UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA INDUSTRIAL

PROYECTO FINAL DE CARRERA

MIGRACIÓN DE LA PLATAFORMA A BORDO DEL ROBOT ASISTENCIAL ASIB0T

AUTOR: *RAÚL PALENCIA LÓPEZ*

TUTOR: *ALBERTO JARDÓN HUETE*

NOVIEMBRE 2009

Agradecimientos

En primer lugar, me gustaría agradecer a todos los miembros del grupo de trabajo que hemos formado *ASIBOT* durante estos 3 últimos años (Alberto Jardón, Santiago Martínez, Víctor Placer, Carlos Pérez , Juan González y Martin Stoelen) por lo que he aprendido de ellos, por su ayuda incondicional, por su buen humor, por haber entendido el tiempo que le tenía que dedicar a mis exámenes sin exigirme en estos periodos y sobre todo por la fuerte evolución profesional que he tenido junto a ellos y, por supuesto, a Antonio Giménez por apreciar mi trabajo constante y a Carlos Balaguer por la oportunidad que me brindó de trabajar en este proyecto tan gratificante.

No quiero olvidarme del resto de compañeros del Dpto. de Sistemas y automática por darme siempre su ayuda y en muchos casos su amistad.

Gracias a mis padres, Julia López y Ángel Palencia, y a mi hermano, Aitor Palencia, por comprender la absorbente dedicación que le he tenido que dedicar a esta carrera que por fin termino y por saber disfrutar del poco tiempo que les he ofrecido durante todos estos años.

Gracias a mi novia, Noelia García, por comprender la dedicación que me ha requerido este tramo final y hacerse cargo de la responsabilidad de dos, e intentar hacerme ir a por todo cuando todo parecía imposible.

Y por último a mis amigos (Víctor Placer, Leandro Boyano, Néstor Delgado y Carlos González) por haber estado, por estar y porque sé que estarán cuando les necesite.

Resumen

El grupo Robotics-lab de la Universidad Carlos III de Madrid, desarrolló en 2004 un primer prototipo del robot asistencial *ASIBOT*, cuyo nombre en el contexto del proyecto europeo fue *MATS* (*flexible Mechatronic Assistive Technology System to support persons with special needs in all their living and working environments*), cuya finalidad es la ayuda y asistencia a personas discapacitadas y de la tercera edad.

Este proyecto fin de carrera surge al darse la necesidad de actualizar el *hardware* interno a bordo del robot, solventando así problemas encontrados en las fases de pruebas con usuarios reales, y superando las limitaciones del anterior sistema de comunicaciones interno.

En este contexto se ha desarrollado dicho proyecto, cuyo objetivo era la migración de la aplicación original a bordo de *ASIBOT* a un sistema *Linux embebido* con capacidades de tiempo real en una tarjeta *phyCORE PXA270* siendo capaz de dar respuesta a una comunicación interna entre distintos dispositivos basada en el bus *CAN* y una comunicación externa mediante el protocolo *Wi-Fi* 802.11b.

Para ello, ha sido necesario sustituir el sistema operativo anterior, utilizando diferentes *drivers* específicos y diferentes herramientas para su programación. Esto ha implicado una completa migración del código fuente original, mediante múltiples recompilaciones y adaptaciones de los módulos del *kernel* hasta conseguir el correcto funcionamiento del sistema y los dispositivos.

Abstract

The RoboticsLab research group, part of University Carlos III de Madrid, developed a first prototype of the assistive robot ASIBOT in 2004. Its name in European project context was MATS (*flexible Mechatronic Assistive Technology System to support persons with special needs in all their living and working environments*). Its objective is helping and assisting impaired and elder people.

This career-end project is derived from the need of updating the robot's internal hardware. This way, problems found in test phases with real patients and internal communication limitations of the predecessor system were reduced.

This is the context in which the project has been developed. Objectives were integration with a Linux system with real time capacities, on board *phyCORE PXA270*, capable of managing internal communications between modules, such as motor controllers, sensors, etc., based on CAN-bus and external communications using the 802.11b wireless protocol, making the complete migration to the *ASIBOT's* application.

For this, the previous operating system had to be replaced, specific hardware drivers had to be substituted, migration to different APIs had to be performed. This is, tasks implied a complete migration of the original source code at the end of the Project, along with the multiple adaptations of modules and kernel recompilation necessary for the correct functioning of the system and their devices.

Índice General

Lista de Figuras 11

Lista de Tablas 15

1. Introducción 19

1.1. Motivaciones 21

1.2. Objetivos del proyecto 25

1.3. Estructura del documento 26

2. Sistema de partida 27

2.1. Descripción de la arquitectura asistencial 28

2.1.1. Sistema de agarre y conectores 30

2.1.2. Silla de ruedas 32

2.2. Descripción del robot asistencial ASIBOT 33

2.2.1. Estructura mecánica 34

2.2.2. Estructura Hardware 36

2.2.3. Estructura Software 41

2.2.4. Estructura de procesos y comunicaciones 44

3. Acondicionamiento software del PXA270 51

3.1. Instalación del bootloader 54

3.2. Herramientas para la construcción del kernel 59

3.3. Construcción y configuración del sistema 71

3.4. Modos NFS y StandAlone. Automatización del Inicio. 100

4. Migración del software de control de ASIBOT. 111

4.1. Herramienta de desarrollo de la aplicación 112

4.2. Métodos modificados debidos al Sistema Operativo 119

4.3. Módulos añadidos a la aplicación original 126

4.3.1. Implementación de comunicaciones del bus CAN 128

4.3.2. Módulo de procesamiento de mensajes para el Joystick 133

5. Conclusiones y futuros desarrollos 135

5.1. Conclusiones 135

5.2. Futuros desarrollos 138

6. Bibliografía 145

Lista de Figuras

Figura 1. 1 Brazo robótico ASIBOT anclado en una DS 20

Figura 1. 2 Dispositivo de anclaje, Docking Station, del robot ASIBOT 20

Figura 1. 3 Tarjeta CerfBoard Windows CE 3.0 21

Figura 1. 4 Esquema de conexionado entre CerfBoard y amplificadores 22

Figura 1. 5 phyCORE-PXA270 a bordo de ASIBOT 24

Figura 2. 1 Arquitectura completa del sistema 29

Figura 2. 3 Sistema de anclaje y garra de ASIBOT 31

Figura 2. 2 Contactos de Alimentación de Asibot 31

Figura 2. 4 Silla de ruedas con anclaje DS en raíl 33

Figura 2. 5 Distribución de articulaciones en ASIBOT y ASIBOT llenando un vaso . 34

Figura 2. 6 Estructura mecánica del eje central 35

Figura 2. 7 Tubo de fibra de carbono 35

Figura 2. 8 Distribución de elementos en la estructura 36

Figura 2. 9 Driver original y sin carcasa 38

Figura 2. 10 Torque Motor 38

Figura 2. 11 Harmonic Drive 39

Figura 2. 12 Encoder relativo modelo RCML de Renco 39

Figura 2. 13 Esquema de la arquitectura software del sistema 41

Figura 2. 14 Interfaz de joystick 43

Figura 2. 15 Canales de comunicación entre HMI y robot 46

Figura 2. 16 Flujo de información por los diferentes canales 47

Figura 2. 17 Esquema de procesos en el servidor 47

Figura 2. 18 Esquema de procesos en la interfaz gráfica de usuario 49

Figura 3. 1 Particionado de la memoria flash 55

Figura 3. 2 Driver de acceso a registros 56

Figura 3. 3 Programación del bootloader 57

Figura 3. 4 Interfaz del bootloader 58

Figura 3. 5 Interpretación del núcleo 59

Figura 3. 6 Interfaz boardsetup 65

Figura 3. 7 Interfaz Kernelconfig 66

Figura 3. 8 Interfaz menuconfig 67

Figura 3. 9 Opciones de Ptxdist 68

Figura 3. 10 Directorio de componentes del BSP 69

Figura 3. 11 Paquetes instalados 69

Figura 3. 12 Copia del Sistema en root 70

Figura 3. 13 Paquetes software descargados 70

Figura 3. 14 Ficheros de reglas 71

Figura 3. 15 Configuración general 73

Figura 3. 16 Configuración Sistema de Archivos 85

Figura 3. 17 Configuración base 86

Figura 3. 18 Función no definida en RT Preempt 98

Figura 3. 19 Modificación de funciones no definidas 99

Figura 3. 20 USB Wi-Fi modelo WUSB54G 99

Figura 3. 21 Editar reglas del gestor de dispositivos udev 100

Figura 3. 22 Sistema raíz en NFS 101

Figura 3. 23 Modo Stand-Alone 101

Figura 3. 24 Grabación en memoria flash 105

Figura 3. 25 Descarga del driver USB_Wi-Fi 106

Figura 3. 26 Enlaces simbólicos a scritps de inicio 108

Figura 3. 27 Configuración de un general de un script 108

Figura 3. 28 Configuración del script arranque 109

Figura 3. 29 Configuración del script server 110

Figura 4. 1 Declaraciones generadas por eVC 111

Figura 4. 2 HelloWorld aplicación de iniciación 114

Figura 4. 3 Configuración del compilador en eclipse 115

Figura 4. 4 Directorio “include” del kernel construido 116

Figura 4. 5 Visualización de la compilación en la consola de Eclipse 117

Figura 4. 6 Configuración para la depuración 118

Figura 4. 7 Configuración de depuración vía red 119

Figura 4. 8 Programación de Socket en modo pasivo 121

Figura 4. 9 Programación de transmisión y recepción mediante sockets 122

Figura 4. 10 Hilos concurrentes en un proceso 123

Figura 4. 11 Configuración de atributos, espera y cancelación de Threads 124

Figura 4. 12 Bloqueo y desbloqueo de secciones criticas 125

Figura 4. 13 Cambio de función de conversión de datos 126

Figura 4. 14 Diagrama del nuevo esquema de comunicaciones 127

Figura 4. 15 Configuración del socket de conexión CAN 129

Figura 4. 16 Funciones de lectura y escritura del bus CAN 129

Figura 4. 17 Configuración de la estructura de datos de la red CAN 130

Figura 4. 18 Tratamiento de los datos recibidos de la red CAN 130

Figura 4. 19 Función se sincronización con encoder absoluto 131

Figura 4. 20 Función de lectura de encoder absoluto 132

Figura 4. 21 Función de escritura de posición absoluta en el driver 132

Figura 4. 22 Diagrama de bloques con módulo de Joystick 134

Figura 5. 1 Tarjeta RB 100 143

Lista de Tablas

Tabla 3. 1 Configuración del fichero batch de grabación del u-boot 56

Tabla 3. 2 Comandos de construcción de la herramienta 64

Tabla 3. 3 Comandos de construcción del directorio donde se realizará el proyecto 64

Tabla 3. 4 Características hardware de phycore y funcionalidades del BSP 72

Tabla 3. 5 Configuración de General Setup 74

Tabla 3. 6 Habilitar carga de módulos 74

Tabla 3. 7 Configuración del tipo de sistema y del soporte de bus 75

Tabla 3. 8 Características del kernel sin funciones de tiempo real habilitadas 76

Tabla 3. 9 Características del kernel con funciones de tiempo real habilitadas 76

Tabla 3. 10 Opciones de inicio 76

Tabla 3. 11 Configuración de comunicaciones a) 77

Tabla 3. 12 Configuración de comunicaciones b) 78

Tabla 3. 13 Configuración de comunicaciones c) 78

Tabla 3. 14 Configuración genérica de dispositivos drivers. 79

Tabla 3. 15 Configuración de dispositivos driver de red, I2C y SPI 80

Tabla 3. 16 Configuración de dispositivos drivers de tipo USB y HID 81

Tabla 3. 17 Configuración de los drivers de MMC/SD, RTC y soporte para GPIO ... 82

Tabla 3. 18 Configuración de tipos de sistemas de archivos 83

Tabla 3. 19 Sistema de archivos jffs2 84

Tabla 3. 20 Habilitar soporte NFS 84

Tabla 3. 21 Configuración para ptxdist de la arquitectura del proyecto 87

Tabla 3. 22 Opciones de creación imágenes, directorio de archivos y kernel 88

Tabla 3. 23 Herramientas del Shell y la consola 90

Tabla 3. 24 Herramientas de red..... 91

Tabla 3. 25 Utilidades de disco y fichero 92

Tabla 3. 26 Utilidades del bus CAN..... 92

Tabla 3. 27 Aplicaciones 93

Tabla 3. 28 Librerías del sistema 94

Tabla 3. 29 Middleware, librerías gráficas y aplicaciones científicas 94

Tabla 3. 30 Dispositivos operativos en el sistema..... 95

Tabla 3. 31 Configuración del sistema de archivos y aplicaciones disponibles 96

Tabla 3. 32 Cambios en el makefile del driver del dispositivo Wi-Fi 98

1. Introducción

Según recientes datos demográficos la esperanza de vida está aumentando en los últimos tiempos de forma importante. Se estima que en los países industrializados las personas mayores de 65 años tendrán en 2030 una cuota superior al 20% de la población total. Este hecho hace que parte de los desarrollos tecnológicos más avanzados estén encaminados a la mejora de la calidad de vida de estas personas. El desarrollo de sistemas robotizados dotados de un alto nivel de movilidad e inteligencia permitirá alcanzar estos objetivos.

Varios robots de este tipo fueron desarrollados en el pasado, sin embargo, se ha comprobado que tienen grandes limitaciones de uso en entornos domésticos dado que necesitan de grandes y complejos sistemas de control que hace muy difícil su transporte. Además, los robots están anclados permanentemente a una mesa o silla de ruedas limitando, de esta forma, su aplicación en diferentes habitaciones y dependencias domésticas.

Con todos estos datos, se creó el primer prototipo del robot asistencial *ASIBOT*, denominado inicialmente **MATS**, realizado por el grupo *RoboticsLab*, en el departamento de Sistemas y Automática de la Universidad Carlos III de Madrid. El robot es capaz de adaptarse a diferentes entornos de la casa e inclusive desplazarse por la misma. El robot, por ejemplo, puede moverse por las paredes de una habitación, sobre el lavabo, estar anclado a la silla de ruedas y moverse con ella, etc. Para ello, la casa, debe estar equipada con un sencillo sistema de anclajes que sirve tanto para alimentar al robot como para dotarlo de apoyo.

El robot *ASIBOT*, figura 1.1, permite realizar una gran diversidad de tareas domésticas, tales como dar de comer, afeitarse, maquillarse, lavarse los dientes, etc.

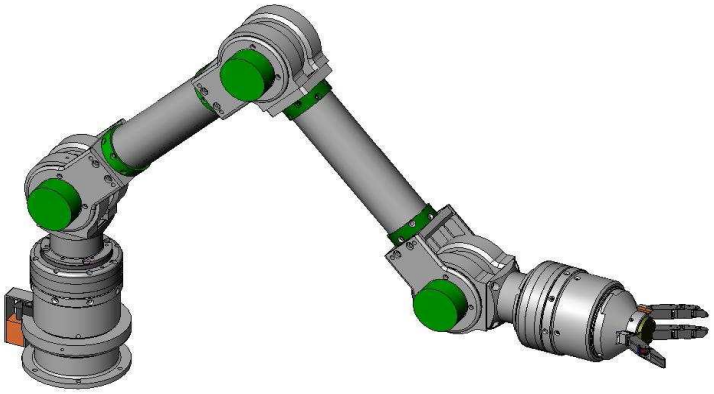


Figura 1. 1 Brazo robótico ASIBOT anclado en una DS

Mediante los anclajes, denominados *Docking Station*, figura 1.2, el robot es capaz de moverse, por ejemplo, de la pared del salón a la silla de ruedas, de la silla a la encimera de la cocina, etc.

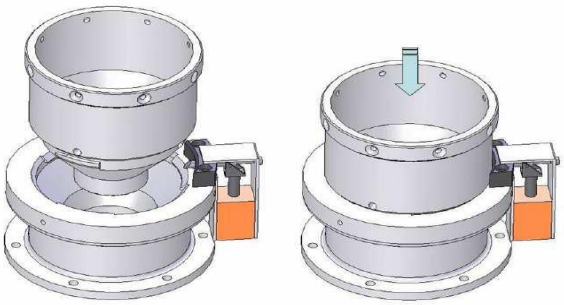


Figura 1. 2 Dispositivo de anclaje, Docking Station, del robot ASIBOT

Se puede describir a *ASIBOT* como un robot asistencial de cinco grados de libertad de estructura simétrica con un mecanismo de anclaje en cada uno de sus extremos a la *Docking Station* con un alcance del orden de 1.3 m. pudiendo transportar un peso de hasta 2 Kg en su extremo. El sistema de control (computadora, electrónica, transmisiones, etc.) se encuentra a bordo del robot y el peso del conjunto no supera los 13.5 Kg. Su reducido peso permite que el robot pueda ser fácilmente transportado por una sola persona, o asistente, a otra ubicación pudiendo ser utilizado en diferentes entornos por diferentes individuos.

La interacción con el robot *ASIBOT* ha sido implementada mediante una comunicación inalámbrica basada en un dispositivo de bajo coste y fácil manejo como es una *PDA*. La comunicación hombre-máquina puede realizarse de diferentes maneras según las diferentes discapacidades del usuario:

- Voz, si la carencia es de movilidad en las manos.
- Manipulación de un joystick, en caso de discapacidades en los brazos.
- Lápiz táctil o mediante el dedo, si sufre una discapacidad en las extremidades inferiores.

El dialogo hombre-robot en el caso de usuarios no técnicos, se efectúa mediante un sencillo sistema de menús orientados a la tarea y basados en iconos gráficos [1], donde es posible realizar desde la manipulación del robot articulación por articulación, hasta la ejecución de tareas predefinidas, programadas con anterioridad .

1.1. Motivaciones

El cerebro del robot utilizado inicialmente estaba formado por un *µPC* denominado *CerfBoard*, embebido en una placa de tan solo 57x69 mm y permitiendo ser alojado en el interior del tubo del brazo del robot *ASIBOT*. Es una placa basada en un microprocesadorde Intel, *StrongARM,* de 32 bits *y* bajo consumo, 2W.

En la figura 1.3 se muestra el dispositivo utilizado junto a una moneda como referencia de su tamaño.

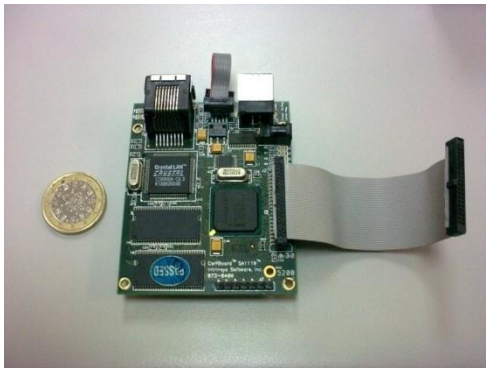


Figura 1. 3 Tarjeta CerfBoard Windows CE 3.0

El microprocesador *StrongARM* es una versión más rápida del diseño del microprocesador *ARM*. Fue fruto de la colaboración en 1995 entre Advanced RISC Machines y *Digital Equipment Corp*.

DEC fabricó varios microprocesadores *StrongARM* basados en el núcleo del procesador *RISC* a 32 bits, que se utilizaban en terminales de redes. A finales del año 1997 Intel compró la tecnología *StrongARM* y poco después anunciaba su intención de utilizar los procesadores *StrongARM* en dispositivos personales como las *PDA's*.

El microprocesador *StrongARM* fue sustituido por el *IntelXScale* de aquí la obsolescencia del sistema implementado en la CerfBoard y uno de los motivos que incitan a la actualización deL *hardware*.

El *µPC* a bordo de ASIBOT se comunica con cada amplificador a través de un puerto serie *RS-232*. Este protocolo define una transmisión *full-duplex* entre dos únicos puntos, por lo que se decidió utilizar un único puerto del *µPC* y compartirlo para todos los amplificadores. La arquitectura, se basaba en la conexión directa del canal de comunicación de la *CerfBoard* a los amplificadores y la *multiplexación* del canal de respuesta de éstos al *µPC* como se muestra en la figura 1.4.

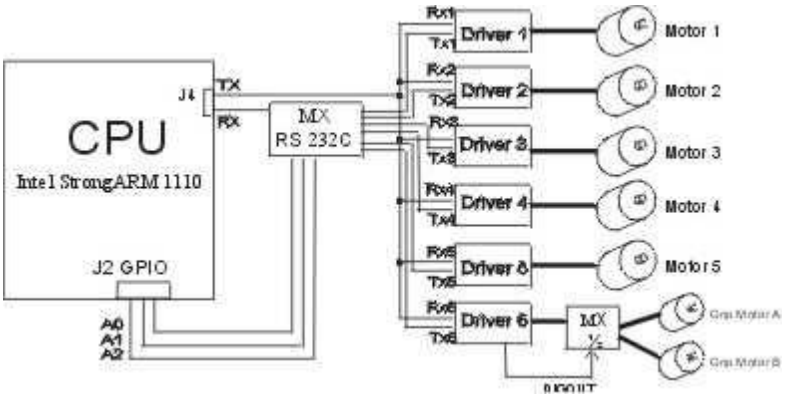


Figura 1. 4 Esquema de conexionado entre CerfBoard y amplificadores

De esta manera se consiguen un esquema y una arquitectura de control a bordo distribuidos, y las referencias tanto de posición como de velocidad se cierran en cada eje.

Sin embargo, este tipo de bus presenta tres inconvenientes importantes:

- El primer inconveniente que presenta esta implementación es la necesidad de realizar la *multiplexación* en la línea de respuesta al microprocesador.

- El segundo problema presente es la velocidad de la comunicación o lo que es lo mismo el tiempo de respuesta. La máxima velocidad permitida por este canal viene limitada por la máxima velocidad de transmisión que soportan los amplificadores que es de 19200 bps. Esta velocidad representa un cuello de botella en la comunicación, disminuyendo la respuesta global del sistema.

- El tercer inconveniente es la incapacidad del sistema operativo del *µPC*, *Windows Embedded CE 3.0*, de control para gestionar el puerto de comunicación serie *RS-232* en tiempo real, por lo que se generan retardos en el procesado de los datos y el protocolo impone serias limitaciones como por ej. la imposibilidad de que los drivers notifiquen eventos de forma asíncrona al *µPC.*

En la actualidad existen sistemas de comunicaciones más potentes y con mayores prestaciones. Al aparecer el mismo modelo de controlador de ejes de idéntico tamaño, y dotado de un sistema de intercambio de datos distinto, se decide cambiar del sistema de comunicaciones interior del robot.

Por lo tanto surge la necesidad de cambiar el sistema a bordo del robot y debe cumplir una serie de requisitos:

- Todo el sistema debe respetar unas ciertas condiciones bastante restrictivas de peso y tamaño debido a las reducidas características del tubo donde se alojan.

- El sistema de comunicaciones internas no debe requerir elementos adicionales que añadan una mayor complejidad.

- El sistema de comunicaciones debe ser un estándar reconocido y con un elevado uso industrial, de esta manera se asegurará la existencia de un elevado abanico de dispositivos diferentes que soporten este tipo de comunicaciones.

- El *µPC* seleccionado deberá cumplir tanto restricciones dimensionales como soportar el sistema de comunicaciones interno.

Siguiendo estos requisitos finalmente se llegó a la elección de los nuevos controladores de ejes porque respetando las mismas especificaciones que los anteriores implementaban el protocolo de comunicaciones bus *CAN*, proporcionando este ciertos beneficios de interés entre los cuales podemos citar:

- Velocidades de transmisión entre 125 Kbps y 1 Mbps, muy superior a la comunicación *RS232*.

- Es un protocolo de comunicaciones normalizado, con lo que se simplifica y economiza la tarea de comunicar subsistemas de diferentes fabricantes sobre una red común o bus.

- Al ser una red con direccionamiento/identificación de los elementos a comunicar, reduce considerablemente el cableado y elimina las conexiones punto a punto, excepto en los terminales.

Por lo tanto, la elección del *µPC* además de cumplir las especificaciones dimensionales debe incorporar al menos un puerto que implemente el bus *CAN*. Cumpliendo estas características se ha llegado a la elección del *µPC* de Phytec, modelo *phyCORE PXA270*, el cual incorpora un microprocesador *PXA270* que es un Intel *Xscale*, el cual posee un puerto que implementa el citado bus y unas dimensiones de 57 x 71.5 mm. En la figura 1.5 se muestra el *µPC* citado junto a una moneda para poder apreciar su tamaño.

Figura 1. 5 phyCORE-PXA270 a bordo de ASIBOT

1.2. Objetivos del proyecto

Los objetivos principales del presente proyecto son:

1. Acondicionar un sistema operativo Linux embebido con capacidades de tiempo real en la tarjeta phyCORE-PXA270.

2. Adaptar el *software* inicial y agregar los módulos *software* necesarios para el aprovechamiento de los nuevos recursos del sistema.

3. Dotar el sistema de una configuración especifica que le confiera autonomía para evitar la necesidad de manipulación del sistema una vez cerrado.

Como se ha comentado anteriormente, el avance de la tecnología nos permitía disponer de nuevos controladores que con unas dimensiones iguales a los anteriormente utilizados disponían de un nuevo estándar, el bus *CAN*, mucho más rápido, fiable y que implementa una gran variedad de comandos a nuestra disposición, características no disponibles en la comunicación *RS232*.

Este mismo avance es el que ha llevado a la obsolescencia del sistema inicial, y ha dado lugar a la necesidad de un cambio de *µPC* utilizando la tarjeta *phyCORE PXA270*. Este modelo de placa soporta dos sistemas operativos, *Windows Embedded CE* 5.0 y Linux.

La opción de utilizar *Windows Embedded* CE 5.0 fue descartada por varias razones:

- Alto coste económico, pues todos los periféricos no estaban disponibles en el BSP (*Board Support Package*) y se requería un *BSP* específico.
- No permitía un acondicionamiento de capacidades en tiempo real.
- Ya se había trabajado anteriormente con un sistema Windows y se quería testear tanto la potencia que podía aportar un sistema Linux como los problemas que podía llevar la utilización del *software* libre.

Por lo tanto se ha optado por un sistema Linux, mencionando la disponibilidad de dos tipos de *BSP's*, uno de ellos proporcionado por la empresa *Viosoft*, con sede en California, el cual no es *OpenSource*, y otro proporcionado por *Pengutronix*, una de las primeras compañías en Alemania que

provee soluciones embebidas basadas en sistemas Linux, cuyo *BSP* si es *OpenSource* e incluye los paquetes necesarios habilitar la funcionalidad requerida en nuestro sistema.

A lo largo del documento se irán describiendo los pasos que se han tenido que llevar a cabo, así como las herramientas necesarias para la implantación del sistema y los cambios u ampliaciones a los que ha dado lugar dicho sistema.

1.3. Estructura del documento

En este primer capítulo se ha realizado una introducción explicando cuales han sido las motivaciones que han dado lugar a la realización del presente proyecto proponiendo para su ejecución una serie de objetivos a cumplimentar en el desarrollo.

En el segundo capítulo se hablará con detalle del estado inicial o estado de partida del robot asistencial *ASIBOT* y su entorno, haciendo una descripción de la arquitectura asistencial y del robot, describiendo tanto su mecánica como su *hardware*, *software* y estructura de procesos.

En el tercer capítulo se describirán los procesos necesarios para configurar, compilar e instalar el sistema operativo en la tarjeta *phyCORE* así como de las herramientas utilizadas para su desarrollo.

En el cuarto capítulo se describirán los mecanismos que hacen posible el desarrollo de la aplicación de control de robot y que ha sido necesario modificar, debido a que la definición de sus funciones y métodos dependen del sistema operativo en el que se ejecutan, para su ejecución en el nuevo sistema desarrollado.

Finalmente se terminará el documento presentando las conclusiones a las que se ha llegado en el desarrollo del proyecto y la propuesta de algunas alternativas interesantes a tener en cuenta para la realización de futuros desarrollos

2. Sistema de partida

El proyecto *ASIBOT*, denominado en el contexto del proyecto europeo *MATS* (*flexible Mechatronic Assistive Technology System to support persons with special needs in all their living and working enviroments*), fue un proyecto financiado por la Unión Europea, bajo el programa *I.S.T.,* coordinado por el profesor *Mike Topping*, de la Universidad de *Staffordshire* (Reino Unido). Para su ejecución se formó un consorcio en el que colaboraron las siguientes entidades:

- *Belgian Centre for Domotics & Immotics*, Bélgica.
- *Centre Docteur Bouffard Vercelli*, Francia.
- *Lunds Universitet*, Suiza.
- *Rehab Robotics Ltd.,* Reino Unido.
- *Scuola Superiore Sant' Anna*, Italia.
- *Helsinki University of Technology*, Finlandia.
- *Innovation Center in Housing for Adapted Movement*, Bélgica.
- *Birmingham Specialist Community Health National Health Service Trust*, Reino Unido.
- Universidad Carlos III de Madrid, España.

El objetivo del proyecto fue concebir, desarrollar, fabricar y evaluar un prototipo de sistema doméstico diseñado para mejorar las condiciones de trabajo y habitabilidad de las personas discapacitadas o ancianas con reducida movilidad. Este objetivo atendía a uno de los principales propósitos de la Comisión Europea en cuanto a construir una sociedad accesible para todo el mundo y sin discriminaciones. El proyecto también aplicaba la nueva tecnología asistencial en entornos domésticos, casas inteligentes (*SMART HOMES*) hospitales y clínicas y ambientes de trabajo.

Otro objetivo del proyecto consistía en buscar no sólo los beneficios sociales, sino también económicos, consiguiendo un producto comercial y no tan sólo un trabajo de laboratorio.

El aspecto clave para el desarrollo del proyecto fue identificar las necesidades y prioridades, mediante la aplicación de métodos de diseño centrados en el usuario, desarrollando soluciones efectivas a los problemas de implementar el concepto del diseño utilizando tecnologías validadas. Evaluando el prototipo del producto en pruebas prácticas en ambientes reales y velando por cumplir las directivas y los estándares europeos de aquellos productos que llegan a ser comercializables.

El sistema podrá reemplazar a los familiares al cuidado del paciente en la realización de variedad de tareas rutinarias y cotidianas, lo que también supone una mejora en la calidad de vida de éstos.

No se puede anticipar que la introducción del sistema suponga una reducción en las oportunidades de empleo para los profesionales del sector asistencial. Muchos servicios de asistencia en toda la comunidad son ya recursos bajo gran demanda y la reducción en las tareas rutinarias permitirá un mejor uso de estos recursos, una especialización de los profesionales y una mejor distribución.

2.1. Descripción de la arquitectura asistencial

El robot *ASIBOT* ha sido diseñado para ser modular y capaz de funcionar tanto en entornos estructurados como no estructurados. Resuelve el problema de la autonomía, común en todos los robots móviles, siendo capaz de navegar por las paredes, los muebles y cruzar puertas moviéndose con precisión y de manera fiable entre habitaciones y entre las distintas plantas de la casa. Tiene la habilidad de transferirse con precisión desde la silla de ruedas o desde los conectores fijos en la pared permitiéndole llevar a cabo cierto número de actividades. Su flexibilidad aporta importantes novedades para el público en general y para el cuidado de discapacitados y mayores con necesidades especiales. Por ejemplo, la modularidad del sistema hace posible el crecimiento y la adaptación del sistema a medida que las condiciones del usuario así lo requieran. Además, la red de conectores del sistema puede ser conectada a una LAN o a internet en un entorno inteligente doméstico, facilitando la monitorización y control, y la descarga de actualizaciones de rutinas de control preprogramadas.

El sistema robotizado permite un significativo avance en el campo de la robótica de servicio y está formado por los siguientes módulos:

- Un entorno adaptado, es decir, una arquitectura diseñada para que encajen robot, conectores de anclaje, unidades de control del sistema, interfaz de usuario, estaciones porta-herramientas, sistemas de comunicaciones y red de alimentación de los conectores.

- Un brazo robótico flexible, capaz de moverse de un conector a otro bajo el sistema de control y capaz de realizar tareas de asistencia y manipular actuadores terminales (*end effector*) y herramientas acopladas a estos últimos, bajo el control del sistema y/o del propio usuario.

- Un vehículo independiente tipo silla de ruedas con movimientos controlados por el usuario. Este vehículo portará el brazo robótico incorporando un conector alojado en un raíl. Portará además el sistema interfaz de control y sistemas de comunicación.

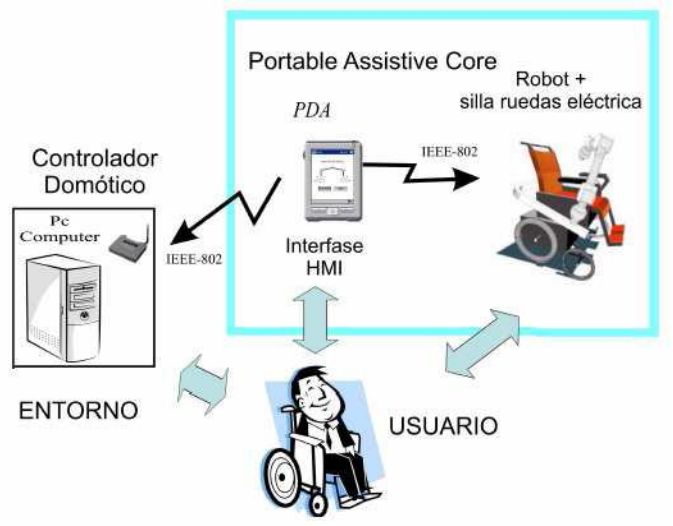


Figura 2. 1 Arquitectura completa del sistema

El sistema incluirá conectores ubicados estratégicamente en el entorno con los necesarios conectores mecánicos (sistema de bayoneta con cerrojo autoblocante) y eléctricos, desde los cuales toma la alimentación el robot mediante un único bus de corriente continua a 24V. Los conectores pueden ser alojados en la pared o bien instalarse sobre un raíl en determinadas localizaciones con el objetivo de prolongar el rango de acción del manipulador. Los conectores permitirán al robot

desplazarse de un lugar de trabajo a otro de forma autónoma siguiendo directivas de movimientos preprogramados según las necesidades del usuario.

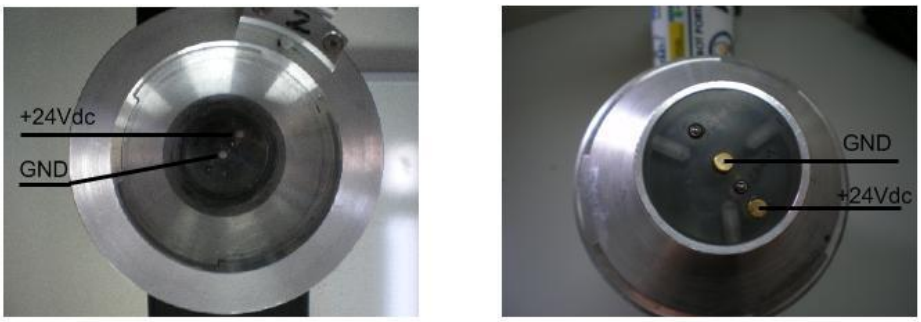
Dadas las características del robot, se establecieron de forma muy concreta que tareas se le exigía que fuese capaz de realizar, asignando prioridades a la hora de implementarlas en función de las directivas del grupo de análisis de especificaciones de usuario, formado por médicos, fisioterapeutas y psicólogos.

El grupo de trabajo formado por arquitectos realizó la adaptación mutua entre el robot y el entorno de trabajo doméstico. Estos grupos generaban las especificaciones funcionales y técnicas que se veían definidas de forma técnica por los equipos de desarrollo de las pinzas, brazo articulado, adaptación de la silla de ruedas comercial para su integración y sistemas de comunicaciones e interfaz de usuario.

2.1.1. Sistema de agarre y conectores

El cometido de la garra o parte extrema del robot (*end effector*) en el proyecto es doble. Primero, ha de servir de pinza o mano capaz de manipular objetos. Segundo, ha de servir como base de fijación para el conjunto del robot cuando se use para anclarse al conector, al raíl o a la silla de ruedas. Además el sistema recibe la alimentación (24v) desde el conector al realizar el anclaje correctamente. Este último punto se consigue mediante un sencillo sistema de confrontación de contactos eléctricos que transmiten la señal que genera la base al robot, figura 2.2. Dado el carácter simétrico y escalador del robot los extremos tienen que realizar alternativamente dos funciones primordiales:

- manipular objetos.
- asegurar el cuerpo del robot al conector de anclaje.



(a) Contactos en Docking -Station (b) Contactos en extremo de ASIBOT

Figura 2. 2 Contactos de Alimentación de Asibot

Estos requisitos funcionales son innovadores y contradictorios entre sí, puesto que obligan a que el extremo del robot trabaje tanto como base como extremo. Lógicamente, lo que en un sitio será bueno en el otro será perjudicial. Por un lado trabajando como base del robot, estructuralmente debe soportar el peso del robot y las fuerzas de reacción del anclaje. Por otro lado, el tamaño y el peso de la pinza deben ser reducidos, y debe ser versátil, es decir, capaz de manipular gran diversidad de objetos.

De cara a poder manipular objetos dispone de tres dedos en el extremo en forma de garras, y en cuanto se necesite anclar el extremo libre, estos dedos se retirarán, apartándose de las superficies donde el extremo realiza el anclaje como puede verse en la figura 2.3.

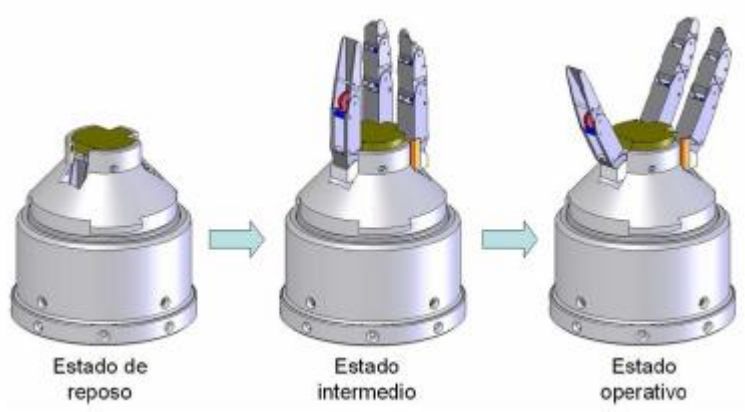


Figura 2. 3 Sistema de anclaje y garra de ASIBOT

Por lo tanto, el extremo es lo suficientemente rígido como para soportar las reacciones del anclaje, pero lo más ligero posible para minimizar éstas y el peso total del robot.

El anclaje posee un perfil cónico en el extremo para poder corregir de forma automática al entrar las desviaciones y errores de posicionamiento tanto del propio extremo del robot como de la instalación de los conectores. Es lo que se conoce como "perfil autocentrante" y está dimensionado de forma que pueda absorber errores de ±2,5mm de radio en posicionamiento y de aproximadamente ± 30' en orientación.

La estación de anclaje o conector exterior al robot (Docking Station) es básicamente un cono hembra réplica del macho que tiene el extremo del robot, figura 2.2 a). La forma cónica de la garra facilita la introducción de esta en el anclaje. Este sistema autocentrante formado por los conos permite garantizar el éxito del anclaje aunque existan errores en el posicionamiento del extremo del robot, del conector exterior o los debidos a la inexactitud en la ortogonalidad de las superficies entre sí. En la parte final del proceso de anclaje y como en todo mecanismo de bayoneta se ha de girar la garra según el eje perpendicular al conector exterior, lo cual se realiza mediante el giro de los ejes más extremos (el uno o el cinco según el punto de anclaje). Este giro provoca el anclaje correcto al llegar a la posición final, saltando un cerrojo de forma automática. Este sistema de autoblocaje impide que el robot se descuelgue de forma accidental debido al propio movimiento del brazo. Para liberar el extremo anclado, y tras comprobar, que el anclaje es total en el otro extremo, se activa un solenoide situado en la base para correr el cerrojo y permitir así el giro de la muñeca del robot y la posterior salida de la garra del conector.

2.1.2. Silla de ruedas

Se dispone de una silla de ruedas eléctrica preparada para poder ser usada mediante un *joystick*. Pensado de manera que se pueda implementar un interfaz basado en PDA para manejar el movimiento del robot y de la silla indistintamente.

A la estructura mecánica de la silla se le acoplará un raíl de lado a lado que pasando por detrás del asiento del ocupante, permita el paso del robot acoplado a un conector, figura 2.4. De esta forma es posible:

- Acoplar el brazo a la silla de ruedas, probando tareas tales como acercar un objeto desde el suelo o desde un lugar elevado hasta el usuario.

- Ampliar enormemente el campo de trabajo del robot sin necesidad de que el usuario desplace su silla de ruedas.

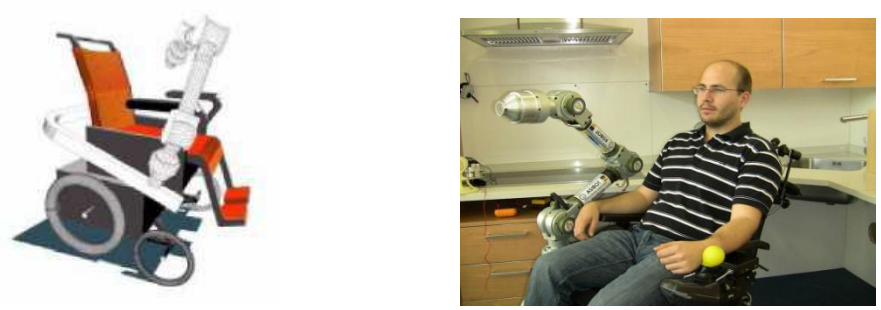


Figura 2. 4 Silla de ruedas con anclaje DS en raíl

Se necesita localizar de forma precisa la silla de ruedas con respecto al conector fijo, para que el robot pueda realizar la transferencia desde la silla de ruedas a un conector localizado en el entorno, por ejemplo en una encimera. En principio la posición de la silla debe ser fijada con precisión y la transferencia solo se puede realizar en determinadas posiciones pero es parte de un trabajo futuro que el sistema tenga la suficiente autonomía para adaptar la trayectoria y realizar así el anclaje por sí solo.

2.2. Descripción del robot asistencial ASIBOT

Si atendemos a la definición de robot escalador que propone el profesor Antonio Giménez [2] de la Universidad Carlos III de Madrid: "Un robot escalador es una máquina móvil que puede realizar tareas de servicio, de forma programada o autónoma, capaz de soportar su peso y propulsarse a sí mismo en su entorno de trabajo, que no tiene por qué ser horizontal. La máquina debe ser capaz de conseguir un alto nivel de sujeción, para que no se caiga ni deslice, en el caso de que se mueva sobre una superficie muy inclinada" y definiendo a los robots asistenciales como un tipo de robots de servicios especializados en asistir a personas discapacitadas o de avanzada edad podemos decir que el robot *ASIBOT* aúna diferentes características de distintas filosofías de diseño

[3] y, por lo tanto se le puede clasificar como un manipulador-escalador autoportado (debido a su reducido peso y tamaño).

Además puede montarse sobre una silla de ruedas cuando las circunstancias lo requieran, convirtiéndose así en un manipulador móvil; o que los dispositivos de agarre son externos al robot, los cuales además solucionan el problema que supone la alimentación en la mayoría de los robot escaladores debido al cableado que limita la distancia a la que se puede desplazar o por el peso y tamaño que imponen las baterías requeridas.

2.2.1. Estructura mecánica

El robot *ASIBOT* es simétrico y consta de cinco grados de libertad o ejes como puede verse en la siguiente figura:

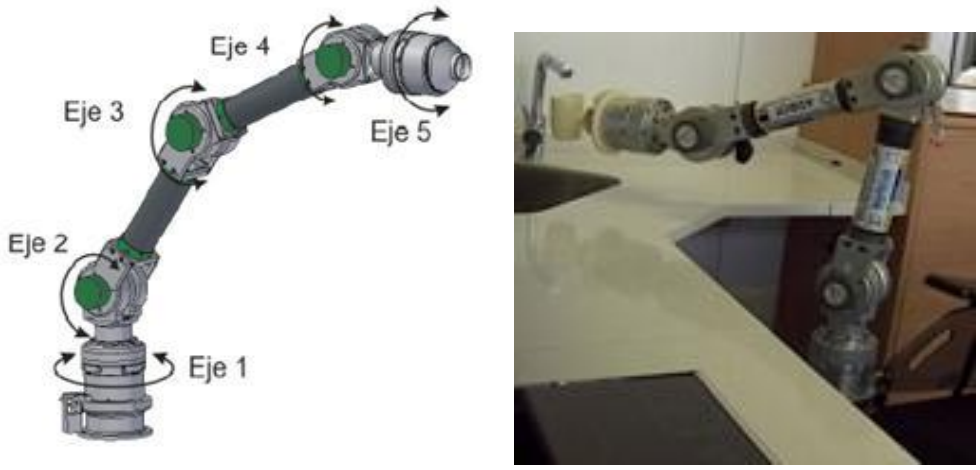


Figura 2. 5 Distribución de articulaciones en ASIBOT y ASIBOT llenando un vaso

Los dos eslabones centrales, a los cuales se les suelen llamar "brazos", alojan en su interior el principal equipamiento electrónico y la unidad de control del robot. Los eslabones más pequeños, las "muñecas", terminan en un mecanismo que le permite anclarse a los dispositivos de agarre. Estos elementos terminales poseen además una pinza con "dedos" que permiten coger objetos. Es importante señalar que los dispositivos de anclaje, o "*Docking Stations*", proporcionan la

alimentación que el robot necesita, por lo que el robot no tiene que cargar con ningún dispositivo de almacenamiento o generación de energía. La ausencia de fuente de energía, generalmente muy pesadas, unido a las materias primas empleadas, aluminio y fibra de carbono, le confiere al robot un peso aproximado de 13.5 Kg.

Con ello, se pueden dividir las partes principales de la estructura del robot de la siguiente forma:

1. Estructuras mecánicas realizadas en aluminio aeronáutico (7075) de alta resistencia similar a la de aceros pero con peso específico muy inferior, donde van alojados los grupos motrices: formados por reductor, el motor, el freno, el sensor de posición y los correspondientes rodamientos del eje motor de cada una de las 5 articulaciones. Estas estructuras se unen en grupos de 2 en las articulaciones 1 y 2, así como en las articulaciones 4 y 5. La estructura mecánica que engloba los componentes del eje restante, figura 2.6., se sitúa en la parte central del robot, por donde pasaría el supuesto eje de simetría, correspondiente a la articulación 3.

Figura 2. 6 Estructura mecánica del eje central

Figura 2. 7 Tubo de fibra de carbono

2. Dos tubos de fibra de carbono, ver figura 2.7., que unen las estructuras anteriormente mencionadas, de material compuesto: fibra de carbono sobre base de resina epoxi, de 2 mm de espesor. La selección del diámetro del tubo y su longitud se eligieron teniendo en cuenta la rigidez necesaria en los eslabones y permitiendo suficiente espacio en su interior para poder alojar los sistemas electrónicos, el procesador a bordo y el sistema de evacuación de calor de la electrónica de potencia.

Se denomina eslabón a cada uno de los dos tubos que son de diferente longitud y un diámetro igual en ambos casos de 60 mm. La diferencia de longitud en los tubos es necesaria para conseguir la simetría en la distribución de las articulaciones ya que la estructura de la articulación 3 no es del todo simétrica [4]. Por ello se han tomado los términos siguientes para referirse a los tubos:

- Tubo largo: Eslabón que une las articulaciones 2 y 3.
- Tubo corto: Eslabón que une las articulaciones 3 y 4.

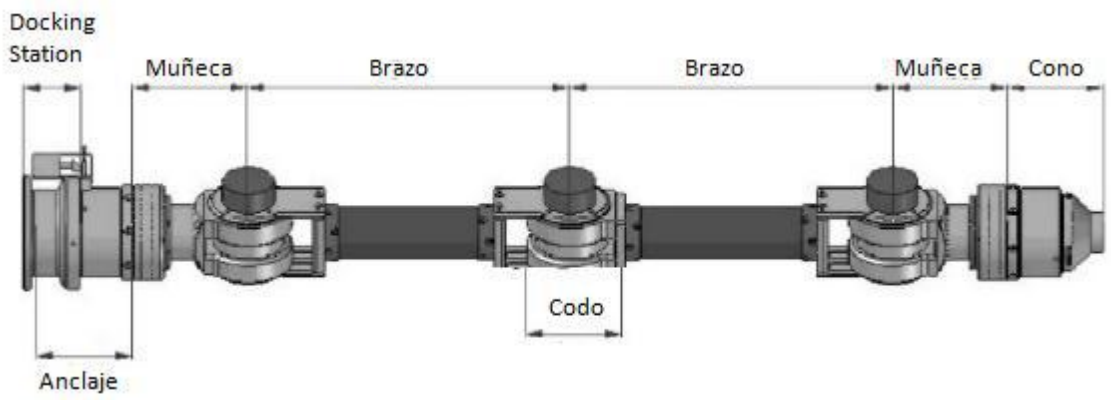


Figura 2. 8 Distribución de elementos en la estructura

En la figura 2.8 se puede ver la división de los elementos de la estructura, que se han mencionado en este apartado.

2.2.2. Estructura Hardware

El robot *ASIBOT* es el desarrollo de un prototipo que integra a bordo todos los sistemas necesarios para su operación, control y comunicaciones, lo cual es algo innovador por no necesitar de un cordón umbilical, algo común en los prototipos de robots escaladores. Debido al empleo de tecnología de miniaturización y sistemas embebidos lo convierte en autoportado.

El sistema de control a bordo está formado por tres subsistemas:

- Ordenador a bordo,
- Actuadores,
- Comunicaciones

Ordenador a bordo

El cerebro del robot está formado por un *µPC* denominado *CerfBoard®* que se asienta sobre una placa de tan solo 57x69 mm, lo cual le permite ser alojado fácilmente en el interior del brazo.

La *CerfBoard* es una placa basada en un microprocesador *StrongARM* de bajo consumo, muy utilizado en tecnologías *hand-held*. Se alimenta a cinco voltios, desde un conversor a bordo del robot.

Esta tarjeta, cuyo sistema operativo es *Windows CE*, realiza principalmente las siguientes tareas: gestión de las comunicaciones externas con la interfaz de usuario, mediante red inalámbrica, interprete de comandos, planificación del movimiento, transformaciones cinemáticas, sincronización de los ejes y comunicación con los controladores de cada eje.

Actuadores

Las tarjetas amplificadoras utilizadas también se alojan en el interior de los brazos. Estas tarjetas han sido extraídas de su carcasa original, como se puede ver en la figura 2.9, para reducirlas a unas dimensiones de 65 x 58 y un peso de tan solo 10 g.

Figura 2. 9 Driver original y sin carcasa

Los actuadores que posee *ASIBOT* consisten unos motores de corriente continua con escobillas, controlados por inducción y con imanes permanentes en el estator. El modelo es el QT-1917ª de *Kollmorgen* y ofrecen una excelente relación par/peso, figura 2.10.

Otra característica de este motor es la ausencia de carcasa exterior, lo cual permite que sean integrados en el diseño de la articulación, ahorrando peso y espacio, algo fundamental debido a las características del robot.

Figura 2. 10 Torque Motor

Como reductores, necesarios casi siempre que se emplean motores eléctricos como actuadores, se emplean *Harmonic-Drives*, ver figura 2.11. Estos dispositivos proporcionan una elevada reducción, un peso y un volumen muy reducido y una excelente transmisión del par. Estos

reductores son muy populares en el campo de la robótica por tener la mejor relación par/peso del mercado, por su elevada precisión y buen rendimiento.

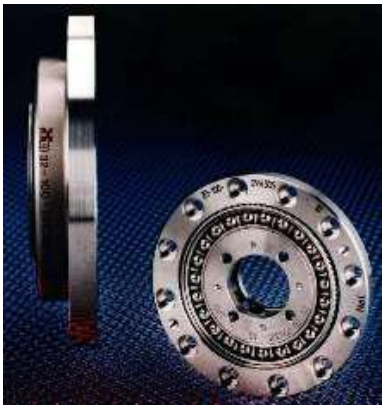


Figura 2. 11 Harmonic Drive

Los actuadores diseñados para el robot *ASIBOT*, también incluyen un freno electromecánico que libera el eje al aplicarle tensión. De este modo, los frenos siempre bloquean el motor salvo cuando se quiera mover un determinado eje. Esto es así por dos motivos: el primero, por seguridad, ya que ante un fallo en la alimentación, el robot tendrá todos los frenos bloqueados y el robot mantendrá la posición evitando así caídas o movimientos descontrolados, por lo que se denominan frenos antifallo; la segunda razón es para evitar que el robot tenga un consumo constante elevado de corriente y que así los motores sólo consuman cuando se vayan a mover.

Por último, el actuador emplea un encoder para cerrar el lazo de control, ya que este dispositivo óptico permite medir posiciones angulares relativas y velocidades del eje. El encoder que se utiliza es el modelo RCML 15 de Renco, figura 2.12, es el más plano del mercado, muy ligero y proporciona una resolución de 1024 ppv.

Figura 2. 12 Encoder relativo modelo RCML de Renco

En este apartado se ha descrito la estructura hardware del robot. En el siguiente apartado se resume brevemente como ésta electrónica procesa los comandos recibidos por la PDA para convertirlos en instrucciones de bajo nivel y de esta manera comunicar dichas instrucciones a los actuadores.

Comunicaciones

La comunicación del robot con el ordenador base se realiza a través de una tarjeta *Compact Flash*, que proporciona acceso mediante protocolo Ethernet radio *IEEE 802-11b*. Gracias a esto la única conexión cableada necesaria del robot es la de alimentación. Esta tarjeta permite a la *CerfBoard* recibir los comandos del usuario y enviar datos de estado al HMI de forma inalámbrica, permitiendo disponer de aun más espacio libre en el entorno del usuario liberándolo del cable de comunicaciones.

El ordenador base con el que se comunica el robot, que es un dispositivo portátil de tipo PDA (Personal Digital Assistant o Pocket PC), tiene como principal objetivo definir el entorno por el que opera el robot y actuar de interfaz entre éste y el usuario. Para ello se definió un conjunto de comandos, un lenguaje especialmente desarrollado (MRL) del que se hablará más adelante, que traducen los deseos del usuario en movimientos concretos, en el caso en el que se desee una teleoperación, y ejecutar tareas preprogramadas o automáticas.

La Cerfboard realiza las diferentes operaciones internas con cada amplificador a través de un puerto serie RS-232. Este protocolo se define para una comunicación full-duplex entre dos únicos puntos, para la que se usa un único puerto del *µPC* y es compartido para todos los amplificadores. La arquitectura definida se basa en la conexión directa del canal de comunicación de la *CerfBoard* a los amplificadores y la multiplexación del canal de respuesta de éstos al *µPC*. Este esquema simplifica notablemente tanto la complejidad del sistema como el cableado interno. Este esquema de conexión de los accionadores con el *µPC* puede visualizarse en la figura 1.4 del capítulo 1.

Para ello se diseñaron unas tarjetas electrónicas para la multiplexación del canal de transmisión y para conectar el motor de las dos pinzas de los extremos a un único amplificador. Esto es posible porque en todo momento hay una pinza que no puede ser utilizada ya que se encuentra en el extremo anclado a una *Docking Station*.

2.2.3. Estructura Software

Todo el sistema software está distribuido entre dos computadoras diferentes:

- La unidad de control del robot.
- La PDA que el usuario emplea como HMI.

Entre estos dos bloques la comunicación es inalámbrica, mediante protocolo *IEEE 802.11b*, para reducir el número de cables y especialmente, no necesitar un "cordón umbilical" en el robot. En la figura 2.13 se muestra un esquema de la arquitectura *software* [5].

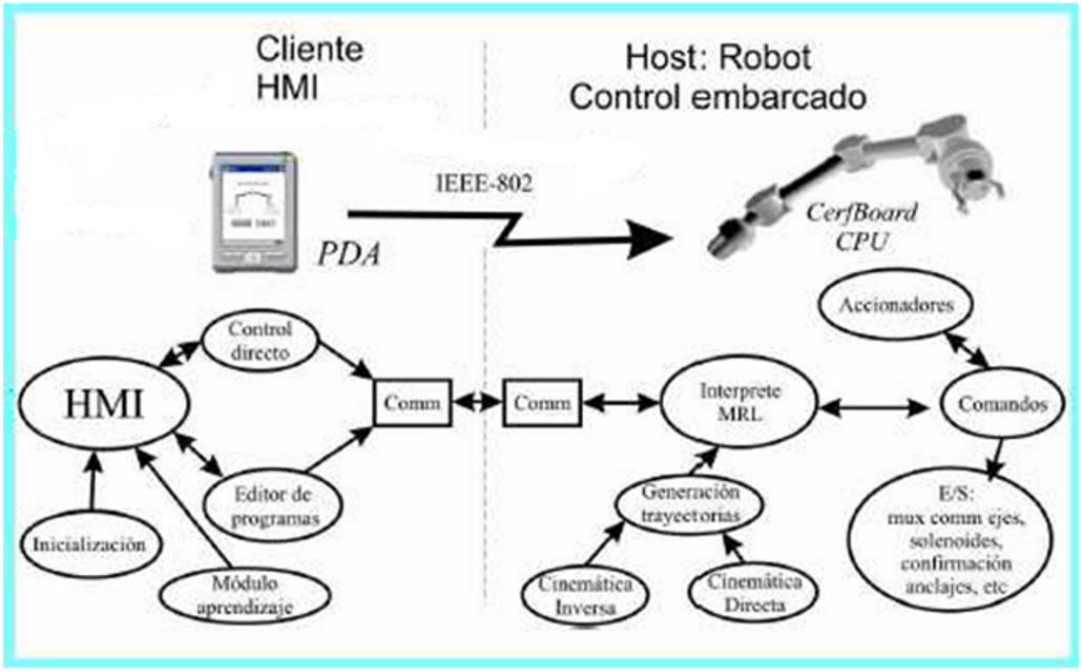


Figura 2. 13 Esquema de la arquitectura software del sistema

HMI/PDA

El interfaz hombre-máquina (*HMI*), instalado en una *PDA*, desempeña un papel fundamental en el sistema. A través de él, el usuario puede controlar el robot y estar informado de su estado o de la tarea que realiza. También dispone de un módulo de reconocimiento de voz para que personas con movilidad manual reducida puedan beneficiarse del robot mediante comandos orales.

El robot es un terminal "tonto" del sistema ya que no dispone de sensores ni de inteligencia suficiente como para tomar decisiones sobre las tareas, ni conoce el entorno ni lo que hace en cada momento. En este sistema es el usuario y el *HMI* los que deben saber en cada momento lo que hace el robot.

El interfaz de la *PDA* presenta dos modos de movimiento diferentes: movimientos preprogramados y movimientos teleoperados. Durante los primeros, los objetos que se manipulen deben encontrarse en posiciones fijas, ya que son tareas programadas mediante guiado y el robot realizará siempre los mismos movimientos e irá a las mismas posiciones; en el segundo tipo de movimiento, éste está directamente controlado por el usuario, con lo que se puede trabajar en todo el espacio de trabajo tanto con movimientos articulares como cartesianos. En cualquiera de ambos casos se dispone de medios con alta prioridad que detienen cualquier acción en curso de manera inmediata.

La forma de interactuar con el *HMI* se realiza a través de dos medios diferentes: mediante una pantalla gráfica táctil, donde se pueden seleccionar diferentes botones e introducir valores; y mediante órdenes vocales.

También existe la posibilidad de manejar el sistema mediante un *joystick* como el mostrado en la figura 2.14, que se conecta a la PDA mediante *bluetooth*. Este *joystick* permite la posibilidad de control del robot mediante los dos modos anteriormente citados.

Figura 2. 14 Interfaz de joystick

La *PDA* tiene instalado como sistema operativo el *MS Windows CE 4.0*. El dispositivo dispone de dos puertos para la comunicación con otros dispositivos: un puerto inalámbrico que implementa el protocolo *802.11b* anteriormente citado y un puerto *bluetooth*.

Software del robot

La unidad de control del robot dispone de: tres puertos serie, de los cuales uno es empleado para la comunicación con los amplificadores de los motores; un puerto *USB*, no utilizado; un conector *Ethernet* y una ranura *compact flash*, empleada por la tarjeta de red inalámbrica. El sistema operativo sobre el que trabaja es el *MS Windows CE 3.0*.

Cuando el robot se inicia, el software comienza un proceso para entablar conexión con un *HMI*; este arranque consiste en iniciar todos los sistemas y quedar a la espera indefinidamente de posibles conexiones de clientes. Por su parte, cuando el programa de la *PDA* se inicia, trata de conectarse con el robot, de tal modo que cuando ambos conectan, comienzan a interactuar siguiendo un protocolo diseñado para la aplicación. A este tipo de comunicación se le denomina cliente-servidor, donde el robot, que está a la espera de conexión, cumple la función de servidor y la *PDA*, que realiza la primera conexión, la de cliente.

La comunicación entre el *HMI* y el robot se establece mediante un lenguaje y un protocolo propio, llamado *MRL* (*MATS Robot Language*), el cual se comentará más adelante. La unidad de control también establece una comunicación directa con los amplificadores por otros motivos, como operaciones de diagnóstico, sincronización, calibración, cambio de parámetros de control y gestión de las entradas y salidas digitales, entre otras.

2.2.4. Estructura de procesos y comunicaciones

Comunicaciones entre elementos del sistema

Los elementos del sistema se encuentran en permanente comunicación e intercambio de información. Se comunican entre sí mediante el mencionado lenguaje *MRL*, el cual consiste en una serie de comandos de diferentes grupos funcionales con parámetros, en texto *ASCII*, para ser transmitidos a través de la red sin problemas de compatibilidades entre formatos. Los principales comandos de este desarrollo son los que se refieren a movimientos que debe realizar el robot; en ellos se especifica el tipo de movimiento, articular, cartesiano, continuo y si es necesario, el punto final al que debe dirigirse el robot. También existen instrucciones para controlar la apertura o cierre de las pinzas en el extremo libre del robot. Por último, para la seguridad en la comunicación se han dispuesto una serie de comandos de eco, de validación de la recepción, etc.

El protocolo de comunicaciones que utiliza es el protocolo *TCP/IP*, ya que este está implementado en todos los sistemas operativos y está universalmente aceptado como estándar de comunicaciones facilitando la compatibilidad entre las diferentes máquinas con distintos sistemas operativos y arquitecturas y, ofreciendo fiabilidad en la transmisión pues el propio protocolo verifica el correcto envío y su recepción. Y más concretamente a bajo nivel se utiliza una tecnología muy probada y afianzada como son: los *sockets*, pues son compatibles con las tecnologías embebidas.

El cliente utiliza el interfaz de usuario para comunicarse con el robot, tanto para enviar órdenes a éste, como para visualizar las respuestas del mismo y el estado del sistema. Dichas órdenes se transforman en los comandos correspondientes al lenguaje que el robot y la interfaz de usuario tienen en común, y son enviados a través de enlace inalámbrico.

El proceso de comunicaciones en el robot consiste en, una vez que ha arrancado su sistema, permanecer a la espera de peticiones de conexión por parte de posibles usuarios. Una vez que se ha realizado la conexión, este queda a la escucha de peticiones en forma de comandos.

En el momento que recibe un comando, se pasa al módulo de análisis, donde se comprueba si este es correcto, tanto su sintaxis como su semántica. Si es validado, es enviado al módulo de ejecución para efectuar su correspondiente tratamiento. Cada comando implica un tratamiento personalizado y dependiente de la semántica del mismo y el estado del sistema.

Algunos implican un cálculo de trayectoria, por lo que necesitan llamar al módulo de cinemática para realizar los cálculos correspondientes y después al planificador de trayectorias, para calcular la trayectoria que ha de seguir el robot y realizar la interpolación de los puntos del espacio por los que este ha de pasar para describir el movimiento deseado. Una vez calculados, serán enviados a los motores para su ejecución y, una vez finalizado se informa mediante mensaje de vuelta al usuario sobre la correcta ejecución del comando o la imposibilidad de realizar dicha acción.

En esta interfaz, el usuario tiene a su disposición un módulo de aprendizaje. Con él se puede enseñar al robot nuevos movimientos y tareas, que serán almacenados para su posterior reproducción, es decir, el usuario puede crear sus propias tareas y, con el tiempo, tener su biblioteca de acciones adaptadas a sus necesidades concretas, de tal forma que el robot se personaliza con el uso.

Canales y flujos de comandos

El intercambio de información se realiza mediante dos canales de comunicaciones entre el interfaz y el robot, llamados canal de comandos y canal de mensajes, como se muestra en la figura 2.15.

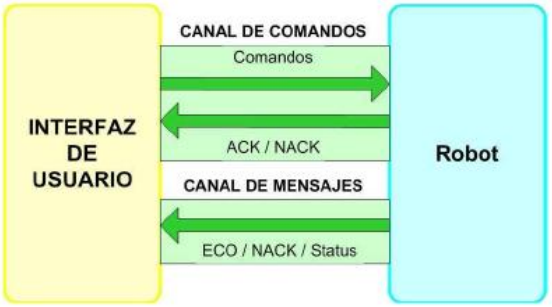


Figura 2. 15 Canales de comunicación entre HMI y robot

El canal de comandos es constantemente monitorizado por el robot, pues es por éste canal por el que recibe las órdenes o comandos a ejecutar. Si el comando es correcto se envía un comando *ACK* por dicho canal, indicando que entiende y reconoce el comando recibido, y este comando se ejecuta si el robot no está atendiendo otro comando o se queda en cola esperando a su ejecución. Si el comando no es correcto se envía un comando *NACK* advirtiendo del error al interfaz.

El canal de mensajes comunica la información de estatus y de ejecución de comandos. Al ejecutar el comando correctamente el canal envía un *ECO* del comando recibido para que el *HMI* sea consciente de que el comando se ha ejecutado. Si el comando no se puede ejecutar correctamente se envía un mensaje de tipo *NACK*. Si el usuario requiere información del sistema, es decir, de estatus, es por este canal por el que se le manda la información solicitada, figura 2.16.

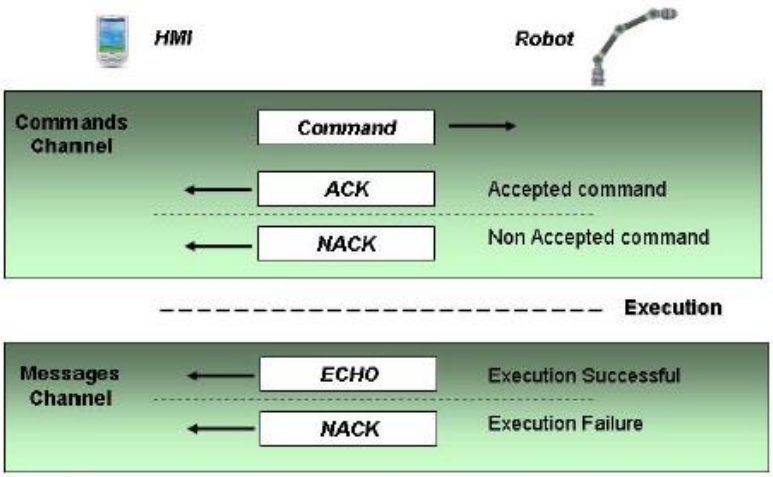


Figura 2. 16 Flujo de información por los diferentes canales

Se puede apreciar que para que se dé el correcto desarrollo de las funcionalidades desarrolladas es necesario que en determinados momentos se ejecuten acciones paralelamente.

Estructura de procesos en el robot

En la figura 2.17 se muestra un esquema del proceso que se realiza en el interior del robot, indicando los diferentes hilos y procesos paralelos que se dan.

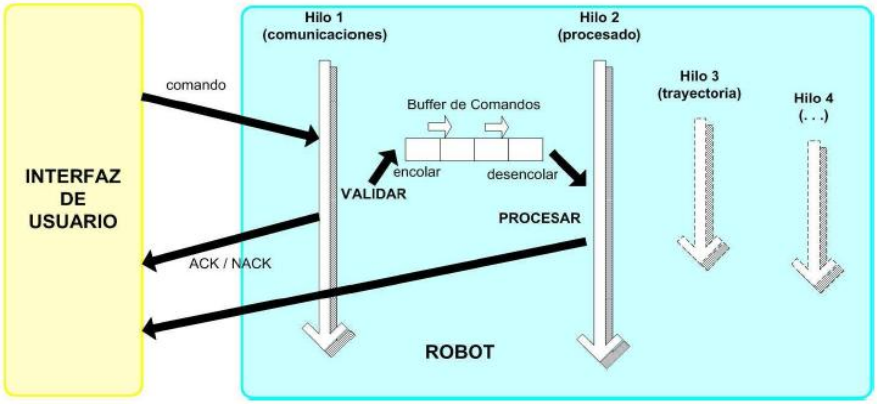


Figura 2. 17 Esquema de procesos en el servidor

En la estructura de procesos en el interior del robot existen dos hilos de ejecución paralelos que están permanentemente activos.

El hilo 1, comunicaciones, se encarga de escuchar permanentemente en espera de recepción de comandos, comprueba si es correcto y envía el comando *ACK* o *NACK* dependiendo de si es correcto o no. Si el comando es correcto se comprueba si la cola no está llena, de otro modo se rechazaría, introduciendo el comando en ella.

El hilo 2, procesado, se encarga de extraer continuamente comandos del buffer y procesarlos. Si no hay comandos que procesar el hilo se queda en estado bloqueado a la espera de comandos para no consumir recursos.

El buffer de comandos es un espacio compartido por los dos hilos citados, por lo que posee mecanismos de secciones críticas para que ambos puedan interactuar y se organizan de forma *FIFO* (*First In First Out*). Todos los comandos excepto el comando *STOP*, que es tratado con la máxima prioridad, son conducidos y extraídos del buffer.

Eventualmente se ejecuta algún proceso paralelo más de carácter temporal para realizar alguna acción concreta, como puede ser el módulo de cinemática para realizar movimientos o calculo de trayectorias.

Estructura de procesos en los interfaces del usuario

Concurrentemente en todo momento hay tres tareas en ejecución, como se puede ver en la figura 2.18. Una de ellas, la tarea principal que inicia las otras dos, se encarga de mantener la interfaz gráfica siempre operativa para atender las peticiones del usuario. Dicha tarea recoge la interacción de este, compone los comandos necesarios para realizar la acción deseada y los introduce en el buffer terminando así su actuación y permitiéndole al usuario el uso nuevamente del interfaz gráfico.

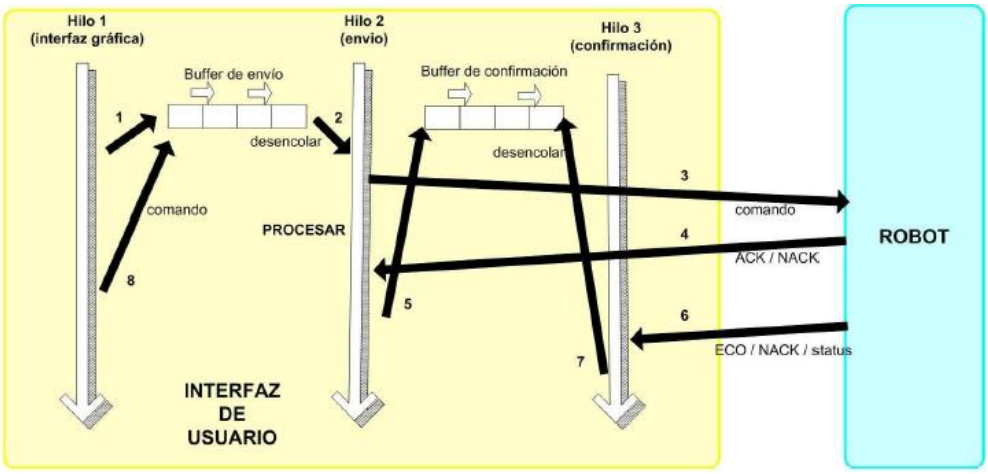


Figura 2. 18 Esquema de procesos en la interfaz gráfica de usuario

Una segunda tarea, envío, monitoriza constantemente el buffer de envío y cuando detecta algún comando lo extrae y lo envía y queda bloqueado a espera de recibir el rechazo o la aceptación por parte del robot. En caso de rechazo del comando, lo vuelve a reenviar por si el buffer del robot estuviese lleno. En caso de aceptación se copia el comando al buffer de confirmación y se repite el procedimiento extrayendo un nuevo comando del buffer de pendientes de enviar o si el buffer esta vacio el hilo queda a la espera.

Por último, un tercer hilo, buffer de confirmación, permanece a la escucha del canal de comunicaciones por el que llegan las confirmaciones de ejecución y los mensajes de información. Cada vez que llega una confirmación o rechazo por este canal, a tarea extrae el mensaje correspondiente de la cola y comprueba si el comando recibido, un eco, coincide con el comando esperado. En tal caso se asume que la ejecución fue correcta y se elimina de la lista, en caso contrario se informará al usuario. Si lo que se recibe es información del estado del robot (comando *status*), se actualizará para mostrarla en la pantalla correspondiente cuando el usuario así lo requiera.

3. Acondicionamiento software del PXA270

El asistente robótico hasta este punto ha sido controlado, como ya se ha comentado, por un *µPC* denominado *CerfBoard*. Este micro además de estar obsoleto únicamente proporcionaba un protocolo *RS-232* adecuado para el control de motores. Los amplificadores delimitaban la velocidad a la que se podía transmitir, produciendo un cuello de botella y además se requería de una multiplexación para gestionar las respuestas de cada una de las articulaciones.

Esta implementación del control de motores mediante multiplexación del puerto serie crea la incapacidad al sistema operativo del *µPC* para gestionar el puerto de comunicaciones serie RS-*232* en tiempo real, por lo que se generan retardos en el procesado de los datos y no se puede asegurar un tiempo de respuesta fijo.

El cambio de la tarjeta microprocesadora además de implicar una actualización tecnológica importante a nivel *hardware* y *software*, proporciona una mejora significativa al implementar un nuevo estándar de comunicaciones, el protocolo de bus *CAN* (*Control Area Network*).

El bus *CAN* [11] se basa en un modelo que describe una relación entre un productor y uno o más consumidores. Es un protocolo orientado a mensajes, es decir la información enviada se descompone en mensajes, a los cuales se les asigna un identificador y se encapsulan en tramas para su transmisión. Cada mensaje tiene un identificador único dentro de la red, con el cual los nodos deciden aceptar o no dicho mensaje. Dentro de sus principales características se encuentran:

- Prioridad de mensajes.
- Garantía de tiempos de latencia.
- Flexibilidad en la configuración.
- Recepción por multidifusión con sincronización de tiempos.
- Sistema robusto en cuanto a consistencia de datos.
- Sistema multimaestro.

- Detección y señalización de errores.
- Retransmisión automática de tramas erróneas.
- Distinción entre errores temporales y fallas permanentes de los nodos de la red, y desconexión autónoma de nodos defectuosos.

Utilizando este protocolo eliminamos las deficiencias producidas por la comunicación implementada anteriormente. Gracias a la implantación del bus *CAN* se han abierto multitud de nuevas oportunidades. Pudiéndose diseñar otros módulos de sensorización basados en micros que implementen este protocolo y así mediante únicamente dos hilos que recorren la longitud del robot se pueden ir conectando los dispositivos sensoriales en la red CAN. Un ejemplo de esta sensorización consiste en un módulo de encoders absolutos diseñados para conocer el movimiento absoluto de las articulaciones del robot desde el encendido del sistema.

El sistema operativo utilizado por la placa *CerfBoard, como ya se ha comentado,* era Windows *Embedded CE 3.0*, el cuál cumplía con los requisitos de modularidad y tamaño necesarios para el entorno, siendo construido a medida, agregando solamente los módulos necesarios para el soporte de la aplicación y dejando fuera todo aquello que no fuese necesario. Una gran característica de este sistema es que *Windows CE* soporta el modelo de programación *Win32*, sin embargo sólo soporta un subconjunto de todo el mundo *Win32*, por lo tanto las aplicaciones no son totalmente compatibles entre *Windows* y *Windows CE*.

El sistema que se va a instalar en la tarjeta *phyCORE PXA270* es un sistema Linux aunque por razones obvias (disposición de 256Kb SRAM, 64Mb SDRAM, y 32Mb NOR Flash) se ha de construir una versión reducida en tamaño denominada empotrada, en inglés *Embedded Linux [12]*.

Una gran ventaja respecto de *Windows CE*, es que las aplicaciones programadas en los sistemas *Linux* se pueden utilizar indistintamente en un *PC* de sobremesa o en la tarjeta que vamos a acondicionar, siempre y cuando no se utilicen librerías en la programación no soportadas por la arquitectura, aclarando como contrapartida, que las aplicaciones en Linux, generalmente deben ser compiladas en sus plataformas específicas para su correcta ejecución.

Los sistemas *Linux [13]*, a diferencia de *Windows*, permiten una gran accesibilidad a todos los ficheros, periféricos, y al *hardware* que conforma el ordenador lo que facilita la interacción con todos sus dispositivos [14]. Es decir, cualquier tipo de fichero puede ser modificado en un sistema *Linux* pudiendo llegar a tener grandes problemas por configuraciones erróneas.

Otra característica de estos sistemas es que mayoritariamente son *OpenSource* o código abierto, es el término con el que se conoce al software distribuido y desarrollado libremente. Tiene un punto de vista orientado a los beneficios prácticos de compartir el código. La idea es sencilla, cuando los programadores (en internet) pueden leer, modificar y redistribuir el código fuente de un programa, éste evoluciona, se desarrolla y mejora. Los usuarios lo adaptan a sus necesidades, corrigen sus errores a una velocidad impresionante, mayor a la aplicada en el desarrollo de software convencional o cerrado, dando como resultado la producción de un mejor *software*. Como contrapartida a ello existe el inconveniente de la poca dedicación a la documentación del trabajo realizado.

Este sistema operativo también permite ser configurado y acondicionado con capacidades de tiempo real [15]. Un sistema de tiempo real debe producir una salida, como respuesta a una entrada, en un tiempo específico.

La mayoría de los sistemas operativos son diseñados para tener buen rendimiento de procesamiento y buen tiempo de respuesta. La gran mayoría tienen un planificador de trabajos equitativo para seleccionar el siguiente proceso a ejecutarse. Sin embargo, un *SO* de tiempo real debe ser capaz de planificar procesos para cumplir los plazos requeridos por las aplicaciones. Esto implica un planificador de trabajos el cual puede que no sea equitativo pero si correcto para seleccionar el próximo proceso a ejecutar en función de las prioridades de dichos procesos.

Se dice que el sistema permite ser acondicionado con capacidades de tiempo real porque implementa, utiliza y pone a disposición del programador las funciones y mecanismos para desarrollar aplicaciones que puedan cumplir ciertos requisitos de tiempo real o mejor dicho permitir los mínimos tiempos de latencia posible.

La comunicación con el exterior se realizaba mediante una tarjeta *Compact Flash Wireless*, siendo sustituido este modelo por un dispositivo *USB Wi-Fi*.

A lo largo del capítulo se describirá como se ha conseguido realizar la instalación del sistema, que herramientas *software* y *hardware* han sido necesarias para la puesta en marcha de todos los dispositivos necesarios.

3.1. Instalación del bootloader

En cualquier *PC* ya sea de sobremesa, portátil, empotrado, industrial, etc.. existe un primer programa que se inicia al activar el interruptor de encendido de la máquina, este programa se denomina *bootloader*, se puede describir como un programa sencillo que gestiona los recursos mínimos para arrancar la máquina y es diseñado exclusivamente para preparar todo lo que necesita el sistema operativo para funcionar.

En los ordenadores habituales, llamaremos así en este caso a los ordenadores de sobremesa o portátiles, el *bootloader*, aunque suelen tener una BIOS que es más genérica y a diferencia de éste no desaparece al cargar el *kernel* sino que sirve como capa de abstracción al hardware para algunos servicios, está almacenado en memoria *ROM* o *RAM* y el sistema operativo y las aplicaciones usuales se encuentran en dispositivos de almacenamiento no volátiles como son: discos duros, CD-ROM, dispositivos de memoria *USB*, etc... la localización del sistema operativo en estas unidades es la causante del tiempo necesario para el arranque del sistema operativo. Por ejemplo, si el tiempo necesario para que el ordenador este preparado para contestar a los requerimientos del exterior es aproximadamente de 1 minuto, 15 segundos serán empleados por los cargadores preliminares y el resto por la carga del sistema operativo, esto no es permisible en sistemas empotrados que deben arrancar casi instantáneamente, por ejemplo, no podemos esperar un minuto a que se active el sistema de control de nuestro automóvil cada vez que arrancamos, por lo tanto, tienen el sistema operativo en la memoria *ROM* o memoria *flash*, mismo sitio en el que se encuentra el *bootloader*, gracias a lo cual pueden ser ejecutados de forma casi instantánea (típicamente en pocos segundos).

Por lo tanto, en nuestro sistema tanto el *bootloader* como el sistema operativo se almacenan en la memoria *flash*, proporcionando también el beneficio de que es reprogramable, es decir permite el borrado y la programación de nuevas aplicaciones, tanto el *bootloader* como el sistema operativo, de interés en el caso de nuevas actualizaciones o necesidad de probar aplicaciones distintas, como podría ser el instalar un sistema *Windows* o un sistema *Linux* distinto.

El *bootloader* permite detener el proceso de arranque, como se explicará posteriormente, si la acción es reinstalar el *kernel* o realizar automáticamente el proceso de arranque para utilizar el sistema ya instalado.

El *bootloader* que requiere nuestro sistema para descargar el *kernel* de *Linux* es el *Universal Bootloader*, *U-Boot*. Como ya se ha dicho la tarjeta *phyCORE-PXA270* posee 32 MB dividida en 128 bloques individuales cada uno de los cuales de 256 kB, y el *U-Boot* tiene que ser instalado en el primer bloque de 256 kB, es decir, ocupará desde la dirección *0* a la *0x40000 como se muestra en la figura 3.1*.

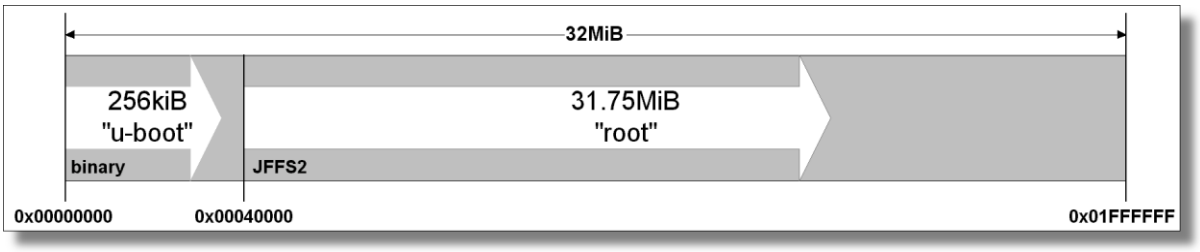


Figura 3. 1 Particionado de la memoria flash

Para grabar el *bootloader* se necesitan los siguientes dispositivos *hardware*:

- *PhyCORE PXA270*.
- Kit de desarrollo de la tarjeta *phyCORE PXA270*.
- Adaptador de alimentación de 12 VDC, 3.3 A.
- Cable *null-modem* RS-*232* (No imprescindible).
- Adaptador de *JTAG* a cable plano de 20 pines (conector del kit de desarrollo).
- Adaptador de *JTAG* al conector del puerto paralelo del *PC* usado.

Además necesitaremos un *PC* basado en *Windows* ya que los programas necesarios para realizar esta operación han sido facilitados en este entorno.

La primera operación a realizar será la de instalar el driver *GIVEIO.sys*, figura 3.2, en el sistema *Windows XP*, este driver es usado para permitir en el *PC* tener acceso a los registros del *hardware* y a los puertos de entrada y salidas. Es decir, primero deberemos instalar este driver con una aplicación llamada *LoadDrv2_XP.exe*, para después conectar el conector *JTAG* al conector del kit de desarrollo y este al adaptador del cable paralelo que se conecta al puerto *LPT1* del ordenador.

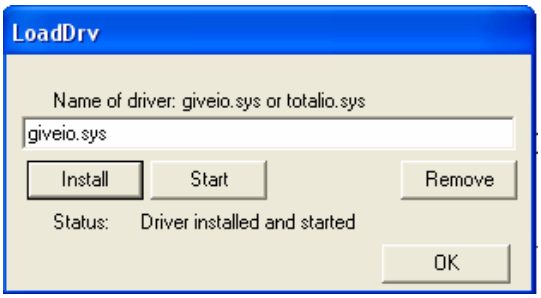


Figura 3. 2 Driver de acceso a registros

En este punto ya podemos programar el *bootloader* mediante el conector *JTAG*. Esto se realiza mediante un archivo de procesamiento por lotes denominado *batch*. Se trata de un archivo de texto sin formato, guardado con la extensión *BAT* que contiene un conjunto de comandos de *MS-DOS*. Los comandos contenidos son ejecutados en grupo de forma secuencial permitiendo automatizar diversas tareas.

El archivo *batch* debe contener para su ejecución la configuración que se muestra en la tabla 3.1.

Jflashmm bulbcx u-boot-1.2.0-phytec-pcm027-2.bin P 0	
Jflashmm	Programador JTAG genérico de memoria flash que usa la configuración de los archivos de datos que definen la plataforma a programar.
Bulbcx.dat	Nombre del archivo de datos que describe la plataforma.
u-boot-1.2.0-phytec-pcm027-2.bin	Nombre de la imagen en binario que se desea programar.
P	Los parámetros serán contados y dependientes de la posición. También indica que la operación se programará y verificará.
0	Dirección hexadecimal donde debe comenzar la programación.

Tabla 3. 1 Configuración del fichero batch de grabación del u-boot

En la pantalla *command* del ordenador *host* podremos ver como el programador reconoce la tarjeta y lleva a cabo la tarea de programación de la imagen en la memoria *flash*, figura 3.3.

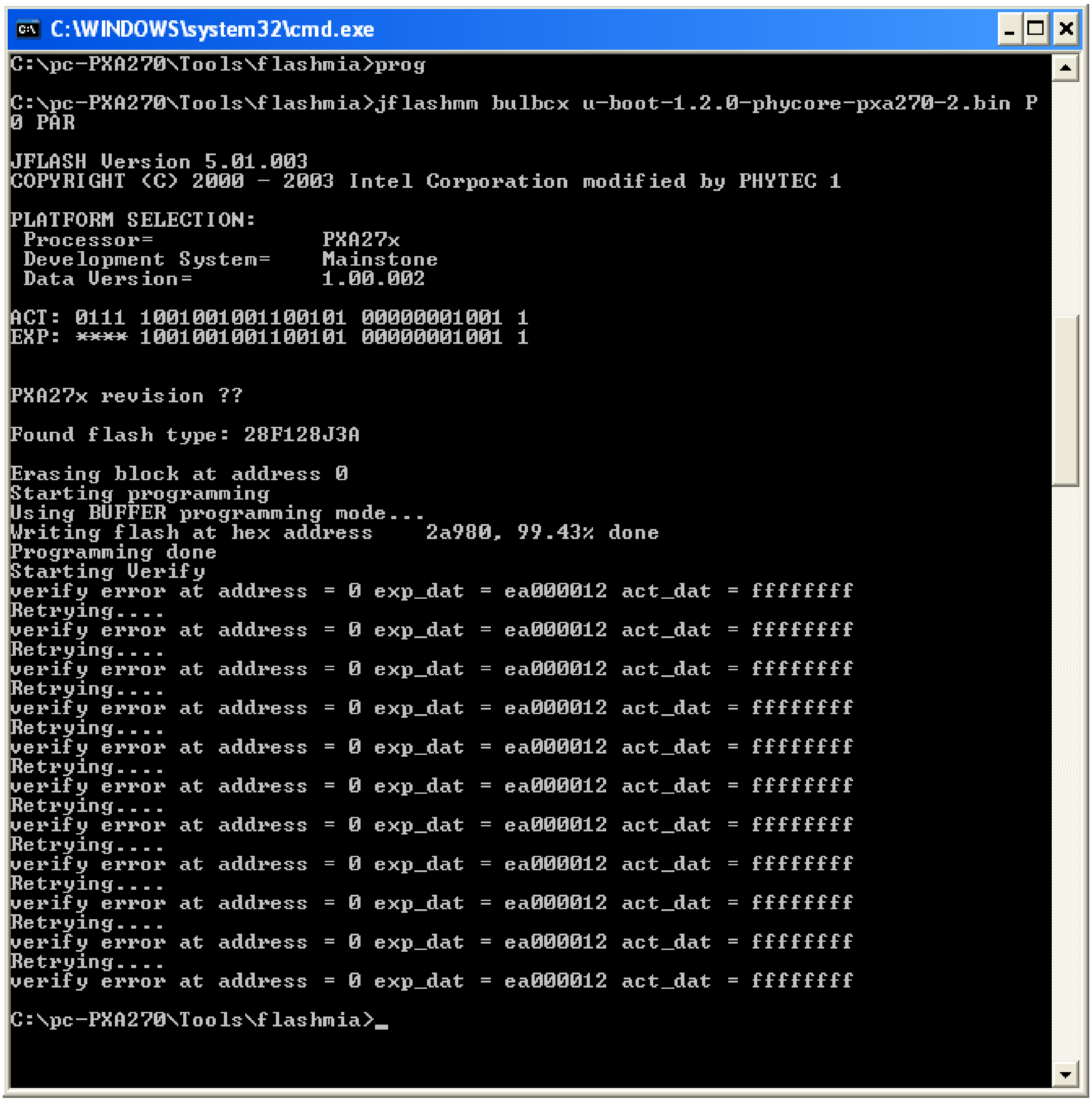


Figura 3. 3 Programación del bootloader

Mediante la conexión serie *RS-232 null-modem* configuramos, por ejemplo, el *hyperterminal*, de Windows, o el *Komport*, en Linux, como se muestra a continuación:

"115200 baud, 1 Start bit, 8 data bits, 1 stop bit, no parity and no flow control"

A continuación se visualizan los mensajes que envía la tarjeta tras ejecutar el *bootloader*, y de la misma forma mediante en comando *printenv* las variables que han sido inicializadas en el bootloader, de las cuales dependerá después la descarga de la imagen del sistema mediante *tftpboot* y su programación en memoria, figura 3. 4.

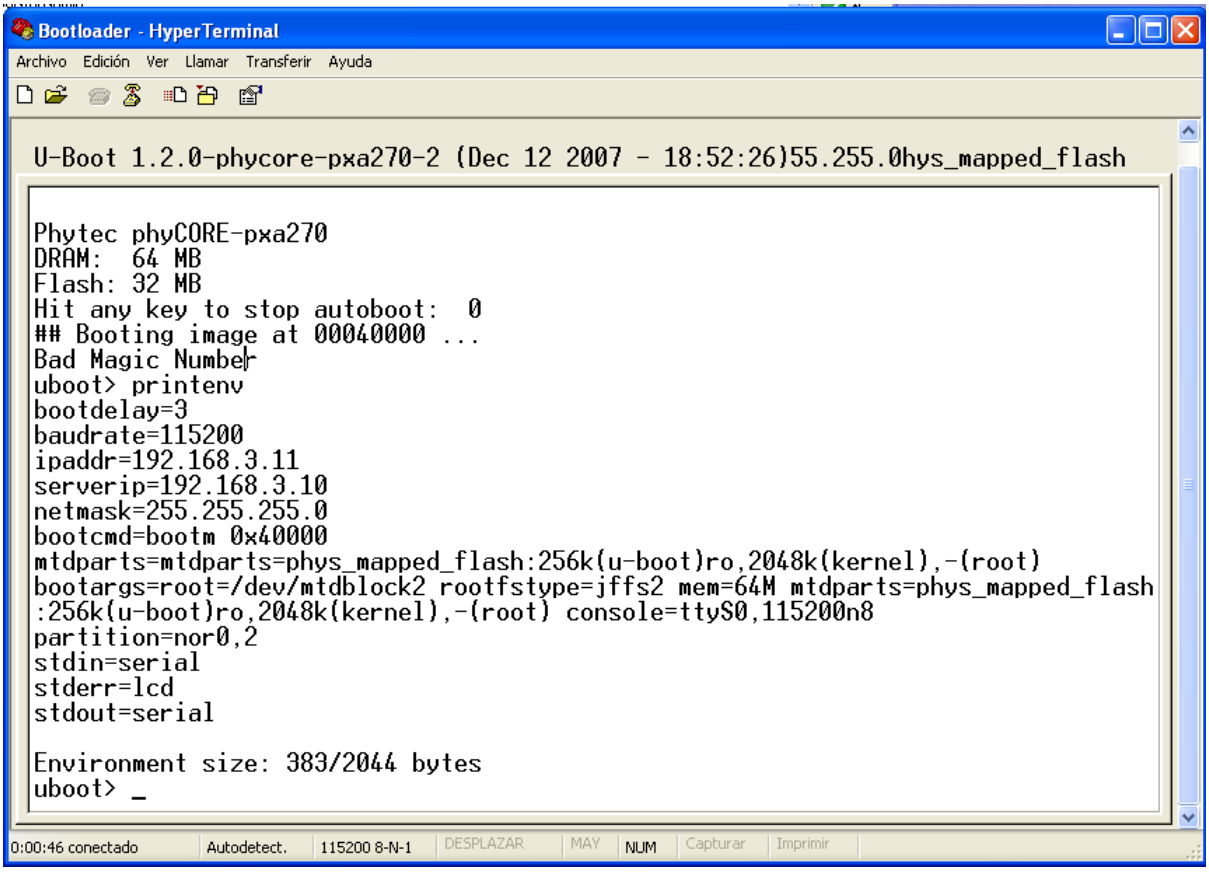


Figura 3. 4 Interfaz del bootloader

En la figura 3.4 se puede comprobar cómo en la cuarta línea se localiza la opción de detener el arranque del sistema durante un tiempo de 3 segundos, especificado en la variable *bootdelay*, transcurridos los cuales al no tener ningún sistema en memoria se introduce en el interfaz del *U-Boot*. También se puede observar la configuración predeterminada con la cual se configura la comunicación serie y direcciones *IP* tanto de la propia tarjeta como del *host* desde donde se realizará la descarga mediante *tftpboot* de la imagen que vaya a ser grabada en la memoria *flash*. A lo largo de los siguientes apartados se volverá a retomar esta configuración y se explicarán las variables que sean concernientes.

3.2. Herramientas para la construcción del kernel

Para implementar un *software* o aplicación cuyo objetivo principal es el control de todos los ejes o motores del robot, e interpretar y responder a los estímulos externos recibidos mediante los sensores distribuidos en el sistema la primera necesidad es habilitar todos los dispositivos *hardware* necesarios mediante la grabación del núcleo o *kernel* del sistema embebido.

El núcleo es la parte fundamental de un sistema operativo. Es el software responsable de facilitar a los distintos programas el acceso seguro al hardware de la computadora, figura 3.5, o dicho de una manera más simple, es el encargado de gestionar los recursos integrados en la tarjeta, a través de servicios de llamada al sistema.

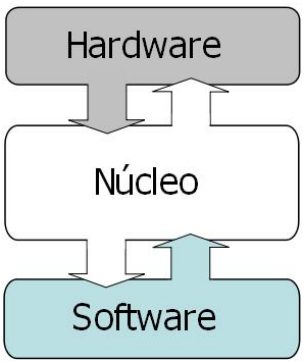


Figura 3. 5 Interpretación del núcleo

Como hay muchos programas y el acceso al *hardware* es limitado, el núcleo también se encarga de decidir qué programa podrá hacer uso de un dispositivo de *hardware* y durante cuánto tiempo, lo que se conoce como multiplexado. Como acceder al *hardware* directamente es realmente complejo, los núcleos suelen implementar una serie de abstracciones del *hardware*, lo que permite esconder la complejidad, y proporciona una interfaz limpia y uniforme al *hardware* subyacente, facilitando su uso para el programador.

El núcleo del sistema operativo *Linux* es un programa escrito casi en su totalidad en lenguaje C, con excepción de una parte del manejo de interrupciones, expresada en el lenguaje ensamblador del procesador en el que opera. Las funciones del núcleo son permitir la existencia de un ambiente

en el que sea posible atender a varios usuarios y múltiples tareas de forma concurrente, repartiendo al procesador entre todos ellos, e intentando mantener en grado óptimo la atención individual.

De esta explicación se puede entender la necesidad de utilizar un compilador específico de la máquina en la que se trabaja, puesto que la función del compilador es traducir el programa, en este caso el núcleo, escrito en el lenguaje de programación utilizado por el programador a otro lenguaje de programación que la máquina es capaz de interpretar, normalmente este segundo lenguaje es código máquina.

En el *µPC* donde queremos instalar el sistema operativo no se utilizará ningún compilador por varias razones:

- Porque inicialmente no existe sistema operativo sobre el que compilar, por lo que primero se debe crear en otro *PC* para después instalarlo en el *µPC*.
- Limitaciones de espacio en la tarjeta *phyCORE.* El espacio mínimo necesario esta definido por el tamaño del compilador, los códigos fuentes del *kernel* (en inglés *kernel-sources*) y las librerías necesarias para el desarrollo del proceso lo que sobrepasa el espacio disponible ya que un *kernel* de *Linux* ocupa aproximadamente 45 MB.
- Reducida velocidad de procesamiento, 520 Mhz.

Todo ello comparado con la posibilidad de realizar todos estos procesos en un *PC* de sobremesa con mucho más espacio y una considerable superior potencia. Este proceso se realizará mediante el uso de un compilador cruzado. Un compilador cruzado se diferencia del primero en que este tipo de compiladores generan código para un sistema distinto al sistema donde se ejecuta, es decir, podremos compilar en un *PC* de sobremesa las fuentes del *kernel* una vez configuradas para que se ejecuten en la tarjeta *phyCORE PXA270*.

Considerando únicamente las alternativas *OpenSource*, para construir el *kernel* tenemos dos opciones y aunque en ambas es necesario el compilador cruzado, será necesario o no disponer de herramientas complementarias en dicho proceso.

Además en ambas opciones deberemos preparar nuestro PC, en el cual está instalada la distribución *Debian* de *Linux*, para poder desarrollar las funciones necesarias que implique la compilación del *kernel*, esto es, añadir ciertas librerías y complementos que el sistema operativo no

tiene por defecto. Para ello habrá que buscar estos paquetes en la red, descargarlos, compilarlos e instalarlos.

Por fortuna, entre las ventajas que presenta la distribución *Debian* de Linux, usada en este caso, se encuentra la de poseer una herramienta denominada *APT*, *Advanced Package Tool*. *APT* es un sistema de gestión de paquetes de software desarrollado por el *APT_Team* del proyecto *Debian* bajo licencia *GNU/GPL*. Es el sistema de gestión de paquetes más avanzado en la actualidad y el que más flexibilidad y potencia posee para entornos de red. La utilidad para usar desde la línea de comandos que hace de interfaz entre el usuario y *APT* es *apt-get*. Los lugares de donde se realizan las descargas en la web se indican en el fichero */etc/apt/sources.list*. Aunque esta herramienta es muy útil habrá algunos paquetes que deban ser buscados manualmente ya sea por obsolescencia o por inexistencia de los paquetes en las ubicaciones señaladas. A continuación se exponen algunos de los paquetes necesarios para la preparación del *host* antes de compilar el *kernel*.

- *Libxml2-devel*: Esta librería permite manipular archivos *XML*. Requerido para instalar Eclipse.
- *Phyton-devel*: Incluye archivos y librerías necesarias para construir módulos *Phyton*. Requerido para instalar eclipse.
- *Tftp*: *Trivial File Transfer Protocol*. Es normalmente usado para iniciar estaciones de trabajo sin suficiente espacio en disco y obtener o guardar archivos de configuración mediante red.
- *Qt3-devel*: Necesario para compilar programas con *QT3*. Requerido para instalar eclipse.
- *Tcl/Tk Development System*: Lenguaje muy compacto y flexible para tipos especiales de *hardware*, por lo que es el preferido para desarrollos embebidos. Necesario para la construcción de la herramienta que asiste la generación del sistema embebido.
- *C/C++ Compiler and Tools*. Necesario para la compilación de los paquetes.

La primera alternativa a considerar es la que ofrece el kit de desarrollo de la propia tarjeta phyCORE, que viene acompañado de un *CD* con documentación, manuales *hardware* y *software*; herramientas para la instalación del *bootloader*, jflashmm; un paquete de instalación el cual instala automáticamente el programa Eclipse, del que se hablará más adelante, y un compilador cruzado

con las librerías requeridas para poder realizar la compilación cruzada (en inglés *cross-compiler toolchain* y *cross-compiled libraries*); y un *kernel* de *Linux* versión 2.6.10.

Esta alternativa consiste en compilar directamente el *kernel* utilizando el compilador cruzado y editando el fichero *Makefile* del *kernel* (fichero de dependencias y reglas para construir un programa de manera automatizada mediante la herramienta *make*) o agregando las variables como argumentos de la herramienta *make* para que tenga en cuenta la arquitectura para la que se realiza la compilación y el tipo de compilador. Para llevar a cabo esta alternativa se debe escribir en la línea de comandos del *host* las siguientes líneas:

- Configuración del *kernel* para la arquitectura *arm*:

  "make menuconfig ARCH=arm"

- Compilación del *kernel* mediante el compilador cruzado:

  "make ARCH=arm CROSS_COMPILE=arm-softfloat-linux-gnu- uImage"

De esta manera los códigos fuente del *kernel* con las opciones implementadas para la arquitectura *ARM* serán compilados y el nuevo *kernel* denominado *uImage* se encontrará en el directorio de las fuentes: *arch/arm/boot.*

Al ser un *kernel* estándar hay que deshabilitar muchas opciones de configuración para poder compilarlo sin problemas y que funcione en la tarjeta *phyCORE PXA270* y al programarlo en la memoria el resultado es que numerosos dispositivos no funcionaran, como por ejemplo, el puerto *USB* o el puerto del bus *CAN*. Para solucionar esto habría que buscar los drivers uno por uno y compilarlos para la arquitectura en cuestión y aún así los resultados no son exitosos.

La segunda alternativa nos da la solución a la mayoría de estos problemas pues aunque el proceso inicial es más largo constituye una herramienta de automatización en la construcción del sistema embebido. Esta opción es proporcionada, como se ha citado anteriormente, por *Pengutronix [16]*, una de las primeras compañías en Alemania que provee soluciones embebidas basadas en sistemas *Linux*, esta empresa provee varios paquetes:

- Herramienta *Ptxdist*, Ptxdist-1.0.1.tgz: Es la herramienta principal del paquete que proporciona *Pengutronix*. Mediante esta herramienta el cliente configura su sistema a medida. Facilita la construcción de una distribución *Linux* para un sistema embebido de manera totalmente reproducible, es decir, totalmente automatizado. Por cada

componente software que se quiere añadir hay un archivo de reglas, el cual especifica las acciones que han de ser tomadas para preparar y compilar dichos paquetes.

- *Ptxdist-1.0.1-patches*: Los parches son necesarios pues todos los paquetes no son OpenSource, especialmente los referentes a compiladores cruzados, por lo que con frecuencia es necesario "parchear" el *software* original. Aclarar que un parche es código que se agrega al programa original para arreglar un código erróneo, actualizarlo, agregar funcionalidad al programa, etc... Aunque los programas y los parches se encuentran en diferentes paquetes, *Ptxdist* proporciona un mecanismo automático para aplicar los parches a los respectivos programas durante la descompresión del paquete.

- *Oselas.Toolchain-1.1.1*, *Ptxdist* no posee un *toolchain*, denominado así el conjunto de herramientas que complementan en lo necesario a las librerías requeridas para poder utilizar el compilador, preconstruido en binario. Sin embargo, es capaz, por sí solo, de construir las toolchain necesarias que dependen del proyecto a desarrollar.

- *Oselas.BSP-phyCORE-PXA270-4*, este paquete es el único específico del *hardware* en cuestión. Un *BSP*, *Board Support Package*, es la implementación de un código específico correspondiente a una tarjeta en concreto en un sistema operativo. La configuración de estos paquetes debe de coincidir con la del *bootloader* para que el sistema operativo pueda ser cargado y dispone de los drivers correspondientes a la mayoría de los periféricos de la tarjeta.

En resumen, estos paquetes proporcionan una herramienta que por un lado permite la configuración del *kernel* del sistema operativo y del sistema de archivos raíz, automatizando estos procesos y la creación de las imágenes correspondientes para poder ser grabados en la memoria *flash* y por otro lado dispone de varios compiladores cruzados correspondientes a diferentes arquitecturas; un paquete que construye el *toolchain* necesario, de acuerdo al compilador perteneciente a la tarjeta para la que se vaya a construir el sistema; y un paquete de soporte especifico de los drivers de la tarjeta en cuestión.

En las tablas 3.2 y 3.3 se muestra la relación entre los paquetes que se han de construir y los comandos que obtienen dicha construcción.

Instalación de la herramienta Ptxdist	***Construcción del toolchain requerido***
tar –zxf ptxdist-1.0.1.tgz *tar –zxf ptxdist-1.0.1-patches.tgz* *cd ptxdist-1.0.1* *./configure && make* *make install*	*//Cambio de permisos en el directorio* *// de instalación* *su* *<passwor_root>* *mkdir /opt/OSELAS.Toolchain-1.1.1* *chown <username> /opt/OSELAS.Toolchain.1.1.1* *chmod a+rwx /opt/OSELAS.Toolchain.1.1.1* *// Construcción del toolchain* *tar xf OSELAS.Toolchain-1.1.1.tar.bz2* *cd OSELAS.Toolchain-1.1.1* *ptxdist select ptxconfigs/arm-iwmmx-linux-gnueabi_gcc-4.1.2_glibc-2.5_linux-2.6.18.ptxconfig* *ptxdist go*

Tabla 3. 2 Comandos de construcción de la herramienta

OSELAS.BSP-phyCORE-PXA270-4
tar –zxf OSELAS.BSP-phyCORE-PXA270-4.tar.gz *cd OSELAS.BSP-phyCORE-PXA270-4* *//Construcción inicial de la imagen ligera* *ptxdist select ptxconfig.light Especificación de la versión ligera a construir* *ptxdist toochain /opt/OSELAS.Toolchain-1.1.1/gcc-4.1.2-glibc-2.5-kernel-2.6.18/arm-iwmmx-linux-gnueabi/bin* *ptxdist go Comando de construcción del sistema* *ptxidst images Con el sistema construido se crean las imágenes JFFS2*

Tabla 3. 3 Comandos de construcción del directorio donde se realizará el proyecto

La herramienta *ptxdist* ofrece diferentes opciones o comandos de uso. A continuación comentaremos brevemente alguna de las opciones más importantes:

- *ptxdist boardsetup*: Proporciona un interfaz para definir la configuración de las variables de inicio de la tarjeta. Es importante darse cuenta, de que si tenemos un arrancador ya instalado en la placa, sus variables de configuración deben coincidir con las de este fichero sino la grabación del *kernel* en la memoria será inviable, figura 3.6.

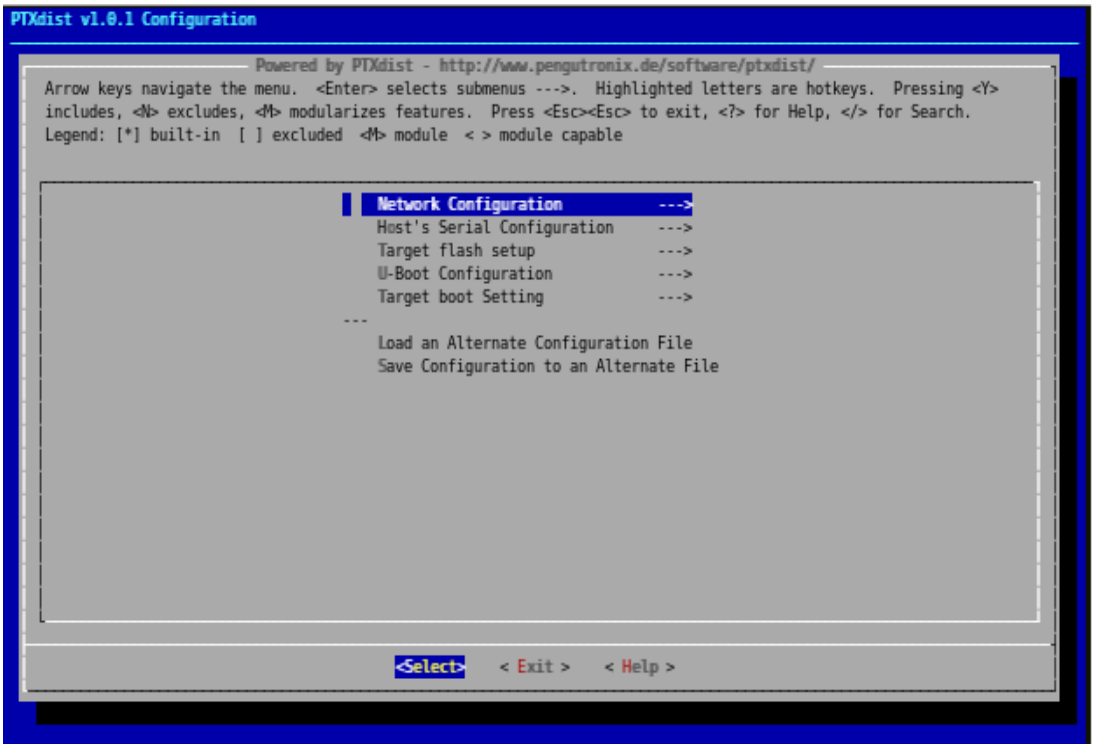


Figura 3. 6 Interfaz boardsetup

- *ptxdist select <config>*: Si tenemos varias configuraciones del sistema guardadas como por ejemplo "*ptxconfig.light y ptxconfig.full*" mediante esta opción selecciona con cual se va a trabajar.

- *ptxdist toolchain <path>*: Se debe seleccionar el *toolchain* que se va a utilizar porque los mismos paquetes pueden ser construidos con diferentes *toolchain* y se necesita uno específico válido para la tarjeta que se desarrolla.

- *Ptxdist kernelconfig*: Este comando proporciona el interfaz de configuración del *kernel*. Es la acción equivalente a *make menuconfig* que es utilizado normalmente para la configuración del *kernel*, figura 3.7.

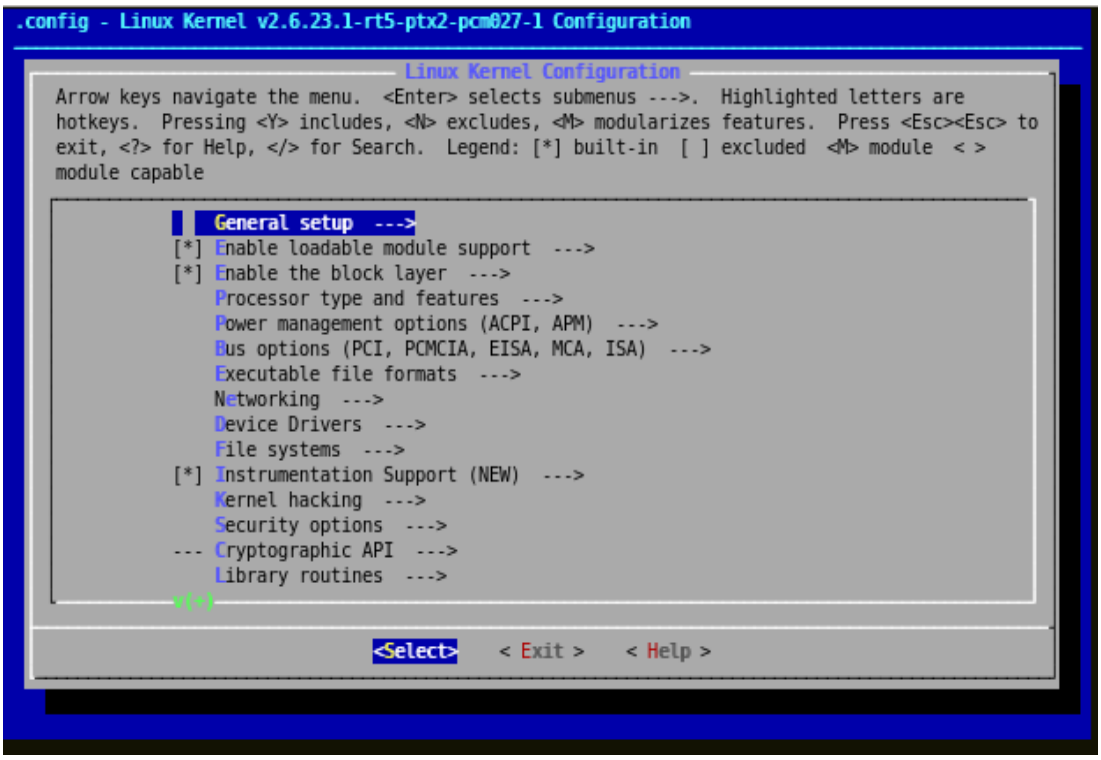


Figura 3. 7 Interfaz Kernelconfig

- *Ptxdist menuconfig*: El sistema de archivos, como ya se ha comentado, también se construye de manera automatizada. Este comando proporciona un interfaz similar al de configuración del *kernel* pero sirve para configurar el sistema de archivos raíz. Las herramientas y librerías seleccionadas serán todas comprimidas en una única imagen para su grabación en la tarjeta. Esto no quiere decir que todos los paquetes estén disponibles en este interfaz, pero si es necesario algún paquete que no esté disponible en esta selección podría agregarse posteriormente, siempre que este paquete haya sido diseñado para que trabaje en esta arquitectura, al directorio raíz o hacerle parte del proceso automático editando los ficheros correspondientes, figura 3.8.

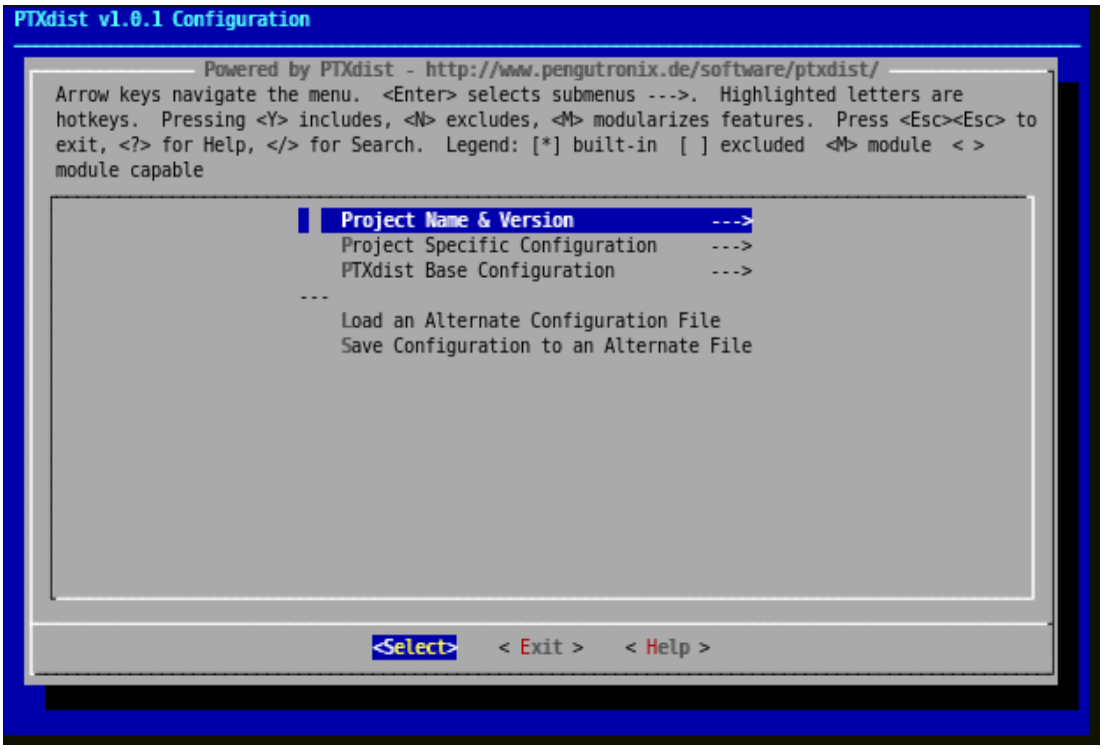


Figura 3. 8 Interfaz menuconfig

- *Ptxdist images*: Una vez configurado el *kernel* y el sistema de archivos raíz este comando crea las imágenes *uImage* (kernel) y *root.jffs2* (sistema de archivos raíz)y el fichero *u-boot.* Aunque no se ha mencionado anteriormente es importante tener en cuenta que las imágenes que se crean poseen la extensión característica de ficheros con soporte para transacciones especializado en memoria *flash*. Es decir, se crea una imagen con el formato requerido para trabajar en la tarjeta.

En la figura 3.9 se muestra una imagen obtenida a través del *Shell* de Linux donde se muestran todas estas opciones.

```
PTXdist 1.0.1                   Build System for Embedded Linux Systems

  ptxdist <action [args]> [options]

Setup and Project Actions:

  setup                         setup per-user preferences
  boardsetup                    setup per-board preferences

  projects                      show available projects
  clone <from> <to>             create a new project, cloned from <from>.

  menuconfig                    configure the root filesystem
  oldconfig                     run 'make oldconfig' on ptxconfig file
  kernelconfig                  configure the kernel

  toolchain <path>              select this toolchain (path to binaries)
  select <config>               if there is no ptxconfig file you can
                                select one of several configs to be used
Build Actions:

  go                            start building the current project

  get <package>                 get packet sources
  extract <package>             extract packet
  prepare <package>             run configure stages for packet
  compile <package>             compile the sources
  install <package>             install host side components into sysroot/
  targetinstall <package>       install files for target into root/
  clean <package>               cleanup packet
  autobuild                     search for "autobuild" scripts and run them
  drop <package>.<stage>        mark a stage of a packet as unbuilt

  images                        build images for target system

Clean Actions:

  clean                         cleanup build-host and build-cross dirs
  distclean                     cleanup everything
  (clean images)                cleanup images directory
  clean root                    cleanup root directory for target
  (clean project)               cleanup project specific packages
  (clean maintainer)            maintainerclean

Misc

  run                           start a previously built native image

  --native                      build with native compiler instead of cross
  --version                     print out ptxdist version

  (svn up)                      run "svn update" in topdir and project dir
  (svn stat)                    run "svn stat" in topdir and project dir

  test <testname>               run tests

  newpacket <type>              create a new packet Makefile in a rules dir
                                type can be one of:
                                target, host, host-existing-target,
                                cross, cross-existing-target, source,
                                kernel_driver, simple
```

Figura 3. 9 Opciones de Ptxdist

Una vez se ha extraído el *BSP* de la tarjeta y se ha construido la imagen primaria es posible empezar a modificar dicha imagen eliminando configuraciones innecesarias y agregando paquetes complementarios, esto será descrito detenidamente en el siguiente apartado. En la figura 3.10 se muestra el directorio de componentes del *BSP* donde se va a construir el proyecto:

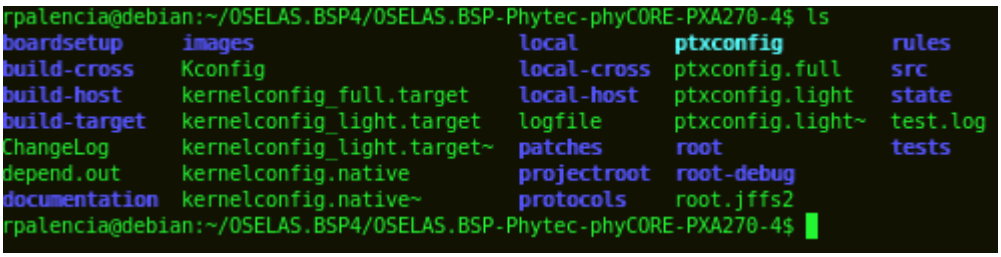


Figura 3. 10 Directorio de componentes del BSP

En este directorio destacan una serie de carpetas y archivos que se comentarán brevemente a continuación:

- *Boardsetup*: En esta carpeta se encuentra el archivo de configuración del *u-boot*.
- *Build-target*: Carpeta donde se encuentran todos los paquetes descomprimidos y compilados que han sido utilizados para construir la imagen del sistema de la tarjeta, incluido el *kernel* de *Linux*, figura 3.11.

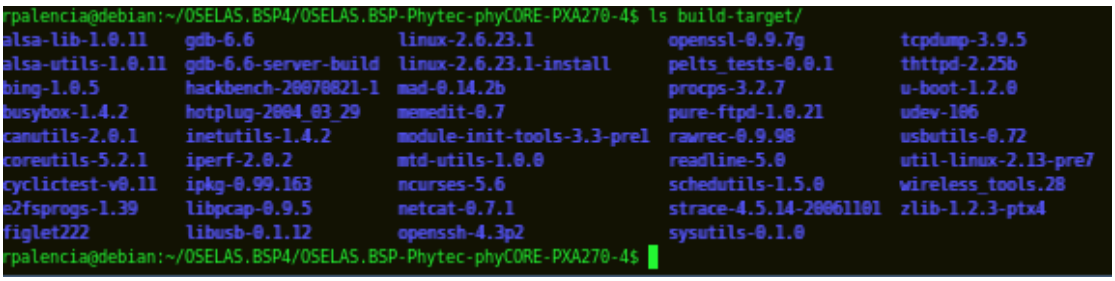


Figura 3. 11 Paquetes instalados

- *Images*: Todo el software en paquetes individuales es guardado en esta carpeta con extensión ipkg. Este formato es utilizado para la construcción de las imágenes flash del sistema embebido. También se guardan en esta carpeta las imágenes creadas *uImage*, *root.jffs2* y *u-boot*.

- *Patches*: Todos los parches y modificaciones que han sido aplicados al núcleo y que posibilitan la funcionalidad de los dispositivos de la tarjeta son guardados en esta carpeta.

- *Root*: En esta carpeta se realiza una copia idéntica a la que se obtendrá en la tarjeta, figura 3.12. Esto puede servir para verificar el sistema que se va a instalar o lo más habitual para trabajar con este sistema durante la etapa de pruebas mediante *Net File System*, *NFS*, de modo que la tarjeta al arrancar cargará a través de la red esta carpeta localizada en otro ordenador como si de su sistema operativo se tratase. Este modo se describirá con mayor detalle posteriormente.

```
rpalencia@debian:~/OSELAS.BSP4/OSELAS.BSP-Phytec-phyCORE-PXA270-4$ ls root/
bin  dev  etc  home  lib  linuxrc  mnt  proc  sbin  sys  tmp  usr  var
```

Figura 3. 12 Copia del Sistema en root

- *Src*: El primer paso que lleva a cabo el proceso de construcción del sistema es la búsqueda del archivo que posteriormente va a compilar. Su búsqueda esta predeterminada, por lo que intenta descargar el paquete desde una ubicación determinada y la descarga la realiza en este directorio, figura 3.13. Es posible que el archivo no este disponible en la dirección indicada o bien haya cambiado su nombre por lo que se requiere que el usuario lo encuentre y lo descargue a este directorio.

```
rpalencia@debian:~/OSELAS.BSP4/OSELAS.BSP-Phytec-phyCORE-PXA270-4$ ls src
alsa-lib-1.0.11.tar.bz2        hotplug-2004_03_29.tar.gz          ncurses-5.6.tar.gz          strace-4.5.14-20061101.tar.bz2
alsa-utils-1.0.11.tar.bz2      inetutils-1.4.2.tar.gz             netcat-0.7.1.tar.gz         sysutils-0.1.0.tar.gz
bing-1.0.5.tar.gz              iperf-2.0.2.tar.gz                 openssh-4.3p2.tar.gz        tcpdump-3.9.5.tar.gz
busybox-1.4.2.tar.bz2          ipkg-0.99.163.tar.gz               openssl-0.9.7g.tar.gz       thttpd-2.25b.tar.gz
canutils-2.0.1.tar.bz2         ipkg-utils-050831.tar.gz           pelts-1.0.12.tar.bz2        u-boot-1.2.0.tar.bz2
coreutils-5.2.1.tar.bz2        libpcap-0.9.5.tar.gz               pelts_tests-0.0.1.tar.bz2   u-boot-mkimage-1.1.6.tar.bz2
cyclictest-v0.11.tar.bz2       libusb-0.1.12.tar.gz               pkg-config-0.21.tar.gz      udev-106.tar.bz2
e2fsprogs-1.39.tar.gz          linux-2.6.23.1.tar.bz2             procps-3.2.7.tar.gz         usbutils-0.72.tar.gz
fakeroot_1.5.10.tar.gz         mad-0.14.2b.tar.gz                 pure-ftpd-1.0.21.tar.bz2    util-linux-2.13-pre7.tar.bz2
figlet222.tar.gz               memedit-0.7.tar.gz                 rawrec-0.9.98.tar.gz        wireless_tools.28.tar.gz
gdb-6.6.tar.bz2                module-init-tools-3.3-pre1.tar.bz2 readline-5.0.tar.gz         zlib-1.2.3-ptx4.tar.bz2
hackbench-20070821-1.tar.bz2   mtd-utils-1.0.0.tar.gz             schedutils-1.5.0.tar.gz
```

Figura 3. 13 Paquetes software descargados

- *Rules*: Los archivos se descargan desde una localización ubicada en la red, se almacenan, se extraen, se compilan y se instalan en los directorios asignados para la creación de la imagen, todos estos pasos se encuentran definidos en unos archivos de reglas almacenados en esta carpeta, figura 3.14.

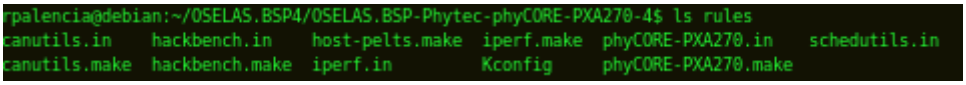


Figura 3. 14 Ficheros de reglas

3.3. Construcción y configuración del sistema

La finalidad de este apartado es explicar en que consiste la configuración del kernel y del sistema de archivos raíz y porqué se deben configurar hacia unas especificaciones concretas limitando su tamaño y optimizando su funcionalidad.

Al contrario de lo que pueda parecer el *kernel* de Linux a instalar en un *PC* de sobremesa es exactamente el mismo que el *kernel* de *Linux* que se va a instalar en el sistema embebido. Básicamente la diferencia principal entre ambos es la necesidad de parchear el sistema que se construye con un soporte específico definido por las características de la tarjeta *phyCORE PXA270* y deshabilitar ciertas características del *kernel* no necesarias o agregar módulos requeridos en el sistema y no estándar para los sistemas en general.

De esta manera nos interesa evaluar 3 bloques diferentes:

- Características que provee la tarjeta *phyCORE* PXA270 a nivel hardware, ya que provee más dispositivos de los que necesitamos.
- Características a las que da soporte el *BSP* de *OSELAS*, el cuál, aunque habilita numerosos dispositivos de la tarjeta *phyCORE* no da funcionalidad completa, pej. el *USB* cliente.
- Características que son o no de interés para desarrollar el sistema requerido.

A continuación, se expondrá en la tabla 3.4 las características hardware de la tarjeta *phyCORE PXA270* y las funcionalidades a las que da soporte el *BSP* de *OSELAS* para luego ir viendo las opciones de las que deberá constar el núcleo.

Caracterísitcas de PhyCORE PXA270	***Funcionalidades ofrecidas por el BSP de OSELAS***
Marvell Xscale PXA270 (ARM V5TE	Support PCM-990 base board
520 Mhz core frequency	Kernel 2.6.23.1
Wireless MMXTM	RT-Preempt support for hard realtime
Quick capture camera interface	On board flash memory support for kernel and root file system
Memory Management Unit (MMU)	Network support through BSD sockets for on board CAN controller
Memory and DMA controllers	On chip I2C support
Integrated LCD controller	On chip SPI support
Matrix Keyboard controller	I2C realtime clock support for correct system time
USB 1.1 host/ USB 1.1 client/ OTG	I2C EEPROM acess support
2x PCMCIA	Some GPIO (depend on baseboard)
2x MMC/SD/SDIO (1 x card slot)	CAN bus debug support
USIM card interface	USB host 1.1
I2C/I2S/AC97/ 3xSSP serial ports	MMC/SD card support
4xPWM	U-boot is now part of the BSP
3xUART (1 at RS232)/IRDA	LCD and Xsupport
RTC with external battery backup	AC97 based audio
SJA1000 2.0b CAN controller	Support for CPU's PWM and EventGPIO feature
10/100 Mbps LAN91C111 Ethernet	
Configuration storage EEPROM for	
SPI GPIO Expander	
Memory configuration: • *256 Kb internal SRAM* • *SDRAM: 64 to 128 Mb* • *NOR flash: 32 to 64 Mb* • I2C EEPROM: 4Kb	
JTAG interface	

Tabla 3. 4 Características hardware de phycore y funcionalidades del BSP

Mostradas estas características se debe acceder a la configuración del *kernel* mediante el comando *ptxdist kernelconfig* en la carpeta *OSELAS.BSP-Phytec-phyCORE-PXA270-4* y se visualizará en la pantalla la figura 3.15 mostrada a continuación.

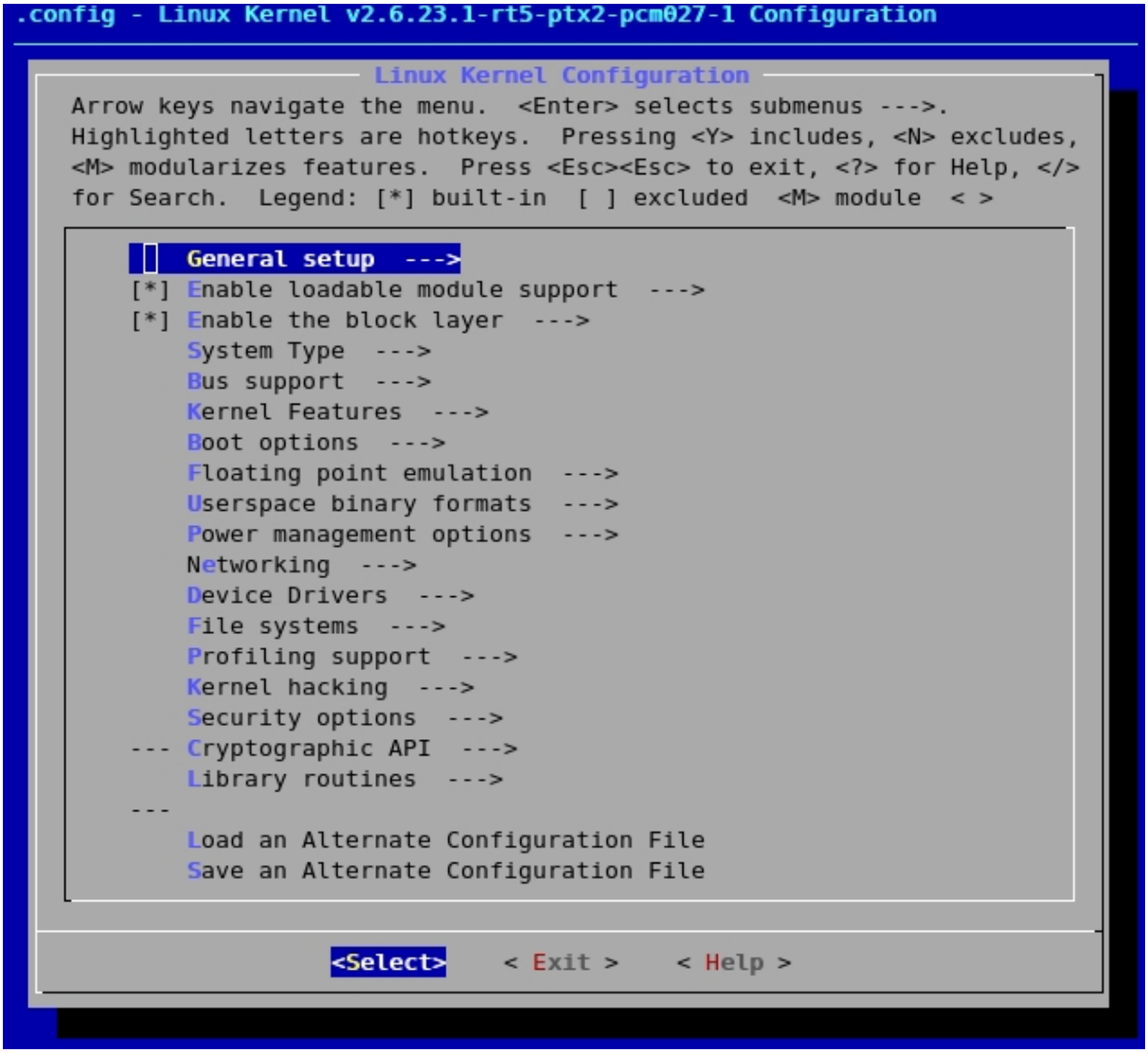


Figura 3. 15 Configuración general

La configuración de *General setup* puede verse en la tabla 3.5. Esta configuración ha sido realizada completamente por el *BSP* y puede apreciarse la diferencia entre una configuración para un ordenador de sobremesa en el cual es requerido, por ejemplo, el soporte *initrd* que es un sistema de archivos temporal para realizar los arreglos necesarios para montar el sistema raíz y no es necesario en este sistema, y cómo se realiza una configuración estándar para pequeños sistemas, *small systems*.

General Setup→
[*]Prompt for development and/or incomplete code/drivers
() Local version – append to kernel release
[*]Automatically append version information to the version string
[ ]Support for paging of anonymous memory (swap)
[*]System V IPC
[*]POSIX Message Queues
[ ]BSD Process Accounting
[ ]Export task/process statistics through netlink (EXPERIMENTAL)
[ ]User Namespaces (EXPERIMENTAL)
[ ]Auditing support
< >Kernel .config support
(14)Kernel log buffer size (16=>64KB, 17=>128KB)
[ ]Create deprecated sysfs files
[ ]Kernel->user space relay support (formerly relayfs)
[ ]Initial RAM filesystem and RAM disk (initramfs/initrd) support
[ ]Optimize for size (Look out for broken compilers!)
[ ]Enable concurrent radix tree operations (EXPERIMENTAL)
[*]Configure standard kernel features (for small systems) →
[*]Enable 16-bit UID system calls
[ ]Sysctl syscall support
[*]Load all symbols for debugging/ksymoops
[ ] Do an extra kallsyms pass
[*]Support for hot-pluggable devices
[*]Enable support for printk
[*]BUG() support
[*]Enable ELF core dumps
[*]Enable full-sized data structures for core
[*]Enable futex support
[*]Configure eventpoll support
[*]Enable signalfd() system call
[*]Enable enventfd() system call
[*]Use full shmem filesystem
[*]Enable VM event counters for /proc/vmstat
Choose SLAB allocator (SLAB)→

Tabla 3. 5 Configuración de General Setup

Se debe configurar el permitir la carga de módulos no cargados al inicio por el sistema, tabla 3.6, pues como se verá posteriormente será un requisito indispensable ya que el módulo para las comunicaciones inalámbricas debe ser cargado al iniciar el sistema operativo.

[*]Enable loadable module support→
[*]Module unloading
[ ] Forced module unloading
[ ]Module versioning support
[ ]Source checksum for all modules
[ ]Automatic kernel module loading

Tabla 3. 6 Habilitar carga de módulos

A continuación se selecciona el tipo de sistema para el que se configura el núcleo, es decir, basado en *PXA2xx* y la tarjeta en cuestión que es la *PHYTEC-phyCORE-PXA270 (PCM-027).* La opción de *display* es deshabilitada ya que para este sistema no se requiere ningún dispositivo de visualización y se habilitan como módulos el soporte para *PWM* y el grupo de entradas y salidas digitales *GPIO*. Puede observarse un soporte *iWMMXT*, el cuál, aunque no es modificable y viene integrado se debe a un requerimiento del compilador cruzado que se va a utilizar. En el soporte para *BUS* está disponible el soporte para *PCMCIA*, pero es innecesario en nuestro sistema por lo que se deshabilita, tabla 3.7.

System Type →
ARM system type (PXA2xx-based) →
Intel PXA2xx Implementations →
Select target board (PHYTEC phyCORE-PXA270 (PCM-027)) →
[*]PHYTEC phyCORE-PXA270 (PCM-027) Baseboard
Choose your display (no display) →
[*]PHYTEC phyCORE-PXA270 (PCM-027) gpio expander board
<M>Support for Pulse Width Modulation device on PXA27x
<M>Support for GPIO based even notification on PXA27x
Boot options
Power management
Processor Type
Processor Features
[ ]Support Thumb user binaries
[ ]Disable D-Cache (C-bit)
Enable iWMMXt support
Bus support →
PCCARD (PCMCIA/CardBus) support →
<>PCCard (PCMCIA/CardBus) support

Tabla 3. 7 Configuración del tipo de sistema y del soporte de bus

A partir del *kernel* 2.6.21 ha salido una nueva tecnología basada en el tiempo real que acondiciona el sistema con una mayor eficiencia, está opción se denomina *Tickless System (Dynamic Ticks)*. Es en esta sección del *kernel* donde se habilitan las capacidades para tiempo real. En la tabla 3.8 se aprecia la configuración en un modo que soporta configuración de *Desktop*, es decir, las latencias son altas, donde se entiende latencias altas como esperas de milisegundos.

*--- **Kernel Features** →*
[*] Tickless System (Dynamic Ticks)
[*]High Resolution Timer Support
Preemption Mode (Voluntary Kernel Preemption (Desktop)) →
Thread Softirqs
[*]Thread Hardirqs
RCU implementation type: (Classic RCU) →
[ ] Enable priority boosting of RCU read-side critical section
[ ]Enable tracing for RCU – currently stats in debugfs
[*]Use the ARM EABI to compile the kernel
[ ] Allow old ABI binaries to run with this kernel (EXPERIMENTAL)
Memory model (Flat Memory) →
[ ]64 bit Memory and IO resources (EXPERIMENTAL)
[ ]Timer and CPU usage LEDs
Consistent DMA size (2 MiByte) →

Tabla 3. 8 Características del kernel sin funciones de tiempo real habilitadas

Sin embargo, si se cambia a la opción *Preemption Mode (Complete Preemption(Real time))* automáticamente se cambian ciertas opciones, tabla 3.9, ya que deben ser más restrictivas y de esta forma esta configuración asegura que las máximas latencias estarán en el orden de microsegundos.

*--- **Kernel Features** →*
[*] Tickless System (Dynamic Ticks)
[*]High Resolution Timer Support
Preemption Mode (Complete Preemption (Real-Time)) →
Thread Softirqs
Thread Hardirqs
RCU implementation type: (Preemptible RCU) →
Preemptible RCU
[ ] Enable priority boosting of RCU read-side critical section

Tabla 3. 9 Características del kernel con funciones de tiempo real habilitadas

En las opciones de inicio lo único que se configura es la dirección en la que se inicia el arranque que es la dirección *0x0, tabla 3.10.*

***Boot options** →*
(0x0)Compressed ROM boot loader base address
(0x0)Compressed ROM boot loader BSS address
( ) Default kernel command string
[ ] Kernel Execute-In-Place from ROM
[ ] kexec system call (EXPERIMENTAL)

Tabla 3. 10 Opciones de inicio

En la configuración de comunicaciones como se puede ver en la tabla 3.11 de una larga lista de protocolos de red los protocolos necesarios serán el paquete *SOCKET* de *UNIX* y los protocolos de red *TCP/IP* y *DHCP* para la comunicación mediante Ethernet.

Networking →
[*]Networking support
Networking options →
<>Packet socket*
[*] Packet socket: mmapped IO
<>Unix domain sockets*
< >PF_KEY sockets
[*]TCP/IP networking
[ ] IP: multicasting
[ ] IP: advanced router
[*] IP: kernel level autoconfiguration
[*] IP: DHCP support
[ ] IP: BOOTP support
[ ] IP: RARP support
< > IP: tunneling
< > IP: GRE tunnels over IP
[ ] IP: ARP daemon support (EXPERIMENTAL)
[ ] IP: TCP syncookie support (disabled for default)
< > IP: AH transformation
< > IP: ESP transformation
< > IP: IPComp transformation
< > IP: IPsec transport mode
< > IP: IPsec tunnel mode
< > IP: IPsec BEET mode
< > INET: socket monitoring interface
[ ] TCP: advanced congestion control →
[ ] TCP: MD5 Signature Option support (RFC2385) (EXPERIMENTAL)
< > The IPv6 protocol
[ ] Security Marking
[ ] Network packet filtering framework (Netfilter) →
< > The DCCP Protocol (EXPERIMENTAL) →
< > The SCTP Protocol (EXPERIMENTAL) →
< > The TIPC Protocol (EXPERIMENTAL) →
< > Asynchronous Transfer Mode (ATM) (EXPERIMENTAL)
< > 802.1d Ethernet Bridging
< > 802.1Q VLAN Support
< > DECnet Support
< > ANSI/IEEE 802.2 LLC type 2 Support
< > The IPX protocol
< > Appletalk protocol support
< > CCITT X.25 Packet Layer (EXPERIMENTAL)
< > LAPB Data Link Driver (EXPERIMENTAL)
[ ]Amateur Radio support →

Tabla 3. 11 Configuración de comunicaciones a)

Siguiendo con las comunicaciones también se debe habilitar los protocolos para la comunicación del bus *CAN* y el driver para el chipset de la controladora del bus *CAN* que integra la placa *Philips SJA1000*, tabla 3.12

Networking →
<M>CAN bus subsystem support →
<M> Raw CAN Protocol (raw access with CAN-ID filtering)
[ ] Allow non-root users to access Raw CAN Protocol sockets
<M> Broadcast Manager CAN Protocol (with content filtering)
[ ] Allow non-root users to access CAN broadcast manager socket
[M] CAN Core debugging messages
CAN Device Drivers →
< > Virtual Local CAN Interface (vcan)
< > Serial/USB serial CAN Adaptors (slcan)
[ ] CAN devices debugging messages
<M>Philips SJA1000 with device interface
< > Philips SJA1000 legacy HAL interface
< > Intel 82527
< > Bosch CCAN driver
< > Phytec pcm027 can driver (SJA1000)
< > IEEE 802.11i TKIP encryption
<> Software MAC add-on to the IEEE 802.11 networking stack*
[*] Enable full debugging output
< >RF switch subsystem support →
< >Plan Resource Sharing Support (9P2000) (EXPERIMENTAL)

Tabla 3. 12 Configuración de comunicaciones b)

Además de las opciones anteriores puesto que se utilizarán las redes inalámbricas para realizar las comunicaciones con el exterior del robot se deben habilitar los protocolos correspondientes a las comunicaciones que se van a llevar a cabo, es decir, el protocolo *802.11* de comunicación *Wi-Fi*, tabla 3.13.

Networking →
< >IrDA (infrared) subsystem support →
< >Bluetooth subsystem support →
<>RxRPC session sockets
Wireless →
Wireless extensions
<> Generic IEEE 802.11 Networking Stack (mac80211)*
[ ] Enable debugging output
Generic IEEE 802.11 Networking Stack
[ ] Enable full debugging output
IEEE 802.11 WEP encryption (802.1x)
< > IEEE 802.11i CCMP support

Tabla 3. 13 Configuración de comunicaciones c)

Una vez seleccionadas las opciones de red se seleccionaran los dispositivos *drivers* que serán requeridos, tabla 3.14. Habilitamos la posibilidad de uso de *firmware* al driver que lo requiera, necesario para el driver que se utilizará en las comunicaciones inalámbricas. Se habilita el soporte para *MTD Memory Technology Device* usado por las memorias de tipo *flash*.

Device Drivers→
Generic Driver Options →
[*] Select only drivers that don't need compile-time external firmware
[*] Prevent firmware from being built
Userspace firmware loading support
< > Connector – unified userspace <-> kernelspace linker →
<>Memory Technology Device (MTD) support →*
< > Parallel port support →
[*] Block devices →
< > ATA/ATAPI/MFM/RLL support →
SCSI device support →
Serial ATA (prod) and Parallel ATA (experimental) drivers →
[ ] Multiple devices driver support (RAID and LVM) →

Tabla 3. 14 Configuración genérica de dispositivos drivers.

En dispositivos drivers también se habilita la controladora de *Ethernet* de la tarjeta que es la *SMC 91C111*. La comunicación inalámbrica se ha llevado a cabo mediante un *USB Wi-Fi*, sin embargo no se señalará ninguna de las opciones disponibles pues ninguna corresponde al chipset del *USB Wi-Fi* utilizado. Se habilita el protocolo *I2C* ya que aunque no sea directamente utilizado en nuestra aplicación si es utilizado como interfaz del reloj de tiempo real, *RTC*, y como acceso a la *EEPROM*, al igual que el soporte *SPI* que es habilitado por defecto no siendo modificable tabla 3.15.

Device Drivers→
[*] Network device support →
[ ] Netdevice multiple hardware queue support
< > Dummy net driver support
< > Bonding driver support
< > MAC-VLAN support (EXPERIMENTAL)
< > EQL (serial line load balancing) support
< > Universal TUN/TAP device driver support
< > PHY Device support and infrastructure →
[*] Ethernet (10 or 100Mbit) →
Generic Media Independent Interface device support
< > ASIX AX88796 NE2000 clone support
<> SMC 91C9x/91C1xxx support*
< > DM9000 support
< > SMSC LAN911[5678] support
[ ] Ethernet (1000Mbit) →
[ ] Ethernet (10000Mbit) →
Wireless LAN →
USB Network Adapters →

```
        [ ] Wan interfaces support →
        < > PPP (point-to-point protocol) support
        < > SLIP (serial line) support
        < > Traffic Shaper (OBSOLETE)
        < > Network console logging support (EXPERIMENTAL)
< > ISDN support →
    Input device support →
    Character devices →
<*> I2C support →
        <*> I2C device interface
            I2C Algorithms →
            I2C Hardware Bus support →
            Miscellaneous I2C Chip support →
        [ ] I2C Core debugging messages
        [ ] I2C Algorithm debugging messages
        [ ] I2C Bus debugging messages
        [ ] I2C Chip debugging messages
    SPI support →
```

Tabla 3. 15 Configuración de dispositivos driver de red, I2C y SPI

No son necesarios los drivers multimedia, ni ningún tipo de soporte gráfico ni de sonido por lo tanto todos estos drivers han sido desactivados. El soporte *HID* sirve para manejar las entradas y salidas del puerto *USB* de manera general con independencia del dispositivo que sea conectado. Se habilita el soporte para que funcione el *USB host*, se habilita el *Open Host Controller Interface* u *OHCI*, el cuál es un estándar abierto que soporta el *USB* 1.1 y se habilita el uso del *USB* como dispositivo de almacenamiento, tabla 3.16.

```
Device Drivers →
    < > Dallas's 1-wire support →
    < > Hardware Monitoring support →
    [ ] Misc devices - →
        Multifunction device drivers →
    [ ] LED Support →
        Multimedia devices →
        Graphics support →
        Sound →
    [*] HID Devices →
            Generic HID support
        [ ] HID debugging support
            USB Input Devices
        <M>USB Human Interface Device (full HID) support
        [ ] Enable support for iBook/PowerBook/MacBook/MacBookPro special
        [ ] Force feedback support (EXPERIMENTAL)
        [ ] /dev/hiddev raw HID device support
            USB HID Boot Protocol drivers →
    [*] USB support →
        <*> Support for Host-side USB
        [ ] USB verbose debug messages
```

```
        Miscellaneous USB options
[*]     USB device filesystem
[ ]     USB device class-devices (DEPRECATED)
[ ]     Dynamic USB minor allocation (EXPERIMENTAL)
        USB Host Controller Drivers
< >     ISP116X HCD support
<M>     OHCI HCD support
< >     SL811HS HCD support
< >     R8A66597 HCD support
        USB Device Class drivers
        USB Modem (CDC ACM) support
        USB Printer support
        NOTE: USB_STORAGE enables SCSI, and 'SCSI disk support'
        May also be needed; see USB_STORAGE Help for more information
<M>     USB Mass Storage support
[ ]     USB Mass Storage verbose debug
[ ]     Datafab Compact Flash Reader support (EXPERIMENTAL)
[ ]     Freecom USB/ATAPI Bridge support
[ ]     Microtech/ZiO! CompactFlash/SmartMedia support
[ ]     USBAT/USBA02-based storage support (EXPERIMENTAL)
[ ]     SanDisk SDDR-09 (and other SmartMedia) support (EXPERIMENTAL)
[ ]     SanDisk SDDR-55 (SmartMedia support (EXPERIMENTAL)
[ ]     Lexar Jumpshot Compact Flash Reader (EXPERIMENTAL)
[ ]     Olympus MAUSB-10/Fuji DPC-R1 support (EXPERIMENTAL)
[ ]     Support OneTouch Button on Maxtor Hard Drives (EXPERIMENTAL)
[ ]     Support for Rio Karma music player
[ ]     The shared table of common (or usual) storage devices
        USB Imaging devices
< >     USB Mustek MDC800 Digital Camera support (EXPERIMENTAL)
< >     Microtek X6USB scanner support
[ ]     USB Monitor
        USB port drivers
        USB Serial Converter support →
        USB Miscellaneous drivers →
< >     EMI 6|2m USB Audio interface support
< >     EMI 2|6 USB Audio interface support
```

Tabla 3. 16 Configuración de dispositivos drivers de tipo USB y HID

Continuando con los dispositivos drivers se habilita el soporte para tarjetas *MMC/SD* junto con los dispositivos de bloques *MMC* que es como se leen dichas memorias. También se habilita el *RTC* pero únicamente el propio de la tarjeta entre todas las opciones disponibles, este es el chipset que sirve para ambos *Philips PCF8563/ Epson RTC8564*. También destaca la opción de habilitar las características del grupo de entradas y salidas, tabla 3.17.

Device Drivers →
<M> *MMC/SD card support* →
[ ] *MMC debugging*
[ ] *Allow unsafe resume (DANGEROUS)*
MMC/SD Card Drivers
<M> *MMC block device driver*
[*] *Use bounce buffer for simple hosts*
MMC/SD Host Controller Drivers
<M> *Intel PXA25x/26x/27x Multimedia Card Interface support*
<M> *Real Time Clock* →
RTC interfaces
[*] */sys/class/rtc/rtcN (sysfs)*
[*] */proc/driver/rtc (procfs for rtc0)*
[*] */dev/rtcN (character devices)*
[ ] *RTC UIE emulation on dev interface*
< > *Test driver/device*
I2C RTC drivers
< > *Dallas/Maxim DS1307/37/38/39/40, ST M41T00*
< > *Dallas/Maxim DS1672*
< > *Maxim 6900*
< > *Ricoh RS5C372A/B, RV5C386, RV5C387A*
< > *Intersil 1208*
< > *Xicor/Intersil X1205*
<M> *Philips PCF8563/Epson RTC8564*
< > *Philips PCF8583*
< > *ST M41T80 series RTC*
SPI RTC drivers
< > *Ricoh RS5C348A/B*
< > *Maxim 6902*
Platform RTC drivers
< > *PC-style 'CMOS'*
< > *Dallas DS1553*
< > *Simtek STK17TA8*
< > *Dallas DS1742/1743*
< > *ST M48T86/Dallas DS12887*
< > *ST M48T59*
< > *EM Microelectronic V3020*
On-CPU RTC drivers
< > *SA11x0/PXA2xx*
DMA Engine support →
GPIO support →
[*] *Support for GPIO features*

Tabla 3. 17 Configuración de los drivers de MMC/SD, RTC y soporte para GPIO

Dentro de los sistemas de archivos se habilitarán los *Ext2* y *Ext3* típicos formatos de *Linux* y diferentes versiones de *FAT* con la única intención de no tener problemas al usar dispositivos de almacenamiento mientras se están realizando la instalación del sistema y las pruebas con las aplicaciones utilizadas, tabla 3.18.

File systems→
<M> Second extended fs support
[ ] Ext2 extended attributes
[ ] Ext2 execute in place support
<M> Ext3 journalling file system support
[*] Ext3 extended attributes
[ ] Ext3 POSIX Access Control Lists
[ ] Ext3 Security Labels
< > Ext4dev/ext4 extended fs support development (EXPERIMENTAL)
[ ] JBD (ext3) debugging support
< > Reiserfs support
< > JFS filesystem support
< > XFS filesystem support
< > GFS2 file system support
< > OCFS2 file system support
< > ROM file system support
[ ] Inotify file change notification support
[ ] Quota support
[ ] Dnotify support
< > Kernel automounter support
< > Kernel automounter version 4 support
< > Filesystem in Userspace support
CD-ROM/DVD Filesystems →
DOS/FAT/NT Filesystems →
< > MSDOS fs support
<M> VFAT (Windows-95) fs support
(850) Default codepage for FAT
(iso8859-15) Default iocharset for FAT
< > NTFS file system support

Tabla 3. 18 Configuración de tipos de sistemas de archivos

Además lógicamente se debe dar soporte al sistema de archivos típico en memorias flash, es decir, *JFFS2*, tabla 3.19.

File systems→
Miscellaneous filesystems →
< > ADFS file system support (EXPERIMENTAL)
< > Amiga FFS file system support (EXPERIMENTAL)
< > Apple Macintosh file system support (EXPERIMENTAL)
< > Apple Extended HFS file system support
< > BeOS file system (BeFS) support (read only) (EXPERIMENTAL)
< > BFS file system support (EXPERIMENTAL)
< > EFS file system support (read only) (EXPERIMENTAL)
<> Journalling Flash File System v2 (JFFS2) support*
(0) JFFS2 debugging verbosity (0=quiet, 2=noisy)
[*] JFFS2 write-buffering support
[ ] JFFS2 summary support (EXPERIMENTAL)
[ ] JFFS2 XATTR support (EXPERIMENTAL)
[ ] Advanced compression options for JFFS2
< > Compressed ROM file system support (cramfs)
< > FreeVxFS file system support (VERITAS VxFS (TM) compatible)

< > OS/2 HPFS file system support *< > QNX4 file system support (read only)* *< > System V/Xenix/V7/Coherent file system support* *< > UFS file system support (read only)*

Tabla 3. 19 Sistema de archivos jffs2

Se debe habilitar el arranque del sistema de archivos raíz a través de red, *NFS*, por razones que serán explicadas con posterioridad, tabla 3.20.

File systems → *Network File Systems →* *<*> NFS file system support* *[*] Provide NFSv3 client support* *[ ] Provide client support for the NFSv3 ACL protocol extension* *[ ] Provide NFSv4 client support (EXPERIMENTAL)* *[ ] Allow direct I/O on NFS FILES* *< > NFS server support* *[*] Root file system on NFS* *[ ] Support for rpcbind versions 3 & 4 (EXPERIMENTAL)* *< > Secure RPC: Kerberos V mechanism (EXPERIMENTAL)* *< > Secure RPC: SPKM3 mechanism (EXPERIMENTAL)* *< > SMB file system support (to mount Windows shares etc..)* *< > CIFS support (advanced network filesystem for Samba, Window and others)* *< > NCP file sysem support (to mount NetWare volumes)* *< > Coda file system support (advanced network fs)* *< > Andrew File System support (AFS) (EXPERIMENTAL)*

Tabla 3. 20 Habilitar soporte NFS

Una vez configurado todo el sistema se guarda la configuración resultante y se construye la imagen del *kernel* mediante el comando *ptxdist go*, tras esto se crea la imagen que se grabará en la memoria flash mediante el comando *ptxdist images* guardándose en la carpeta *images/.*

A continuación se debe realizar la configuración del sistema de archivos raíz y más generalmente de ciertas características del proyecto que se está elaborando. La manera de proceder es similar a la utilizada para realizar la configuración del *kernel*. Mediante el comando *ptxdist menuconfig* se puede visualizar una pantalla como la figura 3.16, donde se ofrecen tres opciones:

1. Nombre del proyecto y versión que se desarrolla.

2. Configuración especifica del proyecto. En esta opción se pueden habilitar/deshabilitar la herramienta de testeo de redes *iperf* y una utilidad del planificador de tareas de Linux

denominada *chrt* el cual sirve para manipular los atributos de tiempo real de los procesos. Ambas son activadas por defecto.

3. Requiere una exhaustiva configuración del sistema base y se verá detenidamente.

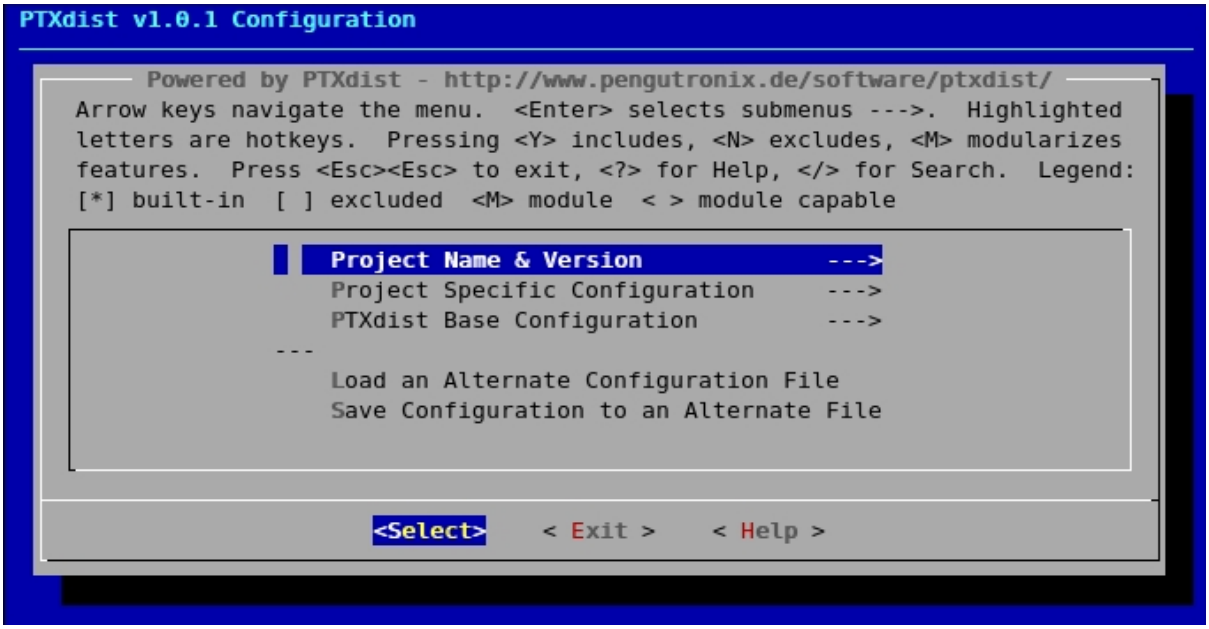


Figura 3. 16 Configuración Sistema de Archivos

El sistema base se divide en tres bloques como se puede apreciar en la figura 3.17. Un primer bloque donde se configuran las características de la arquitectura para la que se construye y las herramientas que intervienen en dicho proceso, un segundo bloque en el que se elige el tipo de imagen a construir, el directorio del sistema y la localización de las carpetas y librerías, y un tercer bloque en el que se seleccionan utilidades o aplicaciones de *Linux* que vayan a ser necesarias en el sistema creado.

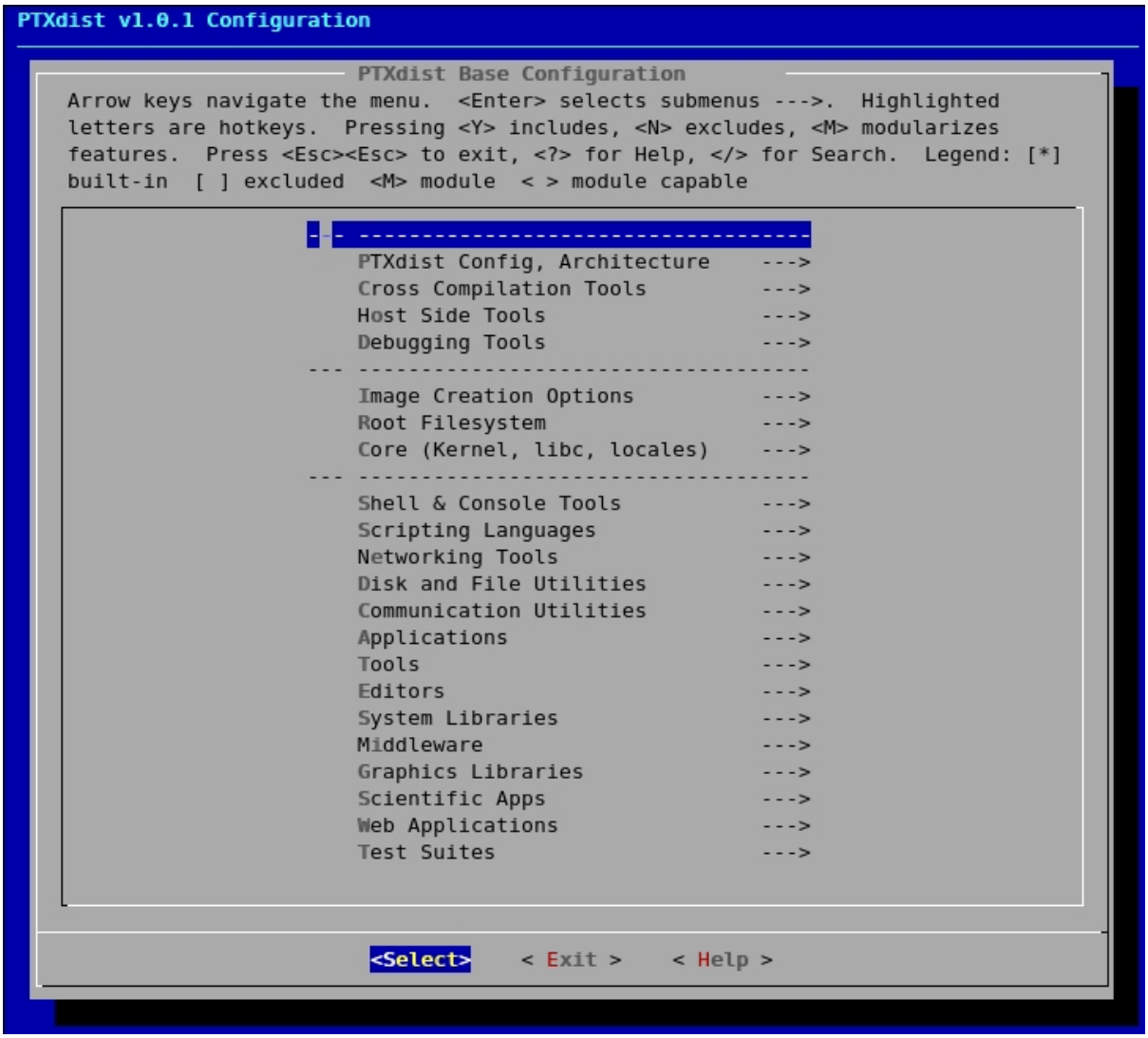


Figura 3. 17 Configuración base

Se puede observar en la tabla 3.21, donde es necesario seleccionar la arquitectura a compilar, tipo de compilador y versión a utilizar, y como son definidas las variables a las que se llama en el proceso de compilación denominadas *FLAGS.* Se muestra como requerida la herramienta que se utilizará en los procesos de depuración de las aplicaciones que se diseñen para ejecutarse en el interior de la tarjeta, denominada *gdbserver*

PTXdist Config, Architecture →	*GNU Toolchain Identification String*
Code Maturity	*(OSELAS.Toolchain-1.1.1) Check for*
[*] Support experimental Stuff	*specific toolchain vendor*
[*] Support even more experimental Stuff	*(4.1.2) Check for specific gcc version*
[ ] Support packets known to be broken	*(arm-iwmmx-linux-gnueabi) GNU*
Target Architecture Options	*Target*
CPU Architecture (ARM) →	***Cross Compilation Tools →***
ARM Family Variant (pxa (little endian)) →	***Host Side Tools →***
Code Optimization (generic arm) →	***Debugging Tools →***
[*] Use Software Floatingpoint	*[ ] cppunit*
Extra Toolchain Options	*[*] gdbserver →*
() Extra CPPFLAGS (cpp)	*[ ] gdb (the real one) →*
() Extra CFLAGS (c)	*[ ] binutils*
() Extra CXXFLAGS (c++)	*[ ] memtest*
() Extra LDFLAGS (ld)	*[ ] memtester*
Paths & Directories	*[ ] ldd*
(${PTXDIST_WORKSPACE}/local) Prefix	*[ ] strace →*
for "install" stage	

Tabla 3. 21 Configuración para ptxdist de la arquitectura del proyecto

En el segundo bloque, que se representa en la tabla 3.22, se puede apreciar la elección del tipo y del nombre de la imagen que se crea sobre el sistema de archivos. Se eligen todos los directorios que se crearán y se deshabilitan los que no vayan a ser necesarios como es el directorio */media*. También se deben habilitar los scripts que se encuentran en el subdirectorio */etc/init.d* que se ejecutan en el inicio del sistema, como es la configuración del servidor Linux de la maquina, y los que se "*llama"* mediante enlaces simbólicos que se encuentran en el subdirectorio */etc/rc.d.* Se debe señalar la versión del *kernel* de *Linux* a utilizar así como la versión del *u-boot* que se construirá y las librerías que vayan a ser necesarias a bordo de la tarjeta, como son las librerías estándar de C++, *libstdc++* y las librerías del compilador *gcc*.

Image creation options →	*(S01_modules) modules link name*
[*] Generate images/root.jffs2	*(S14_banner) banner link name*
Root Filesystem →	*(S08_openssh) openssh link name*
populate rootfs →	*(S05_pureftpd) pureftpd link name*
[*] Create Directories in /	*(S04_thttpd) thttpd link name*
/dev	*(S03_inetd) inetd link name*
[*] create initial device nodes	*(S06_timekeeper) timekeeper link*
[*] /home	*name*
[ ] /media	*config files →*
[*] /mnt	***Core (kernel, libc, locales) →***
[*] /proc	*[ ] klibc →*
[*] /sys	*Linux kernel →*
[*] /tmp	*Build options*
[*] /var	*(2.6.23.1) Linux kernel Version*
start scripts (/etc/init.d) →	*(series) Patch Series File*

[*] Generic Scripts for /etc/init.d *[*] rcS →* *Kind of rcS script (Use generic)* *[*] inetd* *[ ] minimal logrotate* *[*] modules* *[*] networking* *(${PTXDIST_WORKSPACE}/ /projectroot/etc/network/interfaces)* *[*] timekeeper →* *Kind of startup script (Use generic)* *[*] banner →* *(Phytec) vendorname in /etc/init.d/banner* *startscript links (/etc/rc.d) →* *(S00_udev) udev link name*	*(kernelconfig_light.target) kernel config file* *[*] build kernel-modules* *Image Type (uImage) →* *Install options* *[ ] Install kernel into /boot* *[*] Install modules into /lib/modules* *C library →* *[*] glibc →* *gcc libraries →* *libstdc++* *libgcc_s* *[ ] system locales →* *[ ] build U-Boot bootloader for target →* *(1.2.0) U-Boot version* *(phycore_pxa270_config) U-Boot -config target*

Tabla 3. 22 Opciones de creación imágenes, directorio de archivos y kernel

En el tercer bloque, que se puede distinguir en la figura 3.17 de la configuración base, se van a ir comentando detenidamente las herramientas y librerías utilizadas.

En la tabla 3.23 se puede visualizar como se habilitan varios paquetes de herramientas del Shell y la consola, los cuales se definen a continuación:

- *GNU Coreutils* es un paquete de software, desarrollado por el proyecto *GNU* [17], que contiene varias de las herramientas básicas como *cat, ls* y *rm* necesarias para sistemas operativos del tipo Unix. Es una combinación de tres paquetes anteriores: utilidades de ficheros (*fileutils*), utilidades de intérpretes de comandos (*shellutils*) y utilidades de procesadores de texto (*textutils*).

- *Module Init Tools* es un paquete que contiene herramientas para manejar los módulos del núcleo, como por ejemplo: cargar módulos (*insmod o modprobe*), enumerar los módulos cargados (*lsmod*), dar información sobre un módulo (*modinfo*), o descargar un módulo del núcleo cuando no esté en uso (*rmmod*).

- *Hotplug* permite que el sistema reconozca la conexión o desconexión de un dispositivo.

- *Mtd-utils, (Memory Technology Device),* es un subsistema de Linux que provee simples rutinas de lectura, escritura y borrado para manejar dispositivos o memorias *flash*.

- *Top y ps*. El comando *Top* muestra en tiempo real un listado de los procesos que se están ejecutando en el sistema especificando además el porcentaje de memoria que están utilizando, sus *ID's*, usuarios que lo están ejecutando, etc. El comando *ps* imprime por pantalla los procesos que se están ejecutando.

- *Lsbus* y *Systool* son herramientas del sistema que cuando son llamadas sin ningún parámetro presentan los tipos de buses disponibles (*lsbus y systool*) y las clases de dispositivos conectados (*systool*).

- *Fdisk* y *ipcs*. *Fdisk* permite crear particiones en 94 tipos de sistemas de archivos. *Ipcs* es una utilidad que facilita información acerca de las comunicaciones ente los procesos activos, por ejemplo información sobre colas de mensajes, memoria compartida, semáforos, etc.

- *Udev* es el gestor de dispositivos del *kernel* de *Linux*. Es el encargado de dar un nombre fijo a cada dispositivo, no depende del orden en que se conecten, y avisa mediante mensajes *D-BUS* para que cualquier programa del espacio de usuario pueda enterarse cuando un dispositivo se conecta o desconecta. *Firmware helper* es una ayuda para la carga del firmware y tiene que ver con los procesos que maneja *udev* aunque se explicará próximamente.

- *Lsusb* es una utilidad que muestra información sobre los buses *USB* en el sistema y los dispositivos conectados a ellos.

Shell & Console Tools
- *[ ] Bash →*
- *BusyBox →*
- *[*] GNU Coreutils →*
  - *[*] cp*
  - *[*] dd*
  - *[*] md5sum*
  - *[*] seq*
- *[ ] daemonize*
- *[ ] GNU diffutils →*
- *[ ] findutils →*
- *[ ] initng →*
- *[*] Module Init Tools →*
  - *[*] Install insmod on target*
  - *[*] Install rmmod on target*
  - *[*] Install lsmod on target*
  - *[*] Install modinfo on target*
- *[ ] screen →*
- *[ ] setserial →*
- *[ ] sysfsutils →*
- *[*] sysutils →*
  - *[*] install lsbus*
  - *[*] install systool*
- *[*] util-linux →*
  - *[*] fdisk*
  - *[ ] sfdisk*
  - *[ ] cfdisk*
  - *[ ] mkswap*
  - *[ ] swapon*
  - *[*] ipcs*
  - *[ ] readprofile*
  - *[ ] setterm*
- *[*] udev →*
  - *Build options*

[*] Install modprobe on target [*] Install depmod on target [ ] gawk [*] Hotplug → [ ] blacklist [*] firmware agent [ ] . [ ] . [*] mtd-utils → [ ] PCMCIA Utils → [*] procps → [*] top [ ] slabtop [*] ps . .	. Install options . [*] firmware helper [*] USB device id generator Runtime options [*] Install udev.conf Kind of udev.conf (Use generic) → [*] Install user defined rules [*] Install startup script Kind of startup script (Use generic) [ ] xmlstarlet [ ] pciutils [*] usbutils → [*] lsusb

Tabla 3. 23 Herramientas del Shell y la consola

A continuación se definirán las aplicaciones elegidas dentro del subdirectorio de herramientas de red *Network Tools* las cuales se pueden visualizar en la tabla 3.24.

- *OpenSSH*, *Open Secure Shell* es un conjunto de aplicaciones que permiten realizar comunicaciones cifradas a través de una red, usando el protocolo *SSH*, es la alternativa libre y abierta al programa *Secure Shell*, que es *software* propietario.

- *Ping* es la utilidad que comprueba el estado de la conexión con uno o varios equipos remotos por medio de los paquetes de solicitud de eco y de respuesta de eco para determinar si un sistema *IP* específico es accesible en una red.

- *Netcat* permite a través del intérprete de comandos y con una sintaxis sencilla abrir puertos TCP/UDP en un HOST, asociar un S*h*ell a un puerto en concreto y forzar conexiones *UDP/TCP*.

- *Pureftpd* es un servidor *ftp* libre.

- *Tcpdump* es una herramienta en línea de comandos cuya utilidad principal es analizar el tráfico que circula por la red. Permite al usuario capturar y mostrar a tiempo real los paquetes transmitidos y recibidos en la red a la cual el ordenador está conectado.

- *Wireless Tools, WT,* es un conjunto de herramientas que permiten manipular las extensiones inalámbricas *Wireless Extensions (WE)* que son el *API* genérico para manejar

los drivers accediendo desde el espacio de usuario del tipo común de *LANs* inalámbricas. Algunas de estas herramientas son:

- *Iwconfig*, manipula los parámetros característicos inalámbricos.
- *Iwlist*, inicia un escaneo y realiza un listado de frecuencias, *bit-rates*, tipos de encriptación, etc.
- *Iwspy,* muestra la calidad del link/nodo encontrado.
- *Iwpriv,* permite acceder a las extensiones inalámbricas de un driver específico.

```
Scripting Languages→
Network Tools→
        Network Security
    [ ] Dropbear SSH-Server→
    [ ] iptables→
    [ ] iproute2
    [ ] LSH→
        OpenSSL→
    [*] OpenSSH→
            [ ] Install ssh (client)
            [*] Install sshd (server)
            [*]   Install startup script
                  Kind of startup script
                 (Use generic)→
            [ ] Install scp
            [ ] Install sftp-server
            [*] Install ssh-keygen
    [ ] shorewall firewall scripts
        Networkin Apps
    [ ] Apache 2 HTTP Server→
    [ ] betaftpd
    [ ] bind 9→
    [ ] bing
    [ ] Chrony→
    [ ] DHCP→
    [ ] dnsmasq→
    [ ] etherwake
    [ ] eventlog
    [*] inetutils→
              Build options
              Busy box inetd is selected!
            [*] ping
            [ ] rcp
            [ ] rlogind
            [ ] rsh
            [ ] rshd
            [ ] syslogd
            [ ] tftpd
            runtime options
    [ ] libnet
        Libpcap→
    [ ] mii-diag
    [*] netcat→
    [ ] nfsutils→
    [ ] nmap→
    [ ] ntpclient→
    [ ] OpenNTPD→
    [ ] portmapper→
    [ ] ppp→
    [ ] proftpd→
    [*] pureftpd->
            [*] support upload scripts
                Build options
            [ ] virtual hosts
            [*] directory aliases
            [ ] minimal ftpd
                Runtime options
            [*] default config
                Kind of startup (standalone)→
                Kind of startup script (Use generic)→
    [ ] NetKit FTP client
    [ ] rsync→
    [ ] syslog-ng→
    [*] tcpdump→
            [*] enable possibly-buggy SMB printer
            [ ] enable ipv6 (with ipv4) support
    [ ] tcpwrapper→
    [ ] thttpd→
    [ ] wget
        Wireless Tools→
            [*] Wireless Tools
            [*]   Build shared
```

Tabla 3. 24 Herramientas de red

De las herramientas de disco y archivo, tabla 3.25, sólo se ha seleccionado el paquete *e2fsprogs* que es un conjunto de utilidades para mantenimiento de los sistemas de archivos *ext2*, *ext3* y *ext4*, ya que normalmente se considera a este paquete *software* esencial. Dentro de este paquete se usarán:

- *Mke2fs*, usado para crear sistemas de archivos *ext2*, *ext3* y *ext4*.
- *E2fsck*, programa que busca y corrige inconsistencias.

Disk and File Utilities *[ ] bonnie++* *[ ] dosfstools →* *[*] e2fsprogs →* *[*] Install mke2fs* *[*] Install e2fsck* *[ ] Install tune2fs, findfs, e2label*	*[ ] hdparm* *[ ] ipkg →* *[ ] logrotate* *[ ] samba →* *[ ] zip →* *[ ] FUSE*

Tabla 3. 25 Utilidades de disco y fichero

La comunicación se realizará a través del bus *CAN* por lo que se requerirán algunas herramientas de comunicaciones *CAN*, tabla 3.26, tanto para configurar la red como para realizar las pruebas necesarias. Algunas de estas herramientas son:

- *Canconfig*, es utilizado para inicializar y modificar la velocidad de transmisión y recepción del interfaz y la activación o desactivación del interfaz de red *CAN*.
- *Cansend*, sirve para realizar la transmisión de los paquetes mediante el interfaz señalado a la red *CAN*.
- *Candump*, permanece en modo escucha mostrando los paquetes que circulan por la red.
- *Canecho*, muestra un eco de los comandos enviados a través del interfaz indicado.

Communication Utilities *[ ] bluez →* *[*] canutils →* *(2.0.1) Version* *[*] canconfig* *[*] candump* *[*] canecho* *[*] cansend*	*[*] cansequence* *[ ] efax* *[ ] lrzsz* *[ ] mgetty & sendfax →* *[ ] pop3spam* *[ ] smtpclient*

Tabla 3. 26 Utilidades del bus CAN

La herramienta *memedit*, tabla 3.27, aunque no ha sido utilizada podría ser útil ya que permite leer y escribir los registros mapeados en memoria. Las demás aplicaciones disponibles no son requeridas para los objetivos iníciales del sistema por lo tanto son inhabilitados.

Applications	***Tools***
[ ] Alsa Utils	*Figlet*
[ ] GnuPG	*[*] memedit*
[ ] mad →	*[ ] memstat*
[ ] mplayer →	***Editor***
[ ] setmixer	*[ ] hexedit*
[ ] rawrec/rawplay	*[ ] joe*
[ ] midnight commander →	*[ ] nano*
Ffmpeg not supported by strongARM	
[ ] CVS →	

Tabla 3. 27 Aplicaciones

La mayoría de librerías del sistema no se seleccionan o son deshabilitadas ya que en este sistema no se van a realizar los desarrollos, la compilación se va a realizar en el *host* de sobremesa, y tampoco se necesitan librerías de audio como son las *Alsa Libraries*, tabla 3.28. *Termcap* es una librería *software* y base de datos y debe ser habilitada ya que es usada en todos los sistemas de tipo *Linux*. Es la encargada de habilitar la visualización de los programas desde los terminales del ordenador y además contiene en la base de datos las características de distintas formas de poder visualizar el contenido a través de los terminales.

System Libraries	*[ ] libxslt →*
[ ] Alsa Libraries →	*[ ] mysql →*
[ ] boots →	*ncurses →*
[ ] CommonC++ 2 →	*build options*
[ ] Berkeley DB-4.1 →	*[ ] Enable wide char support*
[ ] Berkeley DB-4 (4.4) →	*[*] Enable the big core*
[ ] expat XML parser library	*Install Options*
[ ] gettext (GNU) →	*[*] Install minimal set of termcap data files*
Libusb	*[ ] Install libform on the target*
[ ] libdaemon	*[ ] Install libmenu on the target*
[ ] libelf	*[ ] Install libpanel on the target*
[ ] libezv24	*[ ] PCRE*
[ ] libgsloop	*Readline →*
[ ] libiconv →	*Termcap library to be used (ncurses) →*
[ ] liblist →	*[ ] S-Lang*
[ ] libmqueue	*[ ] sqlite →*

[ ] libnetpbm →	[ ] termcap →
[ ] libpopt →	[ ] xerces →
[ ] librn	Zlib
[ ] libxml2 →	
[ ] libxmlconfig	

Tabla 3. 28 Librerías del sistema

Por último se puede apreciar en la tabla 3.29 la descarga o ligereza del núcleo, como también se ha venido viendo en el resto de tablas. No se utilizan ningún tipo de librerías referentes a entornos gráficos ya que las aplicaciones en el sistema se ejecutarán en modo consola, además los entornos gráficos y los entornos basados en escritorio *Desktop* aumentan las latencias del sistema y no son compatibles con las capacidades de tiempo real.

Middleware	[ ] xterm
[ ] dbus →	X Window Managers
[ ] dbus C++ bindings	[ ] pekwm
Graphics Libraries	Framebuffer
Xorg →	[ ] fbtest
Overall X11 options →	[ ] fbutils →
[ ] Support keyboard mappings	Other Stuff
(/usr/lib) Default location for X data files	[ ] SDL →
[ ] IPv6 support	[ ] libpng
[ ] xorg server →	[ ] libjpeg
[ ] xorg input drivers →	[ ] Fontconfig →
[ ] xorg video drivers →	[ ] tslib (touch library) →
[ ] xorg applications →	[ ] QTopia Core →
[ ] xorg libs →	[ ] i855resolution
[ ] xorg data →	[ ] i915resolution
[ ] xorg Support Libraries →	***Scientific Apps***
X Applications	[ ] fftw →
GTK+ and friends →	[ ] gnuplot →
[ ] FLTK GUI Toolkit →	[ ] libmodbus →
[ ] xli	[ ] libpv →

Tabla 3. 29 Middleware, librerías gráficas y aplicaciones científicas

A lo largo de este apartado se ha explicado y mostrado cómo se debe configurar el sistema para por un lado tener un sistema con todos los dispositivos funcionando correctamente y por otro lado obtener la máxima eficiencia en cuanto a espacio y carga de procesos se refiere, minimizando las opciones en todo lo posible.

A continuación, se hará un resumen de dispositivos y herramientas operativos en el sistema, tablas 3.30 y 3.31.

Dispositivos Operativos en PhyCORE PXA270	
Soporte para la tarjeta PCM-990, Marvell Xscale PXA270, 520Mhz *Kernel 2.6.23.1 y soporte RT-Preempt para tiempo real estricto, "hard realtime"* *Soporte del kernel y del sistema de archivos root en memoria flash* *Soporte de comunicaciones a través de BSD sockets del controlador CAN SJA100* *Soporte I2C y SPI* *Soporte I2C para RTC y acceso a la EEPROM*	*Acceso a GPIO y 1xPWM* *USB host 1.1, MMC/SD , Ethernet SMC91C111 y RS232* *Soporte para Sockets y protocolos TCP/IP, DHCP y 802.11* *Soporte de sistema de archivos ext3* *Soporte de sistema de archivos raíz en red (NFS)*

Tabla 3. 30 Dispositivos operativos en el sistema

Configuración del Sistema de archivos y aplicaciones en PhyCORE PXA270	
GNU Toolchain-1.1.1 *Gcc version 4.1.2* *U-Boot versión 1.2.0* *Shell and Console Tools:* • *GNU Core Utils* • *Module Init Tools* • *Hot-plug* • *Mtd-utils* • *Top y ps* • *Sys-utils*	*CPU Architecture (ARM)* *GNU Target arm-iwmmx-linux-gnueabi* *Debugger Tools:* • *Gdbserver* *Network Tools:* • *OpenSSH* • *Ping* • *Netcat* • *Pureftpd* • *Tcpdump* • *Wireless Tools*

• *Fdisk e ipcs* • *Udev: Firmware Helper* *Disk and File Utilities:* • *Mke2fs* • *E2fsck* *Tools:* • *memedit*	*Communication Utilities:* • *Canutils* *System Libraries:* • *Termcap*

Tabla 3. 31 Configuración del sistema de archivos y aplicaciones disponibles

A pesar de tener un sistema operativo capaz de reconocer todos los dispositivos necesarios para la aplicación que se va a ejecutar a bordo del robot y que se encuentran integrados en la tarjeta *phyCORE*, debido a las comunicaciones exteriores inalámbricas, las cuales hay que integrar en el robot para poder mantener la comunicación con los dispositivos del entorno como la *PDA*, es necesaria la instalación de un *driver* adicional a los que integra el sistema la placa.

El driver a instalar depende del dispositivo que vaya a ser utilizado y del chipset que integra éste, no obstante se utiliza un proceso inverso al normalmente pensado, es decir, no se busca el driver del dispositivo que se va a utilizar una vez que se tiene el dispositivo, sino que se busca un driver que sea operativo en el sistema y después se selecciona el dispositivo que funciona con dicho driver. Esto se realiza así por dos razones fundamentales:

1. Linux en su mayoría no es comercial y por lo tanto todos los dispositivos comerciales no poseen drivers que los hagan funcionar en este sistema operativo. De hecho, la mayoría de los fabricantes no facilitan los drivers de sus dispositivos para utilizarlos en sistemas operativos *Linux*.

2. Aún obteniendo los *drivers* compatibles con sistemas operativos *Linux* dichos drivers han sido diseñados normalmente para ejecutarse en *PC's* de sobremesa, es decir, arquitecturas *x86*, sin embargo, la arquitectura del sistema embebido es *ARM* lo que restringe bastante las opciones disponibles.

El driver utilizado finalmente es el driver *RT73* basado en tarjetas de red inalámbricas *USB*, y es utilizado para dispositivos *USB Wi-Fi* con chipset de Ralink [18].

Normalmente para instalar un driver en un sistema operativo los pasos a seguir son:

1. Compilación del driver. Ejecutar el comando *make* en el directorio donde se encuentra el archivo *makefile*, aclarando que éste es un fichero de texto que utiliza la herramienta *make* para llevar la gestión de la compilación de programas. La compilación crea el módulo del driver, que es un archivo con extensión *.ko*.

2. Instalación del módulo. Una vez compilado el driver el módulo se instala en el sistema mediante la instrucción *make install*.

3. Cargar el módulo. Aunque el módulo este instalado si se desea utilizar hay que activarlo/cargarlo mediante el comando *insmod nombre_modulo.ko*.

En este caso antes de compilar el driver se deben hacer unos cambios en el fichero *makefile*, es decir, debemos indicarle al módulo información que desconoce puesto que está preparado para compilarse en la misma plataforma donde se va a ejecutar y toma las variables respecto del sistema donde está siendo compilado, como en este caso, se realiza una compilación cruzada habrá que facilitar la información sobre el sistema donde se va a ejecutar dicho dispositivo.

- Definir variables que serán exportadas con la información acerca del tipo de arquitectura, *ARCH*, donde se va a ejecutar el módulo y el tipo de compilador cruzado, *CROSS_COMPILE*, que se va a utilizar.

- Redefinir la variable, *CC*, que define el nombre del compilador que se utiliza incluyendo la localización de las librerías que se han utilizado para compilar el sistema operativo.

- Indicar donde se encuentran las fuentes del *kernel*, *kernel_sources*, específicas del *kernel* que se instalará en la tarjeta ya que sino utilizará las fuentes del PC donde se está compilando.

A modo resumen, se puede visualizar en la tabla 3.32 las diferentes líneas que hay que introducir al inicio del *makefile* original para proporcionar la información necesaria durante la compilación. No obstante, los directorios mostrados no deben coincidir exactamente ya que depende de la localización de los directorios del proyecto.

```
ARCH=arm
CROSS_COMPILE=arm-iwmmx-linux-gnueabi-
export ARCH CROSS_COMPILE
INC =/home/rpalencia/OSELAS.BSP4/OSELAS.BSP-Phytec-phyCORE-PXA270-4/build-target/linux-
2.6.23.1/include
CC = arm-iwmmx-linux-gnueabi-gcc-4.1.2 -I$(INC)
KERNDIR=/home/rpalencia/OSELAS.BSP4/OSELAS.BSP-Phytec-phyCORE-PXA270-4/build-target/linux-2.6.23.1
```

Tabla 3. 32 Cambios en el makefile del driver del dispositivo Wi-Fi

Tras modificar el fichero *Makefile* se puede realizar la compilación, siempre y cuando las características del *kernel* se encuentren en modo *Voluntary Kernel Preemption (Desktop)*. Al cambiar las características del *kernel* se deben recompilar todos los módulos, y como se ha indicado anteriormente interesa habilitar las capacidades de tiempo real configurando el sistema en modo *Complete Preemption (Real-Time)*. En este modo, sorprendentemente el driver no permite la compilación y falla en su intento.

Analizando los errores puede comprobarse que el fallo es debido a una función denominada *init_MUTEX_LOCKED(semaphore)* que se encarga de inicializar el semáforo correspondiente y dejarlo bloqueado. Esta función está disponible si no se aplican las capacidades de tiempo real, sin embargo, al aplicar las capacidades de tiempo real dicha función no está definida, esto puede comprobarse en la librería *linux-2.6.23.1/include/Linux/rt_lock.h* donde indica explícitamente en la línea 147 que no existe la función *init_MUTEX_LOCKED(semaphore)* para semáforos de tiempo real, como puede verse en la figura 3.18.

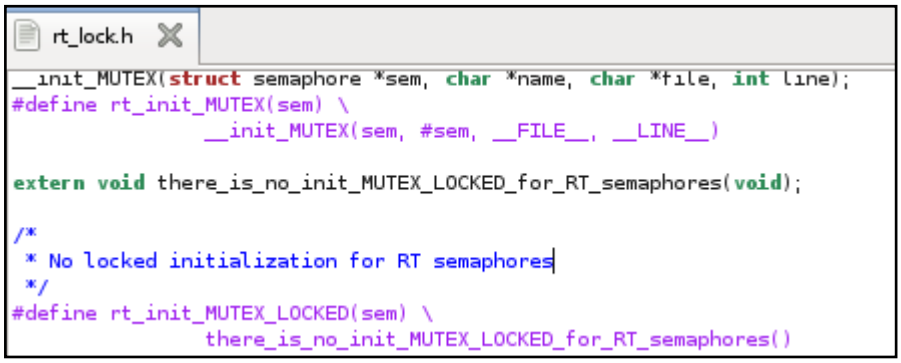


Figura 3. 18 Función no definida en RT Preempt

Para solucionar este error se debe sustituir dicha función por otras dos que complementan la función equivalente de la anterior. Estas dos funciones son:

- *Init_MUTEX(semaphore)*, inicializa el semáforo.
- *Down(semaphore)*, bloquea el semáforo una vez inicializado.

Por lo tanto, entre los diferentes archivos que se compilan en la construcción del driver deben modificarse tres líneas del archivo denominado *rtmp_init.c* como se muestra en la figura 3.19, donde se encuentran comentadas las funciones que no están definidas y escritas las funciones que realizan el equivalente al anterior.

```
//init_MUTEX_LOCKED(&(pAd->usbdev_semaphore));  //Función no definida
init_MUTEX(&(pAd->usbdev_semaphore));     //Funciones que conjuntamente
down(&(pAd->usbdev_semaphore));           //equivalen a la original

//init_MUTEX_LOCKED(&(pAd->mlme_semaphore));    //Función no definida
init_MUTEX(&(pAd->mlme_semaphore));       //Funciones que conjuntamente
down(&(pAd->mlme_semaphore));             //equivalen a la original

//init_MUTEX_LOCKED(&(pAd->RTUSBCmd_semaphore));      //Función no 8efinida
init_MUTEX(&(pAd->RTUSBCmd_semaphore));   //Funciones que conjuntamente
down(&(pAd->RTUSBCmd_semaphore));         //equivalen a la original
```

Figura 3. 19 Modificación de funciones no definidas

Tras realizar los cambios del fichero *Makefile* y la sustitución de funciones no definidas en el código por unas que realizan la acción equivalente, mediante el comando *make* se construirá el módulo *rt73.ko* el cual se utilizará como driver para las comunicaciones inalámbricas del dispositivo *USB* de linksys modelo WUSB54G [19], figura 3.20.

Figura 3. 20 USB Wi-Fi modelo WUSB54G

Aún teniendo el driver correctamente construido, al cargarlo no funcionará por sí sólo ya que al driver le acompaña un *firmware*. Cuando el driver es activado pregunta al gestor de dispositivos *udev* por dicho *firmware* por ello se debe habilitar el *firmware helper*, mediante el comando *ptxdist menuconfig* explicado anteriormente, que asista y permite la carga de *firmware* y además se debe editar el fichero de reglas *udev.rules* que se encuentra en el subdirectorio *projectroot/etc/udev/rules.d/udev.rules*, añadiendo al final la siguiente línea: "*SUBSYSTEM==”firmware”, RUN+=”/sbin/firmware.sh””*, figura 3.21, de esta forma los cambios se harán persistentes en el proyecto y se instalarán con la imagen del sistema.

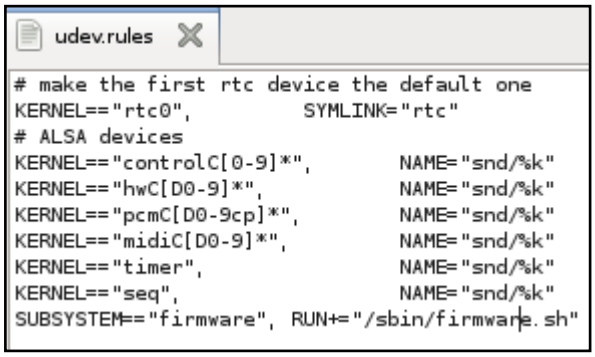


Figura 3. 21 Editar reglas del gestor de dispositivos udev

Tras esto ya se puede proceder a la recompilación del *kernel* teniendo incluida en su configuración la aceptación de *firmware* de los dispositivos para después poder proceder a la carga del driver del dispositivo inalámbrico descrito.

3.4. Modos NFS y StandAlone. Automatización del Inicio.

Una vez compilado el *kernel* y construido tanto la imagen del *kernel* como la imagen del sistema de archivos raíz, existen dos maneras de utilizar la placa:

1. Iniciar la tarjeta desde el ordenador de desarrollo, vía *Ethernet*, mediante *NFS*, figura 3.22.

2. Grabar el sistema en la tarjeta y trabajar en modo *stand-alone*, es decir, que la tarjeta funcione con total independencia, figura 3.23.

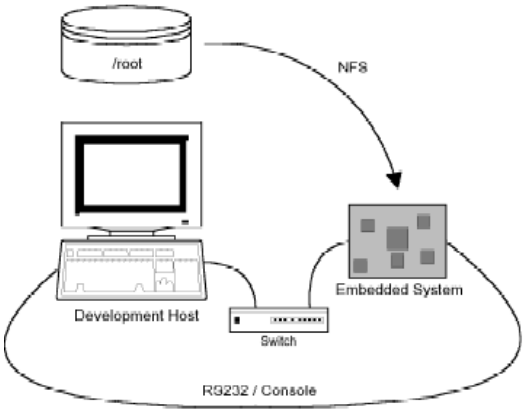


Figura 3. 22 Sistema raíz en NFS

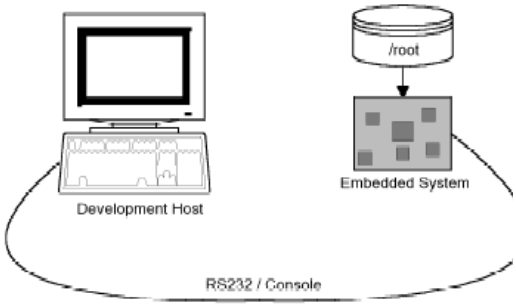


Figura 3. 23 Modo Stand-Alone

En el desarrollo del sistema se han utilizado ambos modos el primero inicialmente, a modo de pruebas, y el segundo para grabar de manera permanente las imágenes y aplicaciones en memoria, por lo tanto se explicarán los procedimientos que se deben llevar a cabo para utilizar cada uno de ellos y las ventajas o desventajas que supone cada uno.

El primer modo se basa en el protocolo *Network File System* es un protocolo de nivel de aplicación, según el modelo *OSI* [20]. Es utilizado para sistemas de archivos distribuidos en un entorno de red de computadoras de área local. Posibilita que distintos sistemas conectados a una misma red accedan a ficheros remotos como si se tratara de locales.

El sistema *NFS* está dividido al menos en dos partes principales: un servidor y uno o más clientes. Los clientes acceden de forma remota a los datos que se encuentran almacenados en el servidor. Las estaciones de trabajo locales utilizan menos espacio de disco debido a que los datos se encuentran centralizados en un único lugar pero pudiendo ser accedidos y modificados por varios usuarios, de tal forma que no es necesario replicar la información.

Todas las operaciones que se realizan mediante este protocolo sobre ficheros son síncronas, lo que significa que la operación sólo retorna cuando el servidor ha completado todo el trabajo asociado para esa operación. En caso de una solicitud de escritura, el servidor escribirá físicamente los datos en el disco, y si es necesario, actualizará la estructura de directorios, antes de devolver una respuesta al cliente. Esto garantiza la integridad de los ficheros.

Por lo tanto utilizando *NFS* el host de desarrollo es conectado a la tarjeta *phyCORE PXA270* con un cable serie *nullmodem* y otro cable *Ethernet*, la tarjeta inicia el *bootloader*, entonces envía

una petición *TFTP* a través de la red e inicia el *kernel* mediante el servidor *TFTP* del *host*. Después de descomprimir el *kernel* en la *RAM* e iniciarse, el *kernel* monta el sistema de archivos raíz vía *NFS* desde la localización original *root/* del *workspace* de *ptxdist*.

Este método presenta la ventaja de que no es necesario *flashear* la memoria, o lo que es lo mismo grabar en *flash* las imágenes, para poder visualizar los archivos en la tarjeta. El directorio que se visualizará será el mismo que se encuentra en la carpeta *root/* como se ha citado anteriormente. Esto es especialmente útil durante la fase de desarrollo del proyecto, ya que aunque se eligen y se deshabilitan las opciones con el mayor criterio posible será un proceso bastante duradero de pruebas en el que se realicen cambios frecuentemente y de esta forma también se consigue un ahorro de tiempo, en cuanto que evitamos el proceso de descarga de imágenes a la placa y grabación en *flash* para visualizarlas como se explicará posteriormente.

El *host* de desarrollo debe ser preparado para hacer uso del procedimiento descrito. Un servidor *TFTP* debe ser instalado y configurado en este *host*, ubicando las imágenes necesarias en lugares específicos del *host* y configurándolo para que de soporte al arranque realizado desde él.

Un servidor *TFTP* usual puede ser el programa *tftpd* que generalmente se puede descargar e instalar mediante el comando de consola *apt-get install tftpd*. Una vez instalado se debe habilitar añadiendo a */etc/inetd.conf* la siguiente línea:

"*tftp dgram udp wait nobody /usr/sbin/tcpd in.tftpd /tftpboot*"

Se debe tener en cuenta que los servidores *TFTP* utilizan la carpeta */tftpboot* como directorio del que servir las imágenes por motivos históricos. Sin embargo, algunos paquetes *Debian GNU/Linux* pueden utilizar otros directorios para cumplir con el estándar de jerarquía de sistemas de ficheros [21]. Por ejemplo, algunos paquetes utilizan por defecto el directorio */var/lib/tftpboot*. Una vez editado se debe recargar el demonio de red para que tenga en cuenta los cambios realizados mediante la ejecución en consola del comando *inetd reload*.

Además del servidor *TFTP* el arranque de la tarjeta *phyCORE* exige que se realice a través de un servidor *DHCP*, o *BOOTP*. *BOOTP* es un protocolo *IP* que informa al ordenador de su dirección *IP* y desde dónde puede obtener una imagen de arranque en la red. *DHCP* (*Dynamic Host Configuration Protocol*, o protocolo de configuración dinámica de equipos) es una extensión de *BOOTP* compatible de éste, pero más flexible.

Se recomienda el uso del paquete *dhcp3-server* en *Debian GNU/Linux*, que como en el caso anterior se consigue mediante el comando *apt-get install dhcp3-server*. Para activar este paquete y configurarlo adecuadamente para la aplicación se ha de editar el fichero */etc/dhcp3/dhcpd.conf*, introduciendo los siguientes datos:

```
Subnet 192.168.3.0 netmask 255.255.0 {
range dynamic-bootp 192.168.3.2 192.168.3.10;
option broadcast-address 192.168.3.255;
server-name "192.168.3.10"
option routers 192.168.3.1;
}
```

Con esta descripción se indica que la subred en la que se van a localizar tanto la tarjeta *phyCORE* como el host de desarrollo va a ser la 192.168.3.0, que el *host* de desarrollo va a tener la *IP* 192.168.3.10, debe ser la misma que se establece en el fichero *uboot*, y va a servir a la tarjeta *phyCORE* una dirección que estará dentro del rango 192.168.3.2-10. Una vez configurado habrá que iniciarlo mediante el comando */etc/init.d/dhcp3-server start*.

Tras esto se deberá ubicar la imagen *TFTP* de arranque que se necesita en el directorio de arranque de *tftpd*, es decir se deberá copiar la imagen al directorio */tftpboot*. Se debe tener en cuenta que para escribir en el directorio */tftpboot* se requieren privilegios de *root*, o usuario administrador, y que el nombre de la imagen definido en *uboot* es *uImage* por lo que habrá que modificar el nombre que en principio se crea como *linuximage*. Los comandos para realizar esto se muestran a continuación:

```
su
password_root
cp images/linuximage /tftpboot/uImage
```

Por último, hay que permitir el acceso al directorio donde se encuentra el sistema de archivos raíz o lo que es lo mismo habilitar su exportación editando el fichero */etc/exports* con la línea siguiente:

```
"/home/rpalencia/OSELAS.BSP-Phytec-phyCORE-PXA270-4/root
192.168.3.9/255.255.255.0(rw,no_root_squash,sync) 192.168.3.9"
```

De esta manera se permite acceder al directorio al dispositivo con la *IP 192.168.3.9*, es decir a la tarjeta *phyORE* que le será servida esta *IP* por el servidor *DHCP*.

Cuando se inicia la tarjeta *phyCORE* a través del terminal serie se debe definir la variable de entorno *nfsrootfs*. Esta variable proporciona la información de la localización del sistema de archivos que se debe cargar, por lo tanto se debe teclear en dicho terminal la inicialización de la variable mediante:

"setenv nfsrootfs '/home/rpalencia/OSELAS.BSP-Phytec-phyCORE-PXA270-4/root'"

Una vez definida la variable *nfsrootfs* es posible iniciar el sistema tecleando en el terminal serie el comando *run bootcmd_net*. Puede suceder que se produzca un fallo al cargar el sistema de archivos y se debe a la falta de permisos ya que todo el proyecto se crea con permisos de usuario pero los dispositivos de la placa solo se pueden ejecutar con permisos de administrador, este error se soluciona haciendo una copia de dispositivos en el sistema de archivos con permisos de administrador como se muestra a continuación:

```
su
password_root
cp –a /dev/{console,null} root/dev/
```

El segundo modo consiste en proveer a la tarjeta *phyCORE* de todos los componentes necesarios para que pueda ejecutarse por sí misma. El *kernel* de *Linux* y el sistema de archivos raíz son grabados en la memoria *flash* de la tarjeta, lo que significa que el único cable necesario a conectar es el *nullmodem*, o el de *Ethernet*, y únicamente si es necesario visualizar lo que ocurre en la tarjeta.

Normalmente después de trabajar con el sistema de archivos durante algún tiempo mediante *NFS*, el sistema de archivos debe grabarse para que se ejecute de manera permanente en la memoria *flash* de la tarjeta *phyCORE* sin volver a requerir una infraestructura de red para realizar sus funciones.

La única preparación necesaria para realizar la grabación de las imágenes en memoria *flash* es una conexión *Ethernet* a la tarjeta embebida y una red desde la cual el *bootloader* pueda obtener las imágenes del servidor *TFTP*. Por lo tanto, para usar este modo, o mejor dicho para realizar la descarga que facilita este modo y así realizar la grabación en memoria flash, es necesario al igual que

anteriormente tener instalado un servidor *TFTP* en el ordenador donde se encuentren las imágenes a descargar.

Antes de indicar el procedimiento para realizar la copia en memoria de las imágenes es vital señalar que la versión 4 del *BSP* de *OSELAS* tiene un error el cual impide el arranque de las imágenes grabadas por una mala configuración de las particiones. Para solucionar esto se debe ejecutar en el directorio de *OSELAS.BSP-Phytec-phyCORE-PXA270-4* el comando *ptxdist boardsetup* y cambiar los valores por defecto en la opción *Target boot Settings* por los siguientes valores:

"0x00000000:0x00040000:0x00440000:0x02000000"

Esta línea indica la dirección donde comienza la memoria flash, *0x00000000* donde es grabado el *bootloader*; donde empieza la partición donde será grabada la imagen del *kernel* y que además es a esta dirección a la que "*salta"* el *bootloader* para arrancar el sistema, *0x00040000*; el comienzo de la partición del sistema de archivos, *0x00440000*; y la dirección en la que finaliza la memoria *flash*, *0x020000000*.

Una vez solucionado este error puede procederse a la grabación en memoria *flash*. Lo primero que se debe hacer es copiar las imágenes linuximage, cambiando su nombre a *uImage*, y *root.jjfs2* al directorio */tftpboot*. A continuación el procedimiento consistirá en descargar las imágenes a la memoria *RAM* de la tarjeta *phyCORE* y a continuación copiarlas en las particiones que les han sido asignadas. En la figura 3.24 pueden visualizarse todos los comandos que deben teclearse para la grabación de las imágenes en la memoria *flash*.

En la consola de comandos del host de desarrollo deben teclearse los siguientes comandos:

```
su
password_root
cp /home/rpalencia/OSELAS.BSP-Phytec-phyCORE-PXA270-4/root/linuximage /tftpboot/uImage
cp /home/rpalencia/OSELAS.BSP-Phytec-phyCORE-PXA270-4/root/root.jffs2 /tftpboot/
```

En el terminal serie deben teclearse los siguientes comandos:

```
tftp 0XA3000000 uImage
cp.b 0XA3000000 0x040000 $(hexadecimal_filesize)
tftp 0xA3000000 root.jffs2
cp.b 0xA3000000 0x440000 $(hexadecimal_filesize)
boot
```

Figura 3. 24 Grabación en memoria flash

Completadas las configuraciones y grabadas en *flash* las imágenes, para obtener el sistema completo de manera que las aplicaciones a bordo del robot puedan ejecutarse con total autonomía deben realizarse unos últimos ajustes ya que el sistema grabado en *flash* no provee estas cualidades de por sí.

Por un lado se debe cargar el módulo del dispositivo *Wi-Fi* para poder ser habilitado, ya que este módulo es independiente de las imágenes, y por otro lado puesto que esta tarjeta va a localizarse a bordo de un robot y la aplicación en su interior debe responder por sí sola debe ser configurada para que sin ayuda de un usuario se activen todos los procesos de manera automática al alimentar el dispositivo.

Lo primero a realizar debe ser la descarga a la tarjeta *phyCORE* el driver compilado del dispositivo *Wi-Fi* y su *firmware*. La descarga se realizará mediante el protocolo de red *FTP* (*File Transfer Protocol o* Protocolo de Transferencia de Archivos). Este protocolo facilita la transferencia de archivos entre sistemas conectados a una red *TCP*, basado en la arquitectura cliente-servidor. Permite desde un equipo cliente la conexión a un servidor para descargar archivos desde él o para enviarle archivos, con independencia del sistema operativo en el que se ejecuten los diferentes equipos.

Conectando el cable *Ethernet* a la tarjeta y al host de desarrollo donde se encuentran los archivos que se quieren descargar, se conectan los ordenadores mediante el protocolo descrito , teniendo en cuenta que la *IP* de conexión es la definida en las opciones que se encuentran en la configuración del *boardsetup*, y se envían los paquetes como se describe en la figura 3.25.

ftp 192.168.3.11
user: root
password: vacio
put /directorio_contenedor/rt73.ko
put /directorio_contenedor/rt73.bin /lib/firmware/

Figura 3. 25 Descarga del driver USB_Wi-Fi

Una vez descargado el driver que activa el dispositivo *Wi-Fi* se pasará a definir la autonomía del sistema. Se definirá autonomía del sistema como la capacidad del mismo de responder a las

acciones externas sin intervención humana exceptuando únicamente esta intervención en el encendido del interruptor que alimenta el sistema. Es decir, al alimentar el robot la tarjeta *phyCORE* se encenderá, su sistema operativo se iniciará, el dispositivo *Wi-Fi* se conectará a la red inalámbrica para mantener las comunicaciones con el exterior, y la aplicación que interpreta los mensajes externos recibidos y controla los dispositivos conectados a la red de comunicaciones interna del bus *CAN* se ejecutará.

La ejecución de estos elementos sin que medie un operador es posible mediante un conjunto de órdenes definidas en archivos denominados s*cripts*. Este archivo es un programa usualmente simple, que generalmente se almacena en un archivo de texto plano y casi siempre son interpretados. El sistema al iniciarse debe ejecutar estos ficheros junto a los definidos por defecto como por ejemplo la carga de todos los drivers del sistema o la configuración de la tarjeta de red.

La ejecución de los ficheros al inicio del sistema tiene que ser indicada en el subdirectorio */etc/rc.d* mediante enlaces simbólicos a los ficheros de interés los cuales deberán localizarse en el subdirectorio */etc/init.d*.

Los enlaces simbólicos deben contener en el nombre un número que determinará el orden de ejecución de cada uno y apuntar al *script* al que se refieren. Primero debe dejarse que se inicien todos los servicios de red y los módulos que habilitan los dispositivos integrados en la tarjeta y a continuación se indicará el inicio de los scripts definidos por las necesidades del proyecto.

En la figura 3.26 puede observarse la prioridad de cada uno de los *scripts* que deben ejecutarse en el inicio del sistema operativo. Puede apreciarse un orden lógico empezando por la activación del gestor de dispositivos *udev* y continuando con los drivers del sistema *modules*; comunicaciones, protocolos y aplicaciones de red como *networking*, *inetd*, *pureftpd*; y habilitación del grupo de entradas y salidas *gpio*. Los *scripts* creados son *arranque* y *server* y comienzan después de habilitar las entradas y salidas porque como se describirá a continuación se hace uso de éstas en el *script*. Los enlaces simbólicos se crean tecleando dentro del subdirectorio */etc/rc.d/* el siguiente comando:

ln –s /etc/init.d/NombreScript S_NºNombreScript

```
root@phyCORE-PXA270:/etc/rc.d ls -n
S00_udev        S02_networking  S04_thttpd      S06_timekeeper  S12_gpio        S14_banner
S01_modules     S03_inetd       S05_pureftpd    S08_openssh     S13_arranque    S15_server
root@phyCORE-PXA270:/etc/rc.d ls -l
lrwxrwxrwx    1 root     root           14 Apr 10  2008 S00_udev -> ../init.d/udev
lrwxrwxrwx    1 root     root           17 Apr 10  2008 S01_modules -> ../init.d/modules
lrwxrwxrwx    1 root     root           20 Apr 10  2008 S02_networking -> ../init.d/networking
lrwxrwxrwx    1 root     root           15 Apr 10  2008 S03_inetd -> ../init.d/inetd
lrwxrwxrwx    1 root     root           16 Apr 10  2008 S04_thttpd -> ../init.d/thttpd
lrwxrwxrwx    1 root     root           19 Apr 10  2008 S05_pureftpd -> ../init.d/pure-ftpd
lrwxrwxrwx    1 root     root           21 Apr 10  2008 S06_timekeeper -> ../init.d/timekeeping
lrwxrwxrwx    1 root     root           17 Apr 10  2008 S08_openssh -> ../init.d/openssh
lrwxrwxrwx    1 root     root           14 Apr 10  2008 S12_gpio -> ../init.d/gpio
lrwxrwxrwx    1 root     root           20 Nov 30 01:05 S13_arranque -> /etc/init.d/arranque
lrwxrwxrwx    1 root     root           16 Apr 10  2008 S14_banner -> ../init.d/banner
lrwxrwxrwx    1 root     root           18 Nov 30 01:05 S15_server -> /etc/init.d/server
```

Figura 3. 26 Enlaces simbólicos a scritps de inicio

Los *scripts* definidos cumplen cada uno un objetivo diferente, pero ambos deben cumplir unas reglas, estas reglas consisten en una línea inicial y final común a todos y además se les debe conceder permisos de ejecución tecleando en la consola dentro del subdirectorio donde esta definido el fichero:

"chmod +x NombreFichero"

En la figura 3.27 se puede apreciar la configuración estándar de cualquier script genérico.

```
#!/bin/sh
#Ordenes a ejecutar por el script
exit
```

Figura 3. 27 Configuración de un general de un script

El fichero que se ejecutará en primer lugar se denomina *arranque* y cumple cuatro funciones:

- Activación de señal digital. Señal indicadora conectada a un *led* que informa de que el sistema operativo se ha iniciado. Esta señal debe ser mapeada en memoria como salida y ser activada a nivel lógico 1.
- Activación de señal *pwm*. Señal de comprobación de funcionamiento que se utilizó durante el desarrollo del *hardware*. Configurada con un periodo de 5 ms y un ciclo de trabajo del 50%.

- Visualización en el terminal serie de la etiqueta "Ejecutándose" lo que sirve de información para tener conocimiento de que el script esta ejecutándose.

- Conexión inalámbrica de la tarjeta mediante el dispositivo *Wi-Fi*. Carga el driver *rt73.ko* que permite el funcionamiento del dispositivo inalámbrico, se activa la tarjeta de dicho dispositivo, se conecta a la red de interés denominada *Connect* y se le asigna una dirección IP.

El contenido del fichero donde constan las cuatro funciones definidas se muestra en la figura 3.28.

```
#!/bin/sh
Echo 11:out:lo > /sys/class/gpio/map_gpio
Echo 1 > /sys/class/gpio/gpio11/level
Echo 5000 > /sys/class/pwm/pwm1/period
Echo 500 > /sys/class/pwm/pwm1/duty
Echo 1 > /sys/class/pwm/pwm1/active
Echo > "Ejecutandose"
Insmod /home/rt73.ko
Ifconfig wlan0 up
Iwconfig wlan0 essid Connect
Ifconfig wlan0 222.222.1.2
exit
```

Figura 3. 28 Configuración del script arranque

El fichero que se ejecuta en segundo lugar es el denominado *server*, figura 3.29, y su único objetivo es iniciar la ejecución de la aplicación que se encargará de que el robot responda a las órdenes recibidas del exterior a través de las comunicaciones inalámbricas, decodificándolas y ejecutándolas mediante el manejo de las comunicaciones internas a través de la red del bus *CAN*.

```
#!/bin/sh
BINARY=/home/AsibotDevelop
test -f$BINARY || {echo "$BINARY not found" >&2; exit 0;}
start_proc(){
        echo -n "Starting Asibot....."
        $BINARY &
        echo "DONE"
stop_proc(){
        echo -n "Stopping Asibot"
        killall server
        echo "DONE"
}
case "$1" in
        start)
                start_proc
```

```
                ;;
        stop)
                stop_proc
                ;;
        restart|force-reload)
                echo -n "Restarting Asibot...."
                stop_proc
                sleep2
                start_proc
exit
```

Figura 3. 29 Configuración del script server

Por lo tanto, para preparar la plataforma *hardware phyCORE* y poder llevar a cabo la migración posterior del código fuente del programa de control del robot ha sido necesario realizar los siguientes pasos:

1. Programar un *bootloader* en la tarjeta *phyCORE*, que asista a la grabación del sistema operativo y realice el arranque.

2. Instalar en el *host* de desarrollo los paquetes de *OSELAS*: La herramienta *ptxdist*, el *toolchain* y el *BSP*.

3. Configurar el *kernel* y el sistema de archivos raíz, y posteriormente crear las imágenes.

4. Construir el módulo del dispositivo inalámbrico utilizado para las comunicaciones externas.

5. Descargar y grabar las imágenes en la tarjeta *phyCORE*, configurando previamente el *host* de desarrollo como servidor *TFTP*.

6. Instalado el sistema operativo en la tarjeta *phyCORE* describir los *scripts* requeridos y descargar a su ubicación en la tarjeta el *firmware* y el *driver* del dispositivo inalámbrico.

De esta manera se ha explicado cómo obtener un sistema lo más eficiente posible haciendo únicamente uso de la memoria y los recursos indispensables para su ejecución, además de poseer la autonomía suficiente para poder ejecutar la aplicación que lo controla sin necesidad de la intervención externa de un usuario.

4. Migración del software de control de ASIBOT.

En la descripción del proyecto realizada hasta el momento se ha explicado cómo construir el sistema *Linux* embebido cumpliendo los requisitos impuestos inicialmente, por lo tanto, la primera parte de la migración, instalar un sistema operativo *Linux* con capacidades de tiempo real y con autonomía suficiente para conectarse a la red inalámbrica e iniciar las aplicaciones necesarias, ha sido concluida.

La herramienta *software* que fue utilizada para el desarrollo de la aplicación inicial a bordo del robot fue *Microsoft Embedded Visual C++ 4.0* y como se ha comentado anteriormente se ejecutaba en el sistema operativo *Windows CE 3.0*.

El uso de esta herramienta y para este sistema operativo condiciona las librerías utilizadas para la programación de la aplicación y la definición de las funciones utilizadas para su implementación.

Por ejemplo, el programa *Microsoft Embedded Visual* al crear los archivos en los que se escribe el código genera unos encabezados que se conocen como micro declaraciones utilizadas para detectar la falta de memoria en el sistema *Windows CE* como se muestra en la figura 4.1. Estas declaraciones no tienen sentido al crear el programa para *Linux* por lo que deben ser eliminadas.

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[]=__file__;
#endif
```

Figura 4. 1 Declaraciones generadas por eVC

Otros cambios que también se han de realizar se deben a la definición de las funciones en el sistema operativo en el que se utilizan por ejemplo el uso de la clase *winsock.h* de *Windows* para la realización de las comunicaciones inalámbricas o el uso de *threads* o hilos para la programación de los procesos concurrentes.

Además también se ha realizado un cambio significativo no debido al cambio de plataforma *software*, como los ejemplos anteriormente citados, sino al cambio de plataforma *hardware*. Este cambio es referente al módulo de comunicaciones interno que aporta el bus *CAN* sustituyendo al módulo de comunicaciones implementado anteriormente mediante el protocolo *RS232* y que da lugar a la implementación de una nueva función de sincronización, lo que será explicado detenidamente más adelante.

La herramienta que se ha utilizado para el desarrollo de la aplicación es el programa *Eclipse*, por lo que primeramente se hará una explicación sobre dicha herramienta para continuar definiendo los cambios más significativos de la migración.

4.1. *Herramienta de desarrollo de la aplicación*

En el desarrollo de la migración de la aplicación original ha sido usada la herramienta de Eclipse por lo que se hará una breve introducción de esta herramienta para entender porque se ha utilizado.

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma, es decir, funcional bajo diferentes sistemas operativos. Esta plataforma, típicamente se utiliza para implementar entornos de desarrollo integrados *IDE*, como el IDE de Java llamado *Java Development Toolkit*.

Los interfaces gráficos de *Eclipse* están implementados por una herramienta denominada *SWT*, *Standard Widget Toolkit*, que consiste en un conjunto de componentes para construir interfaces gráficas en Java, basándose en la utilización de componentes nativos, con lo que adopta un estilo más consistente en todas las plataformas y además el interfaz del espacio de trabajo también depende de una capa intermedia de interfaz gráfica de usuario llamada *JFace* que simplifica la construcción de aplicaciones basadas en *SWT* proporcionando ventanas y elementos de diseño que son utilizados regularmente.

El entorno de desarrollo integrado de Eclipse emplea módulos, en inglés *plug-ins*, para proporcionar toda su funcionalidad consiguiendo mediante este mecanismo de módulos una plataforma ligera para componentes de *software*.

Adicionalmente a los lenguajes de programación como C/C++ y Python, Eclipse permite trabajar con lenguajes para procesado de texto como *LaTeX*, aplicaciones en red como *Telnet* y Sistemas de gestión de bases de datos. Además provee al programador de *frameworks* para el desarrollo de aplicaciones gráficas, definición y manipulación de modelos de *software*, aplicaciones web, etc.

Debido a la estructura de esta herramienta para poder iniciar el desarrollo de aplicaciones utilizando los lenguajes de programación *C/C++*, y en concreto la aplicación que tiene por objeto este proyecto, es necesario instalar los módulos requeridos para este fin. A continuación se expone una lista de los módulos necesarios:

- *Primary CDT plug-in*. Es el módulo principal que proporciona el *framework* de la herramienta de desarrollo de *C/C++*. Es decir, proporciona una estructura de soporte definida, mediante la cual otro proyecto de software puede ser organizado y desarrollado. Proporciona programas, librerías y un lenguaje de ayuda al desarrollo y unión de los diferentes componentes de un proyecto.

- *CDT Feature Eclipse*. Proporciona las características de la herramienta de desarrollo de *C/C++* para *Eclipse*.

- *CDT core*. Define un interfaz simple que provee un nivel de abstracción entre el núcleo y el código del interfaz de usuario.

- *CDT UI*. Es el núcleo del interfaz de usuario. Provee las vistas, los editores y los menús de ayuda.

- *CDT Launch*. Provee los mecanismos de lanzamiento para las herramientas externas , tales como compiladores y depuradores.

- *CDT Debug Core*. Suministra las funciones de depuración.

- *CDT Debug UI*. Vistas, editores y ayudas del modo de depuración.

- *CDT Debug MI*. Enlaza las aplicaciones con los depuradores compatibles.

Instalados todos los módulos anteriores puede empezarse a desarrollar las aplicaciones en *C/C++* pero únicamente funcionales para la arquitectura sobre la que se ha construido *Eclipse* ya que por defecto el compilador es el mismo que se ha utilizado para su construcción, este es el

compilador estándar de *GNU GCC C/C++*. En la figura 4.2 puede verse el interfaz de *Eclipse* con el ejemplo típico de inicio denominado *HelloWorld* mediante el cual se comprueba la correcta instalación de todos los módulos.

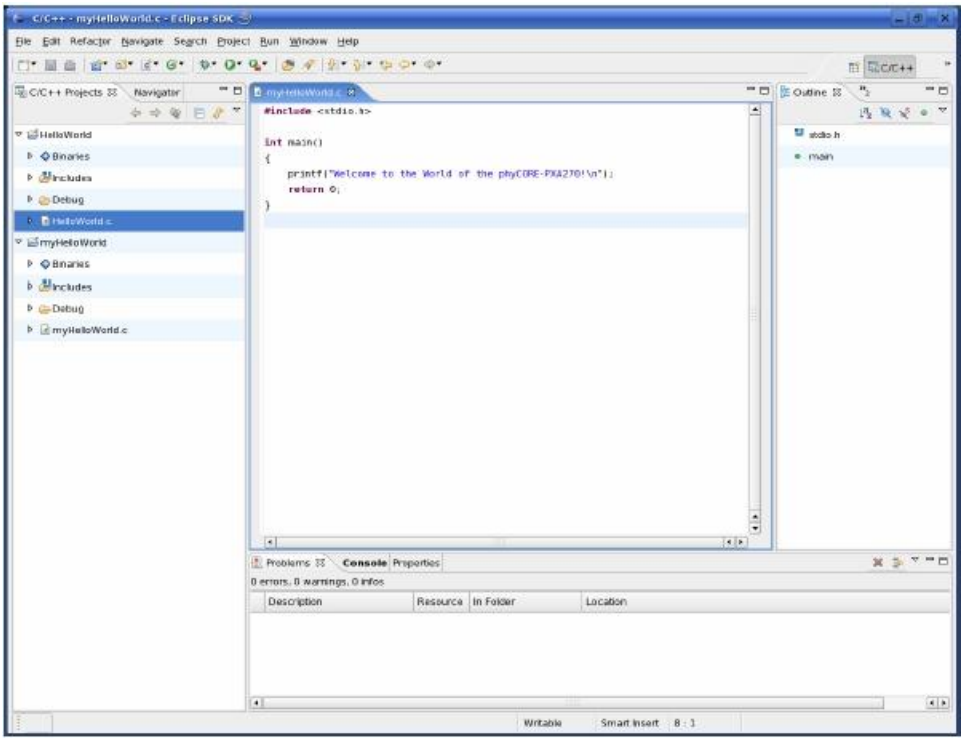


Figura 4. 2 HelloWorld aplicación de iniciación

Por tanto, para construir la aplicación de manera que sea ejecutable en la plataforma embebida, tarjeta *phyCORE PXA270*, se deberán cambiar las opciones del proyecto pues es necesario utilizar las herramientas de compilación cruzada que han sido utilizadas anteriormente para la construcción del sistema operativo.

Para ello, se configurarán las propiedades del proyecto eligiendo la opción *Properties* del proyecto abierto. Como se puede observar en la figura 4.3 se debe informar a *Eclipse* de qué compilador debe usar para realizar la construcción del proyecto, además de incluir la opción *-lpthread* imprescindible para soportar la funcionalidad de multiproceso en la aplicación.

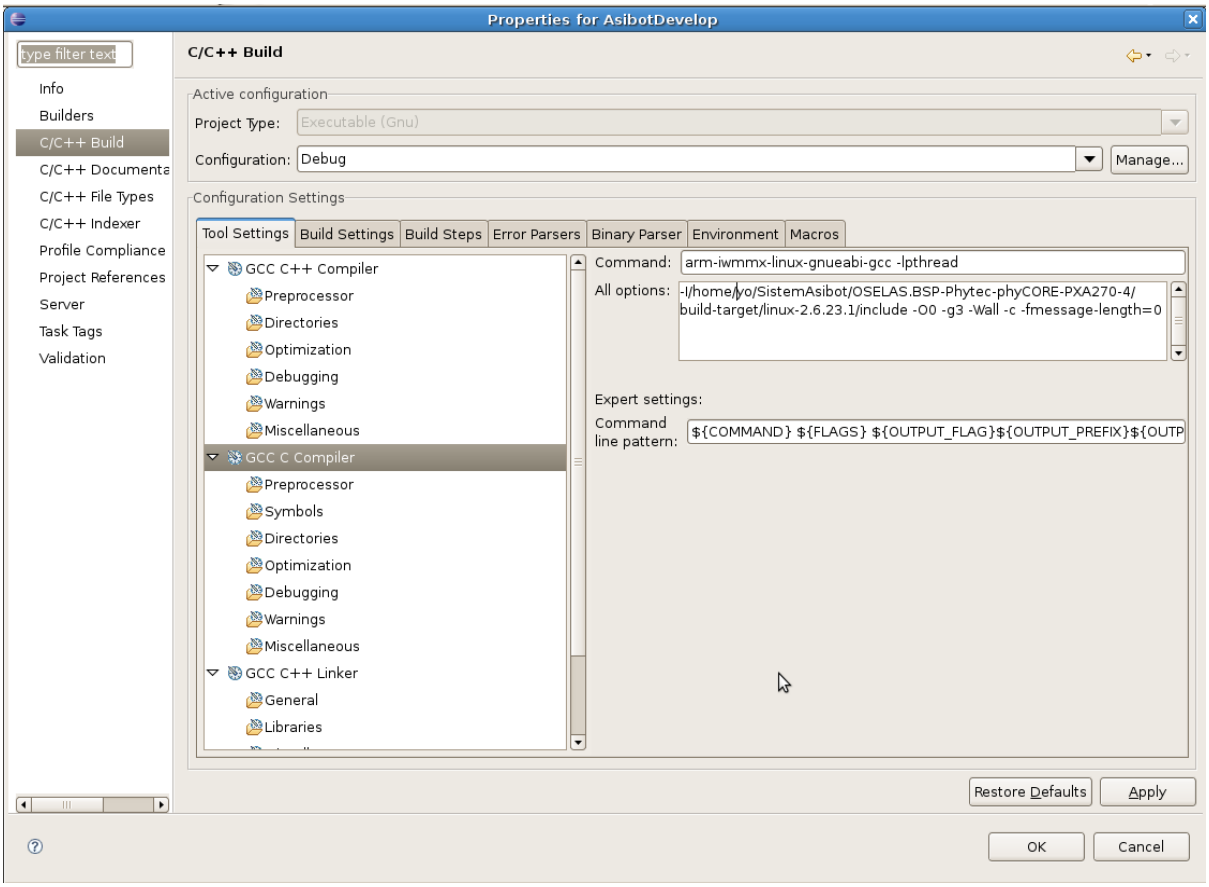


Figura 4. 3 Configuración del compilador en eclipse

Debe ser indicada la localización de las librerías que son necesarias para realizar la compilación y se definen en la opción *Directories*, figura 4.4. En este caso deberá ser incluida la localizaión de la carpeta *include/* del *kernel* del sistema operativo que se ha creado para la tarjeta ya que por defecto incluiría la localización de la carpeta *include/* de la máquina en la que se está trabajando.

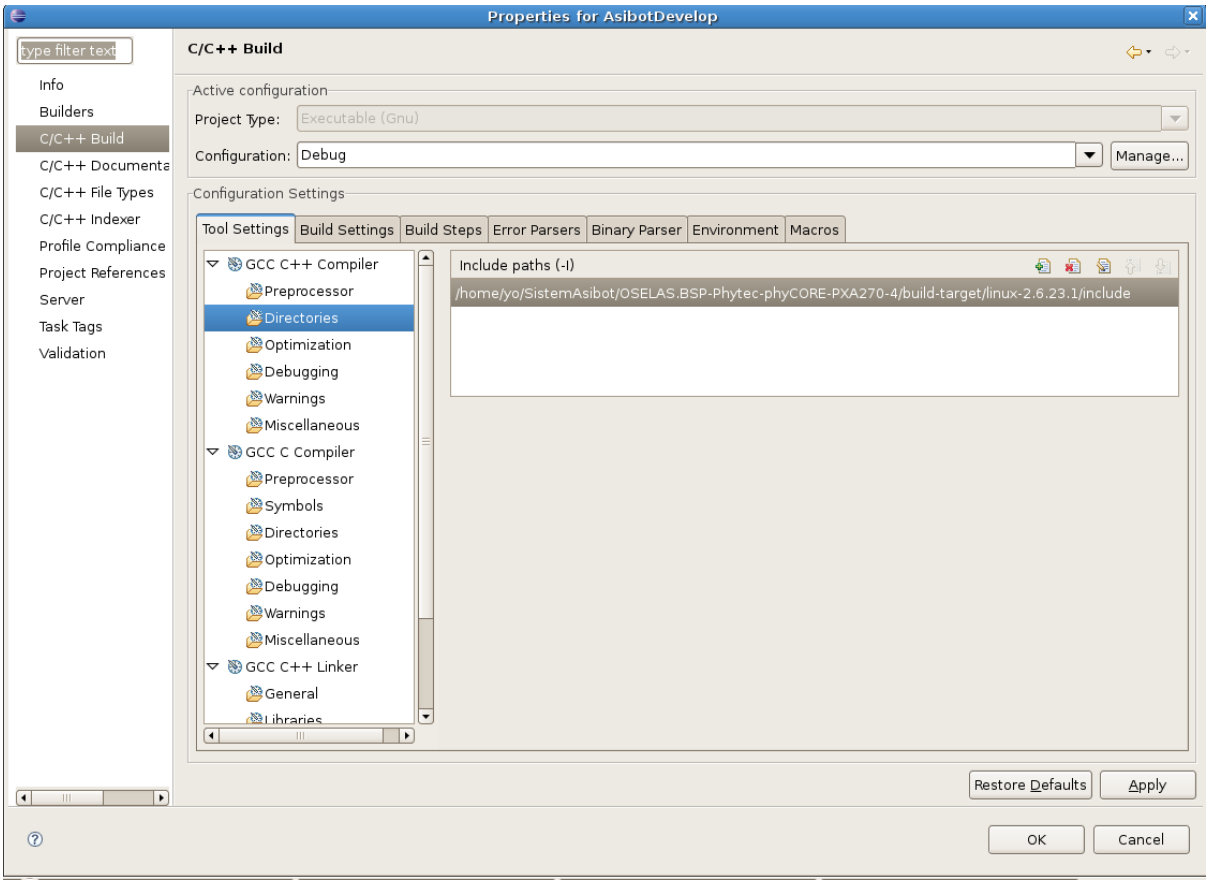


Figura 4. 4 Directorio "include" del kernel construido

Además de definir el compilador de *C/C++* que se va a utilizar deben ser indicados también el *linkador* y el *ensamblador*, en las opciones designadas a ellos, mediante sus nombres *arm-iwmmx-linux-gnueabi-gcc* y *arm-iwmmx-linux-gnueabi-as* respectivamente.

El proyecto puede construirse con dos tipos de modos. Uno de los modos se denomina *Debug* generalmente usado durante el desarrollo de la aplicación y hasta que se han finalizado las pruebas y comprobado que la aplicación funciona correctamente, ya que de esta forma se pueden comprobar los valores que van tomando las variables en tiempo de ejecución. En este modo la aplicación ocupa un espacio superior que la versión final que se denomina *Release*, ya que contiene información extra para poder ir ejecutando el programa paso a paso, dando información de las variables y comprobando los errores en cada punto.

La configuración inicial debe ser realizada para cada uno de los modos y sin dicha configuración el proyecto que se ha desarrollado no funcionaría por causas como por ejemplo la inexistencia de las librerías del controlador de bus *CAN* ya que dicho dispositivo solo existe en la tarjeta *phyCORE* y normalmente no existen en el *PC* de sobremesa que es utilizado durante el desarrollo.

Una vez configuradas las propiedades del proyecto puede apreciarse en la figura 4.6 la correcta compilación del proyecto, para este caso, y su salida en la ventana de la consola.

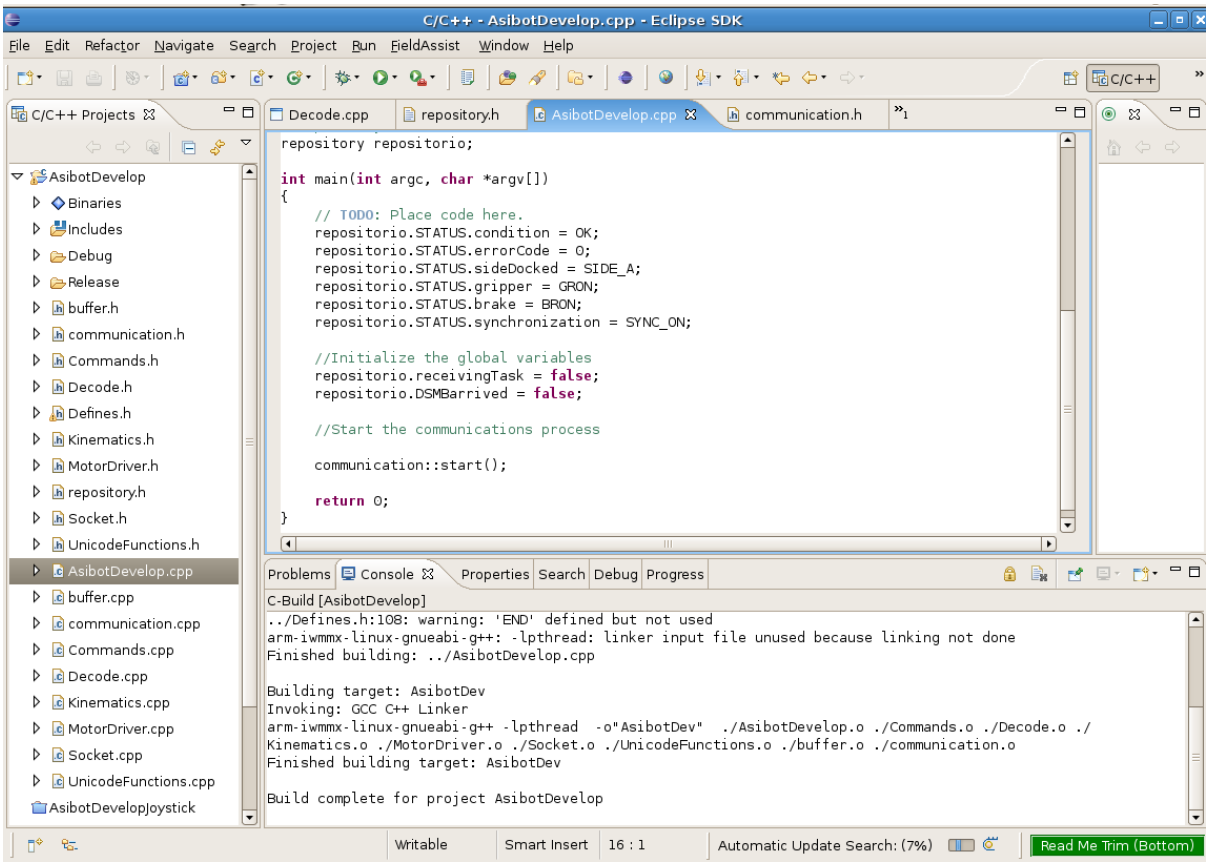


Figura 4. 5 Visualización de la compilación en la consola de Eclipse

Eclipse ofrece una vista de clases en el momento que se esta realizando el desarrollo y una vista de depuración cuando se desea realizar dicho proceso. El proceso de depuración también debe ser configurado cuando se va a llevar a cabo. Como la ejecución de la aplicación no tiene lugar en el *host* de desarrollo para poder depurarlo debe ser utilizada una aplicación llamada *gdbserver*. Esta

utilidad puede ejecutar un programa en una tarjeta remota como es el caso de la *phyCORE* y comunicarse con la utilidad de depuración que se ejecuta en el *host* de desarrollo y se denominada *gdb* utilizando el interfaz de *Eclipse* para la visualización del proceso de visualización.

Como puede verse en la figura 4.6 se configura como depurador que se ejecuta en la tarjeta *phyCORE* el *gdbserver Debugger* y como aplicación que se comunica con éste y se ejecuta mediante *Eclipse* la aplicación *gdb* que se encuentra entre las herramientas construidas en el *Toolchain de Oselas* y se denomina *arm-iwmmx-linux-gnueabi-gdb*.

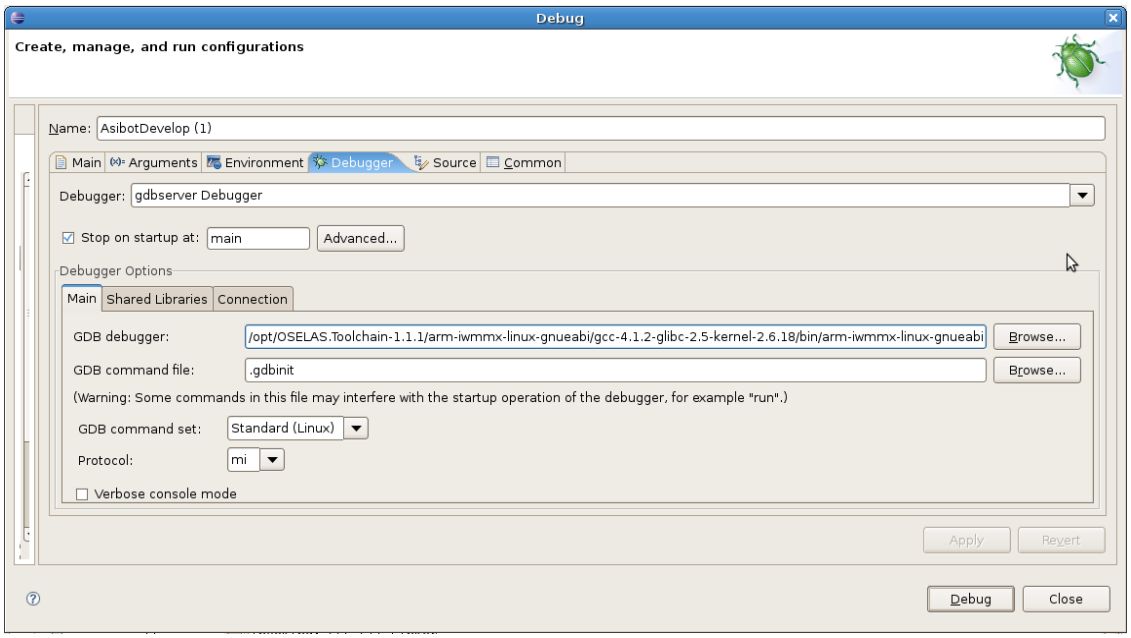


Figura 4. 6 Configuración para la depuración

La comunicación entre la aplicación, que se ejecuta en la tarjeta *phyCORE,* y *Eclipse,* que se encuentra en el host de desarrollo, o lo que es lo mismo, la conexión entre la aplicación *gdb* que se encuentra en el host de desarrollo y el *gdbserver*, que se ejecuta en la tarjeta phyCORE, puede realizarse mediante red o mediante puerto serie.

No cabe duda que la conexión Ethernet es mucho más rápida que la conexión puerto serie por lo que esta última es descartada. Además la tarjeta *phyCORE* estará localizada en el interior del robot ASIBOT lo que significa que aunque los conectores necesarios hayan sido construidos de manera accesible, que el robot realice los movimientos oportunos de manera cableada y conectada

con el host de desarrollo no sería muy útil o más bien muy cómodo, por lo que la comunicación además de ser vía red será inalámbrica gracias al dispositivo *Wi-Fi* instalado en su interior.

Por lo tanto se debe realizar una última configuración en *Eclipse* para indicarle que la comunicación en depuración se hará mediante *TCP/IP*, que el *gdbserver* se localizará en la máquina con la dirección *IP 222.222.1.2* y que la comunicación se mediará a través del puerto *10000*, figura 4.7.

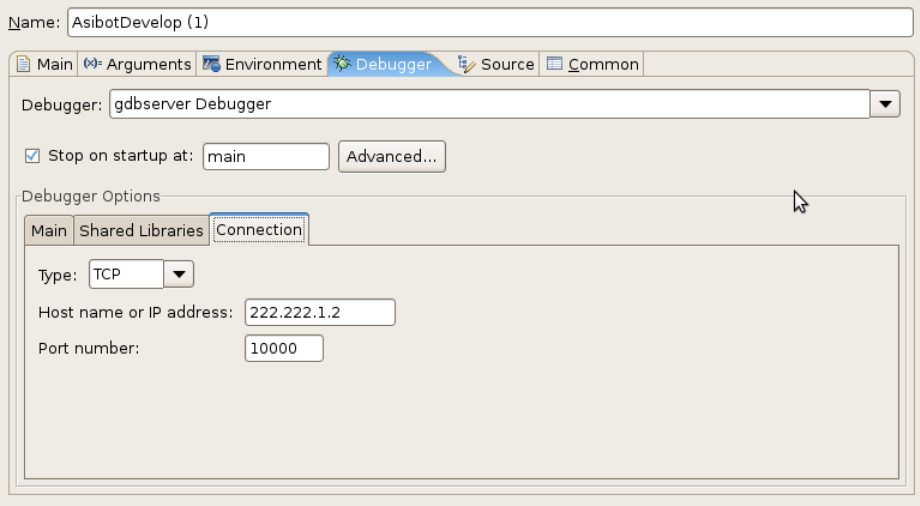


Figura 4. 7 Configuración de depuración vía red

Para iniciar el proceso de depuración se debe ejecutar la aplicación *gdbserver* en la tarjeta *phyCORE*, normalmente mediante una sesión *telnet* o *ssh* y pasarle como argumentos la dirección *IP* donde se encuentra la aplicación *gdb* el puerto a través del cual se realiza la comunicación y la aplicación en desarrollo objeto de la depuración.

4.2. *Métodos modificados debidos al Sistema Operativo*

La aplicación original a bordo del robot asistencial ASIBOT utilizaba tres tipos de mecanismos para hacer posibles las comunicaciones, la programación concurrente y la lectura y escritura del buffer de mensajes, que aunque realizan procesos similares en diferentes sistemas operativos presentan diferencias en cuanto a su uso y la definición de sus funciones.

Estos tres mecanismos que han sido necesarios modificar se denominan *Sockets* para crear y mantener las comunicaciones por red, *Multithreads* o procesos concurrentes para atender a diferentes tareas de manera simultánea y *Critical Sections* o secciones críticas para permitir o bloquear el acceso a una variable libre o en uso. En este orden se comentarán los cambios requeridos en cada uno de estos mecanismos.

La comunicación mediante Sockets creada originalmente sigue el modelo establecido por la librería *winsock.h* lo que se conoce como *Windows Socket*. Esta librería es un interfaz abierto para programación de redes bajo *Microsoft Windows*, es decir, es un *API* que implementa todas las funcionalidades necesarias para programar aplicaciones de red bajo sistemas operativos *Windows*, lo que quiere decir que dichas funciones no son soportadas en un sistema operativo *Linux*.

El modelo de comunicaciones utilizado en el proyecto es el de cliente-servidor, este modelo consiste básicamente en un cliente que realiza peticiones a otro programa, el servidor, que atiende a dichas peticiones. Para que pueda visualizarse esta idea el robot, o la tarjeta *phyCORE*, al encenderse queda a la espera de que un agente externo le envíe una petición de conexión, atendiendo de esta manera a sus peticiones, por lo tanto el robot hace la función de servidor que espera de manera pasiva a las peticiones del cliente, la *PDA*.

Por lo tanto los *sockets* que deben implementarse en la aplicación deben corresponder a la configuración de servidor para atender las peticiones de los clientes. De manera general se explicará a continuación que nombres reciben cada una de las funciones que se deben utilizar, y cuál es su función y el orden que deben seguir [22].

Para crear la comunicación en red lo primero que se debe hacer es crear un nuevo *socket* mediante la llamada a la función *socket*, figura 4.8, la función devuelve un descriptor para identificar el *socket* creado. Los argumentos de la función especifican la familia de protocolos a emplear que en este caso es TCP/IP por lo que se empleará la definición *PF_INET*; el tipo de servicio que puede ser *TCP* o *UDP*, *TCP* a diferencia de *UDP* requiere la confirmación de mensaje pero como contrapartida es más lento que el *UDP,* por lo que será utilizado el *TCP* ya que la confirmación de mensaje recibido provee una mayor seguridad en las comunicaciones y se declara mediante la definición *SOCK_STREAM*; y el número de protocolo que corresponde a la definición *IPPROTO_TCP*. Mediante la llamada a esta función se solicita al sistema operativo que aloje todos los recursos necesarios para una comunicación en red.

Al crear el socket, este no tiene asignada ninguna dirección de red, ni la dirección de red local ni la remota están especificadas. Se debe invocar a la función *bind* para especificar la dirección de red local de un socket. La función toma argumentos que especifican el descriptor del *socket* y la su dirección. Para los protocolos *TCP/IP*, la dirección de red emplea la estructura *sockaddr_in*, la cual tiene campos para especificar tanto la dirección *IP* como el número de puerto a través del cual se realiza la conexión, aunque en este caso se definirá la opción de manera que guarde la dirección *IP* del cliente entrante, mediante *INADDR_ANY*, figura 4.8.

El socket creado y caracterizado todavía no esta preparado para ser utilizado como *socket pasivo*, listo para ser usado como servidor, hasta llamar a la función *listen* y prepararlo para aceptar conexiones. Como al recibir una petición de conexión es posible que en pocos milisegundos vuelva a recibir otra el servidor debe pasar a l*isten* un argumento que le dice al sistema operativo que encole las peticiones de conexión para un *socket* dado. Por lo tanto, el primer argumento de la llamada especifica el *socket* a ser puesto en modo pasivo, mientras que el segundo especifica el tamaño de la cola de conexiones a ser empleada con dicho *socket*, figura 4.8.

Tras invocar a la función *socket* para crear el *socket*, a *bind* para especificar la dirección de red asignada a dicho *socket* y su estructura de datos definida, y a *listen* para poner el *socket* en modo pasivo; se llamará a la función *accept* para extraer la petición de conexión. Esta función crea un nuevo socket por cada nueva petición de conexión, y devuelve el descriptor del nuevo *socket* a la aplicación que la invocó. El servidor utiliza el nuevo *socket* únicamente para la nueva conexión, y emplea el *socket* original para aceptar conexiones adicionales. Una vez aceptada una conexión, el servidor puede enviar y recibir datos a través del nuevo *socket*, figura 4.8. Concluido el uso del nuevo socket, el servidor lo cierra, la función *accept* elimina la siguiente petición de conexión de la cola o bloquea al proceso que la invoca hasta la llegada de una nueva petición de conexión.

```
#define port ”63003”
int maxpending=3;
struct sockaddr_in StructServer, StructClient;
unsigned int client=sizeof(StructClient);
int IdServer= socket(PF_INET,SOCK_STREAM,IPPROTO_TCP)
StructServer.sin_family=AF_INET;
StructServer.sin_addr.s_addr=htonl(INADDR_ANY);
StructServer.sin_port=htons(port);
bind(IdServer,(struct sockaddr *)&StructServer,sizeof(StructServer));
listen(StructServer,maxpending);
int IdClient=accept(StructServer, (struct sockaddr *)&StructClient,&client)
```

Figura 4. 8 Programación de Socket en modo pasivo

Programado el socket de comunicación para el servidor solo falta saber cómo es posible enviar y recibir información a través de él. Esto se realiza mediante las funciones de lectura y escritura *recv* y *send*. Los argumentos de estas funciones se definen especificando el identificador del *socket* a través del cual se quiere mandar un mensaje o desde el cual se quiere recibir el mensaje, el buffer donde se escribe la información y la longitud de la información que se envía o que se espera recibir, figura 4.9.

```
#define MAX_ROBOT_COMMAND_LENGTH "50"
char text[MAX_ROBOT_COMMAND_LENGTH]="";
int length=strlen(text)+1;
recv(IdClient,text,MAX_ROBOT_COMMAND_LENGTH,0);
send(IdClient,text,length,0);
```

Figura 4. 9 Programación de transmisión y recepción mediante sockets

Explicada la programación necesaria para modificar la comunicación de red con el exterior se explicará el cambio de la programación en la parte de los procesos concurrentes para que se puedan llevar a cabo en el sistema *Linux*.

Para explicar porque no son compatibles las funciones que implementan los *threads,* o hilos, en ambos sistemas operativos se debe entender qué es un *thread* y como se crea. Un hilo de ejecución es el resultado de crear un proceso dentro de un programa que ya está en ejecución, por lo que se podría decir que a partir de la creación de ese hilo, dos procesos se están ejecutando de manera concurrente. Esta implementación de hilos y procesos difiere de un sistema operativo a otro porque cada sistema operativo proporciona un *API* distinto de programación para interactuar con los recursos del sistema, como es la memoria compartida, interrupciones en el tiempo e implementación del tiempo.

Antes de continuar es importante especificar que aunque se hable de hilos concurrentes, esto no es cierto ya que un único procesador no puede ejecutar simultáneamente dos órdenes, ni atender a dos peticiones simultáneamente, para ello se necesitarían dos procesadores. Por lo tanto lo que se hace en realidad es procesar *trozos* sucesivamente de cada proceso que se está ejecutando y lo que se pretende al crear varios hilos es que ambos puedan atender peticiones diferentes sin que se bloqueen ambas tareas porque una de ellas necesite esperar a la ejecución de un evento, para ello se deben definir las prioridades que cada hilo tiene dentro del sistema. En la figura 4.10 puede verse un proceso en el que se ejecutan dos hilos concurrentes apreciando como uno de ellos se procesa en el transcurso de una pausa del otro.

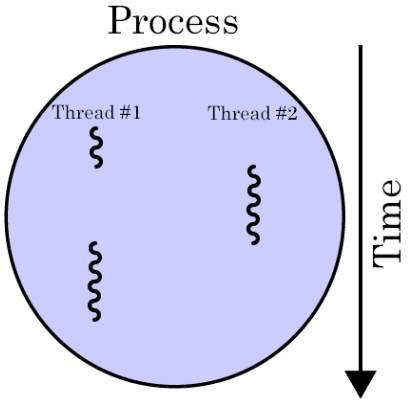


Figura 4. 10 Hilos concurrentes en un proceso

La aplicación que se está migrando posee varios hilos concurrentes en ejecución. El principal permanece a la escucha atendiendo las comunicaciones inalámbricas con el exterior, un hilo secundario que procesa los comandos recibidos por el primero y un tercer hilo que únicamente se ejecuta bajo ciertas condiciones y una vez que finaliza no vuelve a crearse si no se esas condiciones. El tratamiento más complicado de este proceso se da entre el hilo inicial y el secundario ya que el hilo principal lanza el secundario y queda a la espera de recibir un comando desde las comunicaciones externas. El proceso secundario es creado y debe ser pausado, para no consumir recursos innecesarios, hasta que el principal lo active al recibir un comando y éste secundario pueda procesarlo. Todos los comandos son procesados por el hilo secundario excepto un comando de máxima prioridad denominado *STOP* que lo procesa directamente el hilo principal cancelando el procesamiento del hilo secundario.

Crear un hilo requiere primero la inicialización y definición de sus atributos [23]. Primero se debe definir su prioridad y la política de planificación de tareas que va a ser utilizada, figura 4.11.

El planificador es la parte del núcleo que decide que proceso será ejecutado por el procesador a continuación. El planificador de *Linux* ofrece tres políticas de planificación diferentes, una para los procesos normales y dos para aplicaciones en tiempo real en los cuales se puede asignar una prioridad estática. Estas dos políticas de planificación se denominan *SCHED_FIFO (First In First Out)* y *SCHED_RR* ambas son parecidas, por ejemplo, un proceso que ha sido bloqueado por otro de mayor prioridad permanecerá a la espera en la lista pero con una posición que depende de su prioridad y se reanudará cuando todos los procesos de prioridad mayor se hayan ejecutado. La diferencia es que *SCHED_RR* no permite que un proceso que se está ejecutando exceda de un cierto tiempo máximo parándolo y poniéndolo a la lista dependiendo de su prioridad hasta que le vuelva a llegar su turno. Se ha elegido este último por ser más restrictivo sobre las latencias de un proceso.

Se debe declarar que estos atributos definidos serán los válidos cuando se cree el hilo y no serán heredados desde el proceso que crea el hilo mediante la opción *PTHREAD_EXPLICIT_SCHED*, figura 4.11. Y que el hilo a crear no va unido al hilo que lo crea sino que es independiente, es decir, para que este termine no tiene que esperar a que termine el hilo que ha creado, esto debe ser así puesto que cada uno realiza una función distinta y la opción que lo define se denomina *PTHREAD_CREATE_DETACHED*.

Definidas las opciones se crea el hilo que ejecutará la tarea requerida mediante la función *pthread_create*, la cual tiene como argumentos los atributos definidos, la identidad del hilo que se crea y la tarea que ejecuta dicho hilo.

En el hilo creado debe habilitarse su cancelación y además que dicha cancelación se produzca inmediatamente mediante las funciones *pthread_setcancelstate* y pthread*_setcanceltype* ya que como se ha mencionado anteriormente existe un comando de máxima prioridad que cancela el procesamiento de comandos. Tras lanzar el hilo y habilitar su cancelación pasará a un estado de espera para no consumir recursos del sistema mediante la función *pthread_cond_wait* que tiene como argumentos un elemento denominado *mutex* que realiza el bloqueo y una variable que recoge la señal *pthread_cond_signal* que activa la salida de la espera, figura 4.11.

```
//Definición de variables globales
pthread_t thread_id;
pthread_cond_t bloqueo;
pthread_mutex_t mutex1=PTHREAD_MUTEX_INITIALIZER;
//Definición función hilo secundario
void *task (void*);

//Proceso principal

::start{
launchtask;
receive(command);
if(command==STOP){
stoptask;
……..//Borrar bufers…y ejecutar STOP}
pthread_cond_signal(&bloqueo);}

::launchtask(){
Pthread_attr_t Attr;
Pthread_condattr_t atributo;
Pthread_cond_init(&bloqueo,&atributo);
Struct sched_param param;
Pthread_attr_init(&Attr);
Pthread_attr_setschedpolicy(&Attr,SCHED_RR);
Param.sched-priority=PRIORITY;
Pthread_attr_setschedparam (&Attr,&param);
Pthread_attr_setinheritsched
(&Attr,PTHREAD_EXPLICIT_SCHED);
Pthread_attr_setdetachstate
(&Attr,PTHREAD_CREATE_DETACHED);
Pthread_create(&thread_id,&Attr,&task,NULL);}

::stoptask(){
Pthread_cancel(thread_id);}

//Proceso creado

Void *task(void*){
Pthread_setcancelstate
(PTHREAD_CANCEL_ENABLE,NULL);
Pthread_setcanceltype
(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);
Pthread_cond_wait (&bloqueo,&mutex1);
----------//Procesado de commandos
Pthread_exit(NULL);
}
```

Figura 4. 11 Configuración de atributos, espera y cancelación de Threads

Mediante un mecanismo similar al explicado para implementar los *threads* se modificará también la programación que implementa las secciones críticas. La misión de dicho mecanismo consiste en gestionar el acceso a un búfer para realizar las operaciones de lectura/escritura de los datos almacenados.

Cuando el hilo principal recibe un comando lo almacena en el búfer por lo que cuando va a escribir el dato, se bloquea el acceso a la variable en la que se escribe y cuando termina de escribirlo de desbloquea el acceso, de modo que si el procesador de comandos quiere leer un dato si el dato está bloqueado tendrá que esperar y si esta libre lo bloqueará para su lectura y después lo volverá a desbloquear.

Éste bloqueo del acceso a la estructura de datos es posible mediante las funciones *pthread_mutex_lock* y *pthread_mutex_unlock*, con el uso de estas funciones se asegura que sólo un hilo a la vez puede acceder a la estructura de datos bloqueándola y desbloqueándola, figura 4.12. Explicado de manera más simple se puede decir que si el hilo "a" intenta bloquear un *mutex* mientras que el hilo "b" tiene el mismo *mutex* bloqueado, el hilo "a" queda a la espera. Tan pronto como el hilo "b" termine y desbloquee el *mutex*, el hilo "a" será capaz de bloquearlo.

```
Pthread_mutex_t CriticalSection;    //Definición del mutex
::LecturaBuffer(){
pthread_mutex_lock(&CriticalSection);
----------//Funciones de lectura
pthread_mutex_unlock(&CriticalSection);
}
::EscrituraBuffer(){
pthread_mutex_lock(&CriticalSection);
----------//Funciones de escritura
pthread_mutex_unlock(&CriticalSection);
}
```

Figura 4. 12 Bloqueo y desbloqueo de secciones criticas

Básicamente de esta manera todo el código ha sido reutilizable a excepción de alguna función como por ejemplo la función *_itoa* la cual solo es útil en plataformas *Windows* [24]. Esta función convierte un valor entero a una cadena de caracteres con terminación NULL. La función recomendada para reemplazarla es la función *sprintf*. Dicha función escribe la salida a un búfer controlando el formato especificado en su argumento. En la figura 4.13 se puede ver un ejemplo de su uso.

```
//Conversión de entero a ASCII en Windows
Int dLectura;
char responseDriver[10] = "";
_itoa (dLectura,responseDriver,10);
Printf ("La conversión a ASCII del entero es:%s",&responseDriver);

//Conversión de entero a ASCII en Linux
Int dLectura;
char responseDriver[10] = "";
sprintf(responseDriver,"%d",dLectura);
Printf ("La conversión a ASCII del entero es:%s",&responseDriver);
```

Figura 4. 13 Cambio de función de conversión de datos

Las ventajas de utilizar *Linux* frente a *Windows* pueden apreciarse en por ejemplo, mayor flexibilidad al definir las propiedades de los atributos en un *thread*, en *Windows* las opciones a configurar en estos métodos son menores, o un total acceso a las librerías y códigos fuentes que forman el sistema, como se ha comentado anteriormente.

Mediante estos procedimientos ha sido posible la conversión del código para que se pueda ejecutar la aplicación original, que se utilizaba en Windows CE 3.0, en el sistema *Linux* diseñado a medida para la tarjeta *phyCORE PXA270*.

4.3. *Módulos añadidos a la aplicación original*

Uno de los motivos que impulsó el desarrollo de este proyecto y que llevo a la sustitución de la tarjeta de control del sistema fue el cambio del bus de comunicaciones interno, sustituyendo las comunicaciones implementadas a través del puerto serie por su implementación en la red del bus *CAN*.

Inicialmente como se explicó en la introducción la topología utilizada para mantener las comunicaciones con los motores de las articulaciones consistía en una multiplexación del bus serie para atender y gestionas los mensajes de los motores.

La multiplexación era necesaria para poder gestionar cada articulación de manera independiente ya que aunque el canal de transmisión era conectado de manera directa, la respuesta de cada driver debía ser multiplexada para poder ser gestionada lo que implicaba un mayor

cableado, perdida de espacio y una lógica adicional para poder manejar las entradas y salidas para realizar la multiplexación.

Con la utilización del bus *CAN* cada mensaje lleva un identificador que corresponde a un dispositivo conectado a la red. De esta forma todos los mensajes viajan a través de la misma red hasta que llegan al receptor del mensaje utilizando para toda la red únicamente un cableado de dos hilos lo que reduce el espacio necesario para su implementación y además la transmisión de la información se realiza en modo diferencial lo que aumenta la seguridad frente a interferencias del entorno. Otra ventaja fundamental reside en la velocidad alcanzada por dicho bus muy superior a la del bus serie.

Con esta nueva implementación de las comunicaciones internas es posible pasar de un esquema de comunicaciones internas como el que se mostraba en la figura 1.4 de la introducción al que se muestra en la figura 4.14.

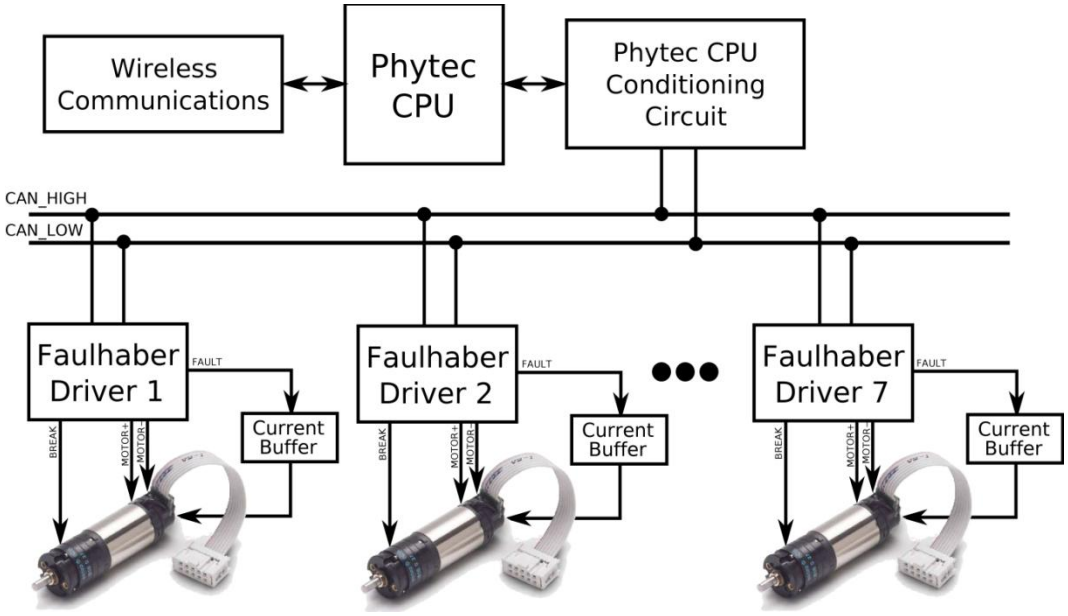


Figura 4. 14 Diagrama del nuevo esquema de comunicaciones

Gracias a esta simplificación es posible ir aumentando los dispositivos en la red, introduciendo sensores para un mayor control del sistema, sin aumentar el cableado que recorre el robot, como por ejemplo la introducción de encoders absolutos para el control de la posición absoluta de los ejes del robot que se explicará más adelante.

Otro módulo añadido es el de control de los mensajes de un Joystick mediante la red *Wi-Fi* ya que hasta el momento el *Joystick* era gestionado por la *PDA* mediante conexión *bluetooth* y esta enviaba los comandos, procedentes del *Joystick*, al robot por lo que el robot realizaba la gestión de dichos comandos de la misma manera que gestionaba cualquier otro comando.

En los siguientes apartados se describirá la programación que ha sido necesaria realizar tanto para las comunicaciones internas del bus *CAN,* como para el proceso de recepción y procesado de comandos del *Joystick*.

4.3.1. Implementación de comunicaciones del bus CAN

Al sustituir el dispositivo de comunicaciones interna ha sido necesario rehacer por completo todo el modulo de comandos del driver, aunque todos los comandos procesados anteriormente se han mantenido por lo tanto se va a pasar de tener un módulo de comunicaciones que necesita gestionar el puerto de comunicaciones *RS232* y un grupo de entradas y salidas digitales para su multiplexación a un módulo que gestiona el envío y la recepción de comandos a través del bus *CAN*.

Hay que decir que la tarjeta no implementa el protocolo *CANOpen* que es un estándar que sigue unas especificaciones para distintos dispositivos, es decir, todas las instrucciones, códigos de error y demás mensajes ya están caracterizados y definidos.

Para poder acceder al puerto de comunicaciones bus *CAN* el sistema implementa un *framework* o lo que es lo mismo una estructura de soporte definida que se denomina *Socket-CAN* y consiste en extender el *API* de *BSD Sockets* al concepto del bus *CAN*, es decir, implementa un interfaz de librerías que permite acceder al bus *CAN* mediante programación de *sockets* como se ha visto en las comunicaciones exteriores aunque las funciones varían ligeramente por lo que a continuación se expondrán algunos ejemplos de su programación.

Aunque se utilicen *sockets* para programar las comunicaciones interna los pasos a seguir son más simples que para programar la comunicación *TCP/IP* ya que para comenzar a enviar o recibir mensajes lo único necesario es crear un *socket* y conectar dicho *socket* al interfaz *CAN* que se va a utilizar para la comunicación.

Lo primero que se ha de hacer para iniciar la comunicación es crear un *socket*, mediante la función *socket*, definiendo la familia de protocolos o *dominio* que debe usar, que en este caso será *Protocol Family CAN* y se define como *PF_CAN,* para esta tarea será necesario definir un socket de tipo *SOCK_RAW* y el protocolo que se usará se define como *CAN_RAW*, figura 4.15.

Una vez creado el socket se le debe conectar con el interfaz de red que va a ser utilizado mediante la función *bind*. Esta función tiene como argumentos el identificador del *socket* creado; un puntero a una estructura de datos, denominada *sockaddr_can* en la que se describen el interfaz y el identificador can; y el tamaño de la estructura, figura 4.15.

```
int bin;
struct sockaddr_can addr;
struct ifreq ifr;
int family= PF_CAN, type=SOCK_RAW, proto=CAN_RAW;
char device[]="can0";
sockfd=socket(family,type,proto);
addr.can_family=family;
strcpy(ifr.ifr_ifrn.ifrn_name,device);
ioctl(sockfd, SIOCGIFINDEX, &ifr); // Asignación del interfaz can0
addr.can_ifindex=ifr.ifr_ifindex;
bin=bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));
```

Figura 4. 15 Configuración del socket de conexión CAN

Las funciones que realizan la lectura y escritura de la red *CAN* son *read* y *write* respectivamente. Ambas poseen básicamente los mismos argumentos que son: el identificador del *socket* de comunicación, una estructura denominada *struct can_frame* de suma importancia y que se definirá a continuación y la longitud de la citada estructura.

```
read(sockfd, &frame, sizeof(frame));
write(sockfd, &frame, sizeof(frame));
```

Figura 4. 16 Funciones de lectura y escritura del bus CAN

Los mensajes que se utilizan en la comunicación son definidos en la estructura *can_frame*. Esta estructura posee tres campos:

- El primer campo define el identificador del mensaje y se denomina *can_id*. Este identificador esta formado por la suma de dos tipos de información que son el tipo de comando enviado, definido en las especificaciones del dispositivo con el que se mantiene la comunicación, y el número de identificación del dispositivo al que se le envía.

- El segundo campo esta formado por un vector de ocho bytes y se denomina *data*. La información que se envía en estos datos es definida, igual que en el caso anterior, por las especificaciones del dispositivo con el que se comunica. Un dato curioso y a la vez de suma importancia es que si se quiere enviar por ejemplo un 3000 en decimal, su valor correspondiente en hexadecimal se sabe que es un *BB8* pero los datos deben ser enviados de menor a mayor importancia, es decir, el primer dato a enviar sería "0x*B8*" y el segundo dato sería 0x*0B*.

- El tercer campo da información sobre la longitud de los datos enviados y se denomina *can_dlc*. Siguiendo con el ejemplo anterior el valor de este campo sería igual a dos.

A continuación en la figura 4.17 se expone un ejemplo de cómo debería rellenarse un dato antes de ser enviado por la red.

```
struct can_frame frame;
int FrameIdType, idMotor;
FrameIdType = 0x300;
idMotor=3;
frame.can_id = FrameIdType + idMotor
frame.data[0]=0x03;
frame.data[1]=0x00;
frame.can_dlc=2;
write(sockfd, &frame, sizeof(frame));
```

Figura 4. 17 Configuración de la estructura de datos de la red CAN

Como se puede comprobar el envío de información es bastante simple, sin embargo, la recepción se complica un poco debido a que los valores en hexadecimal deben irse guardando dato por dato y después debe realizarse la conversión de hexadecimal a entero teniendo en cuenta que el orden de recepción va de los bytes menos significativos a los más significativos.

```
char buf[8][2];
int resp;
read(sockfd, &frame, sizeof(frame))
for (i = 0; i < frame.can_dlc; i++)
sprintf(buf[i],"%02X",frame.data[i]);
hexToInt(buf, &resp);
printf("Respuesta recibida:%d\n",&resp);
```

Figura 4. 18 Tratamiento de los datos recibidos de la red CAN

Gracias a la red *CAN* implementada y siguiendo el modelo mostrado en la figura 4.14 se ha ampliado una funcionalidad del robot instalando unos encoders absolutos comunicados con la tarjeta mediante la red *CAN*. El robot ya poseía encoder en las articulaciones pero son de tipo relativo. Estos son necesarios porque para ejecutar tareas en las cuales los puntos han sido grabados con anterioridad el robot necesita conocer cuál es su posición cero o dicho de otra forma tener una referencia fija respecto a los puntos grabados en otro momento. Para obtener esta posición cero el robot debe realizar ciertos movimientos que se conocen con el nombre de sincronización que consiste en llevar todas sus articulaciones hasta un sensor inductivo situado en uno de sus extremos y después avanzar en sentido contrario la mitad de las vueltas que constituyen todo su rango de movimiento.

Para realizar esta sincronización se necesita un amplio espacio de trabajo y además se obtiene una pérdida de tiempo importante esperando a que concluyan los movimientos. Explicado esto se puede entender la importancia que tienen los encoders absolutos ya que mediante estos dispositivos se puede obtener la posición absoluta en cualquier instante de tiempo sin la necesidad de una sincronización inicial.

La implementación de la función consiste en una lectura de la posición de la articulación y el envío de ésta información sobre la posición a los drivers, figura 4.19.

```
main(){
int valor;
//Lectura de la posición
readabsoluteencoder(2,&valor);
printf("2.....%d\n",valor);
//Escritura de la posición en el driver
writeAbsolutePosition(2,&valor);
}
```

Figura 4. 19 Función se sincronización con encoder absoluto

La función de lectura del encoder absoluto denominada *readabsolutencoder* requiere la configuración del encoder, ya que la comunicación entre la tarjeta *phyCORE* y estos dispositivos se realiza a través del puerto *CAN* de un micro de tipo *PIC* que se encarga de leer la posición del encoder mediante protocolo *SPI Serial Port Interface*, y la petición del envío de la lectura, figura 4.20.

```
readabsoluteencoder(int idAbsEncoder,int *valor)
{
char response[MAX_DRIVER_COMMAND_LENGTH] = "";
int i;
int binario[16];
*valor=0;
//Inicialización del encoder absoluto
frame.can_id= 0x00+0x00;
frame.data[0]=0x01;
frame.data[1]=0x00;
frame.can_dlc=2;
write(sockfd, &frame, sizeof(frame));
usleep(TIME_CAN_ABSOLUTE);
a=1;
```

```
if (a==0){
frame.can_id= 0x600+0x40+idAbsEncoder;
frame.data[0]=0x23;
frame.data[1]=0x01;
frame.data[2]=0x18;
frame.data[3]=0x01;
frame.data[4]=0xC2;
frame.data[5]=0x02;
frame.data[6]=0x00;
frame.data[7]=0x40;
frame.can_dlc=8;
write(sockfd, &frame, sizeof(frame));
usleep(TIME_CAN_ABSOLUTE);
}
//Petición de lectura
Frame.can_id= 0x80+0x00;
frame.can_dlc=0;
write(sockfd, &frame, sizeof(frame));
read(sockfd, &frame, sizeof(frame))
for (i = 0; i < frame.can_dlc; i++)
sprintf(buf[i],"%02X",frame.data[i]);
hexToInt(buf, &response);
usleep(TIME_CAN_ABSOLUTE);

*valor=atoi(response);
}
```

Figura 4. 20 Función de lectura de encoder absoluto

Una vez obtenida la posición absoluta se llama a la función de escritura de la posición denominada *writeAbsolutePosition*. En esta función se realizan las operaciones necesarias para que el cero del encoder absoluto coincida con el cero de los encoder relativos ya que al ser instalado el disco de lectura del encoder absoluto su posición cero puede no coincidir con la posición cero del robot, tras ello se hace la conversión de grados obtenidos a cuentas del encoder relativo teniendo en cuenta el par reductor que tiene el motor ya que los dos encoders, absoluto y relativo, no se encuentran en el mismo lado del eje, figura 4.21.

```
writeAbsolutePosition(int idMotor, int *absolutePosition)
{
int offset[5];
float absoluteDegrees;
int counts;
offset[2]=32082;
offset[3]=46500;
offset[4]=50340;
//Corregimos error inicial de lectura para que coincida el cero de los encoder
//absolutos con el cero de los encoders relativos.
*absolutePosition=*absolutePosition - offset[idMotor];
if (*absolutePosition<0) *absolutePosition= *absolutePosition + 65536;
//Calculamos los grados a los que se encuentra
absoluteDegrees = ((float)(*absolutePosition) / 65536)*360;
//En la posición 0 el encoder relativo sólo girara como máximo
//150 o -150 grados por lo que se necesita identificar en que lado
//de la posición 0 se encuentra.
if(absoluteDegrees>180) absoluteDegrees=absoluteDegrees - 360;
//Calculamos las cuentas correspondientes a esos grados para el encoder relativo
counts= (int) (absoluteDegrees* KREDUCTION);
//Escribimos las cuentas en el driver
homePosition(idMotor, counts);
return true;
}
```

Figura 4. 21 Función de escritura de posición absoluta en el driver

4.3.2. Módulo de procesamiento de mensajes para el Joystick

Inicialmente las comunicaciones entre el *Joystick* y el robot eran asistidas por la *PDA*. La comunicación existente consistía en una conexión *bluetooth* entre el *Joystick* y la *PDA* por lo que la *PDA* enviaba los comandos solicitados por el *Joystick* mediante *Wi-Fi* al robot y éste los trataba de la misma manera que los comandos directamente enviados por la *PDA*.

Actualmente ha sido implementada la comunicación del *Joystick* mediante *Wi-Fi* para poder realizar la transmisión de sus comandos directamente con el robot sin "pasar" a través de la *PDA*. Para habilitar este cambio dentro de la aplicación base se han implementado dos *threads* adicionales a los existentes.

El procesamiento de los comandos que son recibidos desde el *Joystick* es similar a los recibidos por la PDA pero su tratamiento es diferente ya que el *Joystick* únicamente envía comandos, no los recibe, por lo que las confirmaciones de comando correcto y comando procesado no se deben implementar para esta comunicación lo que implica duplicar casi todos los procesos para su procesamiento.

La implementación de este módulo requiere crear dos hilos adicionales desde el proceso principal que replican lo que hacen los dos hilos, hilo principal de recepción e hilo secundado de procesamiento, anteriormente descritos. Por lo tanto, uno de los hilos creados será el encargado de recibir los comandos enviados por el *Joystick*, crear el hilo de procesamiento y enviar la señal de procesamiento de comando a dicho hilo. El hilo creado por éste esperará a la señal del primero para ejecutar el comando recibido.

En el diagrama de bloques de la figura 4.22 se muestra cómo se realiza la duplicación de los procesos para diferenciar el tratamiento que se le da a los comandos recibidos por el Joystick y a los comandos recibidos por la PDA.

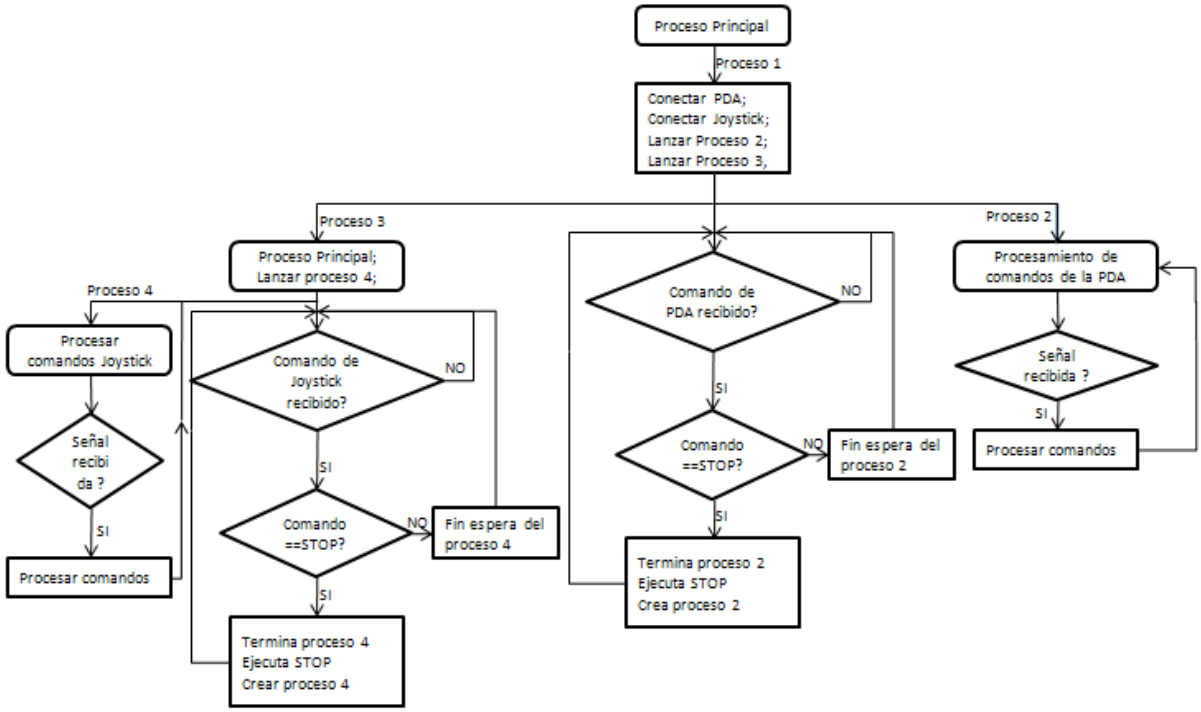


Figura 4. 22 Diagrama de bloques con módulo de Joystick

Los comandos enviados por el *Joystick* tienen la peculiaridad de que por motivos en su modo de transmisión todos los comandos que envía tienen una longitud de 22 bytes por lo tanto al recibir el comando debe hacerse un pequeño procesamiento de filtrado para obtener el comando correcto ya que 22 bytes corresponde a la longitud del comando más largo que podría ser enviado. Una vez obtenido el comando correctamente es procesado del mismo modo que los comandos recibidos por la *PDA* a excepción de cómo se ha mencionado anteriormente los mensajes de confirmación al dispositivo.

5. Conclusiones y futuros desarrollos

5.1. Conclusiones

El proyecto realizado tenía como objetivo principal realizar la migración de la aplicación que se ejecutaba en el robot en un sistema operativo *Windows CE* a bordo de una tarjeta *Cerfboard* a un sistema operativo *Linux* con capacidades de tiempo real en una tarjeta *phyCORE PXA270* con la finalidad de obtener una mayor robustez en el sistema mediante una placa más potente, un sistema operativo más ligero y unas comunicaciones internas, mediante bus *CAN*, más rápidas, con un menor cableado y que posibilitan la conexión de numerosos dispositivos mediante una red únicamente de dos hilos. Para ello se plantearon los siguientes objetivos intermedios:

- Adquirir los conocimientos necesarios sobre entornos *Linux*, aplicaciones embebidas y compilación cruzada.
- Elección de un *BSP* apropiado que soportase las características requeridas.
- Configuración del *host* de desarrollo para la construcción del sistema.
- Elección de las mínimas características que permiten la funcionalidad del núcleo del sistema operativo.
- Elección de aplicaciones y directorios que constituirán la imagen del sistema operativo embebido.
- Búsqueda y puesta en marcha de un dispositivo inalámbrico que permita las comunicaciones externas.
- Desarrollo de las funcionalidades necesarias que doten al sistema de autonomía.
- Desarrollo de aplicaciones básicas para la comprobación de los mecanismos que se van a modificar en la aplicación original debido a los cambios de sistema operativo y plataforma *hardware*.

- Integración de los mecanismos y el módulo de comunicaciones internas en la aplicación completa.
- Comprobación del funcionamiento de la aplicación inicial modificada a bordo del robot.
- Adición de módulos complementarios: integración de encoders absolutos y cambio de comunicaciones del *Joystick*.

Una vez finalizado el proyecto se puede concluir que se han alcanzado los siguientes objetivos:

1. ***Se ha seleccionado, configurado e integrado en la tarjeta phyCORE un sistema operativo que la dota de las funcionalidades básicas requeridas***: Como se ha explicado anteriormente se ha elegido la herramienta de trabajo y el *BSP* proporcionados por *OSELAS* por tratarse de una herramienta que facilita y automatiza el proceso de selección de características del sistema embebido, dota al sistema de la mayoría de las características *hardware* ofrecidas mediante parches que modifican el *kernel* genérico y, finalmente y no menos importante, porque es *OpenSource*, código abierto y gratis. La realización de esta integración ha requerido la configuración del *PC* de sobremesa para utilizarlo como *host* de desarrollo, la grabación en flash de un *bootloader* que arranca el sistema operativo, la selección y configuración de las características que hacen posible interactuar con los dispositivos *hardware* del sistema y dotan al sistema de capacidades de tiempo real y, por supuesto, la grabación de modo permanente de este sistema en la tarjeta *phyCORE*.

2. ***Se ha dotado al sistema de comunicaciones inalámbricas***: Aunque no sea apreciable dotar este sistema de comunicaciones inalámbricas ha sido una tarea ardua de constante búsqueda y numerosos intentos de compilación para la obtención de un driver específico que funcionase en la tarjeta *phyCORE*. Esto se debe a que los códigos fuente se diseñan en general para arquitecturas x86 aunque cada vez se va teniendo más en cuenta a la hora de desarrollar el driver el uso de diferentes arquitecturas con los diferentes dispositivos. Además de la dificultad de encontrar un dispositivo que funcionase con un driver que fuese válido en dicha tarjeta, se necesito cambiar algunas funciones que entraban en conflicto con las capacidades de tiempo real y definir al gestor de dispositivos la localización de su *firmware*. A pesar de todo ello el dispositivo *USB Wi-Fi* está integrado en la tarjeta y se ha comprobado que es un dispositivo totalmente funcional.

3. ***Se ha dotado al sistema de autonomía***: Al encenderse el sistema la aplicación a bordo debe mantenerse a la espera de recibir una conexión entrante mediante comunicación inalámbrica. Esto se ha conseguido mediante la realización de unos ficheros que se ejecutan al iniciarse el sistema operativo cargando los drivers del dispositivo *Wi-Fi*, configurando la conexión y conectándose a la red especificada pudiendo de esta manera ejecutar la aplicación que controla el robot y permanece a la espera de las ordenes externas.

4. ***Se ha migrado la aplicación de control original***: Todos los mecanismos utilizados en *Windows* para realizar la programación de la aplicación y que dependían de las funciones definidas por el interfaz de *Windows* han sido cambiadas por las respectivas funciones que realizan los mecanismos equivalentes en *Linux*, habiéndose probado cada uno de ellos por separado comprobando así su funcionamiento y finalmente integrándolos en la aplicación general. Además se ha modificado por completo el modulo de comunicaciones internas realizado anteriormente mediante protocolo *RS232* y actualmente mediante la red del bus *CAN* habiendo testeado su total funcionalidad a bordo de *ASIBOT*.

5. ***Nuevas funcionalidades***: El uso de la red de bus *CAN* posibilita la integración de numerosos dispositivos debido a la reducción de espacio necesario para su introducción y a la eliminación de la necesidad de una lógica de multiplexación en las comunicaciones. Gracias a esto se ha introducido una nueva funcionalidad de sincronización, basada en encoders absolutos, que disminuye el espacio requerido para la sincronización del robot y el tiempo requerido en realizarla. También se ha añadido un nuevo módulo que procesa de manera específica los comandos recibidos desde un *Joystick* dándoles un tratamiento distinto a los recibidos desde la *PDA*.

En conclusión, se han seleccionado, configurado e instalado los requerimientos necesarios para el funcionamiento del sistema operativo y sus diferentes dispositivos, realizando las modificaciones necesarias en la aplicación para que tenga total funcionalidad en un entorno completamente diferente al anterior, comprobando que el robot ha quedado totalmente operativo y cumpliendo los requisitos iníciales.

5.2. Futuros desarrollos

Este proyecto engloba la integración de algunos de los sistemas que se quieren implantar en un segundo prototipo mecánico de *ASIBOT* utilizando para ello la estructura mecánica del primer prototipo. Para la integración de la nueva tarjeta a bordo de *ASIBOT* además de todo el desarrollo *software* que ello ha conllevado, ha sido necesario rediseñar toda la electrónica a bordo del robot incluido el acondicionamiento de todos los puertos utilizados en la tarjeta. Por lo tanto en este apartado se va a hablar sobre mejoras que se han ido detectando a lo largo del proyecto y pueden ser de gran ayuda. Tenerlas presentes desde el inicio del diseño del segundo prototipo, es muy recomendable.

Implementación de una arquitectura distribuida mediante YARP

Actualmente todos los módulos utilizados para implementar las funcionalidades del robot están definidos mediante diferentes clases. Pero, toda la aplicación se engloba en un único conjunto lo que presenta varios problemas:

1. Aunque se haya comprobado por separado la funcionalidad de cada módulo resulta complicado comprobar el completo funcionamiento de todas las funcionalidades en el conjunto global.

2. La depuración de una aplicación en la que se generan diferentes objetos de cada clase y están ejecutándose ciertos procesos en paralelo con funciones diferentes, como mantener las conexiones con un dispositivo externo o interpretar y procesar los comandos recibidos, es difícil de llevar a cabo de manera exhaustiva precisando de una gran cantidad de tiempo.

A esto hay que sumarle que la implementación de la programación se ha llevado a cabo utilizando directamente las librerías que proporciona el *kernel* del sistema operativo, como *sockets* o *threads*, lo que quiere decir que no existe ningún mecanismo intermedio encargado de solucionar

problemas que se hayan podido pasar por alto lo que aumenta la probabilidad de que se produzcan transiciones por estados no concebidos y den lugar a fallos.

Por estas razones se propone como futura mejora el desarrollo y la implementación de una arquitectura de control distribuida basada en *YARP* [25]. *YARP*, acrónimo de *Yet Another Robot Platform*, es un conjunto de protocolos, librerías y herramientas para mantener módulos y dispositivos desacoplados.

YARP no controla el sistema sino que proporciona el *Middleware* para gestionar la comunicación entre los diferentes módulos u aplicaciones requeridos para realizar la gestión de la arquitectura de control.

De esta manera todos los módulos pueden ser depurados con una menor dificultad uno por uno de manera exhaustiva sin tener que preocuparse de cómo realizar la comunicación entre ellos simplemente preocupándose de la gestión de los datos entre los módulos.

En principio mediante esta gestión los mecanismos implementados mediante *threads* y *sockets* serían innecesarios ahorrando esfuerzos de programación y posibles fallos. Al utilizar este tipo de arquitectura distribuida los módulos pueden ser reemplazados completamente ya que una vez operativa la arquitectura cada módulo actúa como una caja negra en la que solo interesan las entradas y salidas de datos que gestiona cada caja.

Trayectorias mediante uso de la librería KDL

Actualmente el módulo que implementa el cálculo de trayectorias realiza una interpolación en posición lo cual es útil para este primer prototipo, pero para realizar un control más complejo se debería poder realizar la interpolación de trayectorias en velocidad.

Aunque el módulo responde a las especificaciones iníciales, según avanza el proyecto requiere una mayor precisión y control de movimiento basado en modelo dinámico y esquemas de control de fuerza y compensación de gravedad o de impedancia variable.

Para poder seguir avanzando se propone como futuro desarrollo el uso de la librería *KDL, Kinematics and Dynamics Library* del proyecto *OROCOS* [26]. *KDL* es una librería que desarrolla una

aplicación independiente del *framework* utilizada para modelar y calcular cadenas cinemáticas como por ejemplo brazos robóticos, modelos humanos biomecánicos, máquinas herramientas, etc. Esta librería provee clases de objetos geométricos (puntos, cuadrados, líneas, ..), cadenas cinemáticas de varios tipos (series, humanoides, paralelas, móviles,..), y las especificaciones de sus movimientos e interpolaciones, por lo que podría ser de gran ayuda en este desarrollo.

Altenativa a los encoders absolutos

En este proyecto, como se ha comentado se ha implementado una función basada en la lectura de un encoder absoluto lo que ha sido posible gracias al bus de comunicaciones *CAN*. Mediante estos sensores el robot tiene conocimiento de su posición real desde el inicio del sistema sin necesidad de tener que realizar una maniobra de sincronización como se realizaba inicialmente.

Aunque este sistema ha sido probado con total éxito en otros desarrollos éste éxito total es debido a que en otros desarrollos el diseño de la mecánica donde se aloja el lector ha sido tenido en cuenta desde la concepción del prototipo.

Como ya se ha dicho en *ASIBOT* se parte de una estructura mecánica ya definida y los diseños de los soportes para alojar los encoders realizados hasta el momento no han funcionado como se esperaba. Esto se debe a dos motivos principalmente:

- El soporte donde se aloja el encoder es un complemento a la estructura primaria del robot lo que causa la perdida de precisión al realizar el montaje.
- Los diseños son fabricados en una impresora de prototipado rápido que tras imprimir y obtener el diseño en verde se le debe dar un tratamiento con cianocrilato para obtener el endurecimiento del material. Este proceso de fabricación también produce una falta de precisión ya que no es fácil que el cianocrilato sea absorbido por igual en todos los puntos del molde y además este tratamiento provoca un aumento del volumen de la superficie que no es uniforme y depende de su absorción y la correcta limpieza de las piezas.

La posición entre el disco del encoder y el lector exige una gran precisión del orden de tres décimas de milímetro. Por ello puede entenderse que cumplir estas especificaciones resulta difícil tras entender los motivos expuestos.

Por lo tanto el futuro desarrollo que se propone hasta que se fabrique el segundo prototipo de *ASIBOT*, en el cual se tenga en cuenta el diseño de los encoder absolutos, es la conexión de un *PIC* a la red *CAN* cuya misión sea detectar la falta de alimentación en el sistema ante un fallo y grabar las posiciones en memoria *EEPROM* o *FLASH*, memorias no volátiles, para informar al sistema de las últimas posiciones en las que se encontraba antes de su interrupción.

Detección de alimentación en la DS

Una de las características de *ASIBOT* es que posee toda la electrónica necesaria a bordo del sistema obteniendo la alimentación de 24V desde la *Docking Station* a la que permanece anclado, recorriendo esta alimentación toda la longitud del robot hasta llegar al otro extremo.

Lo que se propone es el desarrollo de un mecanismo que detecte en que *Docking* se encuentra anclado el robot lo que puede suponer tres ventajas frente al sistema actual:

- Al detectar si está anclado por el extremoa *A* o por el extremo *B* enviar esta información a través de la red CAN a la aplicación de control. Esta ventaja presenta autonomía respecto al modelo inicial ya que actualmente debe ser el usuario el que indique al robot en que *Docking* se encuentra anclado mediante la *PDA*.

- Tras detectar en qué punto está anclado puede ser desconectada la alimentación del lado contrario. Esto presenta una ventaja de seguridad ya que la tensión del otro punto se encuentra descubierta y aunque pueda resultar difícil es posible que se produzca un cortocircuito en circunstancias inesperadas.

- Para realizar un cambio de *Docking* el robot de manera autónoma debe poder confirmar el correcto anclaje antes de desanclar la *Docking* opuesta. Este dispositivo presenta la ventaja de ofrecer información sobre si el procedimiento se ha llevado a cabo correctamente. Una vez que el cono ha entrado en la *Docking* solo puede asegurarse de manera visual el correcto anclaje. Un error puede llevar a la perdida de alimentación en todo el sistema por una mala conexión al desconectar el cono que estaba bien conectado o en el peor de los casos puede llevar a la caída del robot si resulta no conectado firmemente a ninguno de ambos conos.

Una posible idea puede ser la implementación de dos tarjetas una en cada cono con un *PIC* que comparen la tensión en los extremos. El que detecte la tensión en su extremo deberá dar paso a la alimentación de todo el sistema, el del otro lado tendrá cortado el circuito de alimentación que conecta con el extremo. Estos *PIC*'s deben disponer de un puerto de comunicaciones *CAN* para mantenerse conectados a la red *CAN* y así poder informar a la aplicación de control de cuál de los extremos es el que está conectado.

µPC *alternativo*

La tarjeta *phyCORE* utilizada como cerebro del robot ASIBOT como ya se ha comentado es una arquitectura de tipo *ARM*. Esta arquitectura plantea un inconveniente a nivel software y es que el desarrollo y acondicionamiento del sistema operativo ha sido extenso y complejo.

Este desarrollo es complejo debido a que como se ha comentado en más de una ocasión a lo largo del proyecto los *kernel* genéricos de *Linux* aunque funcionen dentro de la arquitectura no portan los drivers específicos necesarios para poder habilitar los diferentes periféricos integrados en la tarjeta *phyCORE* como son el driver del bus *CAN*, los drivers de entradas y salidas digitales o de la *PWM*, los drivers de la tarjeta *SD* o incluso los del *USB*. Esto se debe a que los dispositivos *hardware* integrados como es obvio no son dispositivos de uso general.

Para poder habilitar dichos periféricos es preciso "parchear", introducir porciones de código, que implementan las funcionalidades especificas de estos dispositivos.

Las aplicaciones o librerías que pueden utilizarse en un *PC* de arquitectura *X86* y que facilitan la realización de procesos de más alto nivel en su mayoría no pueden utilizarse, sin antes realizar algunos cambios, en arquitecturas como la que se ha utilizado ya sea porque estos cambios no son posibles o porque consumen demasiados recursos.

Si al costoso desarrollo *software* se le suma un esfuerzo equivalente para realizar el acondicionamiento de los puertos de la tarjeta, circuito de alimentación y de la tarjeta en general, debe pensarse si los recursos disponibles de tiempo y personal son los suficientes para realizar un desarrollo desde tan bajo nivel, o es más interesante tener una plataforma a la que no haya que dedicarle tantos recursos a su acondicionamiento y permita dedicar estos recursos a tareas como

por ejemplo cálculo de trayectorias, control en bucle cerrado de movimientos, módulos de visión y de voz, etc.

La razón por la que ha sido necesario llevar a cabo todo este trabajo se debe sobre todo a las restrictivas especificaciones dimensionales del interior del robot y a la implementación de la red *CAN* especificaciones únicamente cumplidas en su día por la tarjeta *phyCORE* PXA270.

Al surgir nuevos modelos de tarjetas se propone una nueva tarjeta que cumple, además de las características dimensionales, que es una arquitectura x86 lo que facilitaría la obtención de *software* funcional y que además no requiere acondicionamiento de la electrónica. Esta tarjeta se denomina *RB 100* [27] y puede verse en la figura 5.1.

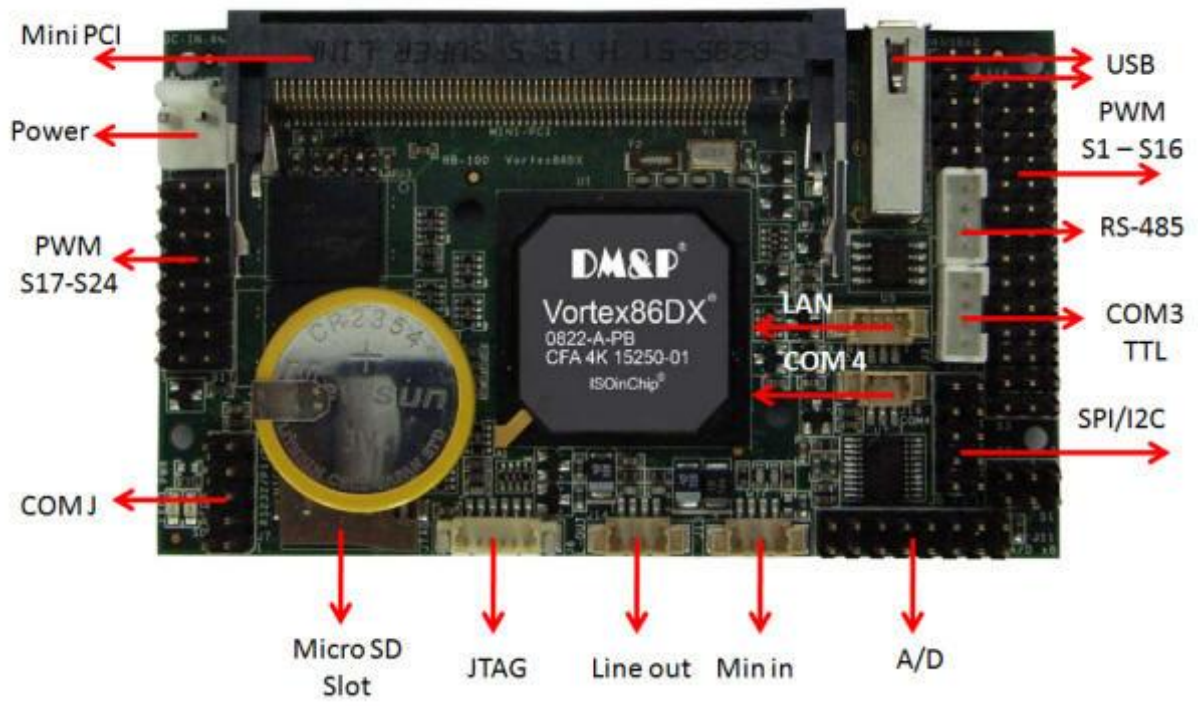


Figura 5. 1 Tarjeta RB 100

Puede notarse que esta tarjeta no implementa el bus *CAN* pero su integración es directa mediante una tarjeta *Mini Pci to CAN*.

Actualmente éste desarrollo y los citados anteriormente están en proceso en el marco de 5 nuevos proyectos final de carrera.

6. Bibliografía

[1] Jardón Huete A.; *Metodología de diseño de robots asistenciales. Aplicación al robot portátil ASIBOT.*; Tesis Universidad Carlos III de Madrid, 2006.

[2] Giménez Fernández A.; *Metodología de Diseño y Control de Robots Escaladores. Aplicación a las Tareas de Inspección*; Tesis Universidad Carlos III de Madrid, 2000.

[3] Barrientos A.; Peñín L.F., Balaguer C., Aracil R.; *Fundamentos de Robótica*; McGraw-Hill, 1997.

[4] Martínez de la Casa Díaz, S.; *Diseño e Implementación de los Sistemas Embarcados del Robot MATS.*; P.F.C. Universidad Carlos III de Madrid, 2005.

[5] Correal Tezanos, R.; *Diseño e implementación de la arquitectura de control del Robot Asistencial MATS.*; P.F.C. Universidad Nacional de Educación a distancia, 2005.

[6] Balaguer C., Giménez A., Jardón A., Cabas R. Correal R. *; Life experimentation of the service robot applications for elderly people care in home environments*. Internatinal Conference on Robotics and Automation. Abril, 2005.

[7] Giménez A., Jardón A., Correal R. y otros; *Service robot applications for elderly people care in home environments*; 2nd International workshop on advances in service robotics, Mayo, 2004.

[8] Balaguer C., Giménez A., Jardón A., Correal R. y otros; *Light weight autonomous climbing robot for elderly and disabled persons services*; International Conference on Field and Service Robotics. Julio, 2003.

[9] Siciliano B.; *Control in Robotics:Open Problems and Future Directions*; Proceedings of the 1998 IEEE Conference on Control Aplication, Trieste, I, pp 81-85, 1998.

[10] Song W.K., Lee H.Y., Kim J.S., Yoon Y.S., Bien Z.; *KARES: Intelligent Rehabilitation Robotic System for the Disabled an the Elderly*; Proceedings of the 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Vol. 20, No. 5, 1998.

[11] CAN in Automation; <http://www.can-cia.org/> [Consulta:20 de abril de 2007].

[12] Lombardo, John; *Embedded Linux*; New Tiders; 2002; Disponible en Web: www.EmbeddedLinuxBook.com.

[13] Bovet, Daniel P.; *Understanding the Linux kernel*; O'Reilly; 2005.

[14] Corbet, Jonathan; *Linux device drivers*; O'Reilly; 2005.

[15] Abbot, Doug; *Linux for embedded and real-time applications*; Newnes; 2003.

[16] Página web de la empresa que proporciona la herramienta de construcción del sistema embebido y el *BSP* de la tarjeta *phyCORE*: <http://www.pengutronix.com> [Consulta: 1 de diciembre de 2006].

[17] Proyecto GNU: <http://www.gnu.org/home.es.html> [Consulta: 10 de Marzo de 2007].

[18] Página web de la empresa Ralink: <http://www.ralinktech.com/> [Consulta : 16 de Mayo de 2007].

[19] Página web del fabricante del dispositivo inalámbrico: <http://www.linksysbycisco.com/US/en/products/WUSB54GC> [Consulta: 30 de Mayo de 2007].

[20] Modelo OSI: <http://es.wikipedia.org/wiki/Modelo_OSI> [Consulta: 20 de Diciembre de 2006].

[21] Estándar de jerarquía de sistema de ficheros: <http://www.pathname.com/fhs/> [Consulta: 10 de enero de 2007].

[22] Walton, Sean; *Linux socket programming*; Sams; 2001.

[23] Tutorial de programación *Poxis threads* en *Linux*: <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html> [Consulta: 10 de Marzo de 2008].

[24] Funciones de manipulación de caracteres: <http://www.lingoport.com/gi/help/gihelp/unsafeMethod/_itoa.htm> [Consulta: 30 de Marzo de 2008].

[25] Página web de *YARP*: <http://eris.liralab.it/yarp/> [Consulta: 10 de abril de 2009].

[26] Página web de *OROCOS*: <http://www.orocos.org/> [Consulta: 15 de marzo de 2009].

[27] Página web del fabricante de la tarjeta *RB 100*: <http://robosavvy.com/site/index.php?option=com_content&task=view&id=183&Itemid=135> [Consulta: 25 de septiembre de 2009].