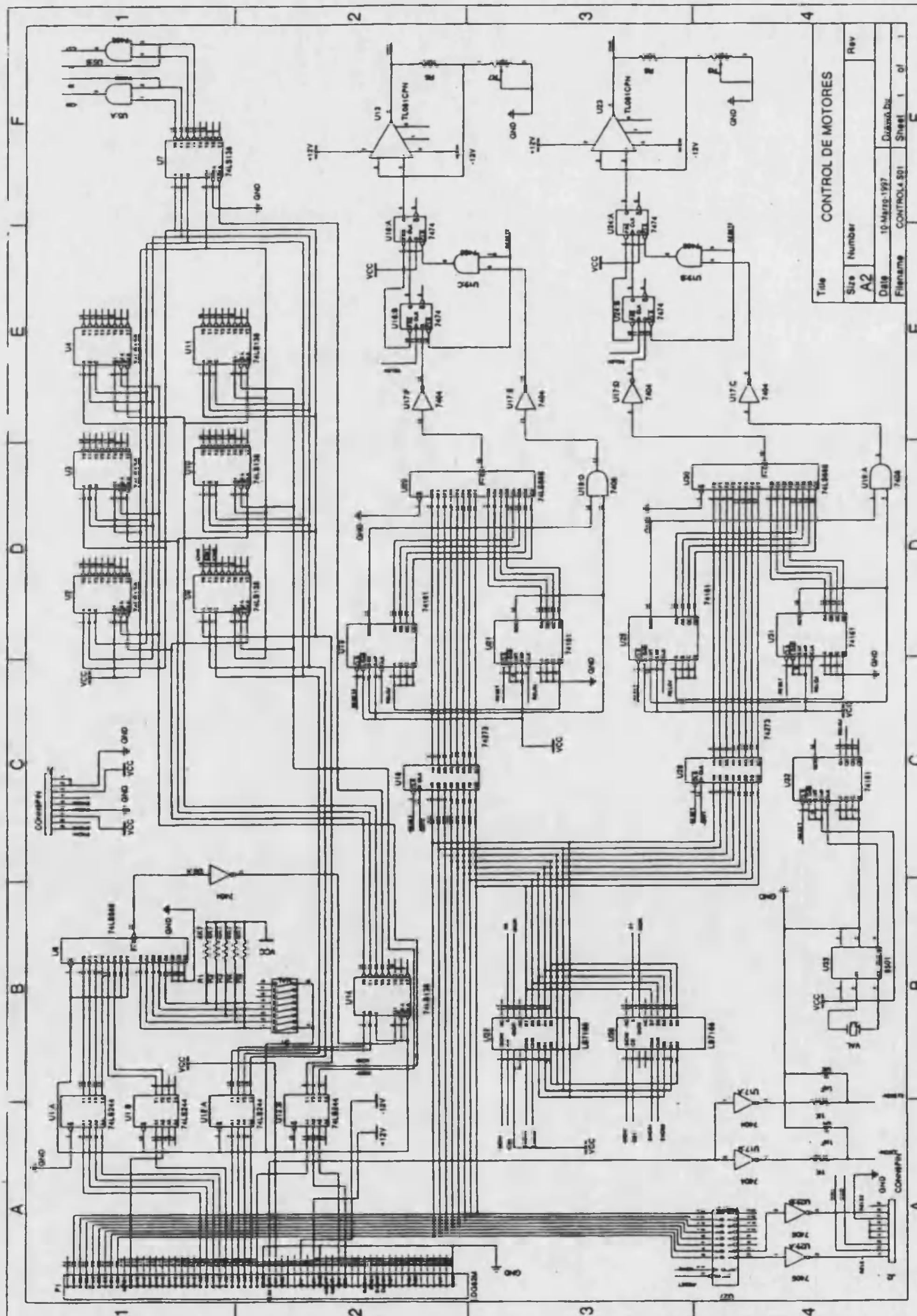
## B.1 Esquemático 1

La siguiente figura representa el esquemático de la tarjeta para generar las señales de R/W en los diferentes registros de E/S.



## B.2 Esquemático 2

La siguiente figura representa el fichero esquemático de la placa para la generación de la señal modulada en anchura de pulso (PWM).

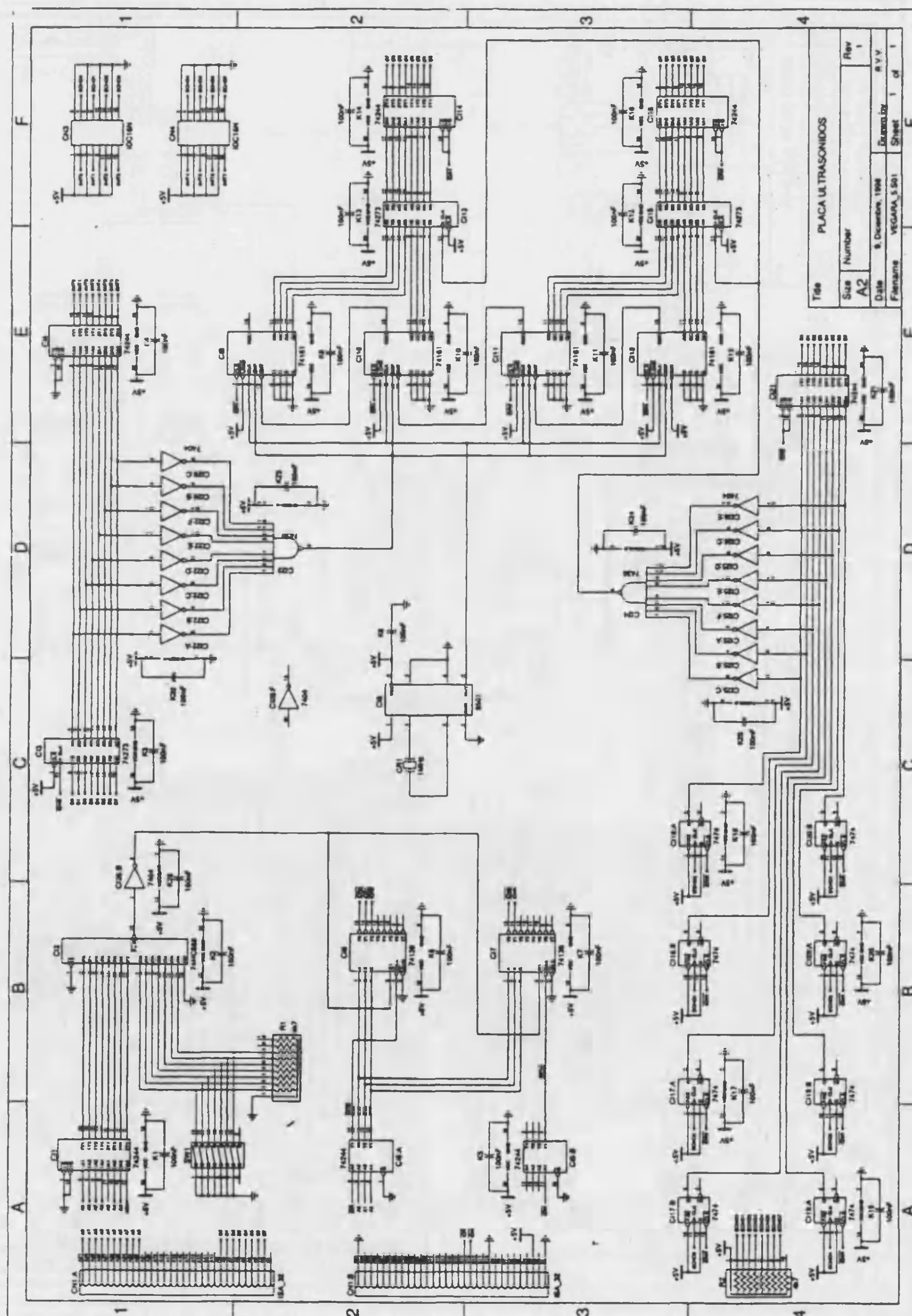


# B.3 Esquemático 3

La siguiente figura representa el fichero esquemático de la placa que gobierna el sistema odométrico (control de los codificadores ópticos incrementales).

## B.4 Esquemático 4

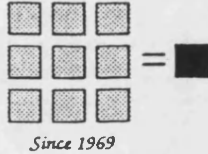
La siguiente figura representa el fichero esquemático de la placa que controla el sistema ultrasónico.



# Apéndice C

# Características del LS7166

A continuación se presentan las hojas de características y modos de funcionamiento del LS7166 comercializado por US Digital.

LSI/CSI  Since 1969

# LS7166

LSI Computer Systems, Inc. 1235 Walt Whitman Road, Melville, NY 11747 (516) 271-0400 FAX (516) 271-0405

*March 1994*

## 24 BIT MULTIMODE COUNTER

**FEATURES:**

- Programmable modes are: Up/Down, Binary, BCD, 24 Hour Clock, Divide-by-N, X1 or X2 or X4 Quadrature and Single Cycle.
- DC to 20 MHZ Count Frequency.
- 8-Bit I/O Bus for Microprocessor Communication and Control.
- 24-Bit comparator for pre-set count comparison.
- Readable status register.
- Input/Output TTL and CMOS compatible.
- 5 Volt operation.
- 20 pin Plastic DIP

**PIN ASSIGNMENT** - TOP VIEW
STANDARD 20 PIN PLASTIC DIP

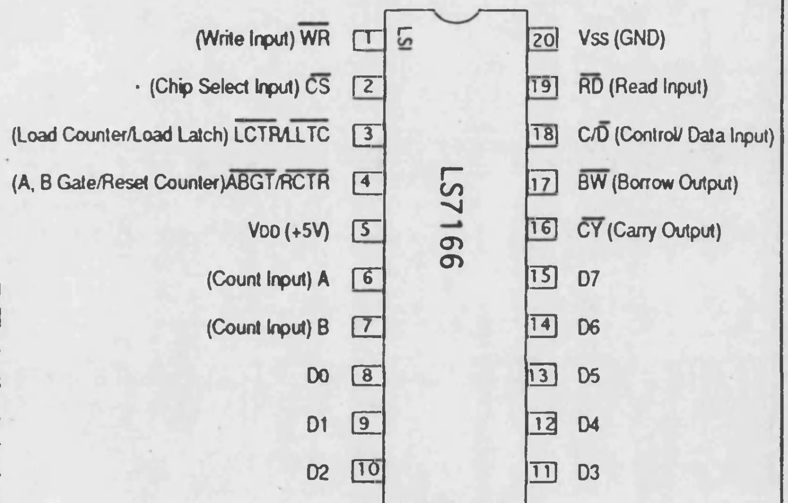


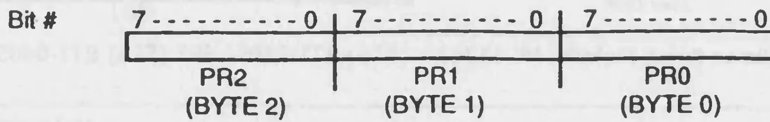

**FIGURE 1**

**GENERAL DESCRIPTION:**
The LS7166 is a monolithic, CMOS Silicon Gate, 24-bit counter that can be programmed to operate in several different modes. The operating mode is set up by writing control words into internal control registers (see Figure 8). There are three 6-bit and one 2-bit control registers for setting up the circuit functional characteristics. In addition to the control registers, there is a 5-bit output status register (OSR) that indicates the current counter status. The LS7166 communicates with external circuits through an 8-bit three state I/O bus. Control and data words are written into the LS7166 through the bus. In addition to the I/O bus, there are a number of discrete inputs and outputs to facilitate instantaneous hardware based control functions and instantaneous status indication.

**REGISTER DESCRIPTION:**
Internal hardware registers are accessible through the I/O bus (D0 - D7) for READ or WRITE when $\overline{CS}$ = 0. The C/$\overline{D}$ input selects between the control registers (C/$\overline{D}$ = 1) and the data registers (C/$\overline{D}$ = 0) during a READ or WRITE operation. (See Table 1)

The information included herein is believed to be accurate and reliable. However, LSI Computer Systems, Inc. assumes no responsibilities for inaccuracies, nor for any infringements of patent rights of others which may result from its use.

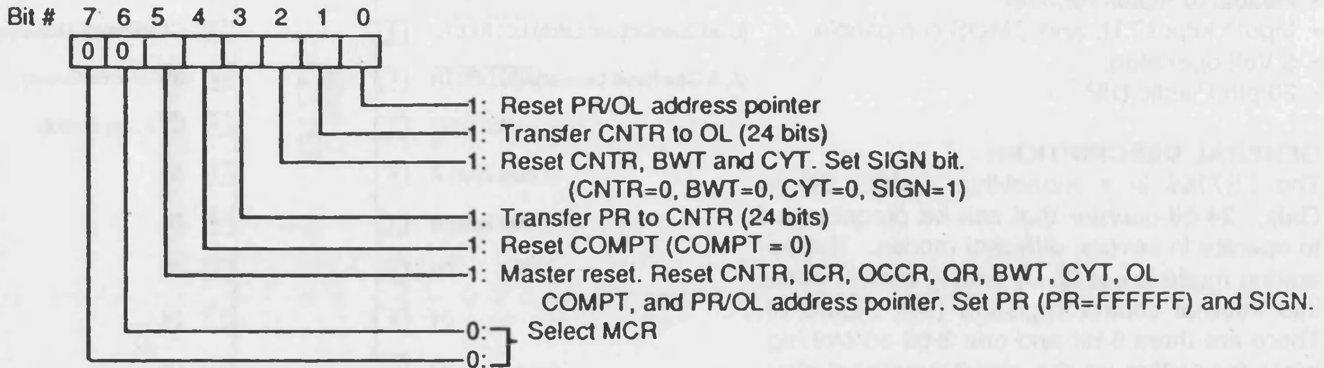
**PR (Preset register)**. The PR is the input port for the CNTR. The CNTR is loaded with a 24 bit data via the PR. The data is first written into the PR in 3 WRITE cycle sequence of Byte 0 (PR0), Byte 1 (PR1) and Byte 2 (PR2). The address pointer for PR0/PR1/PR2 is automatically incremented with each write cycle.
Accessed by: WRITE when C/$\overline{D}$ = 0, $\overline{CS}$ = 0.




Standard Sequence for Loading PR and Reading CNTR:

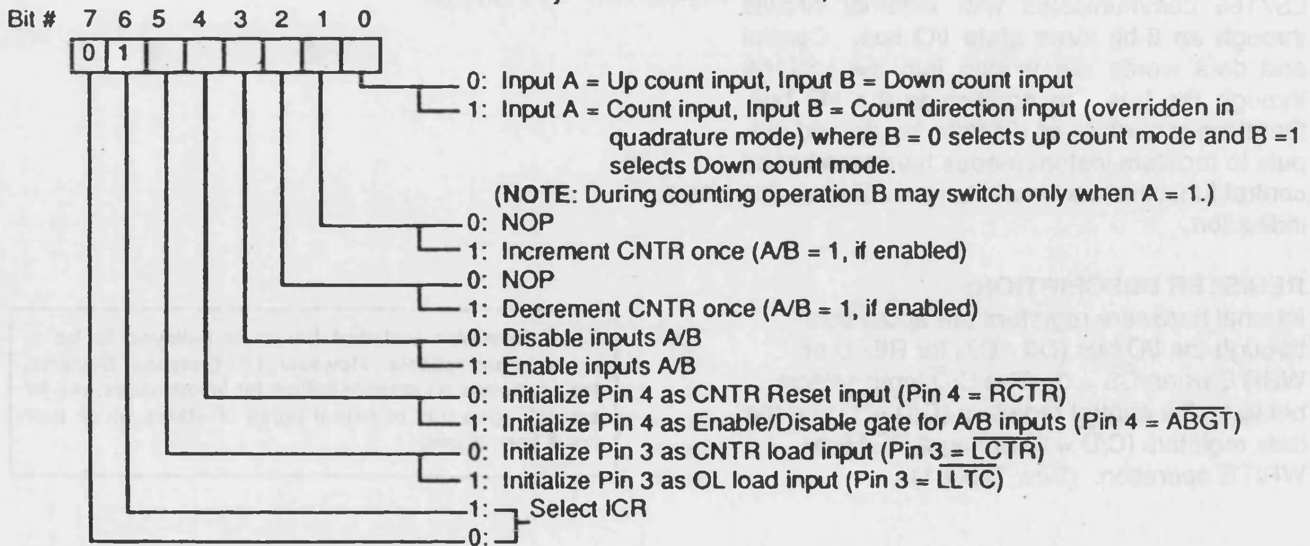
| | |
|---|---|
| 1 ⟶ MCR | ; Reset PR address pointer |
| WRITE PR | ; Load Byte 0 and into PR0 increment address |
| WRITE PR | ; Load Byte 1 and into PR1 increment address |
| WRITE PR | ; Load Byte 2 and into PR3 increment address |
| 8 ⟶ MCR | ; Transfer PR to CNTR |

**MCR (Master Control Register)**. Performs register reset and load operations. Writing a "non-zero" word to MCR does not require a follow-up write of an "all-zero" word to terminate a designated operation.
Accessed by: WRITE when C/$\overline{D}$ = 1, $\overline{CS}$ = 0.




**NOTE**: Control functions may be combined.

**ICR (Input Control Register)**. Initializes counter input operating modes.
Accessed by: WRITE when C/$\overline{D}$ = 1, $\overline{CS}$ = 0.




**NOTE**: Control functions may be combined.

7166-062394-2

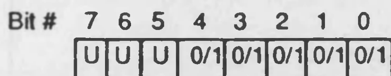
**TABLE 1 - Register Addressing Modes**

| D7 | D6 | C/$\overline{D}$ | $\overline{RD}$ | $\overline{WR}$ | $\overline{CS}$ | COMMENT |
|---|---|---|---|---|---|---|
| X | X | X | X | X | 1 | Disable Chip for READ/WRITE |
| 0 | 0 | 1 | 1 | ⊔ | 0 | Write to Master Control Register (MCR) |
| 0 | 1 | 1 | 1 | ⊔ | 0 | Write to input control register (ICR) |
| 1 | 0 | 1 | 1 | ⊔ | 0 | Write to output/counter control register (OCCR) |
| 1 | 1 | 1 | 1 | ⊔ | 0 | Write to quadrature register (QR) |
| X | X | 0 | 1 | ⊔ | 0 | Write to preset register (PR) and increment register address counter. |
| X | X | 0 | ⊔ | 1 | 0 | Read output latch (OL) and increment register address counter |
| X | X | 1 | ⊔ | 1 | 0 | Read output status register (OSR). |

X = Don't Care

**OSR (Output Status Register).** Indicates CNTR status: Accessed by: READ when C/$\overline{D}$ = 1, $\overline{CS}$ = 0.

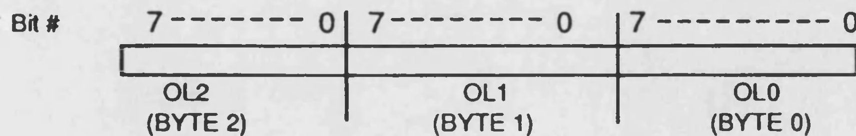
Bit # 7 6 5 4 3 2 1 0




- BWT. Borrow Toggle Flip-Flop. Toggles everytime CNTR underflows generating a borrow.
- CYT. Carry Toggle Flip-Flop. Toggles everytime CNTR overflows generating a carry.
- COMPT. Compare Toggle Flip-Flop. Toggles everytime CNTR equals PR
- SIGN. Sign bit. Reset ( = 0) when CNTR underflows
  Set ( = 1) when CNTR overflows
- UP/DOWN. Count direction indicatior in quadrature mode.
  Reset ( = 0) when counting down
  Set ( = 1) when counting up
  (Forced to 1 in non-quadrature mode)

U = Undefined

**OL(Output latch).** The OL is the output port for the CNTR. The 24 bit CNTR Value at any instant can be accessed by performing a CNTR to OL transfer and then reading the OL in 3 READ cycle sequence of Byte 0 (OL0), Byte 1 (OL1) and Byte 2 (OL2). The address pointer for OL0/OL1/OL2 is automatically incremented with each READ cycle. Accessed by: READ when C/$\overline{D}$ = 0, $\overline{CS}$ = 0.

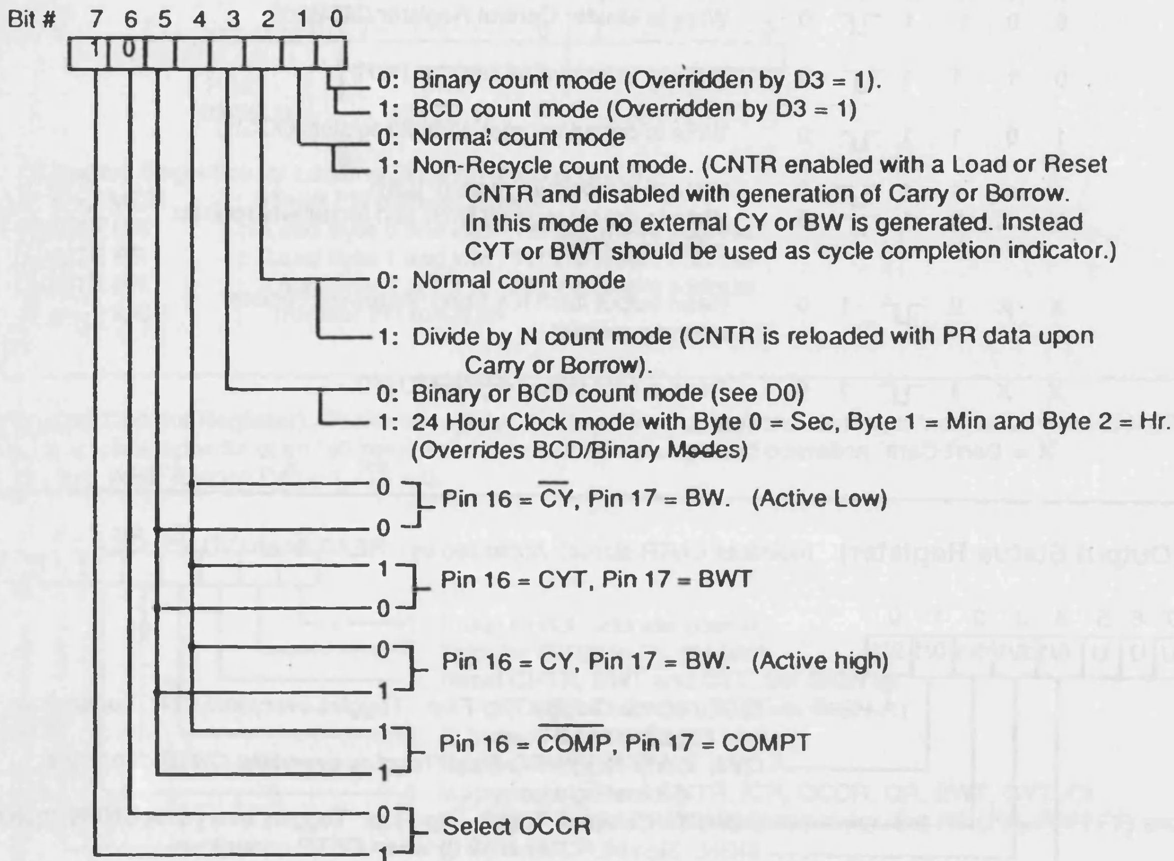



Standard Sequence for Loading and Reading OL:

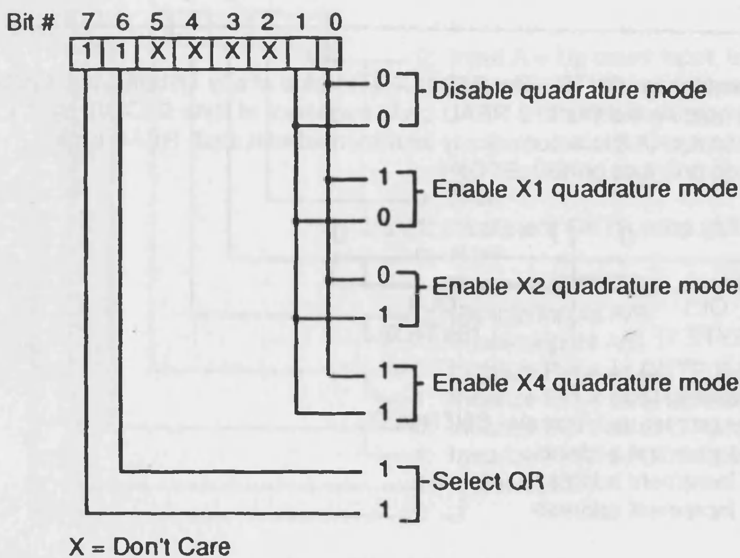
```
3 ——> MCR    ; Reset OL address pointer and Transfer CNTR to OL
READ OL      ; Read Byte 0 and increment address
READ OL      ; Read Byte 1 and increment address
READ OL      ; Read Byte 2 and increment address
```

7166-062394-3

**OCCR (Output Control Register)** Initializes CNTR and output operating modes.
Accessed by : WRITE when C/$\overline{D}$ = 1, $\overline{CS}$ = 0.




**QR (Quadrature Register).** Selects quadrature count mode (See Fig. 7)
Accessed by: WRITE when C/$\overline{D}$ = 1, $\overline{CS}$ = 0.




7166-062394-4

## I/O DESCRIPTION:
## (See REGISTER DESCRIPTION for I/O Prgramming.)

**Data-Bus (D0-D7) (Pin 8-Pin 15).** The 8-line data bus is a three-state I/O bus for interfacing with the system bus.
**$\overline{\text{CS}}$ (Chip Select Input) (Pin 2).** A logical "0" at this input enables the chip for Read and Write.
**$\overline{\text{RD}}$ (Read Input) (Pin 19).** A logical "0" at this input enables the OSR and the OL to be read on the data bus.
**$\overline{\text{WR}}$ (Write Input) (Pin 1)** A logical "0" at this input enables the data bus to be written into the control and data registers.
**C/$\overline{\text{D}}$ (Control/Data Input) (Pin 18).** A logical "1" at this input enables a control word to be written into one of the four control registers or the OSR to be read on the I/O bus. A logical "0" enables a data word to be written into the PR, or the OL to be read on the I/O bus.
**A (Pin 6).** Input A is a programmable count input capable of functioning in three different modes, such as up count input, down count input and quadrature input.
**B (Pin 7).** Input B is also a programmable count input that can be programmed to function either as down count input, or count direction control gate for input A, or quadrature input. When B is programmed as count direction control gate, B = 0 enables A as the Up Count input and B = 1 enables A as the Down Count input.
**$\overline{\text{ABGT}}$/$\overline{\text{RCTR}}$ (PIN 4).** This input can be programmed to function as either inputs A and B enable gate or as external counter reset input. A logical "0" is the active level on this input.

**$\overline{\text{LCTR}}$/$\overline{\text{LLTC}}$ (PIN 3).** This input can be programmed to function as the external load command input for either the CNTR or the OL. When programmed as counter load input, the counter is loaded with the data contained in the PR. When programmed as the OL load input, the OL is loaded with data contained in the CNTR. A logical "0" is the active level on this input.
**$\overline{\text{CY}}$ (Pin 16).** This output can be programmed to serve as one of the following:

A. $\overline{\text{CY}}$. Complemented Carry out (active "0").
B. CY. True Carry out (active "1").
C. CYT. Carry Toggle flip-flop out.
D. $\overline{\text{COMP}}$. Comparator out (active "0")

**$\overline{\text{BW}}$ (Pin 17).** This output can be programmed to serve as one of the following:

A. $\overline{\text{BW}}$. Complemented Borrow out (active "0").
B. BW. True Borrow out (active "1").
C. BWT. Borrow Toggle flip-flop out.
D. COMPT. Comparator Toggle output.

**$V_{DD}$ (Pin 5).** Supply voltage positive terminal.
**$V_{SS}$ (Pin 20).** Supply voltage negative terminal.

### Absolute Maximum Ratings:

| Parameter | Symbol | Values | Unit |
|---|---|---|---|
| Voltage at any input | $V_{IN}$ | $V_{SS}$-.5 to $V_{DD}$+.5 | Volts |
| Operating Temperature | $T_A$ | 0 to +70 | $^oC$ |
| Storage Temperature | $T_{STG}$ | -65 to +150 | $^oC$ |
| Supply Voltage | $V_{DD}$-$V_{SS}$ | +7.0 | Volts |

**DC Electrical Characteristics**. (All voltages referenced to $V_{SS}$.
$T_A$ = 0° to 70°C, $V_{DD}$ = 4.5V to 5.5V, fc = 0, unless otherwise specified)

| Parameter | Symbol | Min. Value | Max.Value | Unit | Remarks |
|---|---|---|---|---|---|
| Supply Voltage | $V_{DD}$ | 4.5 | 5.5 | Volts | - |
| Supply Current | $I_{DD}$ | - | 350 | µA | Outputs open |
| Input Low Voltage | VIL | 0 | 0.8 | Volts | - |
| Input High Voltage | VIH | 2.0 | $V_{DD}$ | Volts | - |
| Output Low Voltage | VOL | - | 0.4 | Volts | 4mA Sink |
| Output High Voltage | VOH | 2.5 | - | Volts | 200µA Source |
| Input Current | - | - | 15 | nA | Leakage Current |
| Output Source Current | ISRC | 200 | - | µA | VOH = 2.5V |
| Output Sink Current | ISINK | 4 | - | mA | VOL = 0.4V |
| Data Bus Off-State Leakage Current | - | - | 15 | nA | - |

**TRANSIENT CHARACTERISTICS** (See Timing Diagrams in Fig. 2 thru Fig. 7, $V_{DD}$ = 4. 5V to 5.5V, $T_A$ = 0° to 70°C, unless otherwise specified)

| Parameter | Symbol | Min.Value | Max.Value | Unit |
|---|---|---|---|---|
| Clock A/B "Low" | $T_{CL}$ | 20 | No Limit | ns |
| Clock A/B "High" | $T_{CH}$ | 30 | No Limit | ns |
| Clock A/B Frequency (See NOTE 1) | fc | 0 | 20 | MHz |
| Clock UP/DN Reversal Delay | $T_{UDD}$ | 100 | - | ns |
| LCTR Positive edge to the next A/B positive or negative edge delay | $T_{LC}$ | 100 | - | ns |
| Clock A/B to CY/BW/COMP "low" propagation delay | $T_{CBL}$ | - | 65 | ns |
| Clock A/B to CY/BW/COMP "high" propagation delay | $T_{CBH}$ | - | 85 | ns |
| LCTR and LLTC pulse width | $T_{LCW}$ | 60 | - | ns |
| Clock A/B to CYT, BWT and COMPT "high" propagation delay | $T_{TFH}$ | - | 100 | ns |
| Clock A/B to CYT, BWT and COMPT "low" progagation delay | $T_{TFL}$ | - | 100 | ns |
| WR pulse width | $T_{WW}$ | 60 | - | ns |
| RD to data out delay ($C_L$=20pF) | $T_R$ | - | 110 | ns |
| CS, RD Terminate to Data-Bus Tri-State | $T_{RT}$ | - | 30 | ns |
| Data-Bus set-up time for WR | $T_{DS}$ | 15 | - | ns |
| Data-Bus hold time for WR | $T_{DH}$ | 30 | - | ns |
| C/D,CS set-up time for RD | $T_{CRS}$ | 0 | - | ns |
| C/D, CS hold time for RD | $T_{CRH}$ | 0 | - | ns |
| C/D set-up time for WR | $T_{CWS}$ | 15 | - | ns |
| C/D hold time for WR | $T_{CWH}$ | 30 | - | ns |
| CS set-up time for WR | $T_{SWS}$ | 15 | - | ns |
| CS holdtime for WR | $T_{SWH}$ | 0 | - | ns |
| **Quadrature Mode:** | | | | |
| Clock A/B Validation delay (See NOTE 2) | $T_{CQV}$ | - | 160 | ns |
| A and B phase delay | $T_{PH}$ | 208 | - | ns |
| Clock A/B frequency | $f_{CQ}$ | - | 1.2 | MHz |
| CY,BW,COMP pulse width | $T_{CBW}$ | 75 | 180 | ns |

**NOTE 1:** A) In Divide by N mode, the maximum clock frequency is 10 MHZ.
B) The maximum frequency for valid CY, BW, CYT, BWT, COMP, COMPT is 10 MHz.

**NOTE 2:** In quadrature mode A/B inputs are filtered and required to be stable for at least $T_{CQV}$ length to be valid.

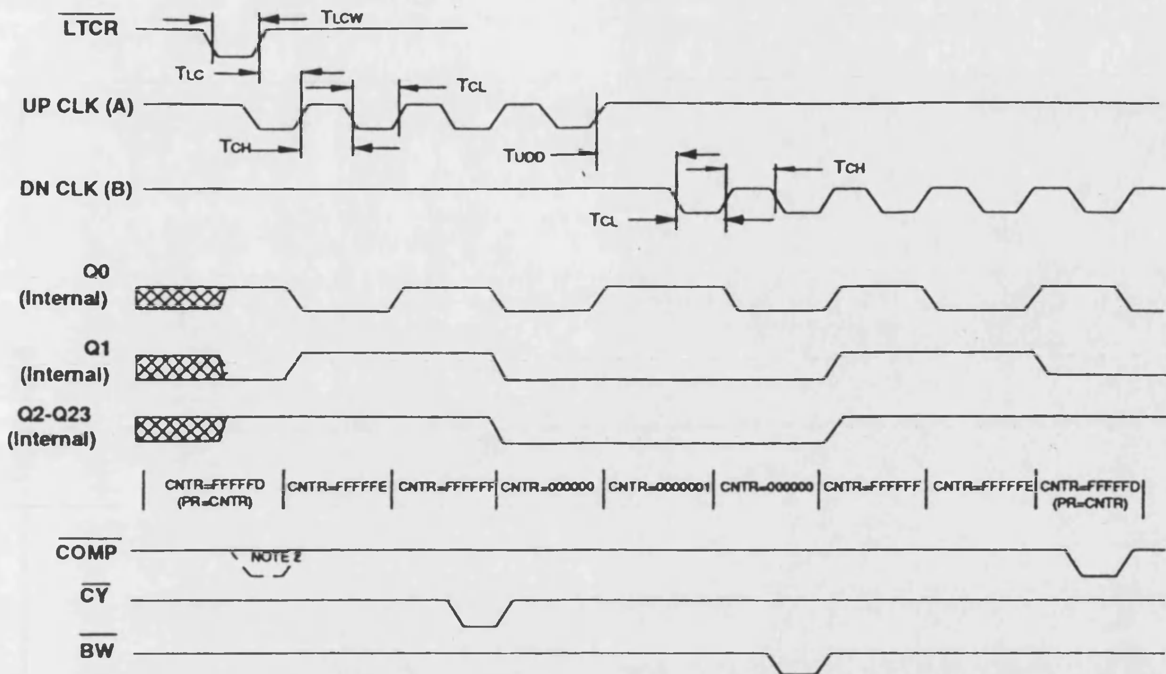



FIGURE 2 . LOAD COUNTER, UP CLOCK, DOWN CLOCK, COMPARE OUT, CARRY, BORROW

**NOTE 1:** The counter in this example is assumed to be operating in the binary mode.
**NOTE 2:** No COMP output is generated here, although PR=CNTR. COMP output is disabled with a counter load command and enabled with the rising edge of the next clock, thus eliminating invalid COMP outputs whenever the CNTR is loaded from the PR.
**NOTE 3:** When UP Clock is active, the DN Clock should be held "HIGH" and vice versa.

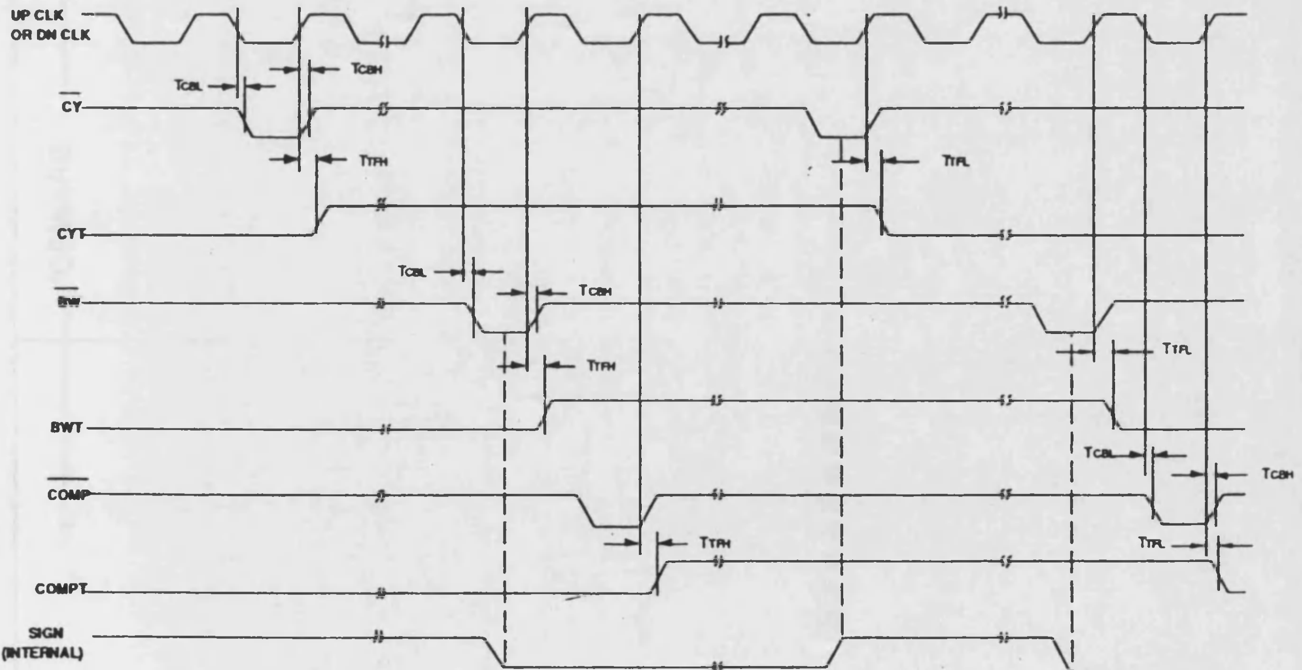



FIGURE 3. CLOCK TO CY/BW OUTPUT PROPAGATION DELAYS

7166-030192-7

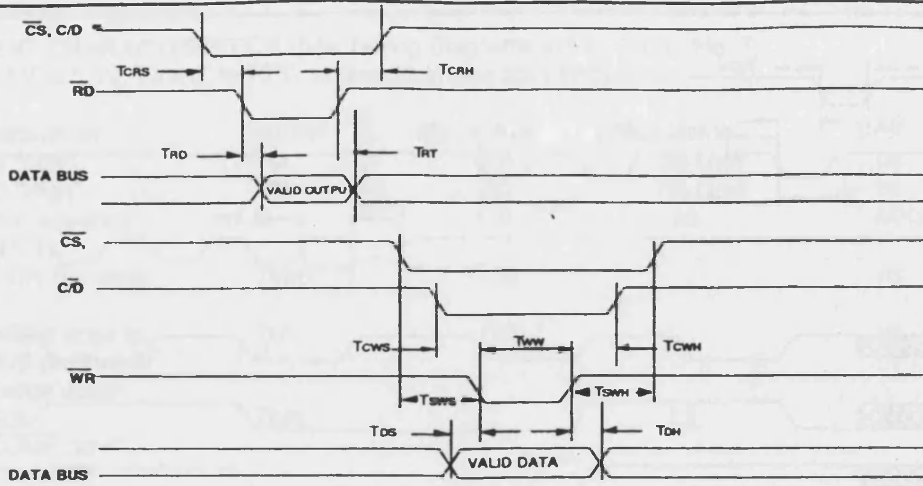
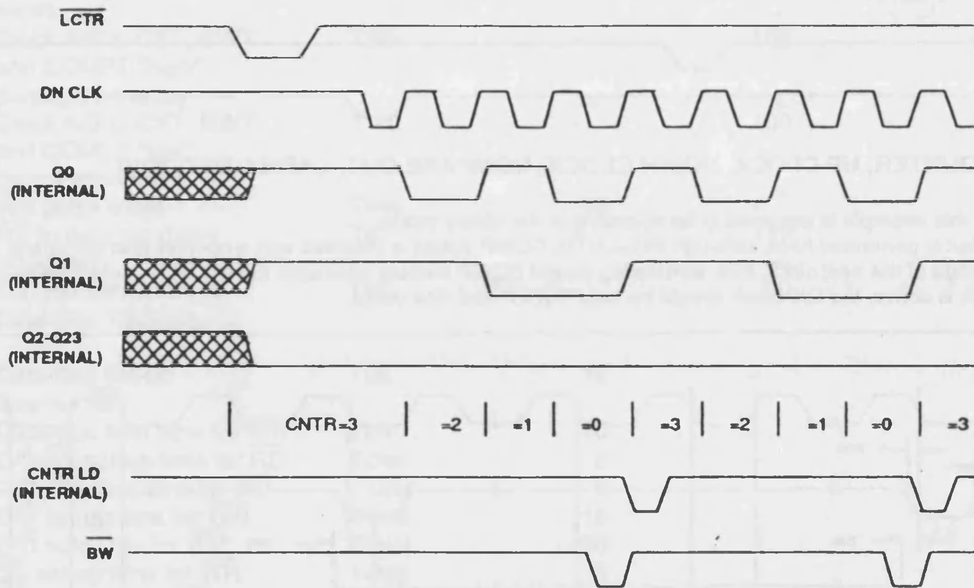



FIGURE 4. READ/WRITE CYCLES




NOTE: EXAMPLE OF DIVIDE BY 4 IN DOWN COUNT MODE

FIGURE 5. DIVIDE BY N MODE

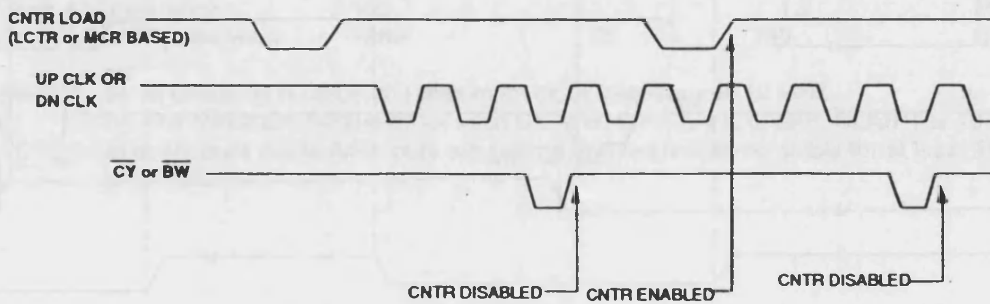



FIGURE 6. CYCLE ONCE MODE

7166-030494-8

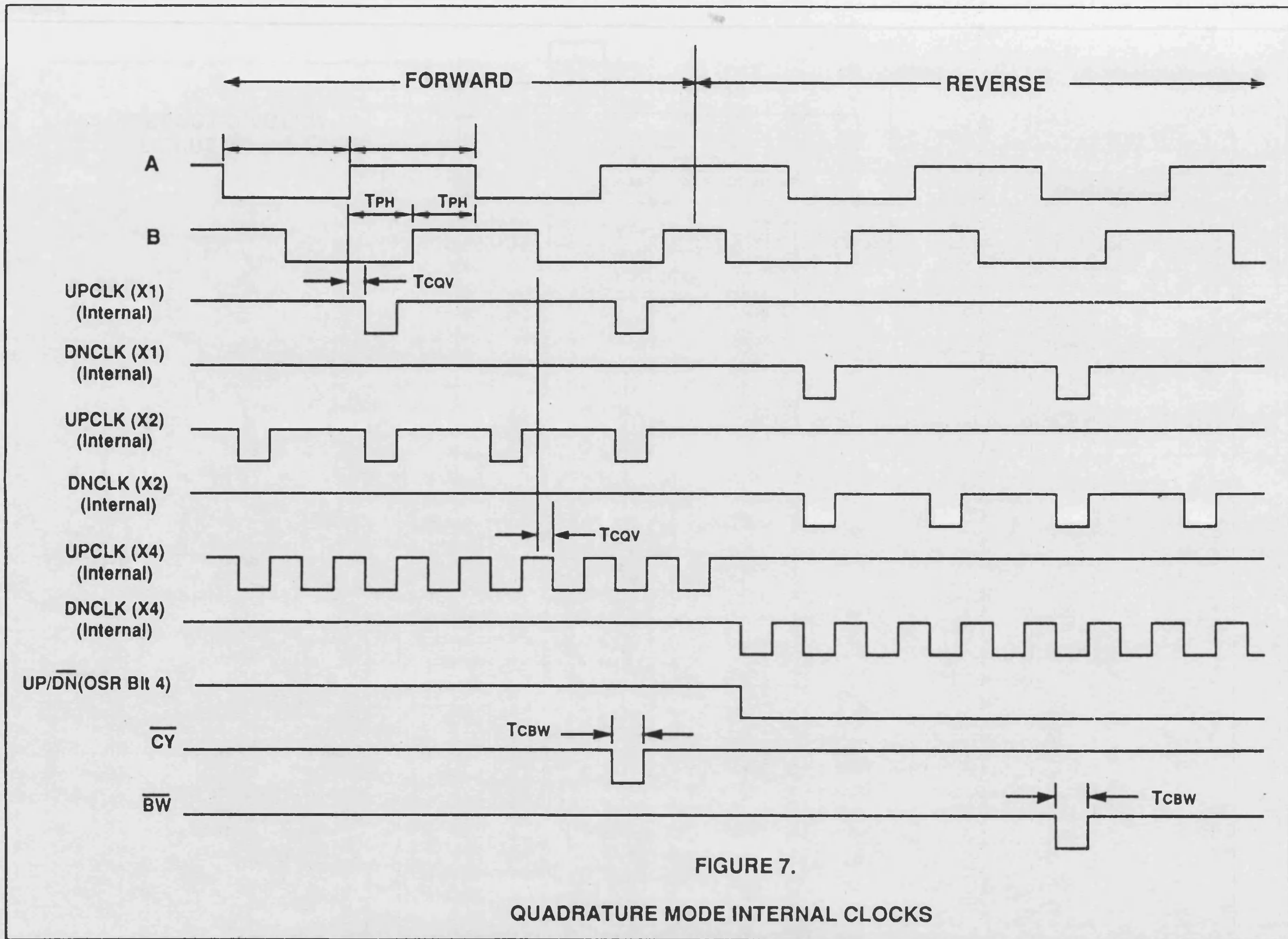



FIGURE 7.

QUADRATURE MODE INTERNAL CLOCKS

7166-030392-9

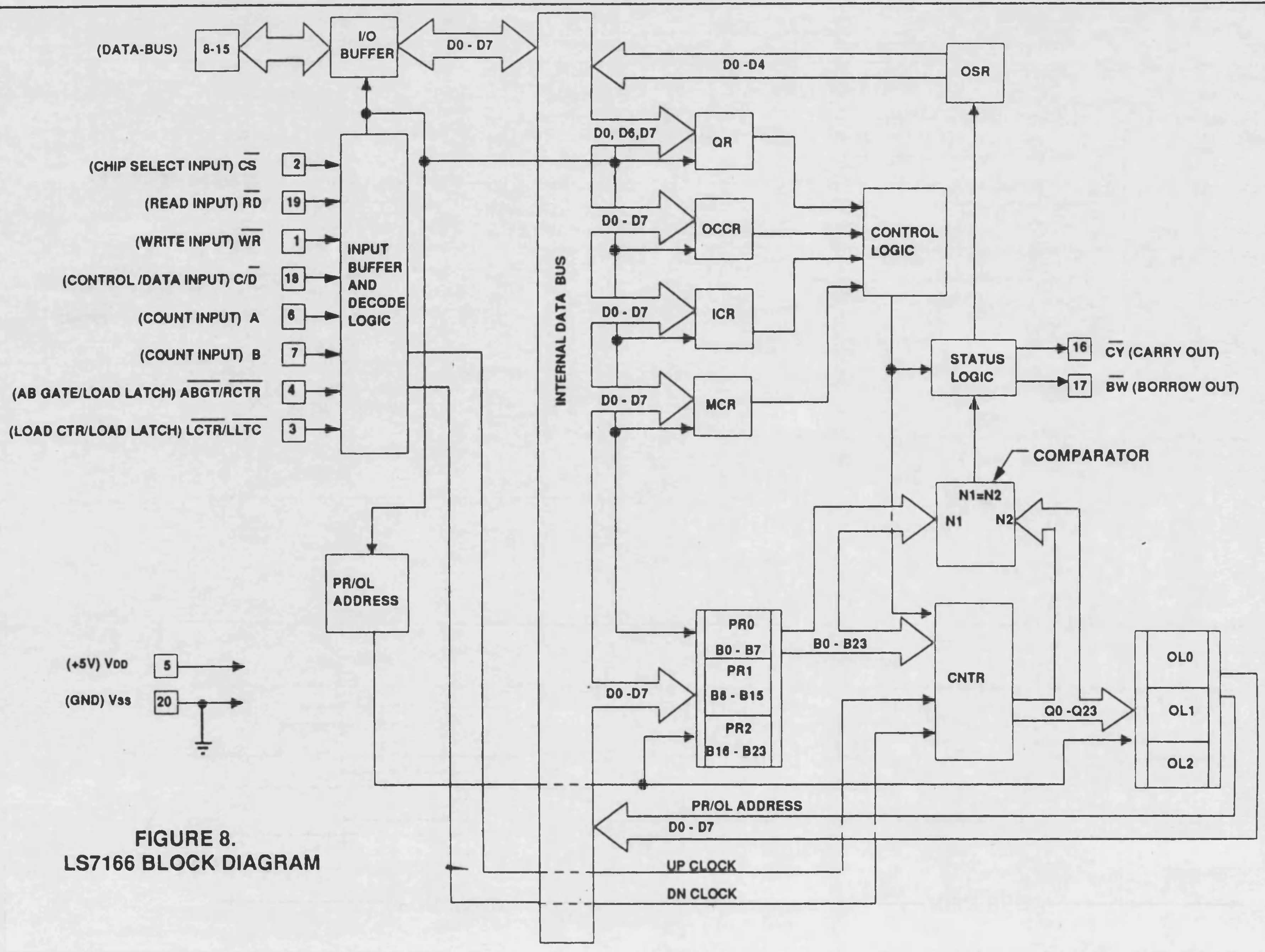
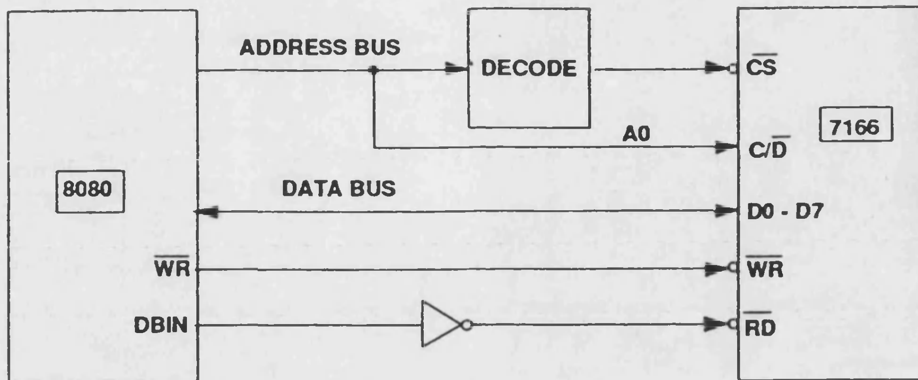
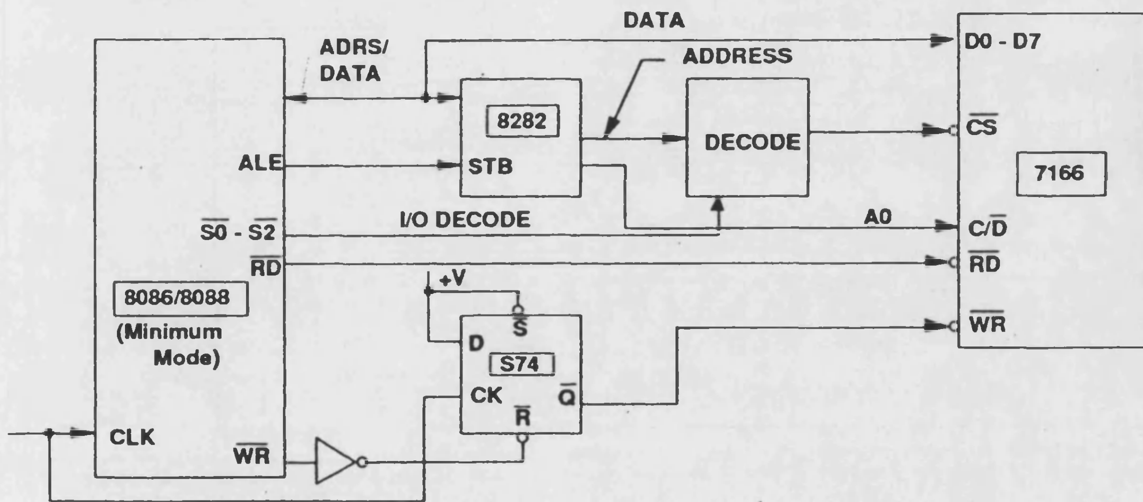
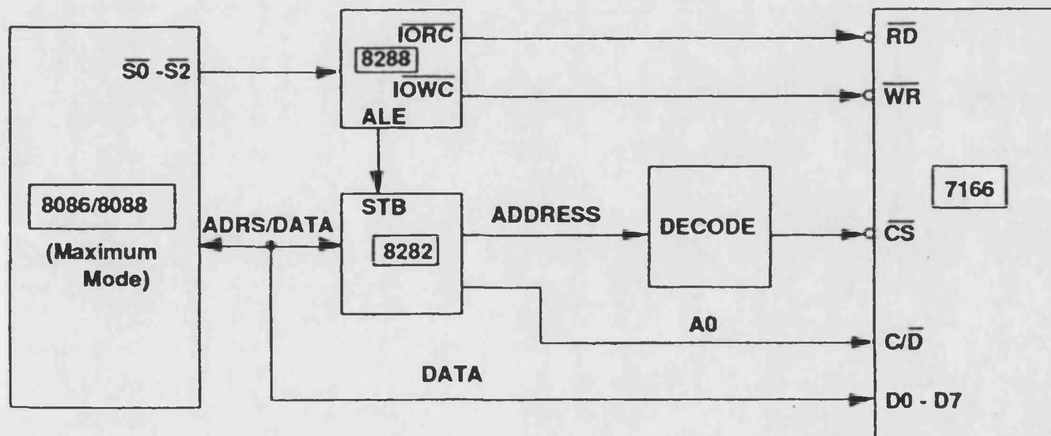



**FIGURE 8.**
**LS7166 BLOCK DIAGRAM**

## LS7166 INTERFACE EXAMPLES

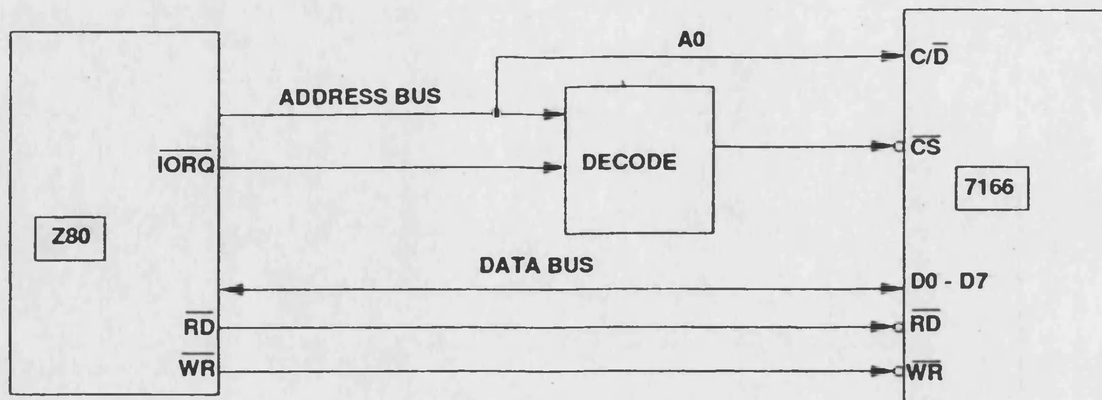


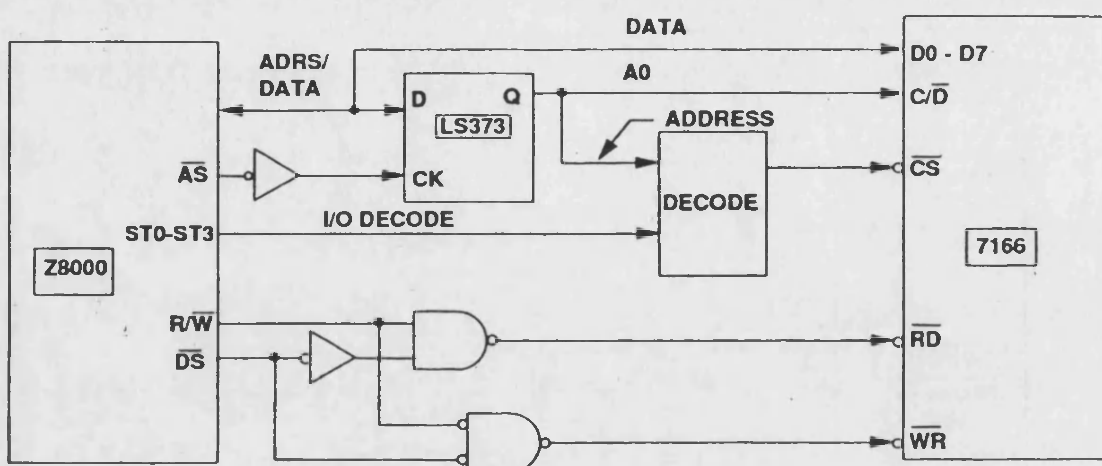



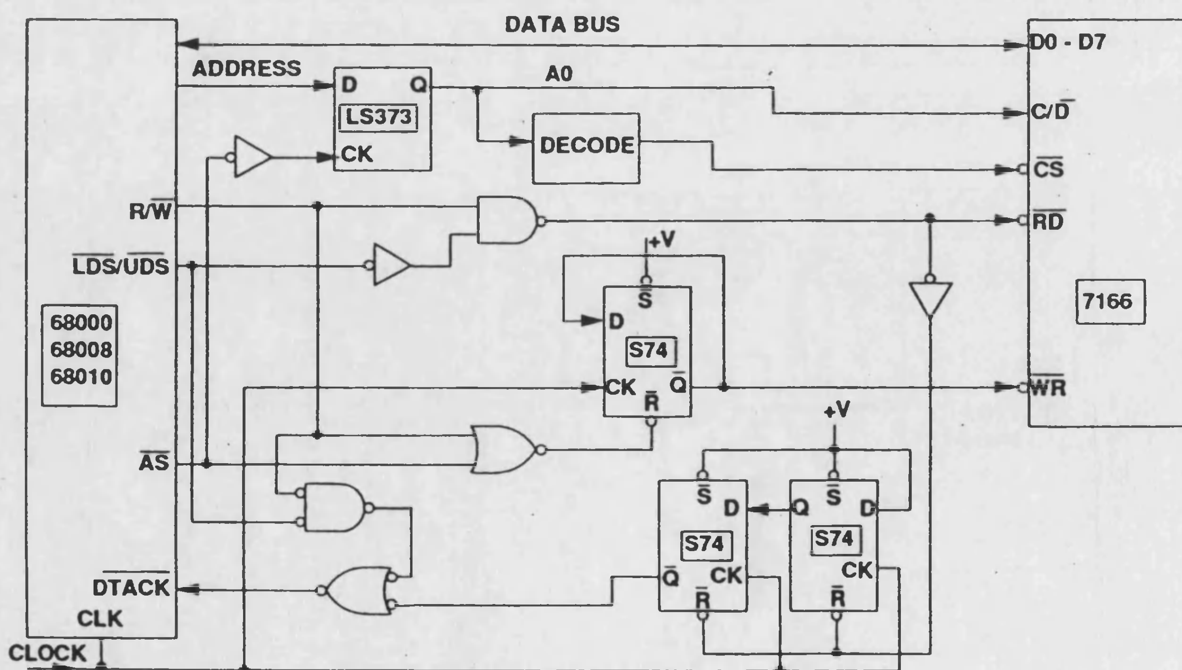




7166-03392-11

# LS7166 INTERFACE EXAMPLES




7166-03392-12

# Bibliografía

[AB] Ronald C. Arkin y Tucker Balch. Aura:principles and practice in review. En *Mobile Robot Laboratory*, Georgia Inst. of Technology.

[AH93] S. Atiya y G. Hager. *Real Time Vision-Based Robot Localization*, volumen 9,Nro 6. 1993.

[Arka] Ronald C. Arkin. Intelligent robotic systems. En *Mobile Robot Laboratory*, Georgia Inst. of Technology.

[Arkb] Ronald C. Arkin. Reactive robotic systems. En *Mobile Robot Laboratory*, Georgia Inst. of Technology.

[BA] Tucker Balch y R. Arkin. Avoiding the past: A simple but effective strategy for reactive navegation. En *Mobile Robot Laboratory*, Georgia Inst. of Technology.

[BC94] M. Betke y L. Curvits. Mobile robot localization using landmarks. En *International Conference on Intelligent Robots and Systems (IROS'94)*, Munich,Germany, Sep 1994.

[BCH88] R.A. Brooks, J.H. Connell, y P. Herbert. A second generation mobile robot. En *M.I.T. AI Lab. AI Memo 1016*, 1988.

[BF94] J. Borenstein y L. Feng. Umbmark - a method for measuring comparing and correcting dead-reckoning errors in mobile robots. En *Technical Report, The University of Michigan UM-MEAM-94*, Dec 1994.

[BF95] J. Borenstein y L. Feng. Measurement and correction of systematic odometry errors in mobile robots. En *IEEE Transactions on Robotics and Automation*, Apr 1995.

[BF96] J. Borenstein y L. Feng. Gyrodometry: A new method for combining data from gyros and odometry in mobile robots. En *IEEE Internat. Conf. on Robotics and Automation*, Minneapolis, Apr 1996.

[BK87] J. Borenstein y Y. Koren. Motion control analysis of a mobile robot. En *Journal of Dynamic Systems, Measurement and Control*, Jun 1987.

[BK89] J. Borenstein y Y. Koren. *Real-Time Obstacle Avoidance for Fast Mobile Robots*, volumen 19. Sep 1989.

[BK91a] J. Borenstein y Y. Koren. *Histogramic In-motion Mapping for Mobile Robot Obstacle Avoidance*, volumen 7,Nro 4. 1991.

[BK91b] J. Borenstein y Y. Koren. *The Vector Fielg Histogram -Fast Obstacle- Avoidance for Mobile Robots*, volumen 7,Nro 3. Jun 1991.

[Bro85] Rodney A. Brooks. A robust layered control system for a mobili robot. En *M.I.T. AI Lab. A.I. Memo 864.*, Sep 1985.

[Bro89] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. En *M.I.T. AI Lab. A.I. Memo 1091.*, Feb 1989.

[Bro91a] R.A. Brooks. Intelligence without reason. En *In Proceedings of the twelveth International Joint Conf. on A.I*, páginas 569–595, San Mateo,California, 1991.

[Bro91b] R.A. Brooks. Intelligence without representation. En *Artificial Intelligence,47*, páginas 139–159, 1991.

[BW93] B. Barsham y H.F. Durrant Whyte. An inertial navigation system for a mobile robot. En *Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems*, Yokohama,Japan, Jul 1993.

[BW95] B. Barsham y H.F. Durrant Whyte. *Inertial Navigation Systems Mobile Robots*, volumen 11,Nro 3. Jun 1995.

[CC92] F. Chenavier y J. Crowley. Position estimation for a mobile robot using vision and odometry. En *Proc. of IEEE Internat. Conf. on Rob. and Autom.*, Nice,France, May 1992.

[CJ94] J. Courtney y A. Jain. Mobile robot localization via classification of multisensor maps. En *Proceedings of the IEEE International Conference on Robotics and Automation*, San Diego,CA, May 1994.

[CK92] C. Cohen y F. Koss. A comprehensive study of three objects triangulation. En *Proceedings of the SPIE Conference on Mobile Robots*, Boston,MA, Nov 1992.

[Cox91] I.J. Cox. Blanche - an experiment in guidanceand and navigation of an autonomous mobile robot. En *IEEE Trans. Robotics ans Automation*, páginas 193–204, 1991.

[CR92] J.L. Crowley y P. Reignier. *Asynchronous Control of Rotation and Translation for a Robot Vehicle*, volumen 10, páginas 243–251. 1992.

[ea90] P. Hoppen et al. Laser-radar mapping and navigation for an autonomuos mobile robot. En *Proceedings of the IEEE International Conference on Robotics and Automation*, Cincinati,OH, May 1990.

[ea93a] J. Vaganay et al. Mobile robot attitude estimation by fusion of inertial data. En *Proceedings of IEEE Internat. Conf. on Robotics and Automation*, Atlanta, May 1993.

[ea93b] J. Vaganay et al. Mobile robot localization by fusing odometric and inertial measurements. En *Topical Meeting on Robotics and Remote Systems*, Knoxville, Apr 1993.

[ea93c] M. Jenkin et al. Global navigation for ark. En *Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems*, Yokohama,Japan, Jul 1993.

[ea94a] E. Stuck et al. Map updating and path planing for real-time mobile robot navigation. En *International Conference on Intelligent Robots and Systems*, Munich,Germany, Sep 1994.

[ea94b] M. Adams et al. Control and localisation of a post distributing mobile robot. En *Internat. Conference on Intelligent Robots and Systems*, Munich,Germany, Sep 1994.

[ea94c] Y. Tonouchi et al. Fusion of dead-reckoning positions witch a workspace model for a mobile robot by bayesian inference. En *Internat. Conference on Intelligent Robots and Systems*, Munich,Germany, Sep 1994.

[ea95] Wilco Oelen et al. Hibrid stabilizing control on a real mobile robot. En *IEEE Robotics and Automation Magazine*, Jun 1995.

[EP94a] Thomas E. y Ewald Von Puttkamer. Exploration of an indoor-environment by an autonomous mobile robot. En *International Conference on Intelligent Robots and Systems (IROS'94)*, Munich,Germany, Sep 1994.

[EP94b] T. Edlinger y E. Puttkamer. Exploration of an indoor environment by an autonomous mobile robot. En *International Conference on Intelligent Robots and Systems (IROS'94)*, Munich,Germany, Sep 1994.

[Etz93] Oren Etzioni. Intelligence without robots (a reply to brooks). En *A.I Magazine*, Dec 1993.

[Eva94] J.M. Evans. Helpmate:an autonomous mobile robot courier for hospitals. En *IROS94*, páginas 1695–1700, Munich, Germany, Sep 1994.

[Eve95] H. Everett. *Sensor for Mobile Robots: Theory and Application*. A.K.Peters,Ltd. Wellesley,MA, 1995.

[GT94] C. Gourley y M. Trivedi. Sensor based obstacle avoidance and mapping for fast mobile robots. En *Proceedings of IEEE Internat. Conference on Robotics and Automation*, San Diego,CA, May 1994.

[Hay69] Patrick J. Hayes. *Robotologic, in Machine Intelligence 5.* eds. Bernard Meltzer and Donald Michie.Edinburgh University Press., 1969.

[Hol91] J. Hollingum. *Caterpillar make the eart move:automatically*, volumen 18,Nro 2, páginas 15–18. Jul 1991.

[HV97] Karen Zita Haigh y Manuela M. Veloso. Hight-level planing and low-level execution: Towards a complete robotic agent. En *In Proceedings of the First International Conference on Autonomous Agents*, Menlo Park, CA, Feb 1997.

[JF93] Joseph L. Jones y Anita Flynn. *Mobile Robots. Inspiration to Implementation.* A.K. Peters, 1993.

[Jor95] K.W. Jorg. *World Modeling for an Autonomous Mobile Robot Using Heterogenous Sensor Information*, volumen 14. 1995.

[KB91] Yoram Koren y Johann Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. En *Proceedings of the IEEE International Conference on Robotics and Automation*, Sacramento, CA, Apr 1991.

[Kla88] P.R. Klarer. Simple 2-d navigation for wheeled vehicles. En *Sandia Report SAND88-0540*, Sandia National Laboratories. Albuquerque, Apr 1988.

[Kor80] Yoram Koren. Cross-coupled biaxial computer control for manufacturing systems. En *Journal of Dynamic Systems, Measurement and Control*, Dec 1980.

[Kro85] B.H. Krogh. A generalized potential field approach to obstacle avoidance control. En *International Robotics Research Conference*, Bethlehem,PA, Aug 1985.

[Kur96] Andreas Kurz. Constructing maps for mobile robot navigation based on ultrasonic range data. En *IEEE Trans. on Systems, Man and Cibernetics*, Apr 1996.

[LS89] G.F. Luger y W.A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems.* The Benjamin/Cummings Publishing Company,Inc., 1989.

[Mad94] J. Maddox. Smart navigation sensors for automatic guided vehicles. En *Sensors*, Apr 1994.

[Mal91] C. Malcon. Behavioural modules in robotic assembly. En *Dep. of A.I, Draft Teaching Paper*, March 1991.

[ME85] H.P. Moravec y A. Elfes. High resolution maps from wide angle sonar. En *Proceedings of the IEEE Conference on Robotics and Automation*, Washington,DC, 1985.

[Mea82] M.Brady y editors. et al. *Robot Motion:Planning and Control. Chapter 3.* MIT Press Series in AI. MIT Press, 1982.

[Min61] Marvin Minsky. Steps towards artificial intelligence. En *in Proc. of the Inst. of Radio Engineers,49*, páginas 39–55, Jan 1961.

[MM92] Y. Mesaki y I. Masuda. A new mobile robot guidance system using optical reflectors. En *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Jul 1992.

[Mor88] H.P. Moravec. Sensor fusion on certainty grids for mobile robots. En *AI Magazine*, Jul 1988.

[MSH89] C. Malcon, T. Smithers, y J. Hallam. An emerging paradigm in robot architecture. En *Intelligent Autonomous systems 2 Conference.*, Amsterdam, Dec 1989.

[Nil94] Nils J. Nilsson. Teleo-reactive programs for agent control. En *Journal of Artificial Intelligence Research*, páginas 139–158, Jan 1994.

[NS89] A.Ñewell y H. Simon. Computer science as empirical enquiry. En *Scientific American*, 1989.

[O.85] Khatib O. Real-time obstacle avoidance for manipulators and mobile robots. En *IEEE International Conference on Robotics and Automation*, St Louis, 1985.

[PG93] D. Parish y R. Grabbe. Robust exterior autonomous navigation. En *Proceedings of the SPIE Conference on Mobile Robots*, Boston,MA, Sep 1993.

[Ren93] W.D. Rencken. Concurrent localization and map building for mobile robots using ultrasonic sensors. En *Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems*, Yokohama,Japan, Jul 1993.

[R.H93] R.H.Byrne. Techniques for autonomous navigation. En *Sandia National Laboratories (report 0457)*, Albuquerque, Sep 1993.

[Sim94] Reid Simmons. Structured control for autonomous robots. En *IEEE Transactions on Robotics and Automation.*, páginas 34–43, Feb 1994.

[Tay91] C. Taylor. Building representations for the environment of a mobile robot from image data. En *Proceedings of the SPIE Conference on Mobile Robots*, Boston,MA, Nov 1991.

[Tro] Troylfes. *Programación Linux.*

[WB] A.D. Webber y D.L. Bisset. Competition and cooperation. a model for behaviour-based robot controllers. En *University of Kent. Canterbury.*

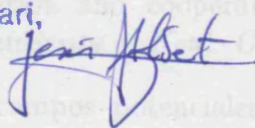
[yJT97] Ricardo P. y Josep Tornero. Uso de campos potenciales artificiales en el guiado de robots móviles. En *Dep. Ing. Sistemas y Automática. Univ. Polit. de Valencia*, 1997.

UNIVERSITAT DE VALÈNCIA FACULTAD DE CIENCIES FISIQUES

Reunit el Tribunal que subscriu, en el dia de la data, acorda d'atorgar, per unanimitat, a aquesta Tesi Doctoral d'En/ Na/ N' Francisco Vegara Meseguer la qualificació d Sobresaliente "cum laude"

València a 15 d Septiembre de 1999

El Secretari,



El President,