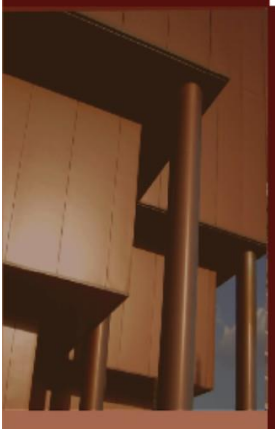
INTEGRACIÓN DE ROS (ROBOT OPERATING SYSTEM) CON ARDUINO Y RASPBERRY PI

Álvaro Ángel Romero Gandul

Tutor: Federico Cuesta Rojo



Dpto. Ingeniería Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2014

Agradecimientos

Resulta difícil expresar en unas pocas líneas tantas sensaciones y emociones que convergen en la finalización de este proyecto fin de carrera y cuánto significa.

En primer lugar me gustaría agradecer a mi familia el apoyo y amor incondicional durante todos estos años. Muy en especial a mis padres, sin los cuales nada de esto hubiera sido posible, a los cuales profeso mi más sincero amor y profunda devoción. A mi padre, Serafín, por representar siempre esa serenidad, humildad, honradez, vocación por su familia y profesión, esa persona con la que siempre da gusto sentarse a hablar de tantos temas como se pueda imaginar e incontables cualidades más en las que siempre he tratado de fijarme para crecer como persona. A mi madre, Rosario, de la cual siempre me he sentido orgulloso de ser su "mellizo", por transmitirme esos valores de lucha, de no rendirme nunca, de pelear hasta el último aliento para conseguir mis sueños y metas, por definirse como la voz de la conciencia, mi "Pepito Grillo" particular sin el cual estaría perdido hoy día. Por todo esto y mucho más, puedo sentir este orgullo y admiración incontenibles además de una envidia sana de dos de las mejores personas que jamás conoceré.

No podría olvidarme de mi hermana pequeña, Ana, mi aliada en la sombra. Pese a que no siempre fuimos uña y carne, el tiempo nos dio la razón y hoy día es de lejos la mejor amiga que puedo llegar a tener. Esa persona a la que siempre he podido recurrir cuando la he necesitado y con la que no me canso de compartir anécdotas y aventuras (*psicosis, ermies, calgon…)*.

A mi novia, Rosa, por constituirse como esa pieza insustituible y de valor incalculable en mi vida. Por haberme hecho crecer como persona desde el primer momento que nos conocimos, por constituir ese apoyo incondicional, ese hombro sobre el que llorar, por transmitir tanta alegría en tantos momentos y por contagiarme de ese positivismo sin el cual no sería posible avanzar. Por haberme ayudado a introducirme en el mundo de la programación, sirviéndome como base para la realización de este proyecto.

Por último y no menos importante, a todos mis amigos y amigas que me han acompañado a lo largo ya no solo de la carrera, si no de tantos años de sana amistad, incontables aventuras, apoyo en malos momentos, risas y variopintas circunstancias. En líneas generales, por respeto y cariño a todos y tantos de ellos, puesto que cada uno ha aportado esa preocupación, cariño, compañerismo y en definitiva amistad de la cual me siento tan orgulloso.

Me gustaría agradecer a mi tutor, Federico Cuesta, la oportunidad de realizar este proyecto fin de carrera bajo su tutela. Por todo cuanto ello ha representado y cuanto he aprendido durante su realización. Por abrirme esa puerta al mundo de la programación y control de robots y descubrir el vasto mundo que hay tras ello.

INDICE ABREVIADO

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

En el presente proyecto se aborda la tarea de la integración de ROS (Robot Operating System) con las plataformas de propósito general Arduino Mega ADK y Raspberry Pi.

ROS se constituye como un *middleware* ya que se trata de un software que asiste a las aplicaciones en concreto diseñadas en este proyecto así como a cualquier aplicación de propósito general diseñada sobre él, para interactuar o comunicarse entre sí y el hardware utilizado. Funciona como una capa de abstracción de software distribuida situada entre las capas de aplicaciones y las capas inferiores. El *middleware* abstrae de la complejidad y heterogeneidad de las redes de comunicaciones subyacentes, así como de los sistemas operativos y lenguajes de programación, proporcionando una API para la fácil programación y manejo de las aplicaciones distribuidas, punto clave para la consecución de este proyecto.

La existencia de plataformas de propósito general, como son Arduino y Raspberry Pi, las cuales se caracterizan por un bajo coste, dimensiones reducidas y alto rendimiento, permiten a la amplia comunidad de desarrolladores de todo el mundo elevar la programación y control de robots a un primer nivel gracias a las facilidades que se les ofrece. Gracias a esto y a la existencia de sistemas operativos dedicados, los cuales ofrecen incontables herramientas de programación y desarrollo, estas plataformas y sistemas se están convirtiendo en una de las opciones más valoradas.

La misión principal que se pretende conseguir con la integración es la de, mediante un sistema distribuido, realizar la programación y control de un robot móvil a través del diseño de aplicaciones que doten a dicho sistema de diferentes características y funcionalidades.

Se recoge también en este proyecto la tarea de dar a conocer las plataformas de propósito general utilizadas, profundizando en sus características, resaltando el potencial que ofrecen a la hora del diseño y programación de aplicaciones robot.

2 Elementos integrantes

2.1 ROS

2.1.1 Introducción

ROS (en inglés Robot Operating System, ROS) o Sistema Operativo Robótico, es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. ROS se desarrolló originalmente en 2007 bajo el nombre de switchyard por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR2). Desde 2008, el desarrollo continúa primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un modelo de desarrollo federado.

Figura 1 Logo ROS

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu (Linux) es el sistema soportado aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows considerados como experimentales).

ROS tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés stacks) que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

ROS proporciona librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones robóticas, abstracción de hardware, los controladores de dispositivo, bibliotecas, visualizadores, paso de mensajes, gestión de paquetes y más, bajo licencia de código abierto BSD. Esta licencia permite libertad para uso comercial e investigador. Las contribuciones de los paquetes en ros-pkg están bajo una gran variedad de licencias diferentes.

2.1.2 Historia

ROS es un proyecto grande con muchos antecesores y colaboradores. La necesidad de un marco de colaboración abierta fue sentida por muchas personas en la comunidad de investigación robótica y muchos proyectos han sido creados para alcanzar este objetivo.

Varios esfuerzos coordinados en la Universidad de Standford a mediados de la década de 2000, de la mano de la Stanford AI Robot (STAIR) y el programa Personal Robots (PR), consiguen la creación de prototipos de sistemas software dinámicos y flexibles, orientados para usos robóticos. En 2007, Willow Garage, un visionario del mundo de la robótica, proveyó abundantes recursos para extender todos estos conceptos más allá y dar vida a implementaciones convenientemente testeadas. El esfuerzo se vio impulsado por innumerables investigadores que contribuyeron con su tiempo y experiencia para dar vida a las ideas del núcleo de ROS y sus paquetes de software fundamentales. En todo momento el software fue desarrollado bajo el uso de la licencia de código abierto BSD y poco a poco se ha convertido en una plataforma ampliamente utilizada en la comunidad de investigación robótica.

Desde el principio, ROS fue desarrollada en múltiples instituciones y para diferentes tipos de robots, incluyendo muchas de las instituciones que habían recibido los PR2 robots de Willow Garage.

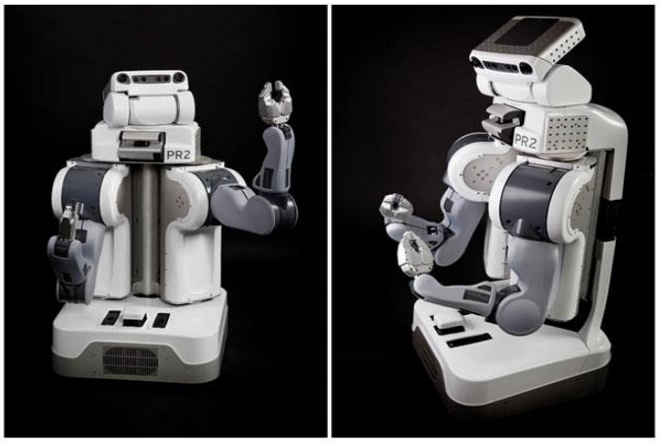


Figura 2 Robot PR2

Figura 3 Willow Garage

Pese a que hubiera sido mucho más sencillo para todos los colaboradores colocar sus códigos en los mismos servidores, a través de los años, el modelo "federado" ha surgido como una de las grandes fortalezas del ecosistema ROS. Cualquier grupo de desarrolladores puede iniciar su propio repositorio de código ROS en sus propios servidores y mantener la plena propiedad y control de los mismos. Si ellos deciden hacer que su repositorio sea público y accesible pueden recibir el reconocimiento y crédito que merecen por sus logros y beneficiarse de la información técnica específica y mejoras como en cualquier proyecto basado en software de código abierto.

2.1.3 Características

2.1.3.1 Componentes del núcleo

Si bien no podemos proporcionar una lista exhaustiva de lo que es el ecosistema de ROS, podemos identificar alguna de las partes centrales de ROS y arrojar algunos datos acerca de su funcionalidad, especificaciones técnicas y de calidad con el fin de proporcionar una mejor idea de lo que ROS puede aportar a un proyecto.

2.1.3.1.1 Infraestructura de comunicaciones

Al más bajo nivel, ROS ofrece una interfaz de paso de mensajes que provee la comunicación entre procesos la cual es comúnmente referida como "middleware".

ROS actúa como "middleware" proporcionando una serie de comodidades:

- Publicar/Subscribir paso anónimo de mensajes
- Grabación y reproducción de mensajes
- Petición/respuesta a llamadas a procedimiento remotos
- Sistema de parámetros distribuidos

2.1.3.1.1.1 Paso de mensajes

Un sistema de comunicación es a menudo una de las primeras necesidades que surgen en la implementación de una nueva aplicación robot. El sistema de mensajería integrado en ROS permite ahorrar tiempo mediante la gestión de los detalles de la comunicación entre los nodos distribuidos a través del mecanismo publicación/subscripción anónimo. Otra ventaja de utilizar un sistema de paso de mensajes es que te obliga a implementar interfaces claras entre los nodos en su sistema, mejorando así la encapsulación y la promoción de la reutilización del código. La estructura de dichas interfaces se define en el mensaje IDL (Lenguaje de descripción de interfaz). Dicho lenguaje es estándar y las funciones de ejecución nativa son capaces de interpretar esta descripción basada en texto corto descriptivo del tipo de dato dentro del mensaje. Los generadores de código para cada lenguaje entonces generan una

implementación nativa de manera que los datos son automáticamente serializados y deserializados por ROS y las funciones que componen sus librerías al utilizar funciones para el envío y la recepción de los mismos.

Esto genera un ahorro considerable en complejidad de programación. Donde el en apartado anterior se refería a la necesidad de dividir los datos en arrays de bytes para su reconstrucción en destino, el uso de las funciones que el IDL y ROS pone a nuestro alcance supone un ahorro de 137 líneas de código en C++ y 96 en Python.

2.1.3.1.1.2 Grabación y reproducción de mensajes

Debido a que el sistema de publicación/subscripción es anónimo y asíncrono, los datos se pueden capturar y volver a reproducir sin ningún tipo de cambio en el código fácilmente. Por ejemplo, imaginemos que tenemos una tarea A que lee datos de un sensor; a su vez se está desarrollando la tarea B, que procesa los datos producidos en A. ROS hace que sea fácil capturar los datos publicados en A en un archivo y, posteriormente, volver a publicarlos para presentarlos más adelante. La abstracción del paso de mensajes permite que la tarea B sea agnóstica con respecto a la fuente de datos, que podrían ser de tareas A o de un registro. Se trata de un patrón de diseño de gran alcance que puede reducir significativamente el esfuerzo de desarrollo y promover la flexibilidad y la modularidad del sistema.

2.1.3.1.1.3 Llamadas a procedimientos remotos

La naturaleza asíncrona de publicación/subscripción funciona eficientemente para muchas de las necesidades de comunicación robótica pero a veces se requieren interacciones síncronas de petición/respuesta. El middleware ROS proporciona esta capacidad utilizando servicios. Al igual que los topics, los datos que se envían entre los procesos en una llamada a un servicio se definen con el mismo mensaje IDL.

2.1.3.1.1.4 Sistema de parámetros distribuidos

El middleware ROS proporciona una manera para que las tareas compartan información de configuración a través de un almacén global de clave-valor. El sistema permite fácilmente modificar los parámetro de una tarea e incluso cambiar los parámetros de configuración de una tarea desde otra.

2.1.3.1.2 Características específicas robot

Además de los componentes del núcleo middleware, ROS proporciona unas librerías comunes robóticas y herramientas que hará que nuestro robot funciones rápidamente. Estas son algunas de las capacidades específicas para robots que ROS ofrece:

- Definición de mensaje estándar para Robots
- Librería de la geometría robot
- Lenguaje de descripción de robot
- Llamadas a procedimiento remoto
- Diagnóstico
- Estimación de la posición
- Localización
- Cartografía
- Navegación

2.1.3.1.2.1 Definición de mensaje estándar para robots

Tras años de desarrollo se ha llegado a un conjunto de formatos de mensaje estándar que cubren la mayor parte de los casos de uso común en la robótica. Existen definiciones de mensajes de conceptos geométricos, para sensores y para datos de navegación entre otros. Mediante el uso de estos mensajes en la aplicación, nuestro código interactuará sin problemas con el ecosistema ROS, desde herramientas de desarrollo a librerías de capacidades.

2.1.3.1.2.2 Librería de la geometría robot

Un reto común entre los proyectos de robótica es mantener un seguimiento de dónde están las diferentes partes de un robot unas con respecto a otras. Este problema es esencialmente importante para los robots humanoides con partes móviles. Este problema es abordado en ROS con la librería *tf*, la cual prestará atención a dónde se encuentra cada parte de nuestro sistema robot.

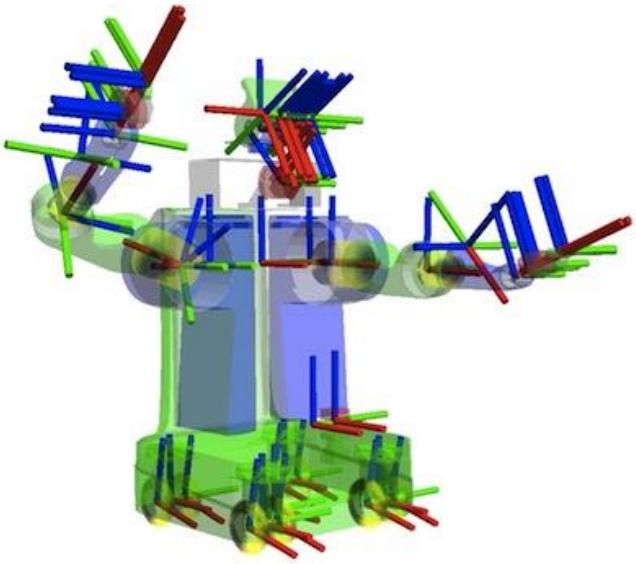


Figura 4 Geometría Robot

Diseñada pensando en la eficacia, la librería *tf* se utiliza para administras, coordinar y transformar los datos para robots con más de un centenar de grados de libertad con tasas de actualización de cientos de Hertz. Permite a su vez definir las transformaciones estáticas y dinámicas y maneja el hecho de que los productores y consumidores de la información del sistema se pueden distribuir a través de la red y el hecho de que la información se actualiza a diferentes velocidades.

2.1.3.1.2.3 Lenguaje de descripción robot

Otro problema común de la robótica que ROS resuelve es cómo describir nuestro robot de forma legible por una máquina. ROS proporciona un conjunto de herramientas para describir y modelar nuestro robot para que pueda ser comprendido por el resto de nuestro sistema ROS, incluyendo *tf, robot_state_publisher* y *rviz.* El formato para la descripción del robot de ROS es URDF (Unified Robot Description Format), que consiste en un documento XML en el que se describen las propiedades físicas del robot.

Una vez definidas de esta manera, el robot se puede utilizar fácilmente mediante la librería *tf,* renderizado en tres dimensiones para obtener buenas visualizaciones y se utiliza junto a los simuladores y planificadores de movimiento.

2.1.3.1.2.4 Llamadas a procedimiento remoto

Mientras que los *topics* (publicación/subscripción anónima) y servicios (llamadas a procedimiento remoto) cubren la mayor parte de los casos de uso de la comunicación en la robótica, a veces se necesita para iniciar un comportamiento "goal-seeking", monitorear el progreso, capacidad de anticipación y capacidad para notificar la consecución de un objetivo. ROS proporciona acciones a tal fin. Dichas acciones son como los servicios excepto que pueden reportar los avances antes de devolver la respuesta final y pueda ser anulada por el llamador. Una acción es un concepto muy poderoso en el ecosistema ROS.

2.1.3.1.2.5 Diagnóstico

ROS proporciona una forma estándar para producir, recopilar y añadir diagnósticos globales sobre el robot para que, a simple vista, se pueda observar el estado del robot y determinar la forma de abordar los problemas a medida que puedan aparecer.

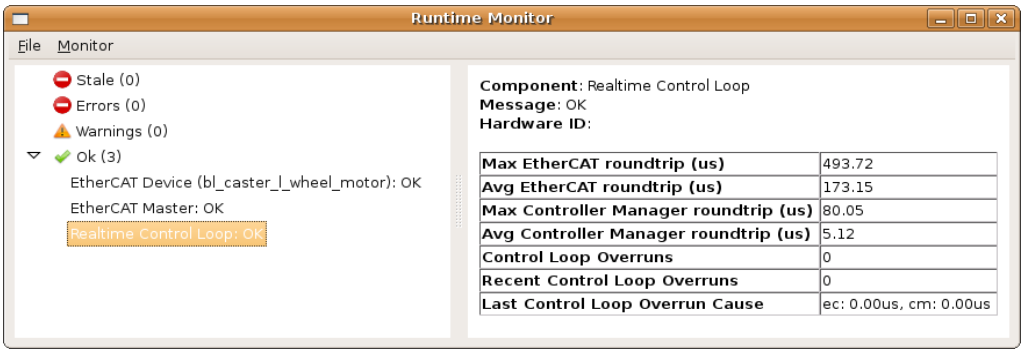


Figura 5 Herramienta de diagnóstico

2.1.3.1.2.6 Estimación de la posición, navegación y localización

ROS proporciona además paquetes que solucionan los problemas clásicos de la robótica como la estimación de la posición, la localización en un mapa, la construcción de un mapa e incluso la navegación móvil.

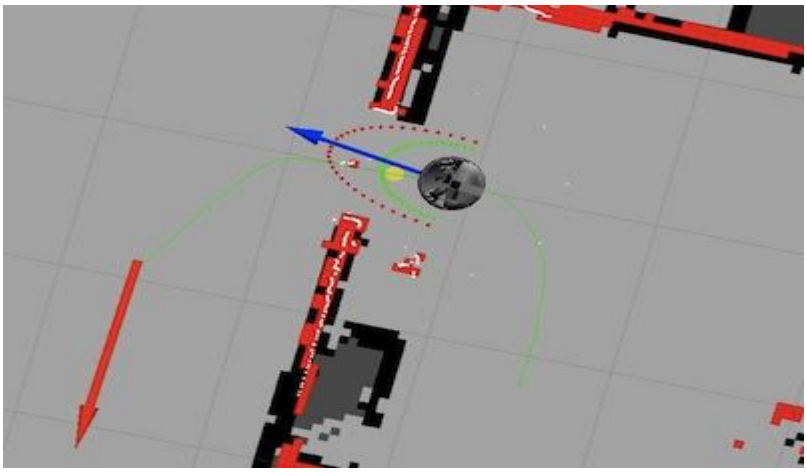


Figura 6 ROS Navigation

2.1.3.1.3 Herramientas

Uno de los rasgos más característicos de ROS es el poderoso conjunto de herramientas de desarrollo. Estas herramientas apoyan la introspección, la depuración, el trazado y visualización del estado del sistema en desarrollo. El mecanismo *publicador/subscriptor* permite la introspección de forma espontánea de los datos que fluyen a través del sistema, por lo que es fácil comprender y depurar los problemas que se produzcan. Las herramientas de ROS se aprovechan d esta capacidad de introspección a través de una extensa colección de utilidades de línea de comandos y gráficos que simplifican el desarrollo y depuración.

2.1.3.1.3.1 Herramientas en línea de comandos

Todas las funcionalidades del núcleo y herramientas de introspección pueden ser utilizadas a través de uno de las más de 45 herramientas en línea de comandos sin necesidad de una interfaz. Si por el contrario preferimos utilizar herramientas gráficas, rvz y rqt proveen una similar (y extendida) funcionalidad.

2.1.3.1.3.2 Rviz

Tal vez la herramienta más conocida en ROS. Rviz proporciona visualización tridimensional, de propósito general, de muchos tipos de datos de los sensores y cualquier robot descrito de la forma URDF.

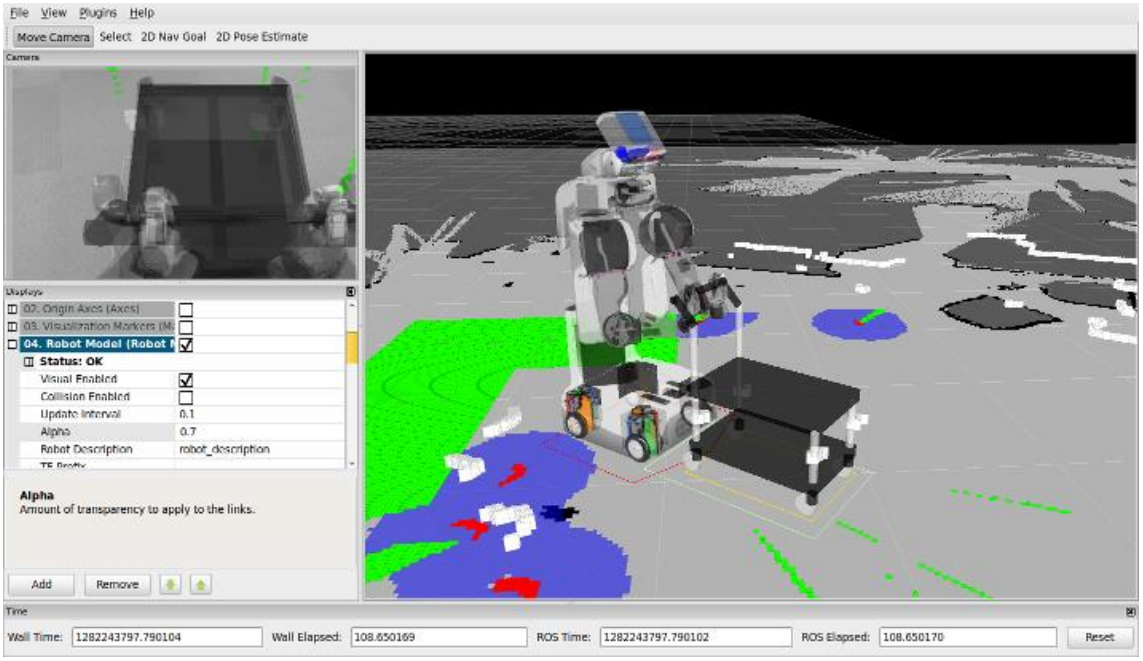


Figura 7 rviz

Rviz puede visualizar muchos de los tipos de mensajes comunes previstos en ROS. A su vez utiliza la información de la biblioteca *tf* para mostrar todos los datos de sensores en un sistema de coordenadas común así como una representación tridimensional de nuestro robot. La ventaja de poder ver todos los datos en una misma aplicación reside en poder analizar rápidamente lo que ve nuestro robot e identificar posibles problemas.

2.1.3.1.3.3 rqt

ROS proporciona *rqt*, un marco basado en Qt para el desarrollo de interfaces gráficas para nuestro robot. Se pueden crear interfaces personalizadas componiendo y configurando la extensa biblioteca de pluggins *rqt* así como introducir nuevos componentes de interfaz escribiendo nuestros propios pluggins *rqt*.

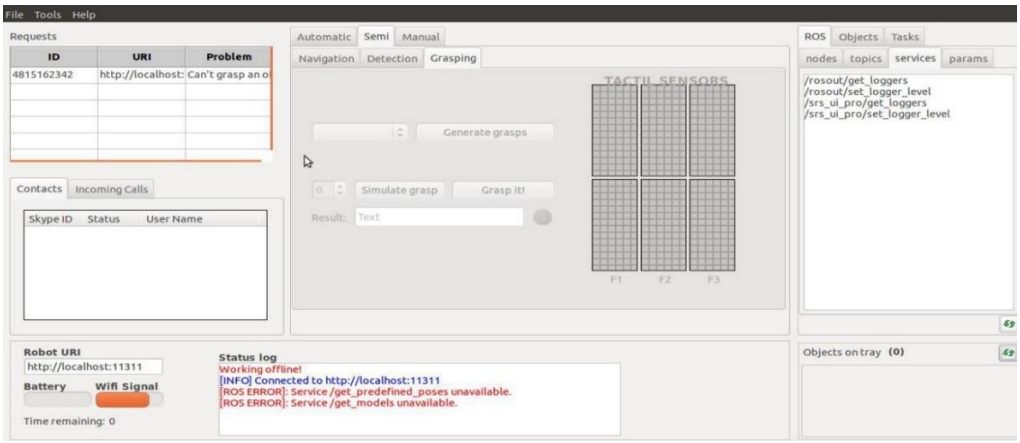


Figura 8 rqt

El pluggin *rqt_graph* ofrece la introspección y la visualización de un sistema ROS en vivo, mostrando los nodos y las conexiones entre ellos, lo que permite fácilmente depurar y entender nuestro sistema funcionando y su estructura.

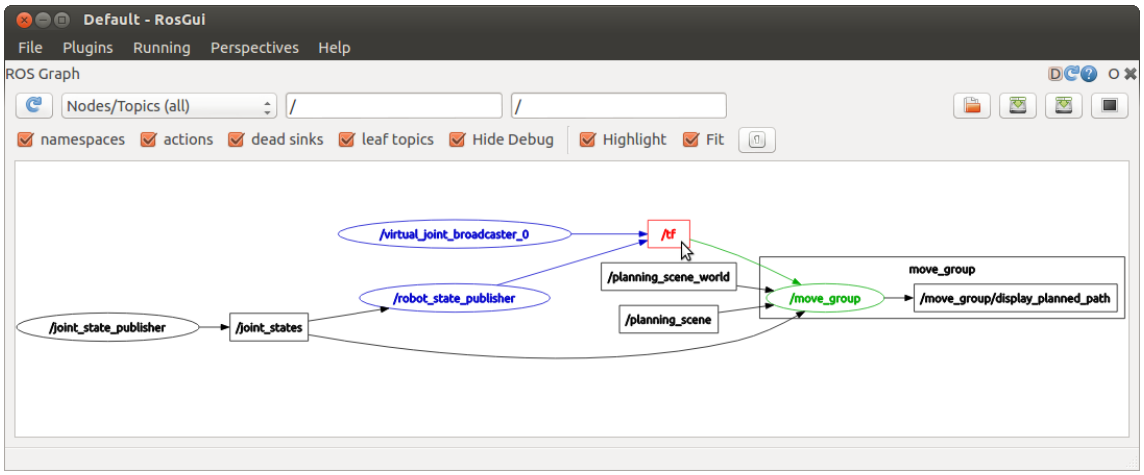


Figura 9 Pluggin rqt_graph

Con el pluggin *rqt_plot*, podemos supervisar codificadores, voltajes o cualquier cosa que se pueda representar como un número variable con el tiempo. Dicho pluggin nos permite elegir el "backend" de trazado que mejor se adapte a nuestras necesidades.

Para la monitorización y uso de *topics* tenemos *rqt_topic* y *rqt_publisher*. El primero permite supervisar y la introspección de cualquier número de temas que se publican en el sistema. El último permite publicar nuestros propios mensajes a cualquier tema, lo que facilita la experimentación ad hoc con el sistema.

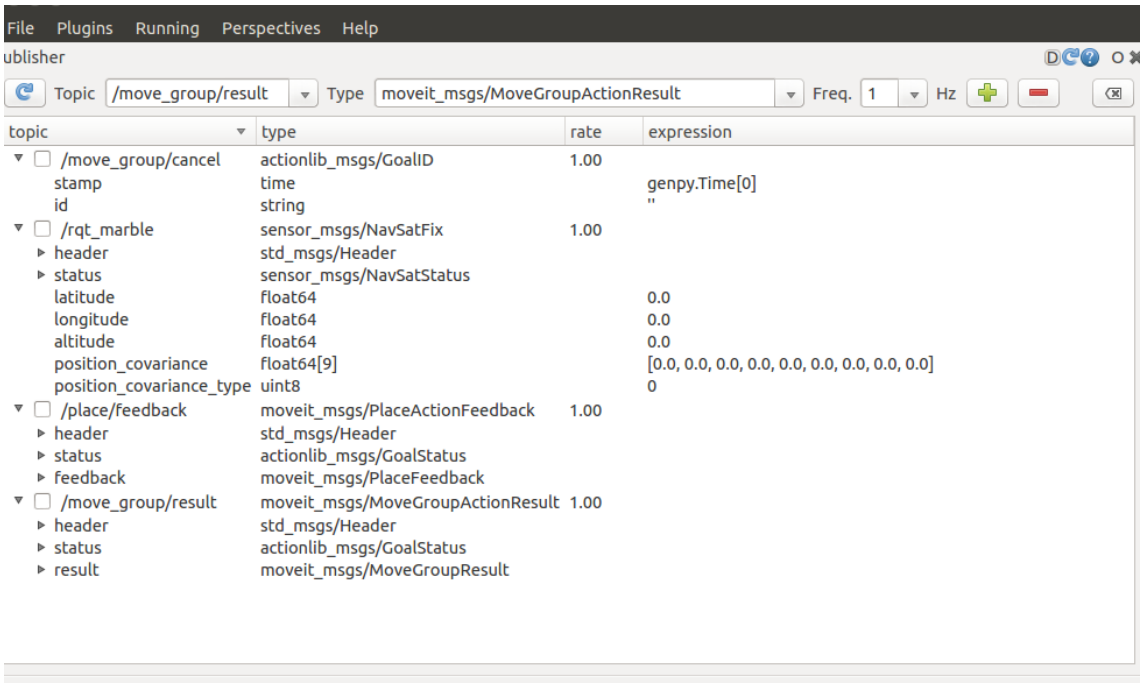


Figura 10 Pluggin rqt_publisher

Para el registro de datos y la reproducción de los mismos ROS utiliza el formato de "bolsa". Los archivos "bolsa" se pueden crear y acceder de forma gráfica a través del pluggin *rqt_bag*.

2.1.4 Historial de versiones

Una versión de ROS puede ser incompatible con otra. Normalmente las versiones están referidas por un sobrenombre en vez de por una versión numérica.

- September 4, 2013 - Hydro Medusa
- December 31, 2012 - Groovy Galapagos
- April 23, 2012 - Fuerte
- Aug 30, 2011 - Electric Emys
- March 2, 2011 - Diamondback
- August 3, 2010 - C Turtle
- March 1, 2010 - Box Turtle
- January 22, 2010 - ROS 1.0

La versión elegida para la realización de este proyecto fue ROS Groovy Galapagos debido a su estado de desarrollo a la hora de comenzar el proyecto y su integración con los demás elementos utilizados a lo largo del mismo: Raspberry Pi y Arduino MEGA ADK.

2.1.5 Conceptos básicos de ROS

A continuación se definen una serie de conceptos y terminología básica de ROS que se utiliza a lo largo de todo el proyecto y sin los cuales resultaría complicado entender correctamente el significado.

2.1.5.1 Roscore

El servicio *roscore* está constituido por una serie de nodos y programas que son pre-requisitos del sistema base ROS. Se ha de tener una instancia de *roscore* funcionando para que los nodos de ROS puedan comunicarse. Se inicia mediante el comando siguiente invocado desde línea de comandos:

```
$ roscore
```

2.1.5.2 Nodos

Un nodo es un proceso que realiza la computación. Los nodos se combinan juntos en un grafo y se comunican entre sí utilizando los topics, servicios RPC y el servidor de parámetros. Los nodos están pensados para operar una escala muy fina; un sistema de control robot suele contener muchos nodos.

El uso de los nodos en ROS proporciona varios beneficios para el sistema global. Proporcionan tolerancia adicional a los errores en la forma en la que los incidentes son aislados en los nodos individualmente. La complejidad del código se reduce en comparación con códigos monolíticos.

Un nodo de ROS está escrito mediante el uso de una biblioteca de cliente de ROS, tales como roscpp o rospy.

2.1.5.3 Topics

Los topics son conocidos como los buses a través de los cuales se intercambian mensajes entre los nodos. Tienen una semántica de publicación/suscripción anónima, lo cual desacopla la producción de información de su consumo. En general, los nodos no son conscientes de que se están comunicando. Si un nodo está interesado en un tipo de dato en concreto se subscribe al topic en cuestión. Los nodos que generan datos los publican en el topic correspondiente. Pueden existir varios publicadores y subscriptores a un mismo topic.

Los topics están pensados para trabajar de forma unidireccional estableciendo la comunicación.

Las herramientas más comunes para trabajar con topics son:

```
$ rostopic list
```

la cual se encarga de listar los topics que están activos actualmente y

```
$ rostopic echo /nombre_del_topic
```

la cual permite mostrar por pantalla los mensajes que se estén publicando en el topic "/topic_name".

2.2 RASPBERRY PI

2.2.1 Introducción

Raspberry Pi es un ordenador de placa reducida o (placa única) (SBC) de bajo costo, desarrollado en Reino Unido por la Fundación Raspberry Pi, con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas.

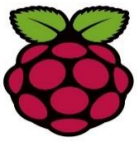


Figura 11 Logo Raspberry

El diseño incluye un System-on-a-chip Broadcom BCM2835, que contiene un procesador central (CPU) ARM1176JZF-S a 700 MHz (el firmware incluye unos modos "Turbo" para que el usuario pueda hacerle overclock de hasta 1 GHz sin perder la garantía), un procesador gráfico (GPU) VideoCore IV, y 512 MiB de memoria RAM (aunque originalmente al ser lanzado eran 256 MiB). El diseño no incluye un disco duro ni unidad de estado sólido, ya que usa una tarjeta SD para el almacenamiento permanente; tampoco incluye fuente de alimentación ni carcasa. El 29 de febrero de 2012 la fundación empezó a aceptar órdenes de compra del modelo B, y el 4 de febrero de 2013 del modelo A.

La fundación da soporte para las descargas de las distribuciones para arquitectura ARM, Raspbian (derivada de Debian), RISC OS, Arch Linux ARM (derivado de Arch Linux) y Pidora (derivado de Fedora); y promueve principalmente el aprendizaje del lenguaje de programación Python, y otros lenguajes como Tiny BASIC, C y Perl.

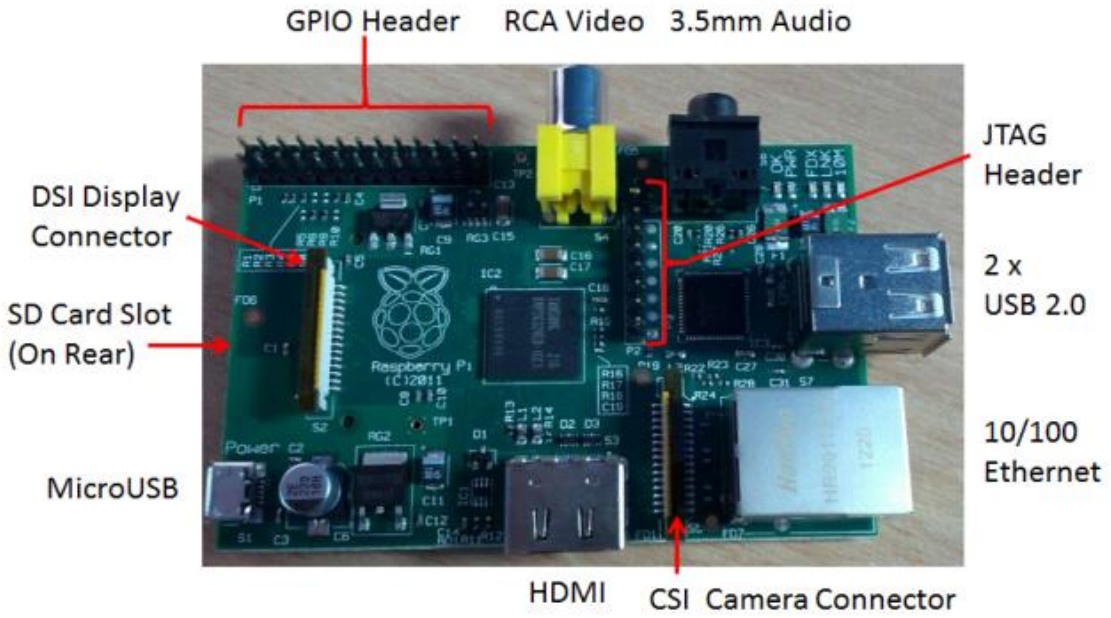


Figura 12 Ordenador Raspberry Pi

2.2.2 Historia

En 2006, los primeros diseños de Raspberry Pi se basaban en el micro controlador Atmel ATmega644. Sus esquemas y el diseño del circuito impreso están disponibles para su descarga pública.

En mayo de 2009, la Fundación Raspberry Pi fue fundada en Caldecote, South Cambridgeshire, Reino Unido como una asociación caritativa que es regulada por la Comisión de Caridad de Inglaterra y Gales.

El administrador de la fundación, Eben Upton, se puso en contacto con un grupo de profesores, académicos y entusiastas de la informática para crear un ordenador con la intención de animar a los niños a aprender informática como lo hizo en 1981 el ordenador Acorn BBC Micro.15 16 El primer prototipo basado en ARM se montó en un módulo del mismo tamaño que una memoria USB. Tenía un puerto USB en un extremo y un puerto HDMI en el otro.

En agosto de 2011, se fabricaron cincuenta placas Alpha, que tenían las mismas características que el modelo B, pero eran un poco más grandes para integrar bien unas interfaces para depuración. En algunas demostraciones se podía ver la placa ejecutando el escritorio LXDE en Debian, Quake 3 a 1080p y vídeo Full HD H.264 a través de la salida HDMI.

En diciembre de 2011, 25 placas Beta del modelo B fueron ensambladas y probadas de un total de 100 placas vacías. El diagrama de componentes de las placas finales sería el mismo que el de esas placas Beta. Se hizo una demostración de la placa beta arrancando Linux, reproduciendo un tráiler de una película a 1080p y ejecutando el benchmark Rightware Samurai OpenGL ES.

Las primeras ventas comenzaron el 29 de febrero de 2012. Premier Farnell vendió toda su existencia de inventario a los pocos minutos del momento de lanzamiento, mientras que RS Components tuvo 100.000 peticiones de interés el primer día. En los seis meses siguientes llegarían a vender 500.000 unidades.

2.2.3 Hardware

El Raspberry PI no viene con un reloj en tiempo real, por lo que el sistema operativo debe usar un servidor de hora en red, o pedir al usuario la hora en el momento de arrancar el ordenador. Se podría añadir un reloj en tiempo real con una batería mediante el uso de la interface I2C.

El modelo utilizado para la elaboración de este proyecto fin de carrera ha sido el Raspberry PI Modelo B Rev.2 cuyas características se muestran en la **tabla 1**.

Características Modelo B	
SoC:	Broadcom BCM2835 (CPU + GPU + DSP + SDRAM + puerto USB)
CPU:	ARM 1176JZF-S a 700 MHz (familia ARM11)[3]
GPU:	Broadcom VideoCore IV , OpenGL ES 2.0, MPEG-2 y VC-1 (con licencia), 1080p30H.264/MPEG-4 AVC
Memoria (SDRAM):	512 MiB (compartidos con la GPU) desde el 15 de octubre de 2012
Puertos USB 2.0:	2 (vía hub USB integrado)
Entradas de vídeo:	Conector MIPI CSI que permite instalar un módulo de cámara desarrollado por la RPF
Salidas de vídeo:	Conector RCA (PAL y NTSC), HDMI (rev1.3 y 1.4), Interfaz DSI para panel LCD[62] [63]
Salidas de audio:	Conector MIPI CSI que permite instalar un módulo de cámara desarrollado por la RPF

Almacenamiento integrado:	Conector RCA (PAL y NTSC), HDMI (rev1.3 y 1.4), Interfaz DSI para panel LCD
Conectividad de red:	10/100 Ethernet (RJ-45) via hub USB
Periféricos de bajo nivel:	8 x GPIO, SPI, I²C, UART
Reloj en tiempo real:	Ninguno
Consumo energético:	700 mA, (3.5 W)
Fuente de alimentación:	5 V vía Micro USB o GPIO header
Dimensiones:	85.60mm × 53.98mm (3.370 × 2.125 inch)
Sistemas operativos soportados:	GNU/Linux: Debian (Raspbian), Fedora (Pidora), Arch Linux (Arch Linux ARM), Slackware Linux. RISC OS

Tabla 1 Características Raspberry Pi

La **figura 13** recoge las interfaces y muestra el *Pinout* del Raspberry PI.

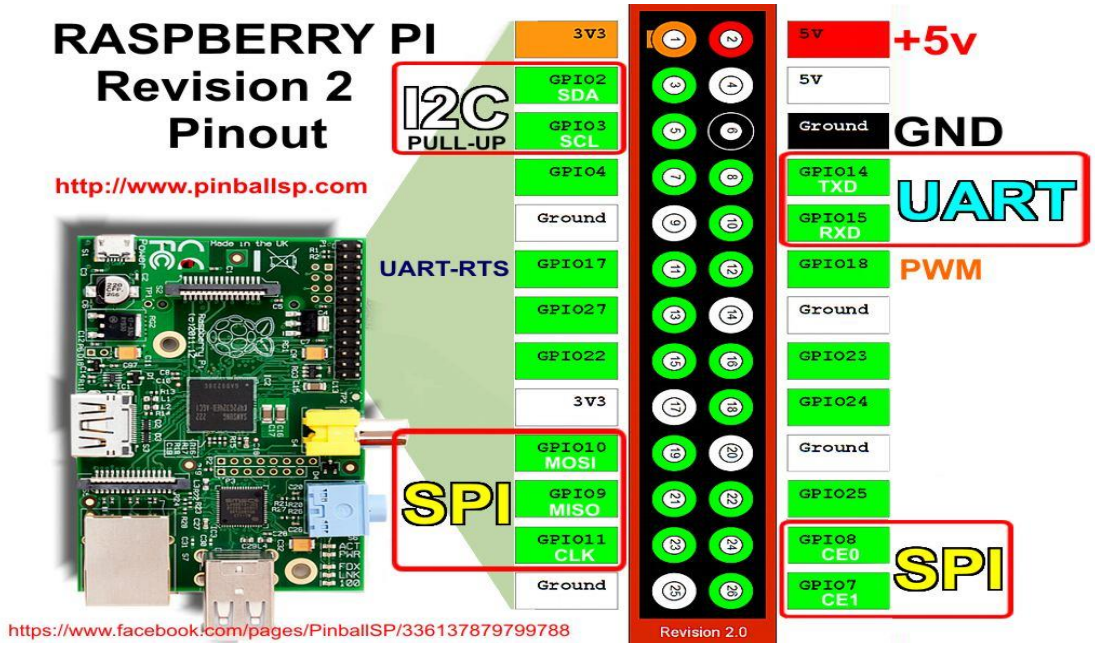


Figura 13 Pinout Raspberry Pi

2.2.4 Conclusiones

A pesar de tratarse de un elemento altamente poderoso y beneficioso a la hora de afrontar el problema del diseño y programación de robots encontramos algunas carencias que nos hacen necesitar otro elemento que nos facilite la tarea (Arduino, el cual veremos más adelante):

- Número de interfaces analógicas reducido
- Número de interfaces digitales reducido
- Vulnerabilidad antes posibles inclemencias como por ejemplo golpes y el soporte sobre el cual se instala el sistema operativo (SDCARD).

2.3 ARDUINO

2.3.1 Introducción

Arduino es una plataforma de hardware libre, basada en una placa con un micro controlador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

Figura 14 Logo Arduino

El hardware consiste en una placa con un micro controlador Atmel AVR y puertos de entrada/salida.4 El micro controlador más usado son el Atmega168, Atmega328, Atmega1280, ATmega8 por su sencillez y bajo coste que permiten el desarrollo de múltiples diseños. Por otro lado el software consiste en un entorno de desarrollo que implementa el lenguaje de programación Processing/Wiring y el cargador de arranque que es ejecutado en la placa.

Se puede utilizar para desarrollar objetos interactivos autónomos o puede ser conectado a software tal como Adobe Flash, Processing, Max/MSP, Pure Data). Las placas se pueden montar a mano o adquirirse. El entorno de desarrollo integrado libre se puede descargar gratuitamente. Puede tomar información del entorno a través de sus entradas y controlar luces, motores y otros actuadores. El micro controlador en la placa Arduino se programa mediante el lenguaje de programación Arduino (basado en Wiring) y el entorno de desarrollo Arduino (basado en Processing). Los proyectos hechos con Arduino pueden ejecutarse sin necesidad de conectar a un computador.

2.3.2 Arduino MEGA ADK

2.3.2.1 Visión general

Para la realización de este proyecto fin de carrera se ha escogido la placa Arduino MEGA ADK. La justificación es sencilla y será comentada en las líneas siguientes.

Figura 15 Arduino Mega ADK Front

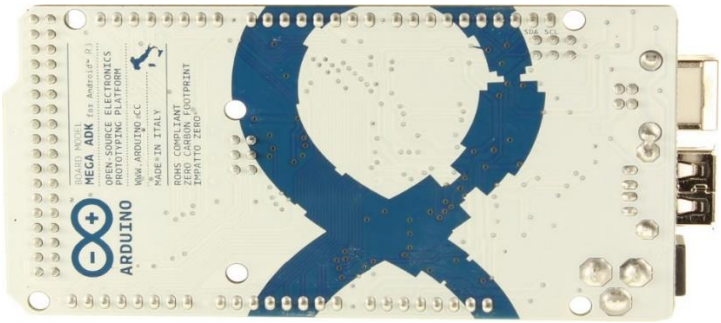


Figura 16 Arduino Mega ADK Back

El Arduino MEGA ADK es una placa basada en el micro controlador ATmega2560. Cuenta con una interfaz host USB para conectar con teléfonos basados en Android, con base en la MAX3421e IC. Dicha interfaz la utilizaremos en el presente proyecto para interconectar las plataformas Raspberry PI y Arduino. Tiene 54 pines de entrada/salida digitales (15 de los cuales pueden ser usadas como salidas PWM), 16 entradas analógicas, 4 UARTS (HW de puerto serie), un oscilador de cristal a 16MHz, una conexión USB, conector de corriente, header ICSP y botón de reset.

Concretamente el Arduino Mega ADK elegido se corresponde con la revisión 3 de la placa la cual trae las siguientes nuevas especificaciones:

- 1.0 Pinout : Añadidos los pines SDA y SCL colocados junto al pin AREF y dos nuevo pines situados cerca del pin RESET, el IOREF que permite a los escudos adaptarse al voltaje facilitado por la placa y un segundo pin reservado para usos futuros.
- Circuito de RESET más fuerte.

2.3.2.2 Resumen de características

En la **tabla 2** se muestran a modo resumen las características principales del Arduino MEGA ADK rev 3:

Micro controlador	ARmega2560
Tensión de funcionamiento	5V
Tensión de entrada recomendada	7-12V

Tensión de entrada límite	6-20V
Pines I/O digitales	54 (15 de los cuales proporcionan salidas PWM)
Pines analógicos de entrada	16
Corriente DC por cada pin I/O	40mA
Corriente DC por el pin 3,3V	50mA
Memoria Flash	256KB (8KB utilizados por el bootloader)
SRAM	8KB
EEPROM	4KB
Velocidad del Reloj	16 MHZ

Tabla 2 Características Arduino Mega ADK

2.3.2.3 Alimentación

El Arduino MEGA ADK puede ser alimentado a través de la conexión USB o mediante una fuente de alimentación externa. La fuente de alimentación se selecciona automáticamente en caso de estar ambas presentes.

Debido a que el MEGA ADK trabaja como un host USB, un teléfono Android conectado a dicha interfaz tratará de extraer energía para cargar su propia batería. Cuando el MEGA ADK se alimenta mediante 500mA a través de USB, dicha cantidad es la total disponible para la alimentación del teléfono y la placa. El regulador externo de alimentación puede suministrar hasta 1500mA. 750mA estarían disponibles para el teléfono conectado en la interfaz y para la placa. Los 750mA adicionales serían utilizados para alimentar actuadores y sensores conectados a la placa.

La placa puede operar con una fuente externa que suministre entre 5,5 y 16V. Si suministra menos de 7V, el pin 5V suministrará menos de 5V y la placa podría comportarse de forma inestable. Se ha comprobado que utilizando un cargador estándar de Smartphone que proporciona 5V y 1.2A alimentando el Raspberry Pi, éste es suficiente para alimentar tanto el Raspberry como el Arduino y que se comporten de forma estable.

Los pines de alimentación son los siguientes:

- VIN: Pin utilizado para alimentar el Arduino cuando se utiliza una fuente de alimentación externa (en vez de la conexión USB de 5V u otra fuente regulada).
- 5V: Este pin proporciona una salida de 5V regulada por la placa. El suministro de tensión a través de los pines 5V o 3,3V no pasa por el regulador interno de la placa y podría ocasionar daños.
- 3,3V: Pin que proporciona una tensión de 3,3V controlados por el regulador de la placa. La máxima corriente es de 50mA.
- GND: Pines de tierra.
- IOREF: Proporciona la referencia de tensión con la que opera un micro controlador. Un escudo configurado puede leer el voltaje de dicho pin y seleccionar la fuente de alimentación adecuada o habilitar traductores de tensión en las salidas para trabajar con los 5V o los 3,3V.

2.3.2.4 Pinout

En la **figura 17** se presenta de forma esquemática la distribución de todos los pines de la placa Arduino MEGA ADK:

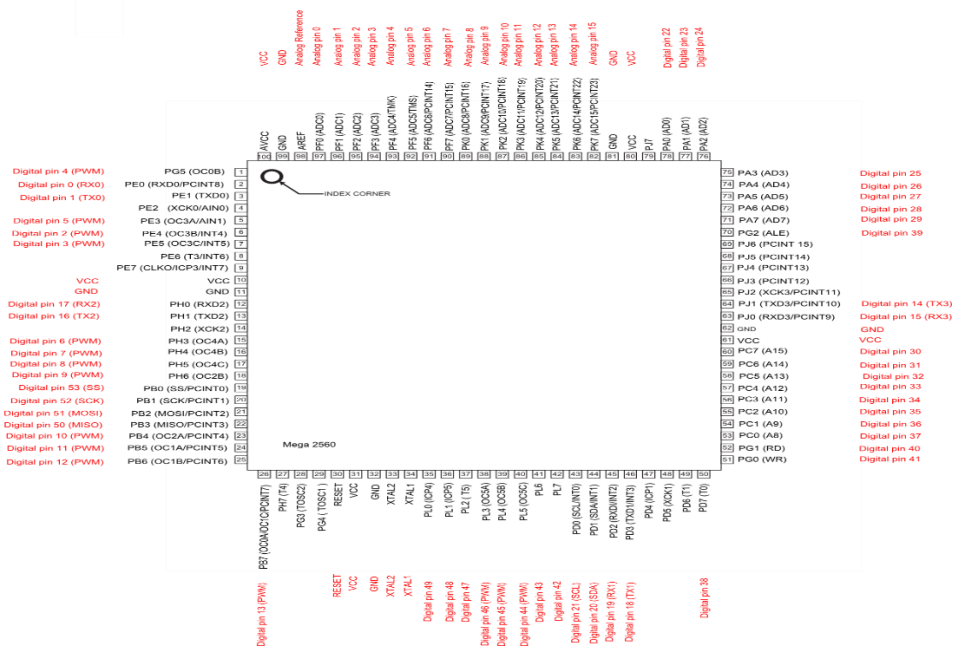


Figura 17 Pinout Arduino Mega ADK

2.3.2.5 Análisis y conclusiones

Tras estudiar las características y analizar qué nos ofrece la placa Arduino MEGA ADK llegamos a la conclusión de que el Arduino Mega ADK, al contar con un gran número de entradas y salidas, se sitúa como una alternativa poderosa a la hora de ser aprovechado para la programación de robots. Se constituye como una plataforma de bajo coste ideal el control de motores, interactuación con sensores, comunicación serie y control hardware gracias a sus oferta de pines que pueden ser usados como PWM, convertidores A/D y el amplio abanico de entradas y salidas digitales.

Las principales ventajas que nos encontramos son:

- Lenguaje de programación C y C++
- Entorno de trabajo compatible con las principales plataformas actuales: Windows, Linux y MacOS.
- Posibilidad de aportar funcionalidad mediante la adición de librerías.
- Comunicación serie USB.
- Plataforma de código abierto
- Disponibilidad de pines I/O tanto digitales como analógicos y funcionalidad
- Amplia comunidad de desarrollo.

De igual manera nos encontramos con ciertas desventajas de esta plataforma las cuales para el proyecto que acometemos no han supuesto deficiencias triviales:

- Tenemos una capacidad limitada a la hora de cargar programas en la placa puesto que tan solo disponemos de 256KB (de los cuales 8KB los necesita el bootloader)
- Procesador de 8 bits
- Limitada frecuencia de reloj (16MHz), la cual podría ser un factor limitante a la hora de trabajar con aplicaciones cuyo parámetro crítico pudiera ser el tiempo de cálculo.

3 PREPARACIÓN DEL ENTORNO DE TRABAJO

Tras este recorrido por los elementos principales que se van a integrar para dar vida a este proyecto llega el momento de pasar a la etapa de instalación y configuración de todos los elementos que vamos a necesitar.

3.1 Instalación y configuración del sistema operativo Raspbian en el Raspberry PI

3.1.1 Raspbian

Raspbian es una distribución del sistema operativo GNU/Linux y por lo tanto libre basado en Debian Wheezy (Debian 7.0) para la placa computadora (SBC) Raspberry Pi, orientado a la enseñanza de informática, optimizado para el hardware presente en la placa.

Figura 18 Raspbian

Técnicamente el sistema operativo es un port no oficial de Debian Wheezy armhf para el procesador (CPU) de Raspberry Pi, con soporte optimizado para cálculos en coma flotante por hardware, lo que permite dar más rendimiento en según qué casos. El port fue necesario al no haber versión Debian Wheezy armhf para la CPU ARMv6 que contiene el Raspberry PI.

Destaca también el menú "raspi-config" que permite configurar el sistema operativo sin tener que modificar archivos de configuración manualmente. Entre sus funciones, permite expandir la partición root para que ocupe toda la tarjeta de memoria, configurar el teclado, aplicar overclock, etc

3.1.2 Requisitos

3.1.2.1 SDCARD y lector

Debido a que el Raspberry Pi utiliza como medio de almacenamiento permanente así como soporte para el sistema operativo una tarjeta de memoria SDCARD es el primero elemento imprescindible para continuar. Se ha utilizado una tarjeta Kingston 8GB class 4, como la de la **figura 19**, la cual se antoja más que suficiente para el desarrollo del proyecto.

Será necesario además un lector de tarjetas para poder trabajar con la SD previo a su colocación en el Raspberry Pi.

Figura 19 Tarjeta SD Kingston 8GB Class4

3.1.2.2 Imagen Wheezy Raspbian

Imprescindible de igual manera descargar una imagen del sistema operativo Wheezy Raspbian para su posterior montado en la tarjeta de memoria mediante el software adecuado.

Para tal efecto debemos dirigirnos a la web www.raspberrypi.org/downloads y descargar la última versión disponible de Wheezy Raspbian tal y como se muestra en la **figura 20**.

Raspbian

Image	2014-01-07-wheezy-raspbian.zip
Torrent	2014-01-07-wheezy-raspbian.zip.torrent
SHA-1 Checksum	9d0afbf932ec22e3c29d793693f58b0406bcab86
Default login	pi / raspberry
Description	A community-created port of Debian wheezy, optimised for the Raspberry Pi
Release Date	2014-01-07
Version	wheezy
Kernel	3.10
URL	Link
Release Notes	release_notes.txt

Figura 20 Versión descargable Wheezy-Raspbian

3.1.2.3 Win32 Disk Imager

Este programa está diseñado para escribir una imagen de disco sin procesar en un dispositivo extraíble o copia de seguridad en un dispositivo extraíble a un archivo de imagen. Lo necesitaremos para escribir la imagen de Wheezy Raspbian en nuestra SDCARD.

Para ello nos dirigiremos a la web http://sourceforge.net/projects/win32diskimager/ y descargarlo. La versión utilizada se corresponde con la destinada a trabajar bajo Windows. Existe de igual manera su versión equivalente para Linux.

3.1.2.4 Hardware y cableado

Inicialmente será requisito un teclado y un ratón para la configuración inicial de nuestro sistema operativo Raspbian. Una vez convenientemente instalado, configurado y conectado a nuestra red doméstica (LAN/WLAN) no será necesario porqué utilizarlos de nuevo pues se accederá al sistema mediante SSH o conexiones a escritorio remoto.

De igual manera será necesario un cable HDMI para su conexión a un monitor/TV para la fase de configuración inicial.

Por último y no menos importante se necesita alimentar el Raspberry Pi. Para ello se ha utilizado un cargador de Smartphone que proporciona 5V de salida y 1.2A que cumplen los requisitos de alimentación y se antojan más que suficientes.

3.1.3 Instalación

Una vez satisfechos los requisitos previos, la instalación en la tarjeta de memoria se antoja sencilla a la vez que rápida gracias al software dedicado para ello.

En el Win32 Disk Imager, **figura 21**, se ha deseleccionar el archivo ".img" en "Image File" y la unidad donde está la tarjeta SD en "Device". Después, pulsar el botón "Write" para comenzar con la carga del sistema en la tarjeta SD. En la parte de abajo de la ventana de Win32 Disk Imager mostrara la velocidad en MB/s del proceso de copia.

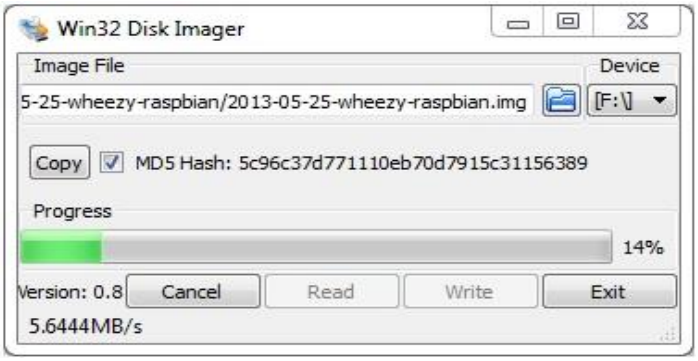


Figura 21 Win32 Disk Imager

Al finalizar se retira la tarjeta SD del ordenador desde el que se ha cargado el sistema y se conecta al Raspberry Pi.

3.1.4 Configuración

Una vez colocada nuestra SD arrancamos el dispositivo y esperamos a que termine de cargar.

Si todo transcurre sin más problemas se llega a la pantalla del menú de configuración *raspi-config* como se muestra en la **figura 22**.

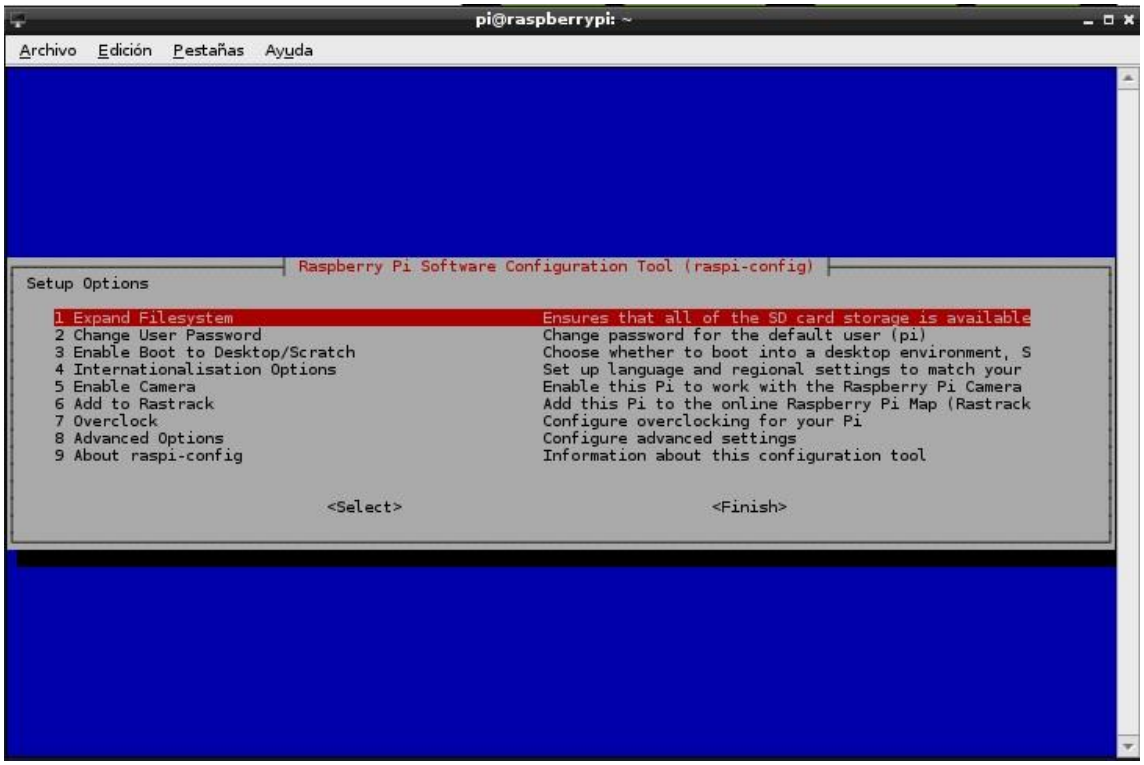


Figura 22 Raspi-Config

En esta pantalla inicial se pueden encontrar importantes opciones a la hora de configurar nuestro Raspbian en Raspberry Pi. Lo primero que se ha de hacer es pulsar sobre la opción 1 *Expand Filesystem* lo cual permitirá expandir el sistema operativo para que utilice todo el espacio disponible en la tarjeta de memoria.

A continuación, en la opción 4, **figura 23**, *Internationalization Options.* Esta opción permitirá configurar el idioma del sistema operativo, la zona horaria que nos corresponde y la distribución del teclado.

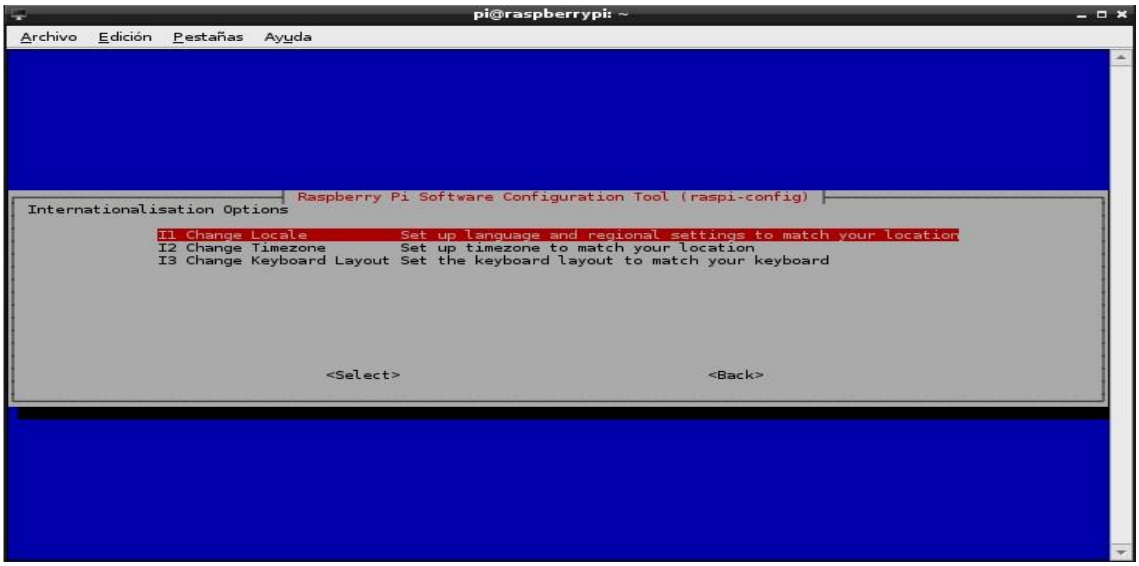


Figura 23 RaspiConfig International Options

La opción I1 permitirá elegir nuestra ubicación para así determinar el lenguaje del sistema operativo, caracteres, moneda, etc., para la cual elegimos *es_ES. UTF-8.*

La opción I2 permitirá seleccionar la zona horaria del sistema para ajustarla a la ciudad en la que nos encontramos.

La opción I3 permitirá cambiar la configuración de nuestro teclado. Si no aparece la marca y modelo del teclado, seleccionar *PC genérico 105 teclas (intl).* A continuación se elige el idioma del teclado y la distribución y se establecerá por defecto *es_ES UTF-8.*

Las siguientes dos ventanas, como las de la **figura 24**, permiten configurar la tecla *AltGr* izquierda y derecha para funciones especiales. Se ha seleccionado la primera opción de forma predeterminada.

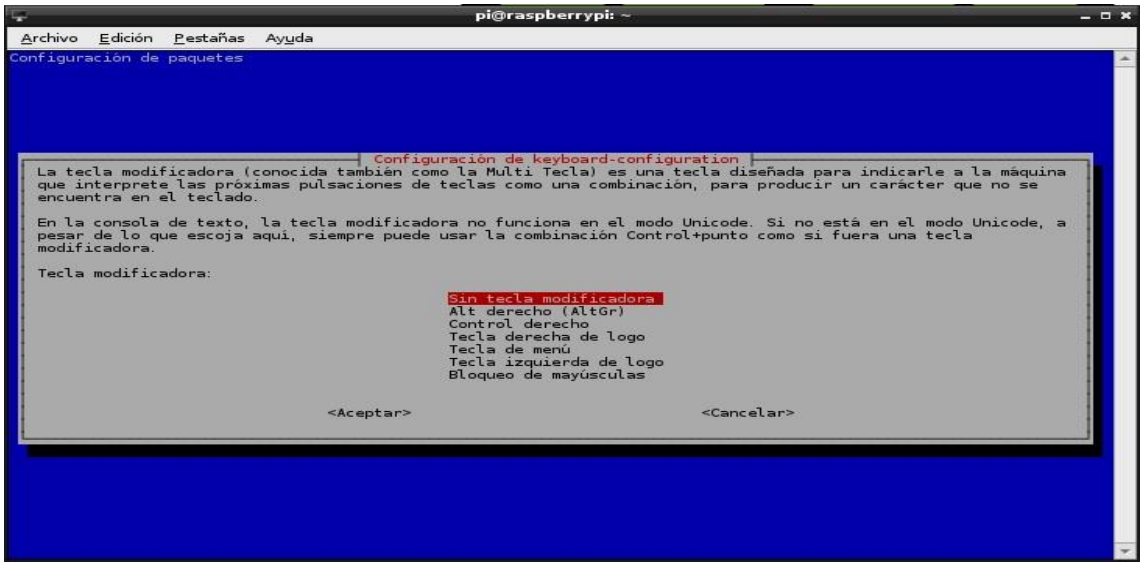


Figura 24 RaspiConfig AltGr

Por último el sistema pregunta si se desea activar *Ctrl+Alt+Retroceso* para terminar el Servidor X (xserver) es decir, cerrar el modo gráfico y pasar a modo consola. Es recomendable pulsar la opción predeterminada *No.*

Existen más funciones en el menú de configuración pero con estos pasos estará listo el sistema operativo para iniciar. Se reinicia desde el menú y tras la carga del sistema operativo se llega a la pantalla de login, donde se introducen los datos:

- **Usuario:** pi
- **Contraseña:** raspberry

Una vez hecho login en el sistema, se invoca el modo gráfico escribiendo en línea de comandos *startx* lo cual nos llevará al escritorio de Raspbian, **figura 25**.

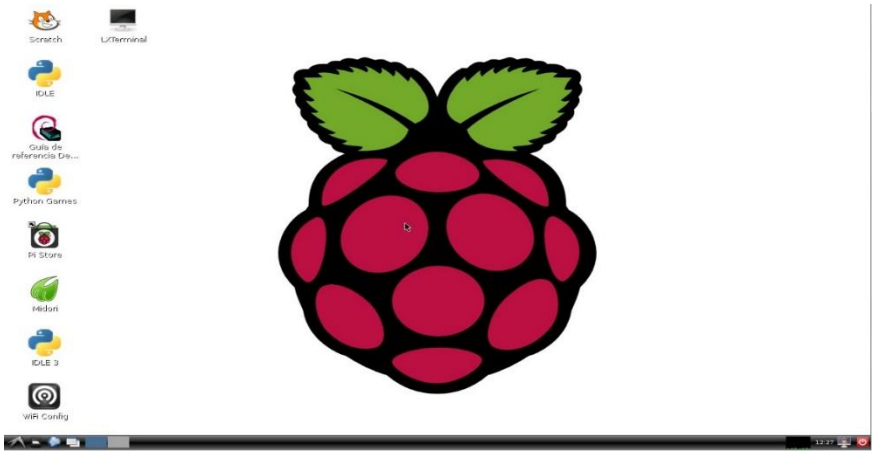


Figura 25 Escritorio Raspbian

3.1.4.1 Configuración interfaz wifi

A continuación y por requisito ligado al objetivo de este proyecto se dota al sistema de conectividad inalámbrica WIFI mediante un módulo inalámbrico nano USB 2.0, gracias al cual se evita la necesidad de tener conectado el Raspberry mediante cable Ethernet.

Para establecer la configuración de red se ha de editar el archivo */etc/network/interfaces* haciendo uso de la herramienta *nano* en un terminal. Se añade en dicho fichero lo siguiente:

```
auto lo
iface lo inet loopback
iface eth0 inet dhcp

allow-hotplug wlan0
auto wlan0

iface wlan0 inet dhcp
        wpa-ssid "Red a utilizar"
        wpa-psk "contraseña"
```

Una vez establecida la configuración reiniciamos la interfaz WIFI de Raspbian en línea de comandos y ya estará listo el acceso inalámbrico al dispositivo a través de la red doméstica mediante sesiones SSH:

```
$ ifdown Wlan0

$ ifup Wlan0
```

3.1.4.2 Configuración escritorio remoto

Si a su vez se desea disponer de la alternativa de conectarnos al Raspberry mediante escritorio remoto se ha de instalar el demonio que se ejecutará en Raspbian y permitirá dichas conexiones. Para ello se ejecuta en un terminal lo siguiente:

```
$ sudo apt-get install xdrp
```

3.2 Instalación del paquete ROS

Una vez instalado y configurado el sistema operativo Raspbian el siguiente paso se corresponde con la instalación de ROS.

3.2.1 ROS Groovy Galapagos

La versión elegida para la realización del proyecto ha sido ROS Groovy Galapagos. Es la sexta gran versión de distribución de ROS y fue lanzada el 31 de diciembre 2012. En esta versión se ha trabajado sobre la infraestructura central de ROS para que sea más fácil de usar, más modular, más escalable, compatible con un mayor número de sistemas operativos/arquitecturas de hardware/robots y lo más importante para involucrar más a la comunidad ROS.

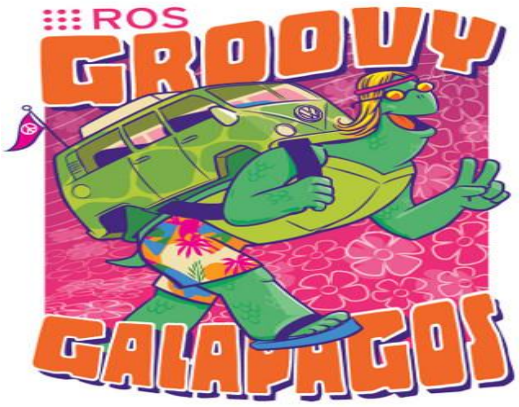


Figura 26 ROS Groogy Galapagos

Las principales característica de esta distribución son las siguientes:

- Migración de los paquetes del núcleo ROS a Gtihub así como el seguimiento de incidencias.
- Introducción de un nuevo sistema de construcción llamado *Catkin* que sustituye completamente al sistema original *Rosbuild.*
- Con la nueva versión de *Catkin* en Groovy, el concepto de *Stacks* ha sido eliminado. Aparece el concepto de *Metapackage* para preservar las ventajas de los stacks.
- Nuevo sistema de publicación de paquetes: *Bloom*, el cual toma los paquetes *catkin* y los publica en un repositorio alojado en www.github.com.
- Nuevas herramientas GUI – *rqt*.
- Rediseño de la herramienta *rviz.*
- Re-escritura de la librería *pluginlib* y *class_loader*. La primera permite crear pluggins en lenguaje de programación C++ mientras que la segunda es un paquete independiente que puede utilizarse para trabajar con pluggins software que no utilizan el sistema de construcción de ROS.
- Transición del entorno *Wt* al entorno *Qt*.

3.2.2 Métodos de instalación

Actualmente existen dos caminos para instalar ROS Groovy sobre Raspbian: construcción desde el código fuente o instalación de los paquetes binarios. Inicialmente se montó con éxito usando el primer camino para la instalación. Posteriormente surgieron problemas con el sistema de ficheros debían y hubo que reinstalar desde cero todo y se siguió el segundo método el cual recientemente había recomendado.

3.2.2.1 Instalación desde el código fuente

La instalación desde el código fuente requiere que se descargue y compile el código fuente por nuestra cuenta. La compilación del código fuente puede llevarnos días si no configuramos un entorno *Chroot* además de ciertas indecencias que hubo que ir solventando durante la instalación.

3.2.2.1.1 Creación del entorno *chroot*

Para construir ROS desde el código fuente es necesario la creación un entorno *chroot* en otro sistema. Dicho sistema debe contar con un sistema operativo Linux o un VirtualBox corriendo un sistema Linux. Esto nos permitirá compilar con mayor memoria y múltiples núcleos de CPU lo cual hará que dicha tarea sea mucho más rápida que si la ejecutamos directamente en el Raspberry Pi.

Para crear el entorno *chroot* necesitaremos el paquete *quemu* para emular el Raspberry Pi. Lo descargamos:

```
$ sudo apt-get install qemu qemu-user qemu-user-static
parted gparted kpartx
```

Ahora ya podemos copiar la tarjeta SD al ordenador mediante *dd* o montándola en una unidad USB como por ejemplo lector de tarjetas.

Se utilizó la imagen de Wheezy Raspbian *2014-01-07-wheezy-raspbian.img, descargada* tal y como se comentó anteriormente. La copiamos a nuestro ordenador corriendo *Ubuntu 12.04 LTS*. Para montar y desmontar la imagen se utilizaron los siguientes scripts:

```
#!/bin/bash

mnt=mount

root_disk=raspi.img

mnt_devices="proc   dev
dev/pts sys"

sudo mkdir "$mnt"

sudo      mount      -o
loop,offset=62914560
$root_disk "$mnt"

for i in $mnt_devices ;
do

sudo mount -o bind /$i
"$mnt"/$i

done
```

```
#!/bin/bash


mnt=mount

root_disk=raspi.img

mnt_devices="proc
dev/pts dev sys"

for i in $mnt_devices;
do

sudo umount "$mnt"/$i

done

sudo umount "$mnt"

sudo rmdir "$mnt"
```

Chroot nos logueará automáticamente como superusuario. Si no se desea hacer login como superusuario podemos cambiar al usuario normal escribiendo:

```
$ su pi
```

3.2.2.1.2 Instalación de dependencias

Lo primero que debemos hacer es añadir el repositorio de Raspbian a */etc/apt/sources.list*. A continuación instalamos las dependencias del núcleo del sistema:

```
$ echo "deb http://ros.raspbian.org/repo/ros/ wheezy main"
>> /etc/apt/sources.list

$ sudo apt-get update

$ sudo apt-get upgrade
```

A continuación se instalarán las dependencias del sistema:

```
$ sudo apt-get install build-essential python-yaml cmake
subversion wget python-setuptools mercurial git-core
libapr1-dev libaprutil1-dev libbz2-dev python-dev
libgtest-dev python-paramiko libboost-all-dev
liblog4cxx10-dev pkg-config python-empy swig python-nose
lsb-release python-pip python-gtk2
```

Una vez instaladas las dependencias del sistema instalamos las herramientas ROS de *bootstrapping:*

```
$ sudo easy_install wstool rospkg rosdep
rosinstall_generator
```

3.2.2.1.3 Instalación de ROS

Ahora continuamos por la construcción de los paquetes del núcleo de ROS.

Primero de todo creamos el área de instalación en el directorio que usaremos para ello:

```
$ sudo mkdir -p /opt/ros/groovy/ros_catkin_ws
```

El siguiente paso es asegurarnos de que *rosdep* se ha iniciado y actualizado. *Rosdep* es una herramienta disponible en línea de comandos que se utiliza para la instalación de las dependencias del sistema.

```
$ sudo rosdep init

$ sudo rosdep update
```

En el momento de la instalación de ROS Groovy Galapagos hemos de aclarar que el sistema de construcción de paquetes ROS se está migrando a uno nuevo: *catkin*. Pero no todos los paquetes se han adaptado aun a este nuevo sistema. Es por ello que primero se han de construir los paquetes del núcleo de ROS y luego el resto.

3.2.2.1.3.1 Creación del Catkin Workspace

Para continuar con la tarea de la construcción de los paquetes del núcleo de ROS se navega hasta el *catkin worskspace (espacio de trabajo catkin)*:

```
$ cd /opt/ros/groovy/ros_catkin_ws
```

A continuación es necesario obtener los paquetes del núcleo de ROS para su posterior instalación. Haciendo uso de la herramienta *wstool* para la variante que

vayamos a instalar. En este caso de las tres disponibles se siguen las recomendaciones de instalación por lo que se instala la versión *mobile*:

```
$ rosinstall_generator --rosdistro groovy mobile > /tmp/groovy_mobile.ri
$ sudo wstool init src -j1 /tmp/groovy_mobile.ri
```

Con estas líneas se agregan casi todos los paquetes de *catkin* para la variante que se ha elegido en el directorio */opt/ros/groovy/ros_catkin_ws/src*. Pasados unos minutos estarán todos los paquetes disponibles en el directorio comentado.

3.2.2.1.4 Dependencias restantes

A continuación hay que ejecutar lo siguiente en línea de comandos:

```
$ sudo rosdep install --from-paths src --ignore-src --rosdistro groovy -yr
```

- **--from-paths:** Le indicamos que queremos instalar las dependencias para un directorio completo de paquetes, en este caso *src*.
- **--ingore-src:** Le indicamos a *rosdep* que no debería intentar instalar paquete alguno de ROS en el directorio *src* desde el gestor de paquetes.
- **--rosdistro:** Comando requerido pues no tenemos establecido el entorno ROS aun.
- **--y:** Este último parámetro indica a *rosdep* que no queremos ser molestados por demasiados mensajes que pueda lanzar el gestor de paquetes.

Transcurridos unos minutos (y tras algunas peticiones de contraseña de superusuario) *rosdep* terminará de instalar las dependencias y se podrá continuar.

3.2.2.1.5 Construcción del *Catkin Workspace*

Una vez que ya están todos los paquetes y las dependencias, el siguiente paso es construir dichos paquetes. Se utilizó el comando *catkin_make_isolated* puesto que existen tanto paquetes *catkin* como paquetes compilados mediante *cmake* en la instalación base. Esto se utilizará para esta primera vez:

```
$ sudo ./src/catkin/bin/catkin_make_isolated --install --install-space /opt/ros/groovy
```

Una vez finalizada la instalación los paquetes deberían encontrarse correctamente instalados en */opt/ros/groovy*.

3.2.2.1.6 Construcción de los paquetes *rosbuild*

Ahora que se dispone de los paquetes *catkin* de ROS creados el siguiente paso es construir el resto de paquetes y *stacks* usando el sistema de construcción *rosbuild.*

3.2.2.1.6.1 Creación del rosbuild workspace

Al igual que hicimos con los paquetes *catkin,* es conveniente construir los paquetes *rosbuild* en un espacio de trabajo o *workspace*. Para ello lo primero que hacemos es crear un *workspace* mediante la herramienta *rows:*

```
$ mkdir ~/ros_ws

$ cd ~/ros_ws

$ rosws init . /opt/ros/groovy/
```

Mediante las líneas anteriores ya tendremos referenciado el *workspace* hacia la localización de instalación *catkin* que hicimos en el paso anterior. Con esto permitimos que el *workspace de ROS* monte el entorno de instalación *catkin* antes de usarlo.

3.2.2.1.6.2 Descarga de los Stacks ROS

Es necesario obtener los componentes *dry* de la variante de instalación elegida, en nuestro caso *mobile.* Por tanto:

```
$ sudo apt-get install libsdl-image1.2-dev

$ rosws merge
http://packages.ros.org/web/rosinstall/generate/dry/raw/groovy
/mobile

$ rosws set laser_geometry git://github.com/ros-
perception/laser_geometry.git --git --version=laser_pipeline-
1.4
```

Por último ya solo falta indicarle a la herramienta *rows* que busque los paquetes:

```
$ rosws update
```

3.2.2.1.6.3 Corrección de las dependencias rotas

Antes de continuar se han de eliminar las dependencias con *common_rosdeps* de cada uno de los archivos que se nos listen introduciendo lo siguiente en línea de comandos:

```
$ grep -r common_rosdeps *
```

3.2.2.1.6.4 Construcción del Stack ROS

Una vez completados los pasos anteriores lo último que nos falta por hacer es el *source* del *workspace* y como último paso construir el *stack.*

```
$ source ~/ros_ws/setup.bash
$ rosmake -a
```

Este método de instalación no es inmediato y nos encontraremos con fallos, repositorios incompletos o no disponibles incluso errores de compatibilidad de versiones de paquetes y aplicaciones. En cualquier caso tras varios intentos frustrados de hacer funcionar ROS al final conseguimos llegar a buen puerto.

3.2.2.2 Método 2: instalación de los paquetes binarios

Tras un tiempo investigando, aprendiendo las funcionalidades ROS y familiarizándome con el entorno surgió un problema con el sistema de ficheros y hubo que empezar desde cero todo el procedimiento.

Para entonces se había creado un repositorio experimental que contenía la mayor parte de la distribución de ROS Groovy, con la falta de algunos de sus paquetes, pero en cualquier caso suficiente y operativa y actualizándose constantemente a las últimas versiones de los paquetes gracias a la comunidad.

3.2.2.2.1 Instalación de los paquetes de ROS

Los pasos para instalar ROS Groovy de esta manera son mucho más sencillos e intuitivos que la compilación desde el código fuente con la única pega de una pérdida insignificante de paquetes con respecto al método anterior de instalación, lo cual no le resta funcionalidad en el proyecto que se ha acometido.

1. Añadimos el repositorio a nuestro *apt sources* desde un terminal de nuestro Raspbian:

```
$ sudo sh -c 'echo "deb http://64.91.227.57/repos/rospbian
wheezy main" > /etc/apt/sources.list.d/rospbian.list'
```

2. Añadimos la clave:

```
$ wget http://64.91.227.57/repos/rospbian.key -O - | sudo apt-
key add -
```

3. Recargamos los repositorios:

```
$ sudo apt-get update
```

4. Instalamos los paquetes ROS. En este caso se corresponden con *ros_comm.* Incluye dependencias con *roscpp, rospy* y todas las herramientas del núcleo de ROS:

```
$ sudo apt-get install ros-groovy-ros-comm
```

Transcurrido un tiempo ROS Groovy quedará íntegramente instalado en nuestro sistema Raspbian y estará listo para utilizarse. No debemos olvidarnos que para que funcionen todos los paquetes instalados y las herramientas de ROS hemos de hacer *source* a los archivos *setup.bash* generados en el directorio de instalación. Para ello:

```
$ source /opt/ros/groovy/setup.bash
```

3.2.2.2.2 Creación del espacio de trabajo de ROS (*catkin workspace)*

Una vez instalados los paquetes del núcleo de ROS el siguiente paso a realizar es la creación del espacio de trabajo (*catkin workspace)*. Para ello se ejecutan las siguientes instrucciones en un terminal:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

A continuación y pese a que el workspace está vacío (no hay paquetes en el directorio *src)* se construye mediante los siguientes comandos:

```
$ cd ~/catkin_ws/
$ catkin_make
```

Una vez finalizado el proceso aparecerán en el *workspace* dos nuevos directorios: *build y devel*. En el directorio *devel* aparecerán ciertos archivos *.sh.* Para el correcto funcionamiento es necesario hacer *source* del nuevo archivo *setup*:

```
$ source devel/setup.bash
```

Con esto queda listo el entorno de trabajo y se puede proceder a la creación de paquetes y mensajes necesarios para el desarrollo del proyecto.

3.3 Edición del archivo .bashrc

Para evitar olvidar hacer source de los archivos setup tanto de ROS como del *catkin workspace* se va a editar el archivo *.bashrc* para que realice esta tarea.

Para ello se ha de buscar dicho archivo en la carpeta del usuario en Raspbian. Para asegurarse de que se encuentra en dicha carpeta se ha de teclear lo siguiente en línea de comandos:

```
$ cd
```

A continuación se han de listar todos los archivos existentes:

```
$ ls -a
```

Como era de esperar se encuentra en esta carpeta el archivo deseado por lo que añadiremos al final del mismo las siguientes líneas con el editor de texto que se desee:

```
source /opt/ros/groovy/setup.bash
source ~/catkin_ws/devel/setup.bash
```

Realizando esta sencilla edición no habrá que preocuparse más por la tarea de hacer source la cual, además, es requisito indispensable para el correcto funcionamiento de ROS y de los paquetes instalados.

3.4 Integración de la placa Arduino con ROS

El último paso para tener el entorno completamente configurado consiste en la integración de la placa Arduino con nuestro Raspberry Pi, la cual ya corre Raspbian con ROS completamente operativo.

Se antoja necesario por tanto establecer la interfaz de comunicación entre el Arduino y el Raspberry Pi. La solución adoptada consiste en utilizar la interfaz USB del Raspberry Pi para conectar ambas plataformas. Pese a que el Arduino necesita una tensión de alimentación recomendada de entre 7 y 12 voltios se ha comprobado que los 5 voltios y 100mA que proporciona el Raspberry Pi a través del USB son más que suficientes para el desarrollo del proyecto y realizar correctamente la comunicación entre ambos.

3.4.1 Rosserial

Solventado el problema de la alimentación de la placa Arduino, el siguiente paso consiste en facilitar la comunicación entre Arduino y Raspberry Pi. Para ello la herramienta *Rosserial* es más que un software, es un protocolo general para enviar y recibir mensajes ROS a través de enlaces serie y, en el caso concreto de este proyecto, a través del puerto USB.

3.4.1.1 Instalación

Desde que apareció la versión Groovy de ROS el software *rosserial* se adaptó a *catkin* por lo que el flujo de trabajo es distinto con respecto a versiones anteriores de ROS. En lugar de ejecutar el generador de librerías sobre cada paquete que se desea usar, se ejecuta una sola vez generando las librerías de todos los paquetes al mismo tiempo.

Para la instalación se navega hasta el *catkin workspace* creado anteriormente y se ejecutan las siguientes instrucciones en líneas de comandos:

```
cd <ws>/src
  git clone https://github.com/ros-drivers/rosserial.git
  cd <ws>
  catkin_make
  catkin_make install
  source <ws>/install/setup.bash
```

<*es*> hace referencia al nombre que se le haya dado al *catkin workspace*. Estos comandos clonan el software *rosserial* desde si repositorio en Github y generan los mensajes *rosserial_msgs* necesarios para la comunicación.

3.4.1.2 Generación de Ros_lib

A continuación se ha de generar *ros_lib*. Dicha carpeta contendrá las librerías necesarias para Arduino y hacer posible la comunicación serie. Este directorio una vez generado se deberá trasladar al sistema donde se instale la herramienta Arduino IDE (en el caso de este proyecto, ordenador de sobremesa bajo Windows 8.1). La ruta donde se debe alojar es */documentos/Arduino/libraries*, al igual que cualquier librería que se vaya a necesitar posteriormente para la compilación en Arduino.

Para la generación de *ros_lib* es necesario navegar o crear un directorio vació en el sistema de ficheros y una vez en él escribir lo siguiente en línea de comandos:

```
cd <some_empty_directory>
  rosrun rosserial_arduino make_libraries.py .
```

3.4.1.3 Arduino IDE

Arduino IDE es el software que proporciona Arduino y que sirve como entorno de programación. En la **figura 27** se comprueba cómo se ha añadido correctamente *ros_lib* al conjunto de librerías de Arduino IDE.

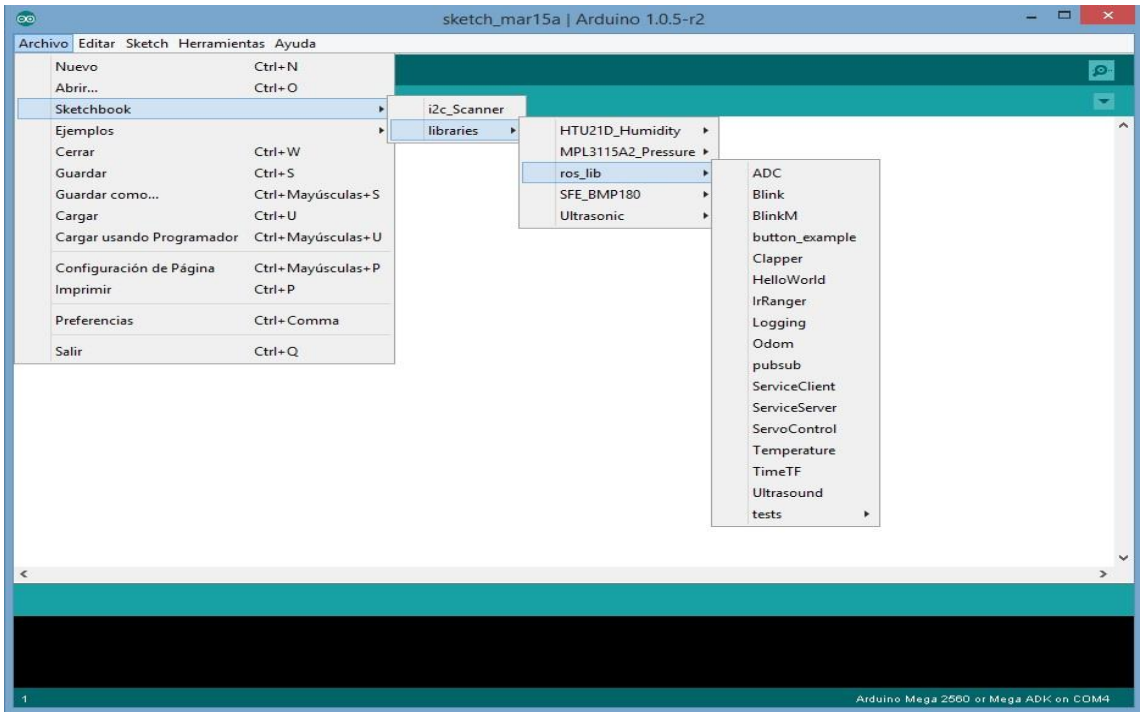


Figura 27 Arduino IDE

Para instalar el software Arduino IDE basta con acceder a la web de Arduino y en el apartado de descargas obtener la última versión estable del software (http://arduino.cc/en/Main/Software#.UySeW_l5N1A) y seguir las instrucciones de instalación.

3.4.1.4 Inicialización de la comunicación serie

Para inicializar una comunicación serie entre el Raspberry Pi y el Arduino es necesario que la interfaz USB del Raspberry Pi se comporte como un puerto serie. Para ello el software *rosserial* provee de las herramientas necesarias para ello.

Dicha herramienta se invoca desde línea de comandos y es la siguiente:

```
rosrun rosserial_python serial_node.py /dev/ttyUSB0
```

4. Robot objetivo

4.1 Descripción y características

La solución adoptada como plataforma destino para su control y programación se corresponde con el modelo *C-8090 Six-Leg Walking Type* fabficado por *CIC Creative.* Dicho modelo se trata de un robot móvil constituido por dos motores completamente independientes y por 6 patas o ruedas a elección del usuario ya que es configurable en este aspecto. Además cuenta con una superficie plana horizontal sobre la cual se colocan todos los elementos integrantes en este proyecto.

La **figura 28** y la **figura 29** recogen el aspecto que tendría dicho robot en función de la configuración elegida.

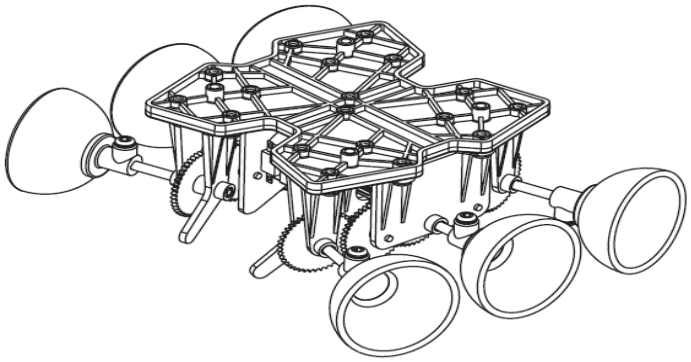


Figura 28 C-8090 Ruedas

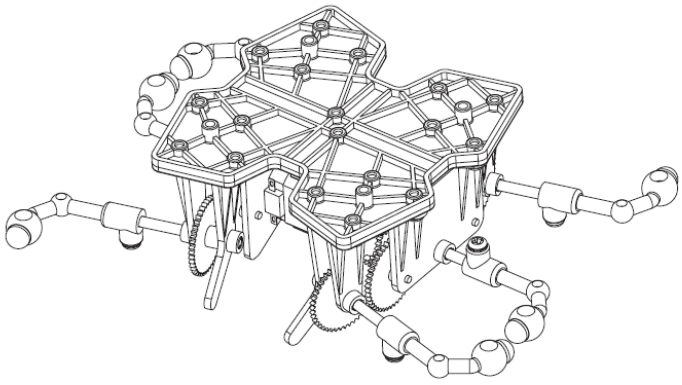


Figura 29 C-8090 Patas

Las características principales que presentan ambos motores que acompañan al robot *C-8090* se recogen en la **tabla 3**.

Características	
Alimentación	5V DC
Velocidad de salida	170 rpm

Tabla 3 Características de los motores

La configuración escogida para realizar a cabo este proyecto es aquella que cuenta con las 6 ruedas, empujadas 3 a 3 por cada motor a través de un sistema de engranajes. Al no disponer de ruedas directrices los giros se implementarán haciendo trabajar a los motores en sentido inverso a la vez.

La alimentación de los motores se realizará mediante un porta-baterías estándar y 4 pilas recargables de 1,5v asociadas a cada motor tal y como se presenta en la **figura 30**.

Figura 30 Alimentación y porta-pilas

4.2 L293D

Para abordar el control de los motores en este proyecto se ha decidido utilizar el circuito L293D del fabricante ST Microelectronics como el de la **figura 31**.

Figura 31 L293D

Está compuesto por 4 circuitos independientes que pueden manejar cargas de potencia media, en especial pequeños motores y cargas inductivas, con la capacidad de controlar corriente hasta 600mA en cada circuito y una tensión entre 4,5V a 36V.

Cualquiera de estos circuitos sirve para configurar un puente H por lo que formando los puentes H completos se puede llevar a cabo el manejo de los dos motores. Gracias a ello el manejo es bidireccional pudiendo establecer el sentido de la marcha así como configurar los giros alterando los sentidos de giro de los motores.

La **figura 32** muestra el pinout de este circuito el cual se encuentra detallado a continuación de la misma.

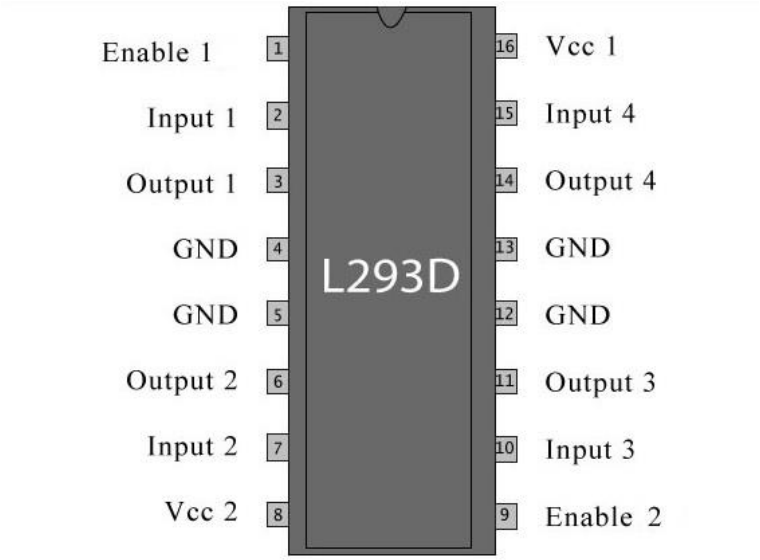


Figura 32 Pinout L293D

A continuación se detallan los pines para su comprensión y enfoque hacia este proyecto:

- 2 Pines *Enable,* los cuales habilitan el paso de potencia hacia los pines *Output.* Hay un por cada motor e irán conectados a uno de los pines PWM de Arduino.
- 4 Pines *Input,* los cuales irá conectados a las salidas digitales de Arduino y que en función del nivel lógico utilizado, HIGH (5V) o LOW (0V), definirán el sentido de giro de los motores.
- Pin *VCC 1*, el cual marca la referencia de nivel lógico alto (HIGH) por lo que irá conectado a uno de los pines referenciados como 5V en Arduino.
- Pin *VCC 2*, el cual provee de alimentación al circuito para distribuirla a los motores. Este pin irá conectado al porta-baterías lo que supondrá una alimentación de 9V en total.

4.3 Esquemas de conexionado

4.3.1 Conexionado al motor

La **figura 33** recoge el esquema de cómo se conectan los motores al L293D.

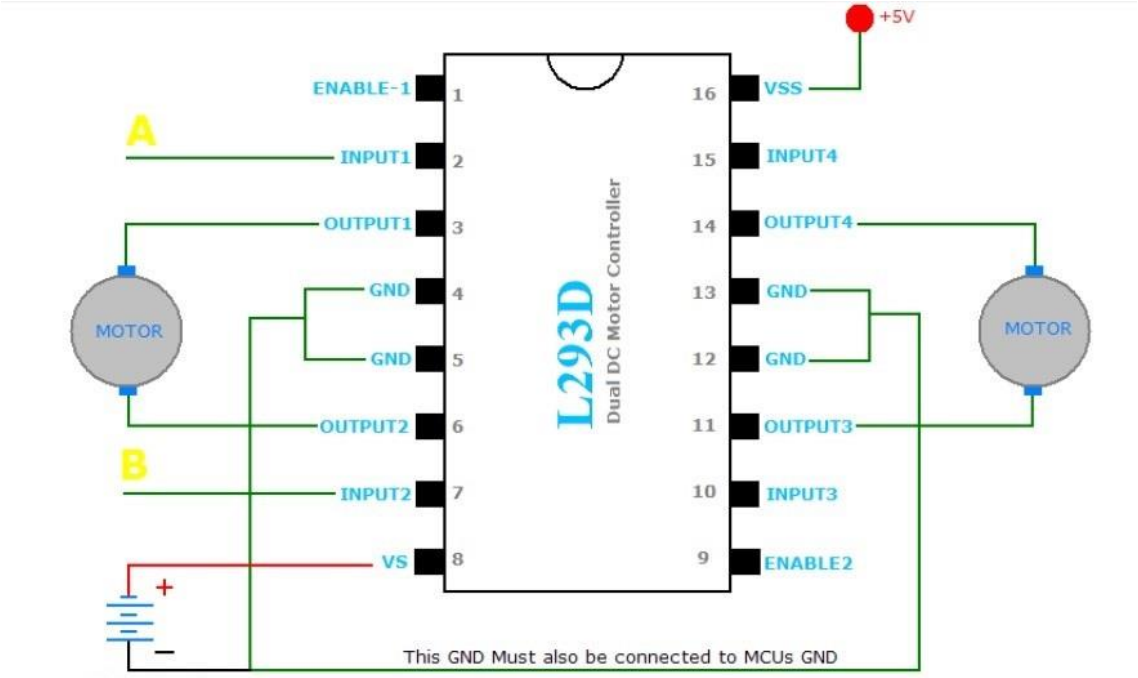


Figura 33Conexionado L293D a los motores

4.3.2 Conexionado Arduino

La **figura 34** recoge el esquema del conexionado del sistema completo al Arduino.

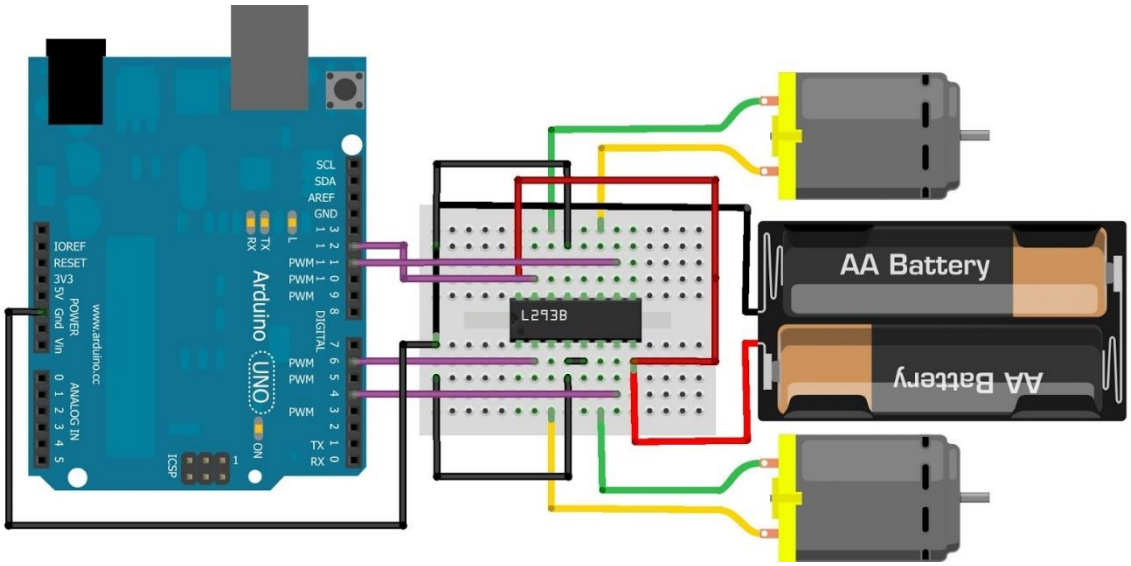


Figura 34 Conexionado Arduino-L293D

4.4 Aplicaciones diseñadas

4.4.1 Aplicación 1

El diseño realizado para esta primera aplicación consiste en un robot que tomará muestras meteorológicas del entorno (temperatura, humedad y presión) mediante unos sensores que utilizarán los pines SDA y SCL del Arduino cuya comunicación será gestionada por la librería *Arduino Wire* a través del bus de datos *i2c*. Contará a su vez con cámara a bordo para la captura de imágenes. Esta primera aplicación consiste en un modelo de robot dirigido hacia una coordenada GPS especificada por el usuario. El posicionamiento GPS vendrá facilitado mediante la integración con ROS de un Smartphone Android el cual publicará los valores de sus sensores on-board que serán recogidos por ROS.

4.4.2 Aplicación 2

En este segundo caso el funcionamiento consistirá en el movimiento libre y autónomo del robot el cual será capaz de evitar los obstáculos que se le vayan presentando mientras va realizando las medidas meteorológicas en el entorno tal y como se describe en la aplicación 1. De igual manera contará con cámara integrada que permitirá al usuario obtener imágenes en todo momento. Dicho sistema será capaz de almacenar la coordenada GPS en la cual se encuentra con un obstáculo para su conocimiento.

Ambos utilizaran sensor de luminosidad proporcionado por el móvil para determinar, en función de la luminosidad existente en el entorno, el modo de funcionamiento.

4.4.3 Comunicación de los sensores: Arduino Wire y bus I2C

4.4.3.1 Arduino Wire

4.4.3.1.1 Descripción

Esta librería permite la comunicación de la placa Arduino con dispositivos I2C/TWI. Está escrita en lenguaje de programación C++ por lo que todas las llamadas que se realizan a funciones que encuentran en esta librería están definidas como métodos en una clase C++.

Al enviar o recibir datos a través del bus *I2C*, la librería *Wire* bloque las demás operaciones de entrada y salida por lo que la comunicación en el bus queda reservada hasta que no termine la operación actual de comunicación, pudiendo así pasar a la siguiente.

4.4.3.1.2 Arduino MEGA ADK

En la placa Arduino MEGA ADK los pines I2C/TWI se encuentran localizados tal y como se indica en la **tabla 4**.

Placa Arduino	**Pines I2C/TWI**
Mega 2560 ADK	20 (SDA) 21(SCL)

Tabla 4 Pines SDA/SCL

En la **figura 35** se resalta la situación de estos pines.

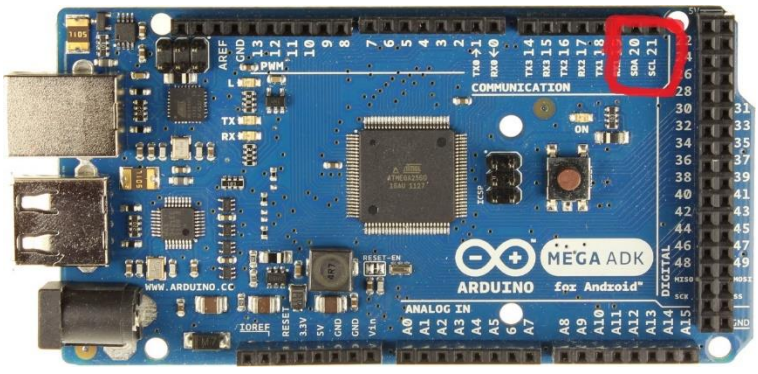


Figura 35 Situación Pines SDA/SCL en Arduino Mega ADK

4.4.3.1.3 Funciones

- begin()

    - Wire.begin() / Wire.begin(address)
        - Inicializa la librería *Wire* y permite unirse al bus *I2C* como maestro o esclavo.
        - Como parámetro opcional se le pasa una dirección de 7 bit (esclavo). Si no se especifica, se une al bus como master.
- requestFrom()
    - Wire.requestFrom()
        - Utilizado por el maestro para solicitar bytes desde un dispositivo esclavo.
        - Se le pasan como parámetros la dirección del esclavo y la cantidad de bytes a solicitar.
- beginTransmission()
    - Wire.beginTransmission(address)
        - Inicializa la transmisión de datos con el dispositivo *I2C* esclavo con la dirección indicada.
        - Necesita de la dirección de 7 bits del dispositivo esclavo para transmitir.
- endTransmission()
    - Wire.endTransmission()
        - Finaliza una transmisión a un dispositivo esclavo iniciada por beginTransmission() y envía los bytes que se pusieron en cola mediante write()
        - Se pasa como parámetro el booleano *stop* el cual en función de si es cierto o no terminará la transmisión o la mantendrá activa.
- write()
    - write(value)/write(string)/write(data,lenght)
        - Escribe los datos de un dispositivo esclavo en respuesta a petición de un maestro.
        - Se le pasarán como parámetros un valor para enviar como un byte, una cadena para enviar como series de bytes o bien un array (colección) de datos para enviarlos como bytes estableciendo el número de estos mediante lenght.
- available()
    - Wire.available()
        - Devuelve el número de bytes disponibles para la recuperación con un read().
        - Llamado en un dispositivo maestro tras una llamada a requestFrom() o bien desde un esclavo tras la llamada onReceive().
- read()
    - Wire.read()
        - Lee un byte que se transmite de un dispositivo esclavo a un maestro tras una llamada a requestFrom() o se transmite de un maestro a un esclavo.
- onReceive()
    - Wire.onReceive(handler)

        - Registra una función que se llamará cuando un dispositivo recibe una transmisión de un maestro.
        - Handler: función que se llamará cuando un esclavo recibe los datos.
- onRequest()
    - Wire.onRequest(handler)
        - Registra una función que será llamada cuando un maestro requiera datos de un dispositivo esclavo.
        - Handler: función a ser llamada. No toma parámetros ni devuelve nada.

4.4.3.2 Bus I2C

4.4.3.2.1 Introducción

I2C es un bus de comunicaciones en serie. Su nombre viene de Inter-Integrated Circuit (Inter-Circuitos Integrados). La versión 1.0 data del año 1992 y la versión 2.1 del año 2000, su diseñador es Philips. La velocidad es de 100 kbit/s en el modo estándar, aunque también permite velocidades de 3.4 Mbit/s.

Los ingenieros de "Philips" vieron la necesidad de la simplificación y normalización de las líneas de datos que viajan entre los diversos circuitos integrados en sus productos. Su solución fue el bus I2C. Esto redujo un gran número de cables a sólo dos (SDA = datos, y SCL = reloj).

4.4.3.2.2 Descripción

Se trata de, un bus bidireccional que utiliza dos líneas, una de datos serie (SDA) y otra de reloj serie (SCL), que requiere resistencias de polarización a positivo (RPA). SCL es la línea de reloj, se utiliza para sincronizar todos los datos SDA de las transferencias durante I²C bus. SDA es la línea de datos.

Las líneas SCL y SDA están conectadas a todos los dispositivos en el I²C bus. Ambas líneas SCL y SDA son del tipo drenador abierto asociados a un transistor de efecto de campo (o FET), es decir, un estado similar al de colector abierto. Esto significa que el chip puede manejar su salida a BAJO, pero no puede manejar a ALTO. Para que la línea pueda ir a ALTO, se deben proporcionar resistencias de polarización a 5V. Necesita de una resistencia de la línea SCL a la línea de 5V y otra de la línea SDA a la línea de 5V.

Sólo necesita un conjunto de resistencias de RPA (pull-up) para todo el I²C bus. Cuidado, no son necesarias para cada dispositivo. La alimentación del sistema, debe tener una masa común, también puede haber una alimentación compartida que, se distribuye entre los distintos dispositivos.

Los dispositivos en el I²C bus son maestros o esclavos. El maestro, es siempre el dispositivo que maneja la línea de reloj SCL. Los esclavos, son los dispositivos que responden al maestro. Un esclavo no puede iniciar una transferencia a través del I²C bus, sólo un maestro puede hacer esa función. Generalmente son, varios esclavos en el I²C bus, sin embargo, normalmente hay un solo maestro. Es posible tener varios maestros, pero es inusual y no se comentará aquí. Por lo tanto, los esclavos, nunca inician una transferencia. Tanto el maestro, como el esclavo puede transferir datos a través del I²C bus, pero la transferencia siempre es controlada por el maestro. En la **figura 36** se observa un esquema de configuración maestro-esclavo donde se representa lo descrito anteriormente.

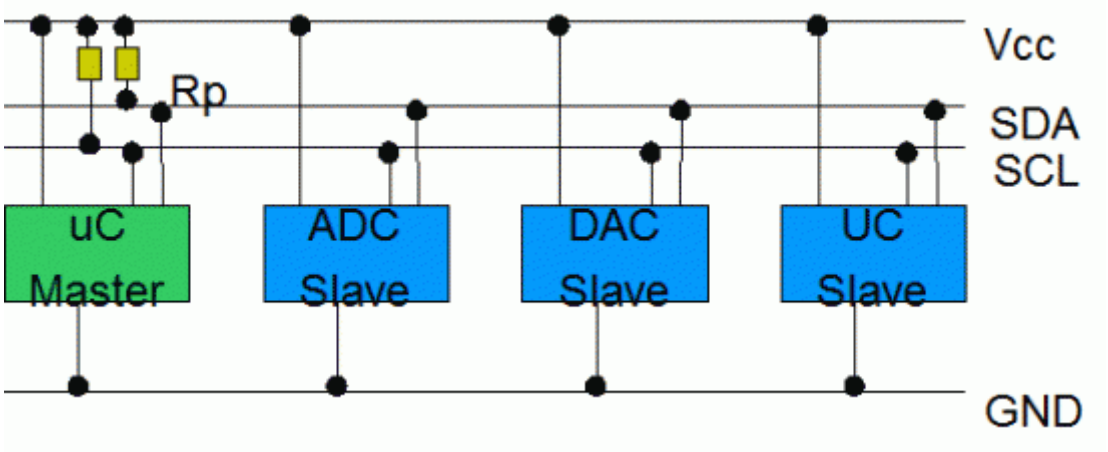


Figura 6 Bus i2c

Todas las direcciones I²C bus son de 7 bits o 10 bits. Esto significa que, se pueden tener hasta 128 dispositivos en el bus I²C, ya que un número de 7bit puede estar entre 0 y 127. El I²C tiene un diseño de espacio de referencia de 7 bits de direcciones, reservado con 16 direcciones, de modo que finalmente, pueden comunicarse en el mismo bus un máximo de 112 nodos. El número máximo de nodos está limitado por el espacio de direcciones y también por la capacidad total de los buses de 400 pF, lo que restringe la práctica de comunicación, a distancias de unos pocos metros.

4.4.3.2.3 Protocolo

La base del protocolo I²C-bus es que, el bus tiene dos funciones para los nodos: maestro y esclavo. El resultado de este sistema produce un anillo compuesto por dos vías (hilos) para el desarrollo de la función de comunicarse entre sí los dispositivos interconectados a las mencionadas vías, esto permite comunicarse entre ellos mismos con un protocolo que, consiste en que, en cada momento hay un maestro y el resto son esclavos. Un símil sería que, -mientras uno habla, el resto escuchan-, o también, uno escribe y el resto leen.

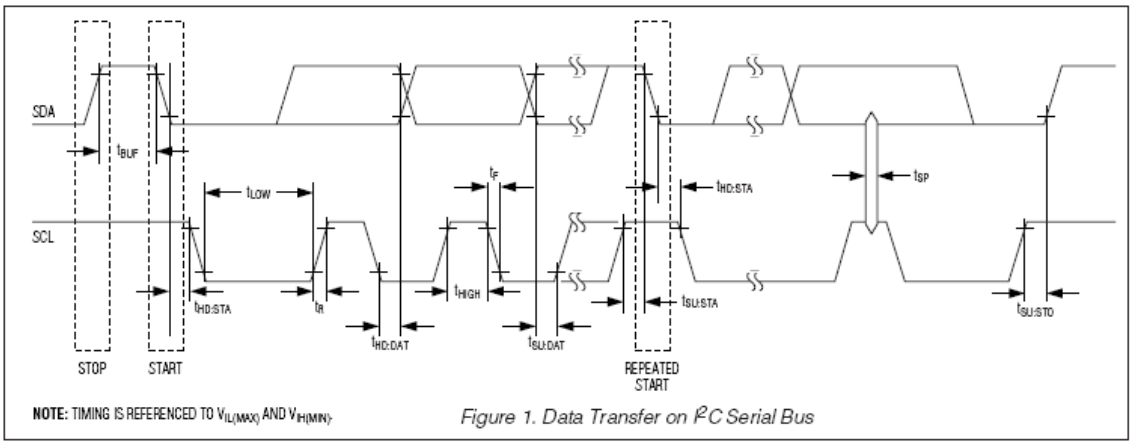


Figura 36 Protocolo de transferencia i2c

En la **figura 37** se puede apreciar que, la condición START está definida, cuando la señal de reloj SCL permanece estable ALTO (H) además, el nivel de la señal de "no reconocimiento", debe ser también ALTA (H), si en ese preciso instante se produce un descenso (flanco de bajada) en la señal de datos, automáticamente se produce la condición de START (inicio) de la transmisión y el dispositivo que la produjo, se convierte en MAESTRO, dando comienzo la transmisión. El pulso de reconocer o reconocimiento, conocido como ACK (del inglés Acknowledge), se logra colocando la línea de datos a un nivel lógico bajo, durante el transcurso del noveno pulso de reloj.

4.4.3.2.4 I2C en Arduino

Actualmente existen muchas aplicaciones, para aplicar con la placa Arduino; como un reloj en tiempo real, potenciómetros digitales, sensores de temperatura, brújulas digitales, chips de memoria, servos, circuitos de radio FM, E/S expansores, controladores de LCD, amplificadores, etc. Y además usted puede tener más de una idea para aplicar este bus, en cualquier momento, como se ha dicho, el número máximo de dispositivos I²C utiliza en un momento dado es de 112.

Si sólo está utilizando un dispositivo I²C, las resistencias RPA, no son necesarias ya que el micro controlador ATMega328 en nuestro Arduino las ha incorporado Sin embargo, si está empleando una serie de dispositivos utilice dos resistencias de 10k ohmios. Como todo, algunas pruebas en un circuito protoboard o prototipo determinarán su necesidad. La longitud máxima de un bus I²C es de alrededor de un metro y es una función de la capacidad del bus.

Cada dispositivo se puede conectar al bus en cualquier orden y, como ya se ha mencionado, los dispositivos pueden ser maestros o esclavos. En nuestra aplicación el IDE Arduino, es el maestro y los dispositivos que "colguemos" en el bus I²C son los esclavos. Podemos escribir datos en un dispositivo o leer datos de un dispositivo.

Al igual que en la mayoría de dispositivos, haremos uso de una librería para Arduino, en este caso <wire.h>. Las funciones que nos permitirán realizar la

comunicación entre maestro y posibles esclavos son las que se presentaron en el apartado anterior.

4.4.3.3 Ejemplo de funcionamiento: Sensor de temperatura

Para comprobar el funcionamiento de lo explicado anteriormente sobre I2C y la librería *Wire* se realiza la medición de temperatura ambiente mediante el sensor TMP102.

4.4.3.3.1 TMP102

El Tmp102 es un pequeño sensor de temperatura digital como el de la **figura 38** que usa como protocolo de comunicación el bus I2C (TWI). Es un sensor muy útil ya que requiere una corriente muy bajara para funcionar. La comunicación con el TMP se logra a través de una interfaz de dos hilos (SDA y SCL).

Figura 37 Sensor de temperatura TMP102

No existe un regulador de tensión a bordo por lo que la tensión suministrada debe estar comprendida entre 1,4 y 3,6VDC. La **tabla 5** recoge las características principales de este sensor.

Características	
Resolución	
Precisión	
Baja corriente estática	• 10uA Activo (max) • 1uA Reposo (max)
Rango de tensión	1.4 – 3.6 VDC
Interfaz de comunicación	2 hilos (SDA,SCL)

Tabla 5 Características TMP102

4.4.3.3.2 Conexionado

El conexionado con el Arduino se realiza conectando el TMP02 al Arduino tal y como se presenta en la **figura 39**.

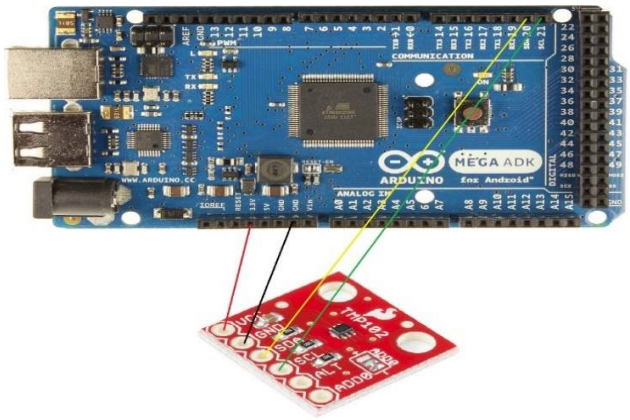


Figura 38 Conexionado TMP102

4.4.3.3.3 Prueba

El primer paso a realizar es, una vez conectada la placa Arduino junto con el sensor TMP102 al Raspberry Pi, llamar al servicio *Roscore* en un terminal el cual permitirá la comunicación entre los nodos.

A continuación se debe inicializar la comunicación serie entre la placa Arduino y ROS en el Raspberry Pi. Para ello se debe escribir lo siguiente en un nuevo terminal:

```
rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Si todo funciona correctamente se obtiene la siguiente salida en dicho terminal, el cual muestra que la conexión ha sido exitosa, la velocidad de la misma y el tipo de mensaje que se está publicando bajo el topic *temperatura* en este caso.

El último paso consiste, en un nuevo terminal, en comprobar que evidentemente se está publicando el topic *temperature*, invocarlo y comprobar su correcto funcionamiento.

Primero se comprueba que se está publicando el *topic:*

```
rostopic list
```

A continuación se van a requerir los datos de dicho topic para que los muestre:

```
echo rostopic /temperature
```

Una vez comprobado que funciona correctamente se da por finalizado cada proceso en cada terminal mediante la combinación de teclas *Ctrl C*.

5 Hardware implicado

En las líneas siguientes se detallan los diferentes componentes que al final han dado forma y funcionalidad al proyecto además de los dos pilares (Raspberry Pi y Arduino).

5.1 Sensor de ultrasonido HC-SR04

La forma operativa de un sensor ultrasónico se basa en un principio similar al sónar que evalúa la distancia de un objeto mediante la interpretación de los ecos de las ondas sonoras ultrasónicas.

Figura 39 Sensor de ultrasonido HC-SR04

5.1.1 Descripción

El módulo *HC-SR04* mide la distancia a objetos situados entre 0 y 400 cm con una precisión de 3cm. Su tamaño compacto, gama superior y facilidad de uso hacen que sea un sensor práctico para la medición de distancia y cartografía.

El módulo puede conectarse fácilmente a los micro controladores donde el desencadenamiento y la medición de la distancia puede realizarse a través de dos pines.

El sensor transmite una onda ultrasónica y produce un impulso de salida de que corresponde al tiempo requerido para que las ráfagas de eco vuelvan al sensor. Mediante la medición de la anchura del pulso de eco, la distancia al objetivo puede ser fácilmente calculada.

5.1.2 Características

El sensor dispone de 4 pines:

- Vcc (5V): Conectable a la interfaz de 5voltios de Arduino. Este pin proporciona la tensión necesaria para la alimentación del circuito receptor y adaptador de la señal de eco.
- Pin GND: Conectable al pin de tierra de Arduino.

- Pin trigger: Conectable a cualquier interfaz digital de Arduino. Este pin será el que marque el pulso de trigger a emitir mediante ultrasonido por el sensor cuya señal será reflejada por los obstáculos que este se encuentre en su camino.
- Pin Echo: En este pin digital Arduino medirá la duración del pulso recibido y su retardo respecto al emitido.

La **tabla 6** recoge las características físicas y las especificaciones eléctricas del módulo *HC-SR04*.

Parámetro	**Características**
VCC	4.8V to 5V (typical) 5.5V (absolute maximum)
Corriente estática	<2 mA
Corriente de funcionamiento	15 mA
Temperatura	0-60º
Pulso de entrada Trigger	Pulso TTL 10uS (5V, ± 2%)
Señal de salida Echo	TTL (0V/5V, ± 2%)
Frecuencia de trabajo	40 KHz
Distancia de detección	2-400 cm
Ángulo eficaz	< 15º
Ángulo de medición	30º

Tabla 6 Características HC-SR04

La **figura 41** muestra las especificaciones físicas así como el rango de alcance y ángulo óptimo de funcionamiento.

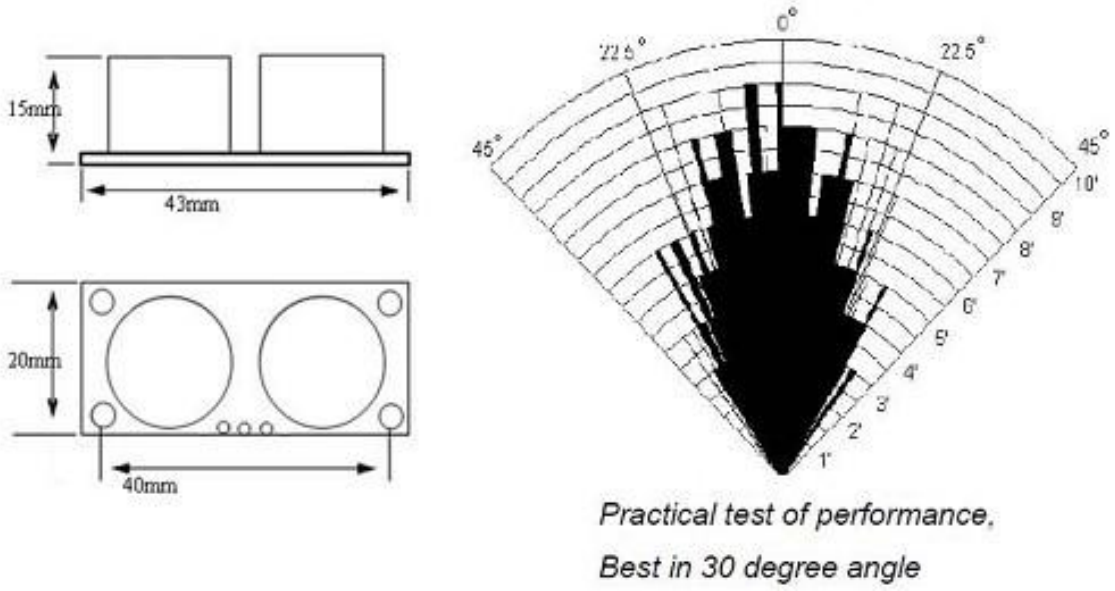


Figura 40 Rango de funcionamiento HC-SR04

5.1.3 Funcionamiento

En la **figura 42** se muestra el diagrama de temporización. Si suministramos un impulso de 10uS a la entrada de disparo Trigger el sensor transmite un pulso de ultrasonido a 40 KHz de 8 ciclos y produce un pulso de salida con una anchura con respecto al anterior que se corresponde con el tiempo requerido por el pulso Trigger en retornar al sensor. Midiendo así la anchura y el retardo de este y teniendo en cuenta la anchura anterior así como la velocidad del sonido en el aire, se puede calcular fácilmente la distancia a un objeto.

El pulso de Echo es el que ayuda a establecer la distancia a un objeto. (150uS – 25ms) (38ms si no hay obstáculo presente).

Para obtener en Arduino la distancia mediante la medida de la longitud en microsegundos en el pin digital asignado se usa la siguiente fórmula:

```
Anchopulso(uS)/58=  distancia
```

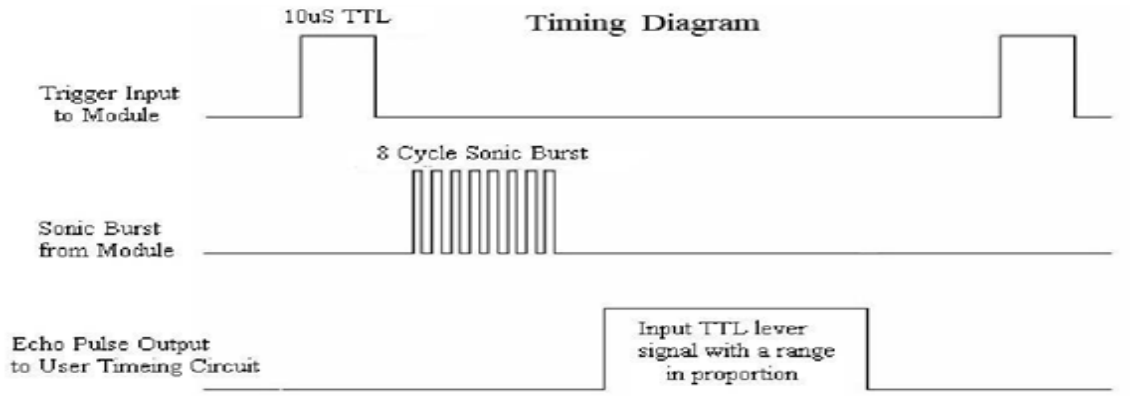


Figura 41 Funcionamiento HC-SR04

5.2 Weather Shield

La placa Weahter Shield, **figura 43**, ha sido el componente elegido para encargarse de realizar las medidas meteorológicas diseñadas en este proyecto. Aparte de la funcionalidad utilizada en este proyecto ofrece interesantes líneas de ampliación y mejora como la localización GPS mediante un módulo acoplable y la recopilación de datos de viento y lluvia mediante una ampliación.

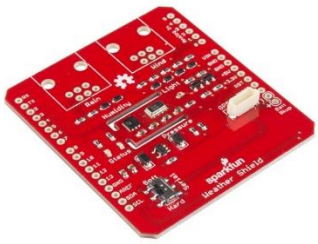


Figura 42 Weather Shield

5.2.1 Descripción

El *Weather Shield* es un escudo para Arduino fácil de usar que permite tener acceso a la información relativa a la presión barométrica, humedad relativa, luminosidad y temperatura ambiente.

Existen igualmente conexiones en este escudo para sensores opcionales tales como la velocidad del viento, dirección, pluviómetro y GPS para la ubicación y precisión horaria.

El *Weather Shield* utiliza el sensor *HTU21D* para las lecturas de humedad relativa, el *MPL3115A2* para la presión barométrica y los sensores de luz ALS-PT19. Todos ellos dependen, para su correcto funcionamiento de las librerías de Arduino MPL3115A2 y HTU21D.

Cada escudo viene con dos espacios disponibles para sendos conectores RJ11 (opcionales para los sensores de lluvia y viento) y un conector GPS de 6 pines (para la conexión opcional del módulo GPS *GP635T*.

5.2.2 Características

El *Weather Shield* puede operar desde 3,3V hasta 16V y lleva a bordo reguladores de voltaje y transductores de señal.

Está conformado por los siguientes elementos:

- Sensor de humedad/temperatura HTU21D
- Sensor de presión barométrica MPL3115A2
- Sensor lumínico ALS-PT19

5.2.2.1 Sensor de humedad/temperatura HTU21D

Figura 43 Sensor de humedad HTU21D

Las características del sensor HTU21D se muestran en la **tabla 7**.

Comunicación	Bus I2C
Precisión en la medida de humedad	±2%
Precisión en la medida de temperatura	±0.3ºC
Rango de funcionamiento (humedad)	0-100% (Evitar contacto agua)
Tensión de funcionamiento	3,3 V

Tabla 7 Características del sensor HTU21D

5.2.2.2 Sensor de presión barométrica MPL3115A2

Figura 44 Sensor de presión MPL3115A2

Las características de este sensor se recogen en la **tabla 8**.

Tensión de alimentación	1.95 – 3.6 V
Alimentación digital	1.6 – 3.6 V
Compensación	Completa digital
Medida directa de presión compensada	20 bit (Pascales)
Altitud	20 bit (metros)
Temperatura	12 bit (Celsius)
Resolución	Hasta 30 cm
Adquisición autónoma de datos	
Eventos programables	
FIFO de 32 muestras	
Posibilidad de almacenamiento de datos hasta 12 días (FIFO)	
Tasa de adquisición de datos	1 seg – 9 horas
Comunicación	Interfaz I2C

Tabla 8 Características del sensor MPL3115A2

5.2.2.3 Sensor lumínico ALS-PT19

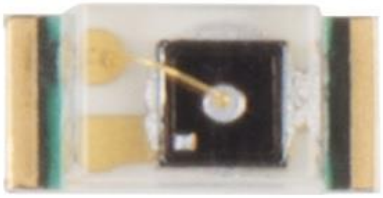


Figura 45 Sensor lumínico ALS-PT19

Se trata de un sensor de luz ambiental SMD capaz de tomar un rango de suministro de tensión de 2.5V a 5.5V el cual también funciona como un transistor NPN. La **tabla 9** recoge las principales características de dicho sensor.

Tensión de alimentación	2.5 – 5.5 V
Temperatura de funcionamiento	-40ºC – 85ºC
Salida analógica	Luz – Corriente
Tamaño	1.7mm(L)*0.8mm(w)*0.6mm(H)

Tabla 9 Características sensor ALS-PT19

5.3 Raspberry Pi Camera

5.3.1 Descripción

La cámara Raspberry Pi, **figura 47**, contiene un sensor de 5 Megapíxeles Omnivision 5647 (2592x1944 píxeles). Se conecta mediante cable de cinta de 15 pines al conector de CSI (MIPI Camera Serial Interface) en el Raspberry. El video y la calidad de imagen que ofrece este dispositivo son mayor que una cámara web de precio similar. Ofrece a su vez a posibilidad de obtener video HD 1080p a 30 fps.

Figura 46 Raspberry Pi Camera

5.3.2 Características

La **tabla 10** recoge las principales características de la Raspberry Pi Camera.

Tipo de sensor	Omnivision OV5647 QSXGA (5 MPx)
Tamaño del sensor	3.67 x 2.74 mm
Resolución	2592 x 1944 px
Tamaño de pixel	1.4 x 1.4 um
Distancia focal	f=3.6mm , f/2.9
Ángulo de visión	54 x 41 grados
Campo de visión	2.0 x 1.33 m – 2m
Full-frame SLR	35 mm
Lente fija	1 m hasta el infinito
Video	1080p a 30 fps con H.264 (AVC)
Fps	Hasta 90 fps en VGA
Tamaño de la placa	25 x 24 mm

Tabla 10 Características Raspberry Pi Camera

Debido a que la distancia focal de la lente es aproximadamente igual que el ancho del sensor, es fácil recordar el campo de visión: a *x* metros de distancia, se pueden ver alrededor de *x* metros en horizontal, asumiendo el modo de imágenes fijas 4:3. El campo horizontal de visión en 1080p se corresponde con el 75% de este último (75%H x 55% V sensor 1:1 a 1920x1080).

5.4 Smartphone Android

5.4.1 Descripción

El siguiente elemento necesario para la consecución y puesta en marcha de este proyecto consiste en un Smartphone con Android. Utilizaremos dicho Smartphone para obtener las lecturas de sus sensores así como el posicionamiento GPS todo ello a través del software *ROS Sensor Drivers*, el cual mediante la publicación de la información de los mismos permitirá subscribirnos desde ROS y utilizarla con fin de cumplir el diseño establecido. Se entrará en detalle en el capítulo siguiente, en el cual se detalla todo el proceso de preparación del entorno y trabajo con los sensores.

El Smartphone utilizado en este caso se trata del LG Nexus 4, **figura 48**. El motivo de esta elección es simple: es el Smartphone disponible a la hora de la realización del proyecto.

Figura 47 LG Google Nexus 4

5.4.2 Características

La **tabla 11** recoge las principales características del Smartphone, donde se han resaltado las características que, por su funcionalidad y cualidades, son importantes en este proyecto.

Pantalla	• 4,7” • Resolución 1280x720 píxeles (320dpi) • WXGA IPS • Gorilla Glass 2
Tamaño	133.9 x 68.7 x 9.1 mm
Peso	139 g
Cámaras	• 8 MP (principal) • 1.3 MP (delantera)
Memoria	• 16 GB almacenamiento • 2 GB Ram
CPU	Qualcomm Snapdragon S4 Pro 1.5 GHz
Sensores	• Micrófono • Acelerómetro • Brújula • Luz ambiental • Barómetro • Giróscopo • GPS
Red	• GSM/UMTS/HSPA+ libre • GSM/EDGE/GPRS (850, 900, 1800, 1900 MHz) • 3G (850, 900, 1700, 1900, 2100 MHz) • HSPA+ 42
Conexiones inalambricas	• Compatible con carga sin cables • Wi-Fi (802.11 a/b/g/n) • NFC (Android Beam)

	• Bluetooth
Conectividad	• Micro USB • SlimPort HDMI • Conector para auriculares de 3,5 mm
Batería	2100 mAh

Tabla 11 Características LG Google Nexus 4

6 Aplicación 1

6.1 Solución adoptada

Para el desarrollo de esta primera aplicación diseñada se ha optado por los siguientes componentes:

- Arduino Mega ADK: el cual se encargará de la gestión y control del motor en función de los datos recibidos desde el Raspberry Pi y de gestionar y publicar los datos meteorológicos a los cuales estará subscrito ROS en el Raspberry.
- Raspberry Pi: corriendo el sistema operativo Raspbian con ROS convenientemente instalado se encargará de recibir los datos de meteorología y actuará como nodo centrar encargado de decidir el movimiento del robot en función de las coordenadas GPS.
- Weather Shield: este elemento se encargará de realizar las mediciones de temperatura, humedad y presión.
- Smartphone Nexus 4: su integración con ROS hará posible la utilización de los sensores de luminosidad, giróscopo y GPS para dotar al robot de dichas funcionalidades.
- Raspberry Pi Camera: la cual permitirá la toma de imágenes del medio.

6.2 Preparación del entorno

Una vez definidos los elementos que van a conformar este primer experimento, prosigue la tarea de su integración, programación y configuración en ROS.

6.2.1 Integración Android: ROS Android Sensors Driver

Para la integración de los sensores de un Smartphone Android con el entorno ROS existe un driver instalable (*apk)* que permite la publicación de los datos de dichos sensores para su utilización en ROS.

La aplicación que permite lo anteriormente comentado se llama ROS *Android Sensors Driver,* **figura 49,** creada por Chad Rockey, se encuentra disponible en Google Play Store. Una vez descargada e instalada se ha de configurar el escenario en ROS para su correcto funcionamiento.

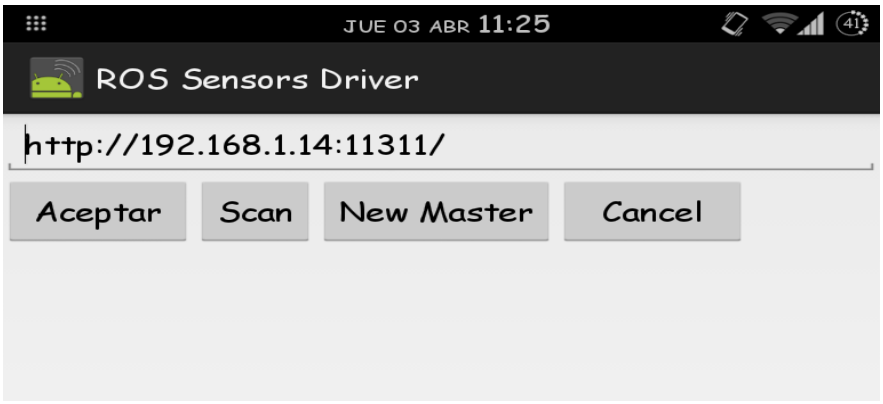


Figura 48 ROS Sensors Driver

Se debe por tanto exportar la dirección IP obtenida por la interfaz WLAN del Raspberry Pi. Se puede obtener fácilmente escribiendo en un terminal:

```
$ ifconfig
```

Una vez obtenida la dirección IP hemos de ejecutar lo siguiente en el mismo terminal (sustituyendo la IP correspondiente):

```
$ exportROS_MASTER_URI=http://192.168.1.14:11311/
```

Ya está configurado el escenario en ROS para que este sea capaz de subscribirse y recibir los datos de los sensores del Smartphone. Para ello tan simple como ejecutar la aplicación y esta comenzará a publicarlos.

6.2.1.2 Prueba de funcionamiento

Para comprobar que se ha configurado correctamente se inicia la aplicación en el Smartphone el cual, al igual que el Raspberry Pi, debe estar en la misma red (SSID).

Tras iniciar la aplicación se debe inicializar el servicio *roscore* en el Raspberry Pi gracias al cual se tendrá acceso a las herramientas de ROS y se podrá comprobar la correcta publicación de topics.

Una vez inicializado *roscore* se hace uso de la herramienta *rostopic list* la cual invocamos desde una nueva pestaña del terminal:

```
$ rostopic list
```

El resultado se presenta en la **figura 50**.

Figura 49 Rostopic list Android

Como era de esperar se están publicando los datos de los sensores del Smartphone Android en sus correspondientes topics. Entre los que se destaca */android/fix*, */android/imu* y */android/illuminance* los cuales contienen los datos relativos al GPS, giróscopo y acelerómetro y sensor de luminosidad que se integran en este proyecto.

Se puede a su vez examinar la distribución de los nodos donde se están publicando los anteriores topics. Para ello basta con invocar desde terminal:

```
$ rosnode list
```

Para asegurar la correcta conexión de cada nodo bastaría con hacer un ping y esperar respuesta.

```
$ rosnode ping /android_sensors_driver_”nombre”
```

Con esto quedaría configurada la publicación de mensajes de los sensores en ROS.

6.2.2 Creación de paquete ROS

El primer paso que se debe dar es la creación de un paquete ROS dentro del *catkin workspace* donde se definirán todos los nuevos servicios, mensajes y programas que darán forma al proyecto.

Para ello se navega hasta el siguiente directorio dentro del *catkin_workspace (*el cual fue creado en el proceso de instalación y configuración):

```
$ cd ~/catkin_ws/src
```

Una vez ahí se ha de ejecutar el siguiente comando:

```
$ catkin_create_pkg alvaro std_msgs rospy roscpp
```

- alvaro: nombre del paquete elegido
- std_msgs: dependencia con los mensajes estándar de ROS los cuales incluyen los tipos de datos primitivos y otras construcciones de mensajes básicos, como arrays.
- rospy: biblioteca de Python para ROS. Favorece la velocidad de ejecución sobre el rendimiento en tiempo de ejecución por lo que los algoritmos pueden ser prototipos y probados dentro de ROS rápidamente.
- roscpp: es una aplicación C++ de ROS. Proporciona una biblioteca de cliente que permite a los programadores de C++ interactuar rápidamente con los topics de ROS, servicios y parámetros. Es la más utilizada y está diseñada para ser la biblioteca de alto rendimiento para ROS.

Tras la ejecución del comando anterior se creará el directorio *alvaro* el cual contendrá dos archivos: *package.xml* y *CMakeLists.txt*.

- package.xml: define las propiedades de cualquier paquete como el nombre, versión, autor, mantenedor y dependencias.
- CMakeLists.txt: es un archivo que sirve como entrada a la herramienta del sistema CMake utilizada para la construcción de paquetes software.

6.2.2.1 Construcción del paquete ROS

Una vez se ha creado el paquete *alvaro* en el paso anterior ahora es necesario construirlo. Para ello hemos de situarnos en el directorio padre del *catkin workspace*, esto es:

```
$ cd ~/catkin_ws/
```

Una vez ahí se ha de ejecutar lo siguiente en línea de comandos:

```
$ catkin_make
```

Tras la compilación y numerosos mensajes de salida provenientes de la herramienta *cmake* y *make*, el proceso concluye exitosamente.

6.2.3 Publicación de datos meteorológicos: creación de un *custom message*

Desde el Weather Shield vamos a publicar 3 mensajes diferentes: temperatura, presión y humedad.

La librería de ROS en la que se incluyen los tipos de mensaje existentes *std_msgs* no contiene ningún tipo de mensaje en la que englobar tres diferentes, pues solo se permite la publicación de un tipo de mensaje a la vez. Por tanto se antoja necesario la creación de un tipo de mensaje customizado (custom message) el cual permitirá dicha tarea.

Por tanto el primer paso consistirá en la creación del nuevo tipo de mensaje. Para ello se ha de crear, mediante el editor de textos deseado por el usuario (en este caso se ha usado nano), el mensaje, al cual hemos llamado *meteo.msg*. Se ha de navegar hasta la siguiente ruta en el *catkin workspace* dentro del paquete creado para este proyecto:

```
$ cd ~/catkin_ws/src/alvaro
```

En esa ruta se debe crear una carpeta que se llame *msg*:

```
~/catkin_ws/src/alvaro$ mkdir msg
```

Una vez creada dicha carpeta se está en disposición de crear el nuevo tipo de mensaje. Se recuerda que contendrá 3 variables las cuales se definirán del tipo float ya que los valores de temperatura presión y humedad contendrán parte decimal. Para ello:

```
~/catkin_ws/src/alvaro/msg$ nano meteo.msg
```

Dentro del nuevo tipo de mensaje se definirán dichas variables de la siguiente forma:

```
float32 temp
float32 hum
float32 press
```

Una vez realizados estos pasos se está en disposición de generar los archivos de código fuente para la definición del mensaje. Para ello en el directorio raíz del *catkin_workspace* invocaremos de nuevo la herramienta *catkin_make:*

```
$ catkin_make
```

Para cualquier archivo ".msg" contenido en el directorio *msg* del paquete alvaro se generará código fuente para los lenguajes de programación soportados.

- C++ : *~/catkin_ws/devel/inlcude/alvaro*
- Python: ~/catkin_ws/devel/lib/python2.7/dist-packages/alvaro/msg

- Lisp: ~/catkin_ws/devel/share/common-lisp/ros/alvaro/msg

6.2.4 Integración de cámara on-board

Para el presente proyecto se han estudiado dos alternativas para dotar de funcionalidad de captura de imágenes al robot objetivo: integrar una webcam USB con ROS y utilizar el accesorio cámara del Raspberry Pi.

6.2.4.1 Integración de web-cam USB con ROS

Como primera alternativa se ha estudiado la posibilidad de utilizar una webcam estándar que pueda estar presente en un hogar corriente.

En este caso se ha trabajado con la Flexxcam de Conceptronics la cual se muestra en la **figura 51**.

Figura 50 Conceptronics Flexxcam

Las principales características se recogen en la **tabla 12**.

Botón Snapshot para hacer fotografías interpoladas de 4 megapíxeles
Incluye la función Face Tracking: la webcam realiza un seguimiento de la cara y la mantiene automáticamente en el centro del cuadro de vídeo
1,3 megapíxeles
Compatible con USB 2.0 y 1.1
Botón Snapshot para captura de imágenes
Micrófono incorporado
Tamaño de imagen: VGA 1280 x 960 píxeles
Exposición electrónica automática: 1/60 a 1/2000 segundos
Frecuencia de imágenes: 35 fps (320 x 240), 30 fps (640 x 480) y 15 fps (1280 x 960)

Trípode flexible para su uso sobre la pantalla LCD y en la mesa
Objetivo ajustable
Corrección automática de la imagen
Alimentación por el puerto USB

Tabla 12 Características Conceptronic Flexxcam

6.2.4.1.1 Instalación de driver ROS y dependencias

El primer paso para lograr la integración de la webcam con el sistema ROS consiste en adaptar un driver que permita su detección y uso. En la comunidad ROS existen varias alternativas disponibles en Github que permiten dotar de compatibilidad al sistema con dicho dispositivo.

El driver elegido en cuestión es *usb_cam* contenido en el conjunto *bosch_drivers* desarrollado por Benjamin Pitzer.

El nodo *usb_cam_node* interactúa con cámaras USB estándar y publica las imágenes en el tipo de mensaje *sensor_msgs::Image*. Utiliza la biblioteca image_transport para permitir el transporte de imagen comprimido.

El primer paso consiste en descargar el driver mediante la herramienta *svn*. Por defecto dicha herramienta no viene instalada en *Raspbian* ni con ROS, por lo que previo a la descarga del driver se ha este componente:

```
$ sudo apt-get install subversion
```

Una vez instalado la herramienta se procede a la instalación de los drivers en la siguiente ruta:

```
$ cd ~/catkin_ws/src
```

En dicha ruta se ejecuta la siguiente instrucción:

```
$ svn co https://bosch-ros-
pkg.svn.sourceforge.net/svnroot/bosch-ros-
pkg/trunk/stacks/bosch_drivers
```

A continuación se han de instalar ciertos paquetes que son pre-requisitos para que funcione correctamente:

```
$ sudo apt-get install libavcodec-dev
$ sudo apt-get install libswscale-dev
$ sudo apt-get install libtinyxml-dev
```

Antes de compilar se han de instalar ciertas dependencias que el paquete *usb_cam* tiene con otros paquetes ROS, más concretamente con el paquete *self_test,* las cuales se encuentran recogidas en el conjunto de paquetes *diagnostics*.

Para ello, sin salir del directorio en el que se invocó la herramienta *svn* anteriormente se ha de ejecutar lo siguiente:

```
$ git clone  https://github.com/ros/diagnostics.git
```

El sistema de diagnóstico está diseñado para recoger información de los controladores de hardware. *Diagnostics Stack* contiene herramientas para la recopilación, edición, análisis y visualización de datos de diagnóstico. Por su parte, el paquete *self_test* permite a los nodos recopilar y publicar datos de diagnóstico.

Una vez finalizada su descarga ya están todas las dependencias convenientemente descargadas en el sistema por lo que el siguiente paso consiste en compilar y construir los paquetes. Para ello se ha de ejecutar el siguiente comando en el directorio raíz del *catkin workspace* para compilar y construir los drivers:

```
$ catkin_make
```

Una vez instalados los paquetes se ha de construir el paquete de los drivers invocando la herramienta *rosmake* dedicada a tal efecto desde el directorio raíz del *catkin_workspace*.

```
$ rosmake
```

6.2.4.1.2 Prueba de funcionamiento y conclusiones

Superada con éxito la fase de instalación el siguiente paso es comprobar el funcionamiento. Para ello se conecta la webcam a una de las dos interfaces USB del Raspberry Pi y a continuación comienza la fase de pruebas.

Para ello se puede escribir un archivo ".launch" con todo el proceso de configuración incluido y evitar así tener que llamar por separado al servicio *roscore* y ejecutar los nodos por separado. El archivo se crea mediante el editor de textos *nano* con el siguiente contenido:

```
$ nano usb_cam-test.launch
```

```
<launch>

  <node   name="usb_cam"   pkg="usb_cam"   type="usb_cam_node"
output="screen" >

    <param name="video_device" value="/dev/video0" />

    <param name="image_width" value="640" />

    <param name="image_height" value="480" />

    <param name="pixel_format" value="yuyv" />

    <param name="camera_frame_id" value="usb_cam" />

    <param name="io_method" value="mmap"/>

  </node>

  <node  name="image_view"  pkg="image_view"  type="image_view"
respawn="false" output="screen">

    <remap from="image" to="/usb_cam/image_raw"/>

    <param name="autosize" value="true" />

  </node>
```

Una vez creado y guardado se procede a su ejecución mediante el comando *roslaunch*:

```
$ roslaunch usb_cam-test.launc
```

Como se observa a continuación la captación de imágenes funciona correctamente y se está enviando video.

```
usb_cam video_device set to [/dev/video0]
usb_cam io_method set to [mmap]
usb_cam image_width set to [640]
usb_cam image_height set to [480]
usb_cam pixel_format set to [yuyv]
usb_cam auto_focus set to [0]
18 frames/sec at 1396606268.777745625
24 frames/sec at 1396606269.775914579
25 frames/sec at 1396606270.776405433
21 frames/sec at 1396606271.811980809
24 frames/sec at 1396606272.811122719
21 frames/sec at 1396606273.806696779
```

El problema en adoptar esta solución reside en las limitaciones hardware del Raspberry Pi ya que solo cuenta con dos puertos USB. Se podría utilizar un HUB para incrementar el número de puertos disponibles pero es necesario que éste sea alimentado externamente por lo que pierde funcionalidad y movilidad el robot diseñado. Otro inconveniente es que la calidad de la imagen recibida no es buena ni en resolución ni en fps por lo que, pese a integrar esta solución, se descarta en favor de la Raspberry Pi Camera.

6.2.4.2 Raspberry Pi Camera

Tal y como se ha comentado en el punto anterior la solución elegida para dotar con la funcionalidad de captura de imágenes a este proyecto ha sido la cámara Raspberry Pi, debido a que el Raspberry Pi cuenta con una interfaz dedicada exclusivamente a ella y no se necesita un hub alimentado.

6.2.4.2.1 Instalación y configuración

6.2.4.2.1.1 Conexión

El primer paso consiste en conectar la Raspberry Pi Camera en la interfaz dedicada situada justo detrás del puerto Ethernet tal y como muestra la **figura 52**.

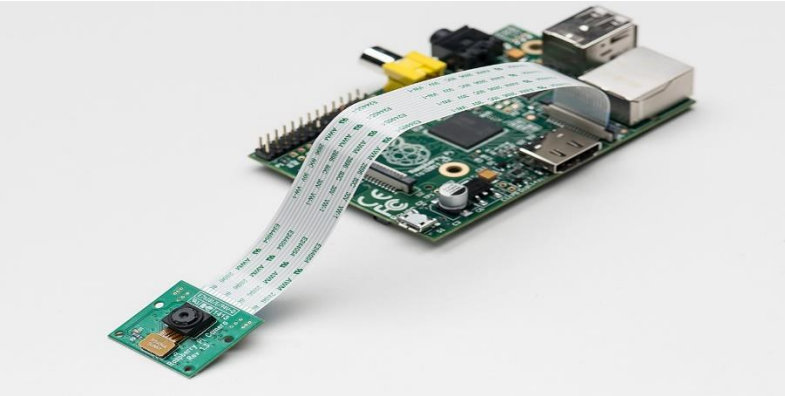


Figura 51 Conexionado Raspberry Pi Camera

6.2.4.1.2 Configuración

Una vez conectada la cámara y encendido el sistema, desde un terminal, se ha de escribir lo siguiente:

```
$ sudo raspi-config
```

Se abrirá el menú de configuración del Raspberry Pi y tan solo será necesario navegar hasta "camera" y activar el servicio como se muestra en la **figura 53**.

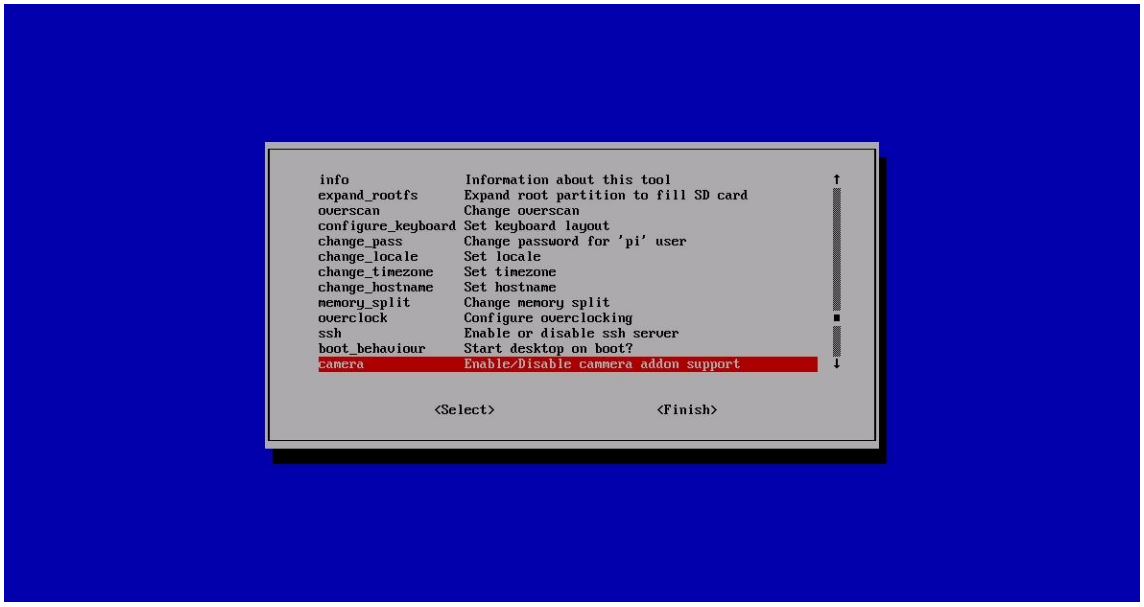


Figura 52 RaspiConfig Enable Camera

Una vez activada, se reinicia el sistema y ya estará disponible para su utilización.

6.2.4.2.2 Prueba de funcionamiento

Existen diferentes posibilidades a la hora de capturar imágenes ya sea tanto en formato, como en calidad y tamaño, tiempo de grabación o incluso la posibilidad de hacer streaming en la red.

6.2.4.2.2.1 Captura de imágenes y video

Raspivid es una aplicación de línea de comandos que permite capturar video utilizando el módulo de la cámara Raspberry Pi. Mientras que, por otro lado, *raspistill*, es la aplicación de línea de comandos que nos permite realizar capturas de imágenes.

Las opciones "-o" o "-output" especifican el nombre del fichero de salida y "-t" o "-timeout" especifican el tiempo que será mostrado en milisegundos. Por defecto viene configurado a 5 segundos por lo que *raspistill* capturará la imagen última del tiempo de exposición.

Las opciones "-d" o "-demo" ejecutan la función demostración que harán un ciclo a través de los diversos efectos de imagen que están disponibles.

Por ejemplo si se desea capturar una imagen en formato jpeg como la mostrada en la figura X, se ha de escribir lo siguiente en línea de comandos:

```
$ raspistill –o imagen.jpg
```

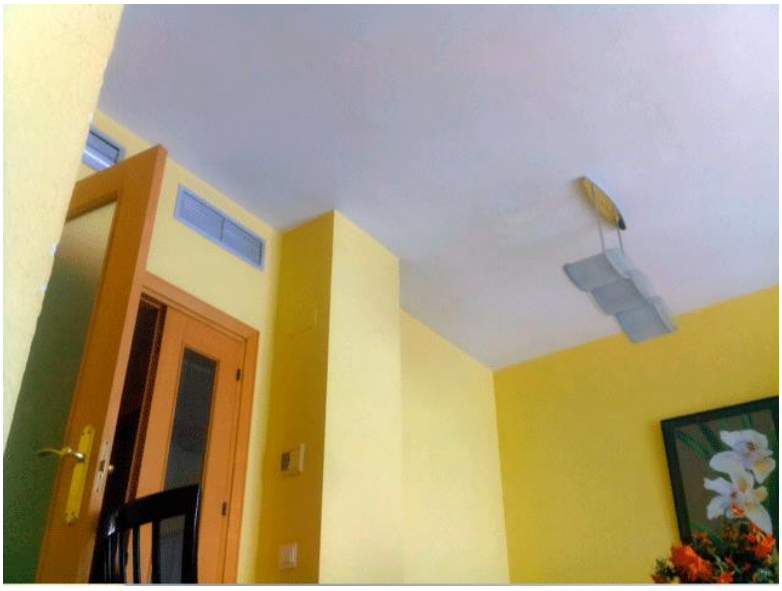


Figura 53 Captura Raspberry Pi Camera

De igual manera si lo que se desea es obtener una grabación de video de 5 segundos codificada en formato h264, se ha de proceder de la siguiente manera:

```
$ raspivid –o video.h264
```

Existen numerosas alternativas y opciones a la hora de correr estas dos herramientas las cuales pueden ser listadas mediante los siguientes comandos:

```
$ raspivid | less
$ raspistill | less
```

6.2.4.2.2.2 Streaming

Existen numerosas alternativas a la hora de hacer streaming desde el Raspberry Pi. Por la que se ha optado es por utilizar la propia herramienta *raspivid* indicándole que ha de enviar el video broadcast a través de la red.

Para ello se necesita la herramienta VLC Player instalada por lo que el primer paso desde el terminal consiste precisamente en ello:

```
$ sudo apt-get install vlc
```

6.2.5 Programación Arduino

A partir de este momento se está en disposición de crear el sketch de Arduino que se encargará de actuar sobre los motores en función de los datos que se publiquen desde el nodo controlador el cual se programará y ejecutará en el Raspberry. De igual manera, dicho sketch contendrá la publicación de los valores de los sensores meteorológicos los cuales irán contenidos en el tipo de mensaje customizado creado para tal fin.

Previo paso a la escritura del sketch se ha de generar el fichero cabecera para el tipo de mensaje customizado para que sea compatible con la placa Arduino.

6.2.5.1 Generación del archivo de cabecera para Arduino

Tal y como se ha comentado anteriormente, es necesario generar el archivo de cabecera para el mensaje customizado de manera que sea posible su uso en Arduino.

Para ello se ha de ejecutar la siguiente instrucción en línea de comandos la cual generará los archivos cabecera para todos los tipos de mensaje que se hayan creado en el directorio que se le indique:

```
$ rosrun rosserial_arduino make_libraries.py <directorio>
```

Una vez generados se han de añadir los archivos de cabecera necesarios a la carpeta *ros_lib* que se encuentra en el ordenador en el que se instaló la herramienta Arduino IDE, en "*documentos/Arduino/libraries"*.

6.2.5.2 Sketch de Arduino

6.2.5.2.1 Librerías y definiciones

Lo primero que hay que añadir en el archivo son las librerías necesarias para el funcionamiento de Arduino con ROS, el archivo cabecera del mensaje customizado, las librerías que permitirán el funcionamiento del Weather Shield así como de sus sensores on-board y la librería *Wire.h* que permitirá la comunicación por los pines SDA/SCL de Arduino con el Weather Shield.

```
#include <Wire.h>
#include <MPL3115A2.h>
#include <HTU21D.h>
#include <ArduinoHardware.h>
#include <alvaro/meteo.h>
#include <ros.h>
#include <std_msgs/Int8.h>
#include <std_msgs/Float32.h>
```

A continuación de definen los pines de los motores que permitirán actuar sobre ellos indicándoles el sentido de rotación así como la velocidad.

```
#define motizqH 17
#define motizqL 18
#define motderH 19
#define motderL 20
#define veloc 6
```

Se configura una pareja de pines digitales para cada motor. Estos pines irán conectados al L293D mediante los pines IN1, IN2, IN3 e IN4. Estos pines controlarán el sentido de giro de los motores del siguiente modo:

- Input 1 “HIGH”, Input2 “LOW” Input 3 “HIGH”, Input 4 “LOW”: ambos motores se moverán en sentido opuesto a las agujas del reloj por lo que el robot avanzará hacia delante.
- Input 1 “HIGH”, Input2 “LOW”, Input 3 “LOW”, Input 4 “HIGH”: el motor derecho girará en sentido de las agujas del reloj mientras el que el izquierdo girará al contrario por lo que el conjunto del robot girará sobre sí mismo hacia la derecha
- Input 1 “LOW”, Input2 “HIGH”, Input 3 “HIGH”, Input 4 “LOW”: el motor izquierdo girará en sentido de las agujas del reloj mientras el que el derecho girará al contrario por lo que el conjunto del robot girará sobre sí mismo hacia la izquierda.
- Input 1 “LOW”, Input2 “HIGH”, Input 3 “LOW”, Input 4 “HIGH”: ambos motores girarán en sentido de las agujas del reloj por lo que el conjunto del robot avanzará hacia atrás.

Estas salidas vendrán controladas por una variable de tipo entero a la cual se subscribe Arduino con el fin de controlar dichos motores. Dicha variable será publicada en el Raspberry Pi mediante el topic “opción”.

Para la obtención de los datos de los sensores de meteorología se harán uso de las funciones destinadas a tal efecto en las librerías correspondientes a cada sensor. Dicho proceso se puede consultar en el ANEXO X.

6.2.5.2.2 Subscriptores

Conforme se reciben datos a través del subscriptor al cual se ha llamado “opt” se ejecuta la función de Callback. El evento de recepción de un dato a través de un suscriptor provoca una interrupción que llama a esta función y que según el valor de la variable “opción” recibida se genera una salida en los pines digitales asignados a las entradas del L293D anteriormente descritas de la siguiente manera:

- Opción=5. Stop
- Opción =1. Avanzar
- Opción=2. Giro a la derecha
- Opción=3. Giro a la izquierda
- Opción=4. Hacia atrás

La función que atiende los eventos del subscriptor es la siguiente:

```
void opcionCallback(const std_msgs::Int8& option)

{ int op;

int a=0;

op=option.data;

//if(digitalRead(led)==LOW)

if(a==0)

{ switch(op)

{case                                                                    0:
digitalWrite(motizqL,LOW);digitalWrite(motizqH,LOW);digitalWri
te(motderH,LOW);digitalWrite(motderL,LOW);digitalWrite(10,HIGH
); break;

case                                                                    1:
digitalWrite(motizqL,HIGH);digitalWrite(motizqH,LOW);digitalWr
ite(motderH,HIGH);digitalWrite(motderL,LOW); break;

case                                                                    2:
digitalWrite(motizqL,HIGH);digitalWrite(motizqH,LOW);digitalWr
ite(motderH,LOW);digitalWrite(motderL,HIGH); break;

case                                                                    3:
digitalWrite(motizqL,LOW);digitalWrite(motizqH,HIGH);digitalWr
ite(motderH,HIGH);digitalWrite(motderL,LOW); break;

case                                                                    4:
digitalWrite(motizqL,LOW);digitalWrite(motizqH,HIGH);digitalWr
ite(motderH,LOW);digitalWrite(motderL,HIGH); break;

default:
digitalWrite(motizqL,LOW);digitalWrite(motizqH,LOW);digitalWri
te(motderH,LOW);digitalWrite(motderL,LOW);

}}else{digitalWrite(motizqL,LOW);digitalWrite(motizqH,LOW);dig
italWrite(motderH,LOW);digitalWrite(motderL,LOW);}}
```

De igual manera se define un subscriptor llamado “vel” el cual recibirá un tipo de dato entero el cual determinará la velocidad de giro para cada uno de los casos anteriormente descritos.

La velocidad se recibirá escalada entre 0 y 256 para su posterior establecimiento en el pin PWM asociado. Para ellos se definen tres posibles valores:

- Velocidad nula 0
- Velocidad media 180
- Velocidad de viaje 255

El bloque de código que sigue se encarga de recibir el valor escalado de la velocidad y transmitirlo a los motores:

```
void velocidadCallback(const std_msgs::Float32& velo)
{
  float velocidadpwm;
  velocidadpwm=velo.data;
  analogWrite(veloc,velocidadpwm);
}
```

6.2.5.2.3 Setup e inicialización

Se han de definir la función que tendrán los pines, inicializar el manejador de ROS y definir los publicadores y subscriptores. Definimos el publicador (el cual publicará los datos de los sensores meteorológicos), los subscriptores, en manejador de ROS y el tipo de mensaje custom de la siguiente forma:

```
alvaro::meteo medidas;
ros::Publisher i2c("i2c", &medidas);
ros::Subscriber<std_msgs::Int8> opt("opcion",&opcionCallback);
ros::Subscriber<std_msgs::Float32>
vel("velocidad",&velocidadCallback);
ros::NodeHandle nh;
```

Donde se ha definido el mensaje “medidas” de tipo meteo que es el que publicará los valores de los sensores de temperatura, se han definido los subscriptores “vel” y “opt” los cuales se subscriben a los topics “velocidad” y “opción” respectivamente y el manejador de ROS nh.

A continuación en la fase de setup se incluirán las funciones de los pines para que el Arduino sepa cómo tratarlos. De igual manera se inicializan y definen el comportamiento de los sensores y funciones del Weather Shield lo cual, se puede comprobar en el Anexo B.

```
    pinMode(motderH,OUTPUT);
    pinMode(motderL,OUTPUT);
      pinMode(motizqH,OUTPUT);
        pinMode(motizqL,OUTPUT);
        pinMode(10,OUTPUT);
        pinMode(veloc,OUTPUT);
  nh.initNode();
  nh.subscribe(opt);   //Subscriptor   para   la   opcion   de
funcionamiento
  nh.subscribe(vel); //Subscriptor para la velocidad
  nh.advertise(i2c); //Publicador para los sensores i2c
```

6.2.5.2.4 Publicación de los datos

Por último, una vez obtenidos los datos de los sensores se han de publicar para su posterior recepción en el Raspberry Pi. Para ello:

```
// Publicación de los datos de los sensores de meteorología
medidas.temp=tempC;
medidas.press=pressure;
medidas.hum=humidity;
i2c.publish( &medidas );
delay (100);
nh.spinOnce();
```

Donde se está asignando a cada una de las variables que componen el mensaje customizado "medidas" su correspondiente valor obtenido de los sensores. A continuación se publica el contenido de dicho mensaje mediante el publicador "i2c".

Finalmente "nh.spinOnce()" permite que no se pierda el sincronismo entre Arduino y Raspberry Pi y se encarga de controlar la recepción de datos y la ejecución de las funciones Callback.

6.2.6 Programación del nodo controlador en Raspberry

Por último ya solo queda el desarrollo del código que se encargará de actuar con los datos que se integrarán con ROS provenientes de los sensores del Smartphone Android que permitirán lograr el objetivo de esta primera aplicación.

Este programa que se va a compilar ejecutará las siguientes tareas:

- Orientación al Norte.
- Alcance de posición GPS deseada.
- Definición de los publicadores "opción" y "velocidad" que indicarán al robot cómo comportarse.
- Establecimiento de las condiciones de marcha o parada en función de la intensidad lumínica.

Para llevar a cabo las tareas descritas, este programa tomará los datos de los topics Android/Imu, Android/Illuminance y Android/fix para conseguir la orientación, el umbral de luz con el cual trabajar y la posición GPS.

6.2.6.1 Definiciones

Se define pues una clase en C++ en la que aparecerán accesibles públicamente los subscriptores y publicadores para que puedan tratarse desde las funciones de Callback. A su vez se definirán dos variables públicas *latitudepublic* y *longitudepublic* que serán accesibles desde la función *main* para que el usuario pueda introducir las coordenadas GPS deseadas.

Las variables "lux" y "orientado" actuarán como banderas las cuales determinarán si el robot se ha orientado correctamente al Norte y si las condiciones de luz ambientales permiten la marcha o por el contrario se determinará la parada. Por tanto su uso será como banderas.

Se definen a su vez los tres subscriptores (*imu, luz y GPS)* a los cuales se hizo referencia anteriormente así como sus correspondientes funciones de Callback las cuales atenderán la recepción de los datos correspondientes a los sensores implicados.

Se definen igualmente los dos publicadores (*vel y opt)* que se encargarán de publicar la velocidad de funcionamiento así como la opción de movimiento (avance, giro, parada o marcha atrás).

6.2.6.2 Algoritmo de orientación al Norte

Dentro del topic Android/imu se utilizará la variable *w* del mensaje *orientation* como valor para establecer la orientación al Norte. La función encargada de llevar a cabo esta tarea es la función de Callback "void imuCallback(const sensor_msgs::Imu& imu_msg)".

La **figura 55** representa un esquema simplificado del algoritmo de orientación al Norte.

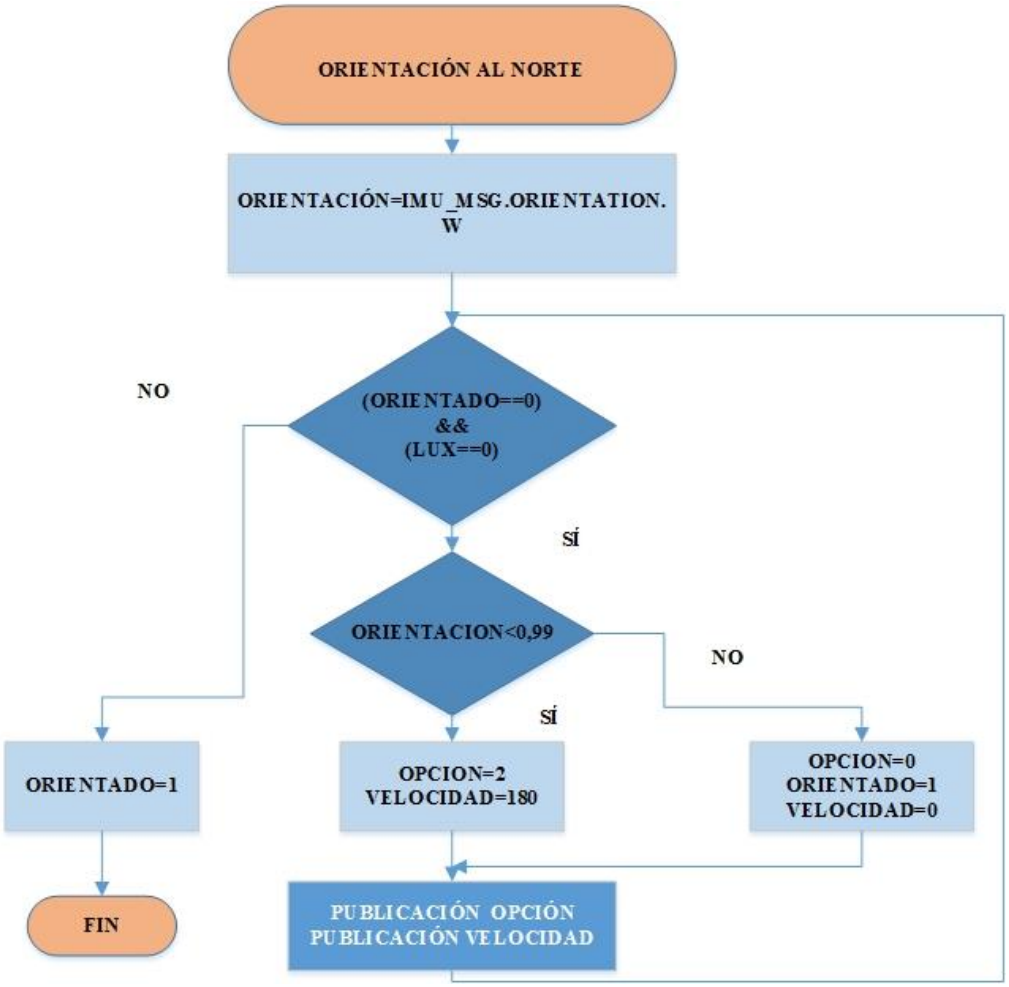


Figura 54 Algoritmo de orientación al Norte

Mientas las condiciones lumínicas permitan continuar con la marcha y el robot no se haya orientado al Norte, se publicará la opción 2 (giro a la derecha) y la velocidad 180 (velocidad media). Una vez orientado el robot al Norte, la bandera *orientado* se pondrá a 1 y se proseguirá con la búsqueda de las coordenadas GPS introducidas por el usuario.

6.2.6.3 Algoritmo de Alcance de la posición GPS

Una vez que el robot se ha orientado al Norte comenzará la búsqueda de las coordenadas GPS introducidas por el usuario.

Para realizar este cometido, el robot se moverá de forma semejante a como lo hace la pieza de ajedrez conocida como "caballo". Esto quiere decir que el robot se moverá sobre la línea de la Latitud aprovechando la orientación al Norte y, en función de las coordenadas deseadas y las actuales del robot, avanzará o retrocederá. Una vez alcanzada la Latitud deseada el robot ejecutará un giro de 90º para alinearse en la

Longitud, y avanzará o retrocederá en función de la Longitud deseada y en la que se encuentre. La **figura 56** representa un esquema simplificado de cada iteración de la función de Callback.

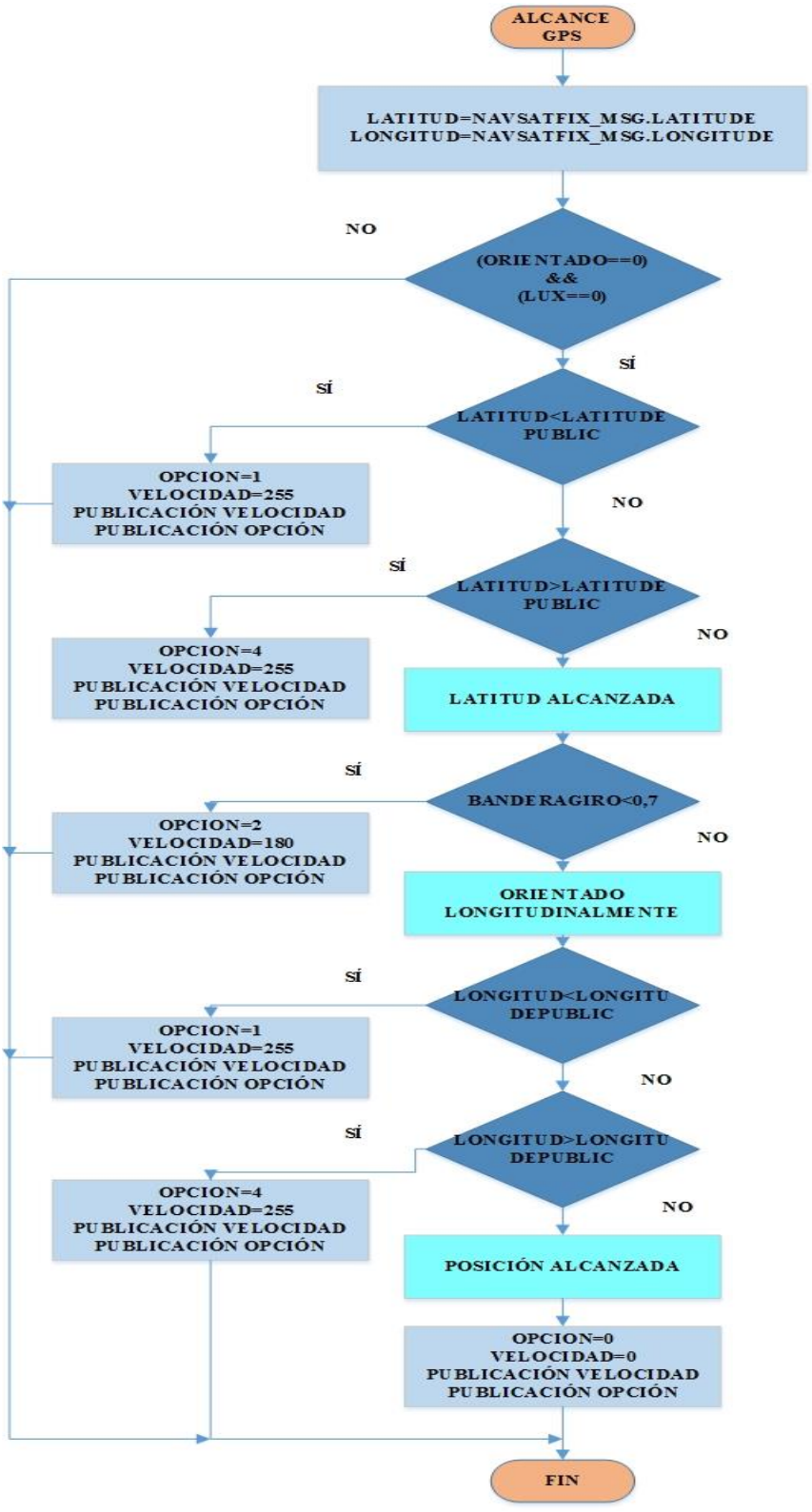


Figura 55 Algoritmo de alcance de posición GPS

6.2.6.4 Algoritmo de control de la iluminación

El robot diseñado funcionará siempre y cuando las condiciones lumínicas sean aceptables y se encuentren dentro de un rango que permita la diferenciación de objetos por parte del ojo humano. Para ellos se ha establecido el valor del umbral sobre el que se decide en 20. Dicho valor se ha elegido realizando pruebas de luminosidad del entorno medidas con el sensor de luz del Smartphone.

En este caso el subscriptor "luz" tomará los datos del topic Android/illuminance en los que se estarán publicando los datos del sensor lumínico. La variable "lux" actuará como bandera la cual indicará al resto del programa cuando seguir operando o no en función de la visibilidad.

La **figura 57** muestra un esquema simplificado del funcionamiento de la función de Callback asociada a este evento.

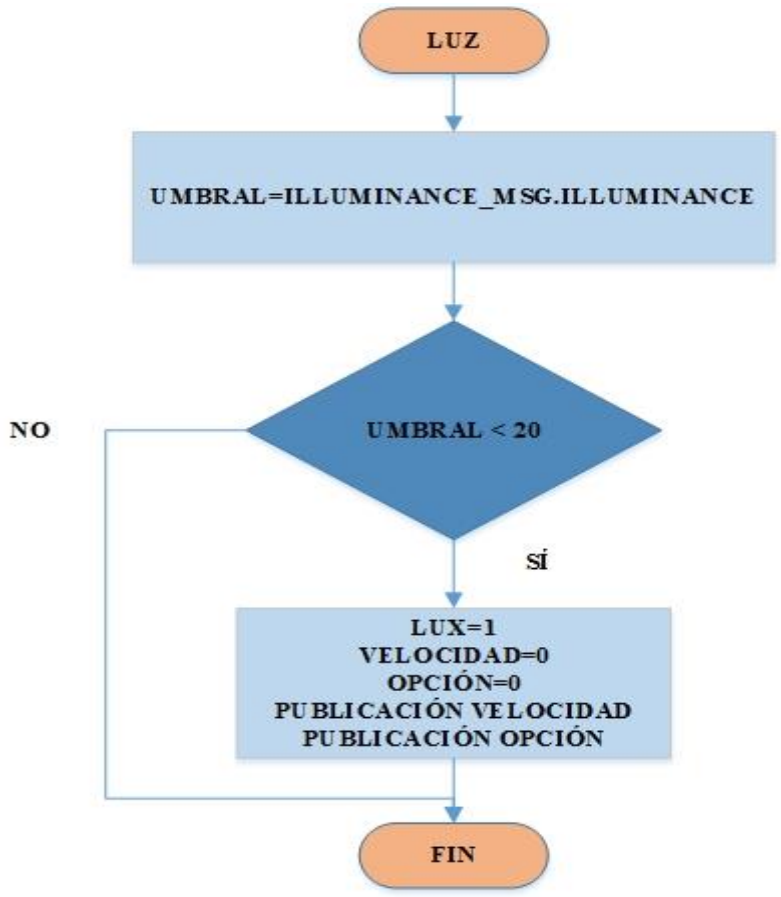


Figura 56 Algoritmo luzCallback

6.2.6.5 Función Main

Por último en la función main, se inicializa el nodo con el nombre "robot1" y establecemos un *rate loop* de 100 milisegundos. Es decir, la rutina de control se ejecutará cada 100 milisegundos, tiempo más que suficiente para la atención de los publicadores y la actuación sobre los motores.

De igual manera se define un objeto de tipo "robot1". Así se llamará al constructor de C++ el cual construirá la clase que se definió al principio del programa. Se utilizarán dos variables locales para trabajar con los valores de coordenadas GPS que se le solicitarán al usuario por pantalla y una vez introducidos se asignarán a las correspondientes variables públicas *longitudepublic* y *latitudepublic*.

6.2.6.6 Compilación

Una vez finalizado el código del programa tan solo resta la fase de compilación. Previo a ella se han de añadir unas líneas de código al archivo CMakeLists.txt al final del mismo.

Dicho archivo se encontraba en la ruta siguiente:

```
~/catkin_ws/src/alvaro/
```

Se han de añadir las siguientes líneas:

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(robot1 src/robot1.cpp)

target_link_libraries(robot1 ${catkin_LIBRARIES})

add_dependencies(robot1 alvaro_generate_messages_cpp)
```

Hecho esto se ha de navegar hasta el directorio raíz del *catkin workspace* y ejecutar la orden de compilación:

```
catkin_make
```

6.3 Prueba de funcionamiento

Una vez se ha compilado con éxito el programa y el sketch de Arduino no arroja error alguno se está en disposición de realizar las pruebas pertinentes sobre la aplicación.

Se lleva a cabo el conexionado de todos los elementos tal y como se ha descrito con anterioridad y se procede como sigue.

- Inicialización del servicio *roscore*
- Inicialización de la comunicación serie entre Arduino y Raspberry:

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

- Ejecución del programa diseñado:

```
$ rosrun alvaro robot1
```

Hecho esto ya está el sistema funcionando y trabajando en ambas plataformas.

Cuando establecemos la comunicación serie se comprueba que se han establecido correctamente los publicadores y subscriptores de Arduino tal y como se muestra en la **figura 58**.

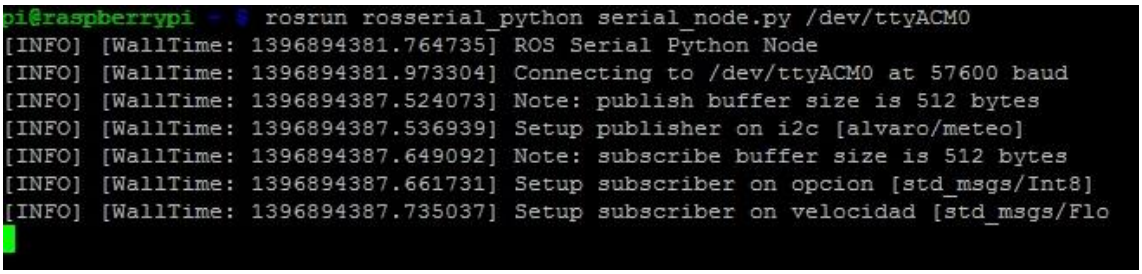


Figura 57 Aplicación 1 Establecimiento de comunicación

A continuación se han de comprobar los topics que se están publicando para cerciorarse de que coinciden con lo diseñado. Para ello ejecutamos la siguiente instrucción en línea de comandos:

```
$ rostopic list
```

El resultado de esta ejecución se muestra en la **figura 59** y pone de manifiesto cómo se están publicando correctamente tanto los sensores del Smartphone como el topic "i2c" (que contiene los datos de los sensores meteorológicos) así como los topics "opción" y "velocidad" que controlan el robot.

```
pi@raspberrypi ~ $ rostopic list
/android/barometric_pressure
/android/fix
/android/illuminance
/android/imu
/android/magnetic_field
/diagnostics
/i2c
/opcion
/rosout
/rosout_agg
/velocidad
pi@raspberrypi ~ $
```

Figura 58 Topics Aplicación 1

La herramienta *rqt_graph* puede ayudar a entender mejor de forma esquemática cómo está funcionando el sistema mostrando los nodos y topics indicando además, cuáles son publicados por cada nodo y a cuales está subscrito cada nodo.

En la **figura 60** se muestra un esquema completo de cómo se están comunicando los nodos y de los topics con los que se está trabajando.

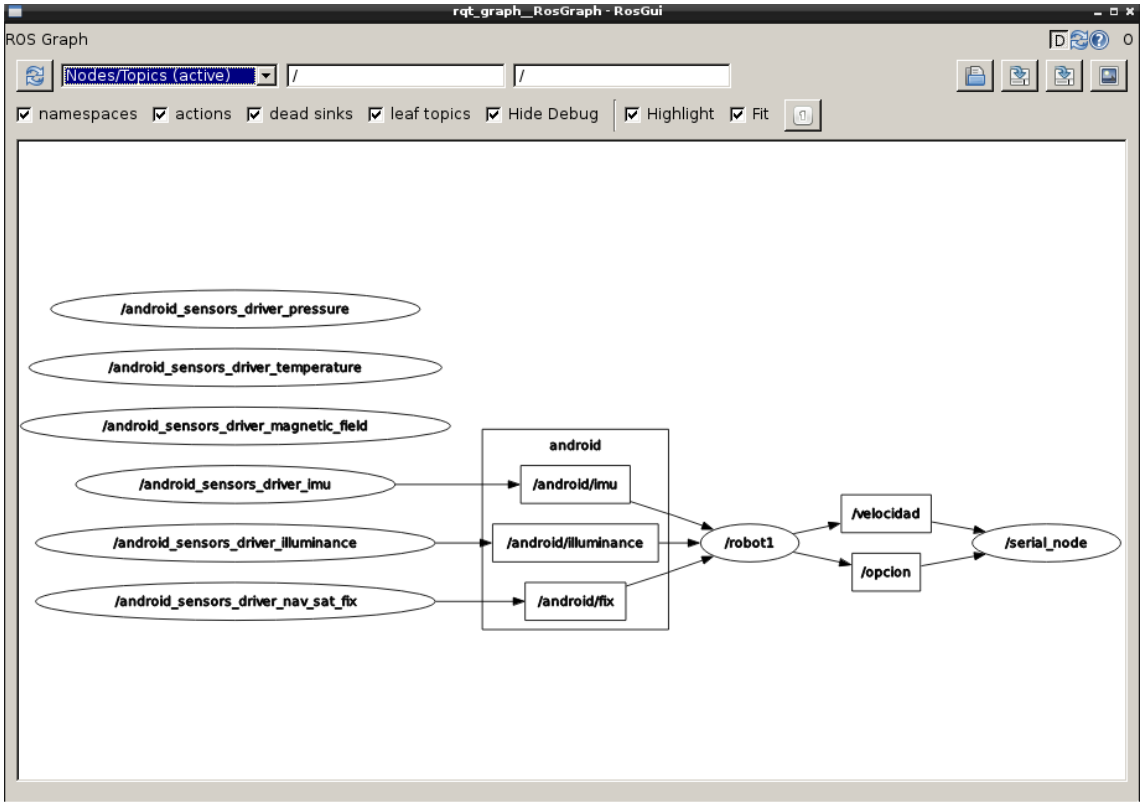


Figura 59 Esquema de funcionamiento Aplicación 1 rqt_graph

De igual manera utilizando los pluggins que ofrece la herramienta *rqt* se puede obtener información útil acerca de todo cuanto acontece en el sistema construido además de constituirse como una herramienta de gran valor a la hora de depurar, analizar y recabar información sobre lo que se está trabajando.

Para obtener más información y explorar las diferentes herramientas que nos ofrece rqt se ha de invocar dicha aplicación desde terminar:

```
$ rqt
```

Una vez abierto el programa, en la pestaña pluggins se pueden utilizar *logger_level* y *console* para obtener información en labores de debugging del sistema. La **figura 61** muestra una captura de información recibida en consola mediante el *logger_level*.

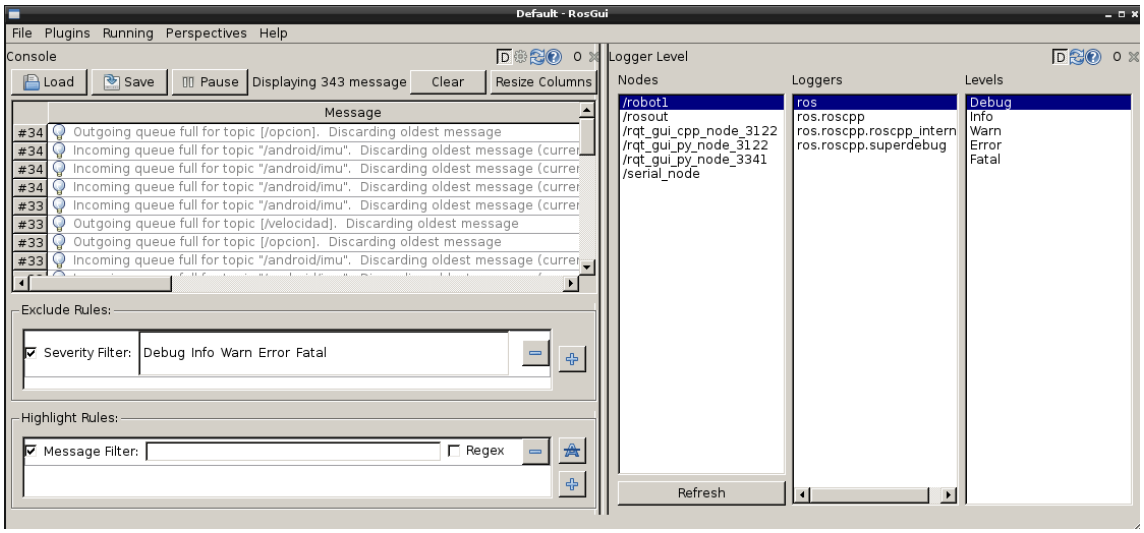


Figura 60 Aplicación 1 rqt_logger_level

De igual manera se puede obtener información acerca de los topics que se están publicando mediante el pluggin *topic introspection* la cual mostrará información relativa a los mismos y los tipos de mensajes que se están publicando como se puede ver en la **figura 62**.

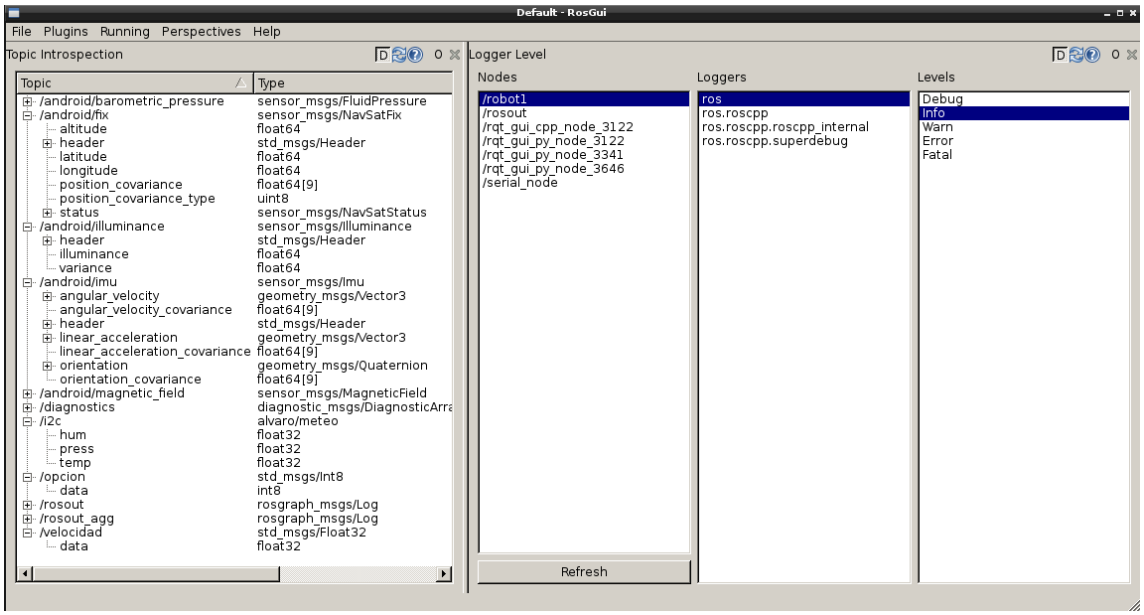


Figura 61 Aplicación 1 rqt_topic_introspection

7 Aplicación 2

7.1 Solución adoptada

Para el desarrollo de esta segunda aplicación diseñada se ha optado por los siguientes componentes:

- Arduino Mega ADK: el cual se encargará de la gestión y control del motor en función de los datos recibidos desde el Raspberry Pi y de gestionar y publicar los datos meteorológicos a los cuales estará subscrito ROS en el Raspberry.
- Raspberry Pi: corriendo el sistema operativo Raspbian con ROS convenientemente instalado se encargará de recibir los datos de meteorología y actuará como nodo centrar encargado de decidir el movimiento del robot en función de las coordenadas GPS.
- Weather Shield: este elemento se encargará de realizar las mediciones de temperatura, humedad y presión.
- Raspberry Pi Camera: la cual permitirá la toma de imágenes del medio.
- Sensores de ultrasonido HCSR04: Se utilizarán 5 sensores que dotarán al sistema con la capacidad de evitar obstáculos.
- Smartphone Android: proporcionará los datos del sensor de luminosidad para determinar si continuar con la marcha o no.

7.2 Preparación del entorno

Introducidos los componentes que conformarán esta segunda aplicación se da paso a la tarea de configuración y desarrollo.

Se aprovechará en gran parte la configuración llevada a cabo en la aplicación anterior tanto para los componentes integrados como para el código desarrollado por lo que en las siguientes líneas se añadirán aquellas partes que resultan novedosas con respecto a la aplicación 1.

7.2.1 Creación del custom message: Publicación de distancias

Para esta segunda aplicación se van a utilizar 5 sensores de ultra sonido gracias a los cuales se podrá determinar la distancia a posibles obstáculos e implementar una rutina de control para evitarlos.

Al igual que en la primera aplicación se ha de crear un mensaje customizado que contenga cinco variables las cuales albergarán los datos de los sensores para su posterior publicación como un tipo de mensaje customizado.

Se creará por tanto en la siguiente ruta el mensaje al cual se ha llamado *sensores.msg:*

```
~/catkin_ws/src/alvaro/msg$ nano sensores.msg
```

Dicho mensaje contendrá como se ha especificado anteriormente 5 variables de tipo float que recogerán las distancias de los sensores:

```
Float32 front
Float32 upleft
Float32 upright
Float32 left
Float32 right
```

Hecho esto, desde el directorio raíz del *catkin workspace* se invoca la orden de compilación:

```
$ catkin_make
```

7.3 Programación de Arduino

De nuevo se está en disposición de elaborar el programa de Arduino que además de controlar los motores y generar los datos relativos a los sensores de meteorología, en este caso obtendrá las distancias de los sensores y las publicará. Para ello el paso previo consiste en generar el archivo cabecera compatible con Arduino para el mensaje *sensores.msg*.

7.3.1 Generación del archivo de cabecera para Arduino

Tal y como se ha comentado anteriormente, es necesario generar el archivo de cabecera para el mensaje customizado de manera que sea posible su uso en Arduino.

Para ello se ha de ejecutar, tal y como se hizo anteriormente, la siguiente instrucción en línea de comandos la cual generará los archivos cabecera para todos los tipos de mensaje que se hayan creado en el directorio que se le indique:

```
$ rosrun rosserial_arduino make_libraries.py <directorio>
```

Una vez generados se vuelven a añadir los archivos de cabecera necesarios a la carpeta *ros_lib* que se encuentra en el ordenador en el que se instaló la herramienta Arduino IDE, en "*documentos/Arduino/libraries"*.

7.3.2 Sketch Arduino

7.3.2.1 Librerías y definiciones

En este nuevo caso se ha de añadir el fichero de cabecera generado para el mensaje customizado.

```
#include <alvaro/sensors.h>
```

A continuación se han de definir los pines de Arduino a los que irán conectados los sensores HCSR04.

```
#define ECHO 37        // Pin para recibir el pulso de eco
#define TRIGGER 36     // Pin para enviar el pulso de disparo
#define ECHO2 35
#define TRIGGER2 34
#define ECHO3 33
#define TRIGGER3 32
#define ECHO4 31
#define TRIGGER4 30
#define TRIGGER5 29
#define ECHO5 28
```

Se definirán a su vez 6 nuevas variables: cinco para almacenar las distancias de cada sensor y una sexta llamada *tiempo* que ayudará en el cálculo de las mismas.

```
unsigned int tiempo, distancia, distancia2, distancia3,
distancia4, distancia5
```

7.3.2.2 Setup e inicialización

A continuación se ha de establecer la función que tendrán los pines para que Arduino sepa cómo actuar sobre ellos, definir un tipo de dato de tipo *sensores (*asociado al custom message) y a su vez inicializar el publicador (hcsr04) que se encargará de enviar los datos relativos a los sensores de proximidad mediante el topic *hcsr04.*

```
alvaro::sensores distancias;

ros::Publisher hcsr04("hcsr04", &distancias);

nh.advertise(hcsr04);//Publicador para los sensores de ultrasonido

pinMode(ECHO,INPUT);

pinMode(ECHO2,INPUT);

pinMode(ECHO3,INPUT);

pinMode(ECHO4,INPUT);

pinMode(ECHO5,INPUT);

pinMode(TRIGGER3,OUTPUT);

pinMode(TRIGGER3,OUTPUT);

pinMode(TRIGGER3,OUTPUT);

pinMode(TRIGGER4,OUTPUT);

pinMode(TRIGGER5,OUTPUT);
```

7.3.2.3 Medida de las distancias

Tras la configuración de los pines se lleva a cabo la medida de las distancias obtenidas de cada sensor. Para ello en la función loop (la cual se ejecuta constantemente) se añaden las siguientes líneas de código que hacen posible la medición de la distancia.

```
digitalWrite(TRIGGER, LOW);

delayMicroseconds(2);

digitalWrite(TRIGGER, HIGH);

delayMicroseconds(10);

digitalWrite(TRIGGER, LOW);

// Calcula la distancia midiendo el tiempo del estado alto del pin ECHO

tiempo = pulseIn(ECHO, HIGH);

// La velocidad del sonido es de 340m/s o 29 microsegundos por centimetro

distancia= tiempo/58;
```

Dicho bloque de código se repetirá para cada uno de los sensores utilizados obteniéndose así las medidas de cada uno de ellos.

7.3.2.4 Publicación de las distancias

Por último, una vez obtenidos los datos de los sensores, se han de publicar para su posterior recepción en el Raspberry Pi. Para ello:

```
// Publicación de los datos de los sensores de proximidad
distancias.front=distancia;
distancias.upleft=distancia2;
distancias.upright=distancia3;
distancias.left=distancia4;
distancias.right=distancia5;
hcsr04.publish( &distancias );
```

Donde se está asignando a cada una de las variables que componen el mensaje customizado "sensores" su correspondiente valor obtenido de los sensores. A continuación se publica el contenido de dicho mensaje mediante el publicador "hcsr04".

7.3.3 Programación del nodo controlador en Raspberry

Por último se ha de escribir en el Raspberry Pi el código del programa que ejecutará las rutinas de control y algoritmos que informarán a Arduino del modo de funcionamiento en función de si se detectan o no obstáculos.

Para ello se ha de crear con el editor de textos deseado el archivo que contendrá todo el código en la siguiente ruta:

```
~/catkin_ws/src/alvaro/src/
```

Este programa que se va a compilar ejecutará las siguientes tareas:

- Algoritmo de detección de obstáculos.
- Definición de los publicadores "opción" y "velocidad" que indicarán al robot cómo comportarse.
- Establecimiento de las condiciones de marcha o parada en función de la intensidad lumínica.

Para llevar a cabo las tareas descritas, este programa tomará los datos de los topics Android/Illuminance y "hcsr04" para obtener los datos relativos al sensor de luz y las distancias a posibles objetos medidas con los sensores de proximidad respectivamente.

7.3.3.1 Definiciones

Se definirá en este caso una nueva clase C++ donde se albergarán los publicadores (que publicarán la opción y velocidad de movimiento) al igual que para la Aplicación 1 y los subscriptores; estos se subscribirán a los topics Android/Illuminance (para leer los datos del sensor de luz) y al topic "hcsr04" para poder acceder a los datos de los sensores de proximidad. De igual manera se definirá la función de Callback asociada al subscriptor que se le ha llamado "dist" que ejecutará el algoritmo de evitación de obstáculos.

7.3.3.2 Algoritmo sorteador de obstáculos

El algoritmo tendrá en cuenta las distancias recibidas de los 5 sensores HC-SR04 mediante el topic "hcsr04". Dichas distancias serán interpretadas en el nodo Master y en función de ellas se tomará una decisión u otra en cuanto al funcionamiento del robot.

Se definen por tanto 3 intervalos de distancias, expresadas en centímetros, que se tendrán en cuenta:

```
#define intervalo1 20

#define intervalo2 30

#define intervalo3 40
```

Cualquier obstáculo en "intervalo3" activará la detección y se anotará dicho evento en una variable bandera llamada "obstáculo". En caso de no detectarse objeto alguno simplemente se dará la orden de avanzar (opción 1 y a máxima velocidad v=255):

```
op=1
v=255
```

Debido a que el robot, por construcción y diseño, avanzará hacia delante, los 3 sensores delanteros deberán ser atendidos con antelación con respecto a los otros dos. Por tanto, se establece un algoritmo de prioridad teniendo en cuenta la distancia recibida por qué sensores.

En el caso de que la distancia a un objeto detectado sea menor que la definida en el intervalo "intervalo1" se le atenderá con mayor prioridad. En esta situación tendrán mayor prioridad igualmente los tres sensores frontales con respecto a los laterales.

7.3.3.2.1 Prioridad 1

Si algún objeto es detectado en el intervalo "intervalo1" es decir, a una distancia menor de 20cm el procedimiento es el siguiente:

- En caso de que sea detectado el obstáculo por alguno de los tres sensores frontales (front, upright o upleft), el robot retrocederá lentamente (opción 4) a velocidad media (v=180). Dicha circunstancia se correspondería con la aparición repentina de un objeto ya que si no habría sido tratado por el algoritmo de prioridad 2.
- Si los que detectan el obstáculo son los sensores laterales (right o left), el robot girará a la izquierda (opción=3) o a la derecha (opción=2) a una velocidad menor (v=155).

7.3.3.2.2 Prioridad 2

En el caso de que no se produzcan las condiciones que contemplan el algoritmo de prioridad 1 se pueden dar los siguientes casos:

- Si un objeto es detectado entre los intervalos "intervalo1" e "intervalo2" (entre 30 y 40 cm) por parte del sensor frontal o alguno de los laterales, el robot girará hacia la izquierda o derecha a velocidad intermedia. Esto quiere decir que si se ha detectado por el sensor frontal derecho (upright) el robot girará a la izquierda (opción=3) mientras que si fue detectado por el frontal izquierdo (upleft) girará a la derecha (opción=2) a velocidad moderada establecida con un dutycicle de 155.
- El último caso ante el que se puede encontrar el robot es que ningún sensor reciba un obstáculo dentro de los rangos definidos por lo que la opción de funcionamiento será avanzar a máxima velocidad (opción=1 y v=255).

La **figura 63** recoge el esquema del algoritmo de evitación de obstáculos utilizado.

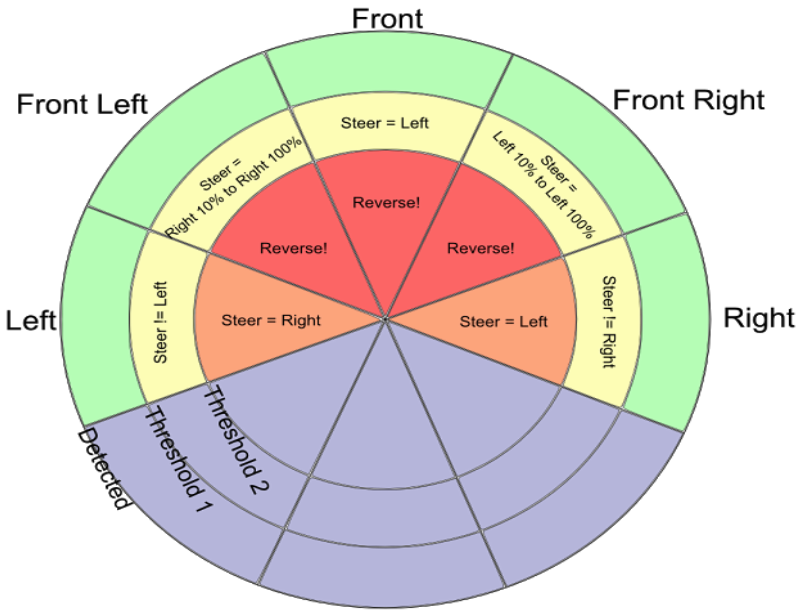


Figura 62 Esquema del algoritmo de evitación de obstáculos

7.3.3.3 Esquema conexionado sensores HC-SR04 con Arduino

La **figura 64** recoge el esquema de conexionado para uno de los sensores HC-SR04 extensible a los 5 utilizados.

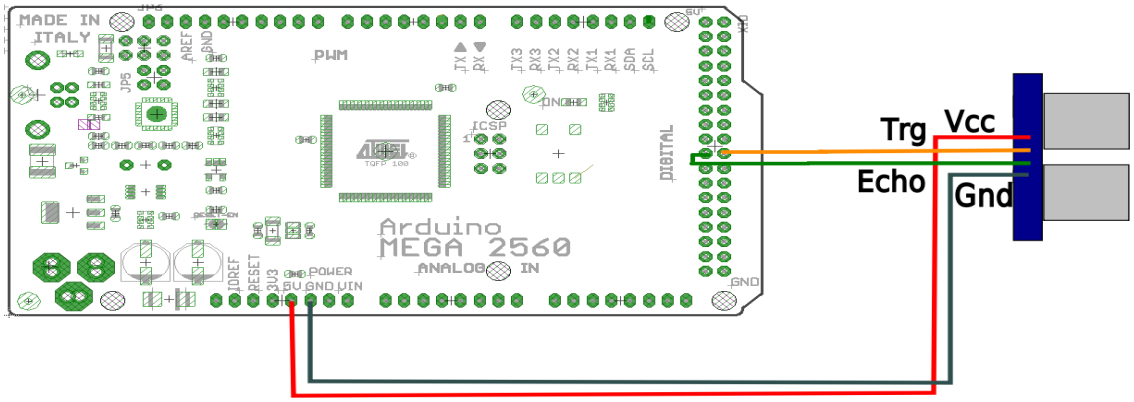


Figura 63 Esquema de conexión del HC-SR04

7.3.3.4 Compilación

Terminado el código del programa, la última tarea consiste en la compilación del mismo. De igual modo que para la Aplicación 1, previo a ella se han de añadir unas líneas de código al archivo CMakeLists.txt al final del mismo.

Dicho archivo se encontraba en la ruta siguiente:

```
~/catkin_ws/src/alvaro/
```

Se han de añadir las siguientes líneas:

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(robot1 src/robot2.cpp)

target_link_libraries(robot2 ${catkin_LIBRARIES})

add_dependencies(robot2 alvaro_generate_messages_cpp)
```

Hecho esto se ha de navegar hasta el directorio raíz del *catkin workspace* y ejecutar la orden de compilación:

```
catkin_make
```

7.3.4 Prueba de funcionamiento

Una vez depurado el código de errores y compilado, cargado el sketch en la placa Arduino sin errores y conectado todos los elementos se está en disposición de pasar a la fase de pruebas y comprobaciones.

Se lleva a cabo el conexionado de todos los elementos tal y como se ha descrito con anterioridad y se procede como sigue.

- Inicialización del servicio *roscore*
- Inicialización de la comunicación serie entre Arduino y Raspberry:

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Ejecución del programa diseñado:

```
$ rosrun alvaro robot2
```

La **figura 65** muestra el servicio *roscore* correctamente inicializado, esencial para cualquier tipo de comprobación y requisito básico de funcionamiento de ROS. De igual forma la **figura 66** muestra cómo la conexión serie entre Arduino y Raspberry Pi se ha realizado correctamente y se han inicializado publicadores y subscriptores pertenecientes al sketch de Arduino.

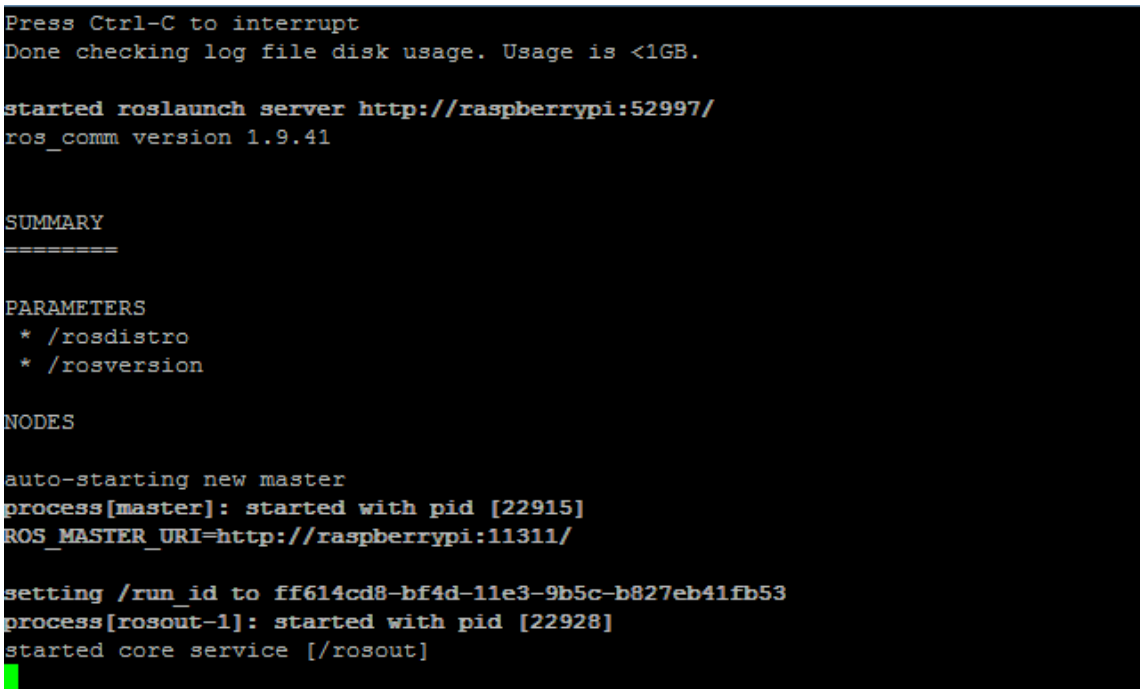


Figura 64 Ejecución del servicio roscore

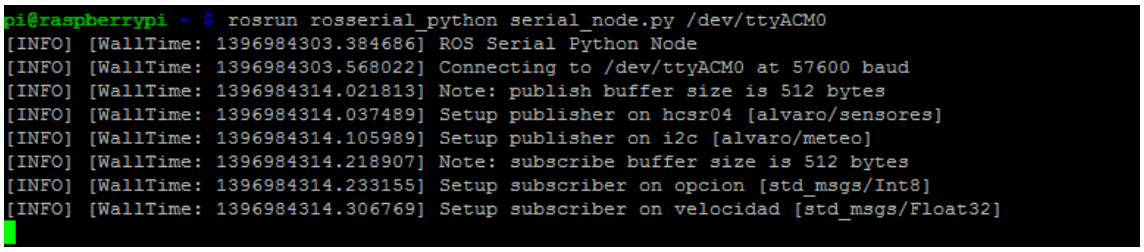


Figura 65 Aplicación 2 Establecimiento de conexión

Se procede a continuación a comprobar la correcta publicación de todos los topics presentes en esta aplicación. **La figura 67** muestra como todos los topics implicados se están publicando sin problemas.

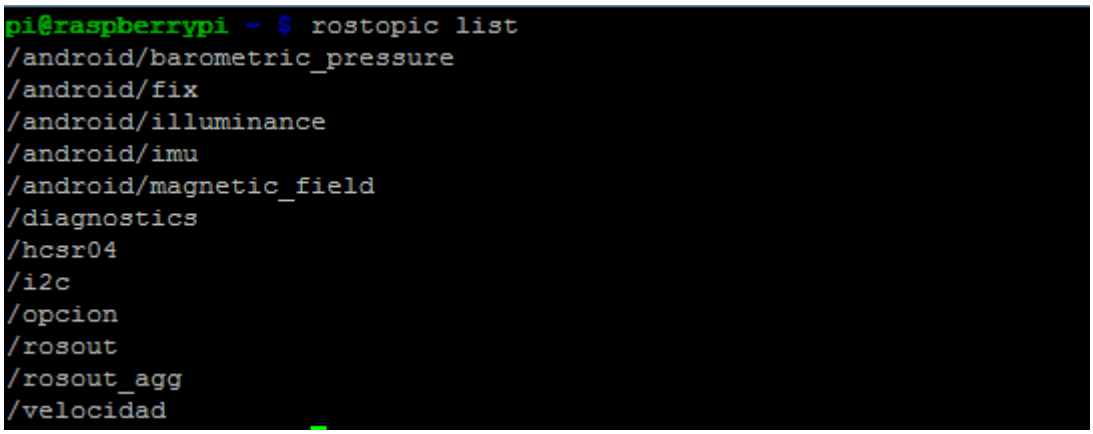


Figura 66 Aplicación 2 rostopic list

Para establecer un esquema de funcionamiento general del sistema se recurre a la herramienta *rqt.* Haciendo uso del pluggin *rqt_graph* se presenta en la **figura 68** una representación gráfica de la comunicación entre los nodos y la publicación y recepción de topics.

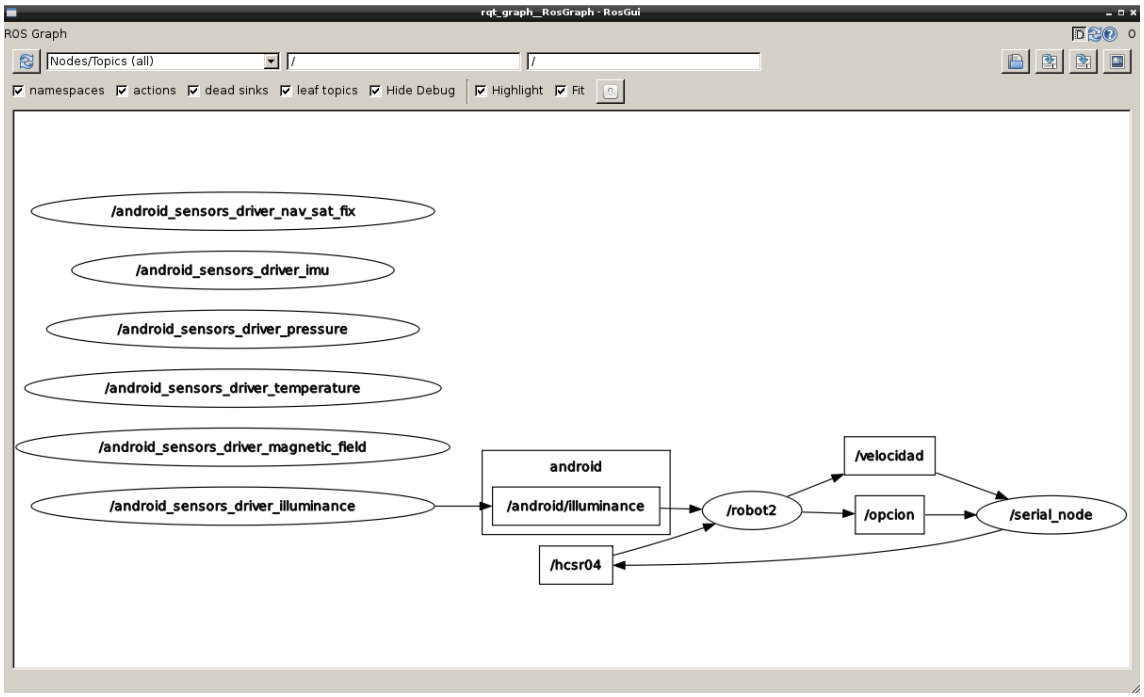


Figura 67 Esquema rqt_graph Aplicación 2

Se puede comprobar como los datos que se están publicando por el sensor de luz se presentan en el topic /android/illuminance al cual se suscribe el nodo "robot2". Este a su vez está publicando los topics "velocidad" y "opcion" a los cuales está suscrito el nodo Arduino (serial_node). Este último está publicando el topic "hcsr04" (además de los valores de los sensores meteorológicos) que contiene las distancias de los sensores de proximidad, al cual está suscrito el nodo "robot2".

8 Conclusiones y líneas de mejora

8.1 Conclusiones

La realización del proyecto llevaba asociada la misión de integrar las plataformas de propósito general Arduino Mega ADK y Raspberry Pi con ROS (Robot Operating System) la cual se ha cumplido de forma exitosa.

Las ventajas y alternativas que ofrece ROS a la hora de abordar el diseño y programación de aplicaciones robot son muy extensas pues al situarse como esa capa de abstracción entre las diferentes plataformas hace mucho más sencillo el establecimiento de conexión, el paso de mensajes y comunicación.

Por otro lado las carencias más importantes que presentaba el Raspberry Pi para la realización de este proyecto,

- Escasez de interfaces digitales.
- Escasez de interfaces analógicas

se solventan gracias a la adición de la placa Arduino Mega ADK dotando al sistema distribuido del hardware y funcionalidades necesarios.

El Raspberry Pi presenta limitaciones en sus especificaciones hardware que lo hacen inviable para llevar a cabo otras tareas más pesadas que requieran de cálculos más complejos y avanzados o de tratamiento de imágenes ya sea tanto en 2D como en 3D.

En cualquier caso la finalización de este proyecto, más que representar un final como tal, va mucho más allá y abre las puertas a unas más que interesantes líneas de investigación y desarrollo que podrían llegar a ser muy útiles de cara a interés personal pero sobre todo abre un sinfín de posibilidades de cara a su continuación y/o trabajo sobre las bases establecidas para la Universidad:

- Realización de laboratorios en el ámbito del control y programación de robots.
- Líneas de desarrollo y programación de aplicaciones robot.
- Investigación y desarrollo de aplicaciones más complejas y completas.
- Impartición y aprendizaje de nuevos conocimientos a nivel de software y hardware.

Finalmente se han desarrollado dos aplicaciones que han hecho uso de las herramientas y posibilidades que ofrecían los elementos integrantes culminando en un sistema distribuido y funcional.

8.2 Líneas futuras

Como se ha comentado anteriormente la finalización de este proyecto no supone un punto y final a nivel de desarrollo o funcionalidades.

El proyecto realizado se podría extender de diferentes formas:

1. En primer lugar y con el objetivo de obtener una plataforma más compacta que ofrezca viabilidad para su montaje y funcionamiento autónomo y una integración limpia con el hardware y plataformas utilizados, se podría diseñar una placa PCB mediante *Altium* que actúe como escudo (shield) para Arduino Mega ADK. Lo que se pretendería sería conseguir la integración limpia y clara del hardware que presente características de fiabilidad de las conexiones, estabilidad y robustez, escalabilidad del diseño e integración.
2. Mejora de las aplicaciones diseñadas depurando los códigos implementados para lograr una mayor depuración incluso integrarlos para dotar de una funcionalidad completa al sistema.
3. Añadir componentes extra que doten al sistema de funcionalidades extra fácilmente adaptables:
    a. Receptor GPS GP-635T integrado en el Weather Shield evitando así tener que usar la localización a través del Smartphone.
    b. Complemento *Wheater Meters*, que dotaría al sistema meteorológico de las funcionalidades de pluviómetro y anemómetro.
4. Investigación y desarrollo de un mejor sistema de localización, salvando así los errores en precisión del GPS.
5. Implementación de herramientas de navegación de ROS tales como Rviz, que permitiría la representación de obstáculos en un mapa autogenerado, introducción de datos de ubicación y planificación de rutas. Para ello se establece como requisito un escáner láser (encarecería el sistema) así como la configuración de las transformadas del robot y la recepción de datos odométricos.

8.2.1 Aplicaciones de los modelos

Aprovechando las herramientas que ROS facilita así como las aplicaciones diseñadas para este proyecto las posibles aplicaciones reales para el sistema podrían ser:

1. Estación meteorológica autónoma para grandes superficies agrarias dotadas de un sistema de cobertura WIFI.
2. Sistema de vigilancia autónomo gracias a la posibilidad de hacer streaming en todo momento.
3. Herramienta de medición de condiciones climatológicas de zonas de difícil acceso para el ser humano.

Las posibilidades terminarían donde la imaginación del desarrollador pudiera llevarle así como cuando se alcanzaran las limitaciones técnicas de los sistemas.

9 Bibliografía

1. ROS Powering the world robots
   http://www.ros.org/
2. Wiki ROS
   http://wiki.ros.org/
3. Questions – ROS Answers: Open Source O&A Forum
   http://answers.ros.org/questions/
4. Raspberry PI
   http://www.raspberrypi.org/
5. Arduino Home Page
   http://www.arduino.cc/
6. Sparkfun Electronics
   https://www.sparkfun.com/
   6.1. Weather Shield DEV-12081 RoHS
      https://www.sparkfun.com/products/12081
7. I2C bus Technical Overview and FAQ
   http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/i2c-bus.html
8. The C++ Resource Network
   http://www.cplusplus.com/
9. Github – Build software better, together
   https://github.com/