UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

ROBÓTICA MÓVIL. ESTUDIO Y CARACTERIZACIÓN DEL ROBOT MÓVIL KJUNIOR DESARROLLO DE APLICACIÓN DE ROBOT LABERINTO

Titulación:	Ingeniería Técnica Industrial esp. Electrónica Industrial
Autor:	Juan Antonio Gómez Cánovas
Director:	José Luis Muñoz Lozano
Departamento:	Ingeniería de Sistemas y Automática
Fecha:	Septiembre 2011

1. Capítulo I : Introducción. Objetivos del proyecto.

1.1 Antecedentes del PFC

Las clases de robótica con D. Juan López Coronado fueron mis primeros contactos técnicos con robots. Preguntándole acerca de desarrollar un proyecto de fin de carrera me propuso ponerme en contacto con D. José Luis Muñoz Lozano, profesor que anteriormente me había dado clases de Regulación Automática, y en las que me había formado una imagen muy positiva acerca de él para que fuera mi tutor del proyecto de fin de carrera. Inmediatamente me puso encima de la mesa varias propuestas. Un robot móvil sencillo fue mi elección. Desde entonces, dediqué gran parte de mi tiempo libre a conocerlo, hacerlo funcionar, buscar aplicaciones hecha para él, y desarrollar algunas simples por mi cuenta.

1.2 Motivaciones

Son muchas y muy variadas las razones por las cuales decidí que quería realizar un proyecto de fin de carrera enfocado en este ámbito. Una clasificación a priori sería en experiencias pasadas, ideas presentes y proyección de futuro.

Como experiencias pasadas, gran parte de la infancia de las generaciones cercanas a la mía (89), ha estado marcada por artefactos tales como el scalextric, los coches teledirigidos a radiocontrol, o robots como el robot Emilio. El mero hecho de comprender cómo funcionan estos artefactos era un argumento bastante válido para mí mismo.

En el presente, ¿quién no desea que las tareas repetitivas y tediosas las desarrolle “otro”? Ser capaz de comprender los robots aspiradora, para poder construir algo parecido, de bajo costo, me llamaba la atención. No se puede decir que fuera mi objetivo principal el desarrollar un amo de casa, pero es una idea idealista, al fin y al cabo, con todo lo que ello conlleva.

Con vistas al futuro, la ciencia robótica está claro que entrará en pleno auge, tanto en el ámbito industrial como en el de investigación. Inherente a esta premisa se encuentra la idea de querer formarme un poco más específicamente en el estudio y comprensión de los robots.

Por todo ello y más era atractiva la idea, así pues, empecé el viaje, sólo veía los 30 siguientes metros de la carretera, pero siempre había y habrá más camino.

1.3 Objetivos del PFC

Este proyecto de fin de carrera tiene por objeto el estudio de la robótica móvil, la caracterización del robot móvil K-Junior y la realización de alguna aplicación para dicho robot, así como facilitar los primeros contactos con el robot KJunior a otros estudiantes.

Otros objetivos para mí mismo han sido, entre otros: entender el funcionamiento de un robot y aplicar los conocimientos adquiridos a lo largo de la carrera en algo práctico; especialmente útiles me han sido las aportaciones de las asignaturas dedicadas a la programación, la electrónica digital, automatización, la instrumentación y el control.

Otra meta es la intención de formarme específicamente en este sector con altas probabilidades de éxito.

1.4 Estructura del documento

El proyecto está dividido en 5 capítulos:

1. Éste primero se dedica a la explicación a grandes rasgos de los antecedentes, motivaciones y objetivos.

2. En este capítulo se hace un repaso histórico de la evolución de la robótica, además contempla una visión del estado actual y futuro de la robótica. Se aborda especialmente el campo de la robótica móvil en el aspecto estructural y sensorial.

3. Contiene las especificaciones técnicas del robot K-Junior. Se da a conocer el funcionamiento del robot y su control. Se realiza un símil entre el robot K-Junior y un ser vivo. Contiene un manual sencillo para comenzar a usar el robot K-Junior y se comentan sus posibles accesorios

4. Se realiza la aplicación de resolución de laberintos con lógica tradicional, contemplando los errores y pérdidas de tiempo durante el camino, así como las buenas soluciones. Se enumeran distintos métodos de resolución de laberintos. Y se introducen dos conceptos que podrían ser útiles para futuros trabajos con K-Junior. La lógica borrosa y el método de interpolación “splines”.

5. Se exponen las conclusiones y algunas ideas para trabajos futuros.

6. Se incluyen la bibliografía referenciada y consultada

   Anexo: Programación en C

2. Capítulo II : Robótica y robots móviles

2.1 Introducción

La robótica es la rama de la ciencia que estudia el diseño y aplicaciones de los robots. Es una rama multidisciplinar al involucrar distintos tipos de conocimientos, como son la electrónica, mecánica e informática, además de hacer uso de otras disciplinas: inteligencia y visión artificial, e ingeniería de control. La unión y desarrollo conjunto de éstas ha dado lugar a lo que hoy día conocemos como robots: creaciones electro-mecánicas que realizan tareas repetitivas, peligrosas, o simplemente imposibles para el ser humano,

En esta primera parte del proyecto de fin de carrera se contempla cómo ha nacido y crecido la robótica, cuál es el estado actual de ésta en la industria y la sociedad, además de realizar una previsión de lo que puede venir en un futuro.

A continuación se darán a conocer las distintas posibles clasificaciones de los robots, según determinados criterios y se profundizará en los sistemas robóticos móviles, pues en este grupo se encuentra el robot K-Junior, en el cual se encuentra enfocado este proyecto de fin de carrera. Se analizarán los distintos sistemas de locomoción, diferenciando unos de otros según complejidad y el uso al que esté destinado el robot.

2.2 Historia, presente y futuro

El afán por la creación de mecanismos imitadores de los del ser humano se remonta varios siglos atrás. Tanto egipcios como griegos construyeron estatuas con sistemas mecánicos e hidráulicos que imitaban algunos movimientos (como alzar el brazo) del ser humano.

Jacques de Vauncanson, en el siglo XVIII, construyó varios músicos mecánicos a tamaño real, como el tocador de flauta travesera y el tamborilero (Figura 1.1 Robot músico), además de un pato autómata, capaz de digerir grano, batir las alas, beber agua y defecar, con más de 400 partes móviles. Es destacable también su invención de un telar automático (programable con tarjetas perforadas), además de una bomba hidráulica.

Figura 1.1 Robot músico

Durante la revolución industrial el ingenio llegó a un punto clave, máximamente para la industria textil, la cual fue una de las más beneficiadas al inventarse: hiladora giratoria de Hargreaves

(1770), la hiladora mecánica de Crompton (1779)), el telar mecánico de Cartwright (1785), el telar de Jacquard (1801), y otros.

A principios del siglo XIX, Henri Maillardert construyó una muñeca mecánica (Figura 1.2 Muñeca mecánica) capaz de dibujar. El mecanismo principal eran unas levas cuya silueta era la programación del autómata.

Figura 1.2 Muñeca mecánica

Todavía no había nacido el término robot. Sus orígenes se remontan a una obra de teatro del dramaturgo checo Karel Capek "Los Robots Universales de Rossum"de 1920, en la cual, el protagonista, Rossum, fabrica y mejora un robot, éste luego se vuelve contra el humano. Son muchas las historias y películas que hacen del robot un enemigo a largo plazo de la humanidad. La palabra checa "robota" significa servidumbre o trabajador forzado, y cuando se tradujo al inglés se convirtió en el término robot. (Figura 1.3 Obras literarias sobre robots)

Isaac Asimov fue uno de los escritores de ciencia ficción en cuyas obras aparecían robots. En sus obras la imagen de robot es la de una máquina bien diseñada y con una seguridad que actúa según los tres principios, denominados por Asimov como "Las tres leyes de la Robótica":

1. Un robot no puede actuar contra un ser humano o, mediante la inacción, que un ser humano sufra daños.

2. Un robot debe de obedecer las órdenes dadas por los seres humanos, salvo que estén en conflictos con la primera ley.

3. Un robot debe proteger su propia existencia, a no ser que esté en conflicto con las dos primeras leyes

Así, la imagen de los robots que nos ofrece Asimov contradice la desconfiada versión de los robots de Capek.

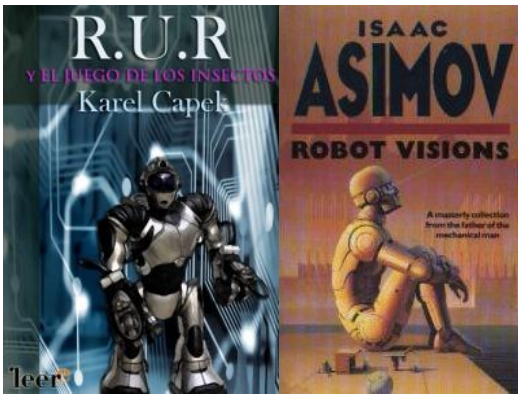


Figura 1.3 Obras literarias sobre robots

Ya en el siglo XX, gracias al desarrollo de la tecnología, en la que se incluyen: tecnología computacional, actuadores de control realimentados, transmisión de potencia a través de engranajes y sobre todo la tecnología en sensores, se han desarrollado robots industriales capaces de reducir notablemente los costos de producción en serie, con múltiples aplicaciones y elevada flexibilidad. (Figura 1.4 Robot moderno)

Figura 1.4 Robot moderno

Hasta este momento los robots de uso no industrial sólo han estado presentes en novelas de ficción, sin embargo, en la actualidad (siglo XXI) ya se han comenzado a comercializar los primeros robots de uso doméstico, como son las aspiradoras automáticas, clasificadas en el grupo de los robots móviles, grupo en el cuál nos centraremos en este PFC. Además, también se está desarrollando la posibilidad de suplir carencias del cuerpo humano, como mutaciones genéticas o derivadas de accidentes, con articulaciones robotizadas (Figura 1.5 Piernas robóticas). Lo más impresionante de

esta tecnología, es que con los impulsos eléctricos producidos por orden de nuestro cerebro, es posible controlar la articulación robótica, por otro lado, es necesaria una intervención para "soldar" nuestros nervios al microcontrolador de la articulación.

Figura 1.5 Piernas robóticas

Otra posibilidad de la robótica en el campo de la biomedicina, es la denominada nanorobótica, o fármacos inteligentes (Figura 1.6 Nanorobótica médica). Se trata de la creación de robots de pequeñísimo tamaño, programados para la inhibición o destrucción de determinadas células/moléculas, pudiendo así destruir células cancerígenas, o distintas afecciones por virus sin dañar otras células como hacen los fármacos actuales. Es una posibilidad que podría revolucionar el mundo de la medicina.

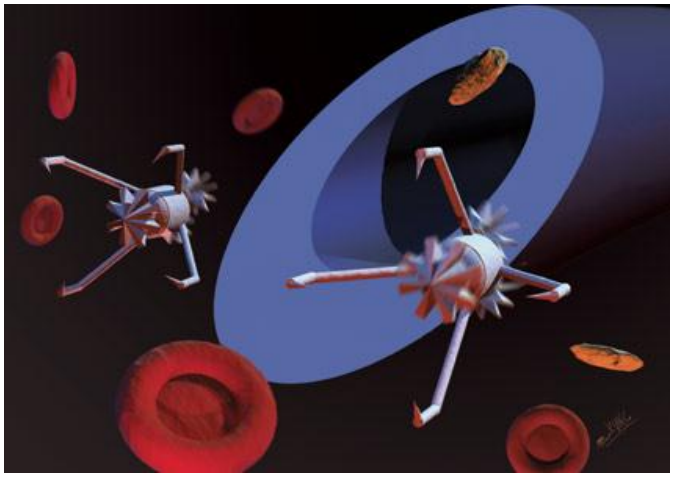


Figura 1.6 Nanorobótica médica

El limitante de un ser humano es la fuerza, el de una máquina, es la inteligencia. La unión de ambos es la cibernética. Cuando las piernas robotizadas estén más desarrolladas, dejará de ser una máquina para alguien que no puede mover las piernas y pasará a ser una herramienta como la bicicleta, la moto, o el coche, que todo el mundo usará. Esta idea tiene connotaciones positivas y/o negativas según sea la interpretación que le demos en un contexto concreto. (Figura 1.7 Cibernética)

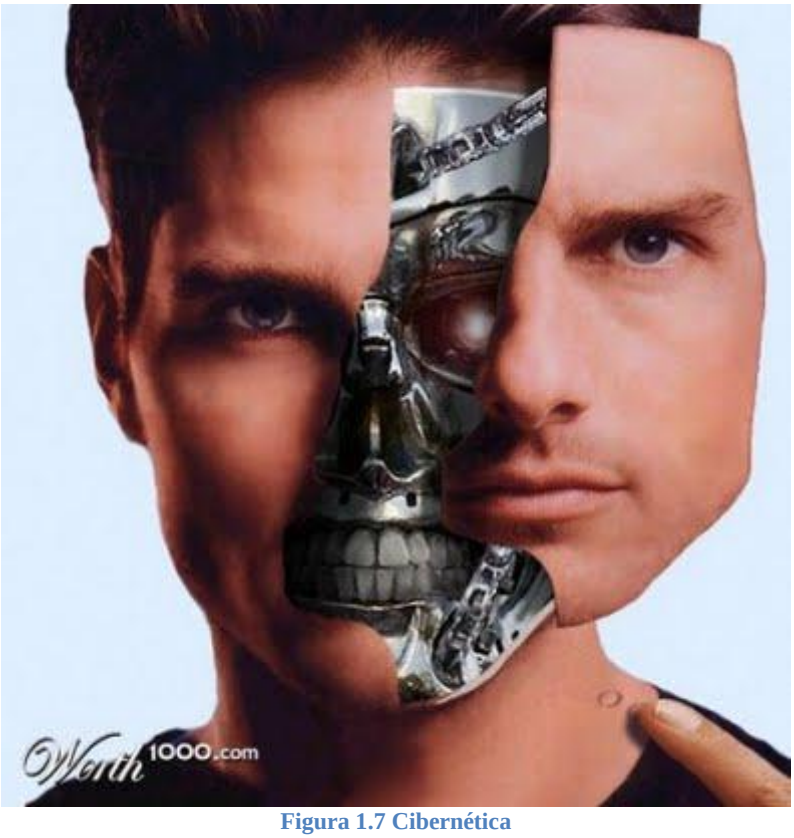


Figura 1.7 Cibernética

Lo que está claro es que la robótica está en plena juventud y le queda mucho por crecer, a pesar todo lo avanzado. Es de preveer que el robot tal y como lo veía Asimov, o como se podía ver en la serie de dibujos animados "Los supersónicos", sea una realidad en un futuro no muy lejano. Acabando así con muchas de las tareas típicas de el/la "amo/a de casa". (Figuras 1.8 Robot de los supersónicos y 1.9 MayorRobot)

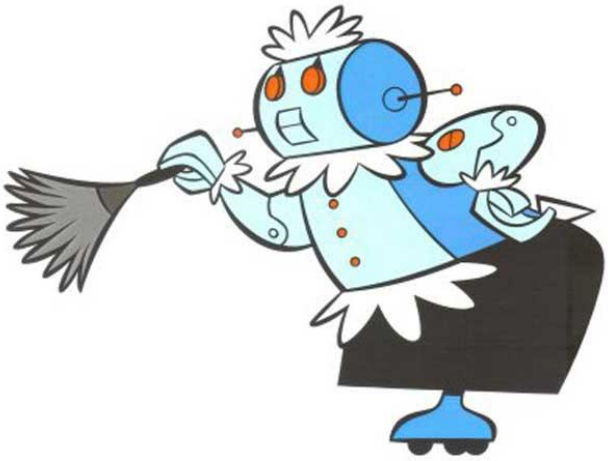


Figura 1.8 Robot de los supersónicos

Figura 1.9 MayorRobot

Ante dicho auge de esta disciplina, no está claro dónde está el techo. Así, cada día, aparecen nuevos robots sorprendentes, subacuáticos, voladores, luchadores, etc. Uno de los que a mí más me ha llamado la atención estos últimos días ha sido el "robot portero", inventado por el alemán Thomas Pfeifer: se trata de un guardameta, al cuál es francamente difícil, marcarle un gol. Consta de una cámara de alta velocidad de captura (más de 30 fps) gracias a la cual, en milésimas de segundo, unos pocos ciclos del microcontrolador, es capaz de conocer la trayectoria del balón y girar un ángulo concreto en función de ésta. Para un humano es imposible darle la suficiente velocidad a la pelota para que el robot no la detecte. Este es un claro ejemplo de lo que es capaz de hacer un robot. (Figura 1.10 Guardameta robótico)

Figura 1.10 Guardameta robótico

2.3 Clasificación

2.3.1 Clasificación en generaciones

La clasificación más usual y aceptada universalmente es la realizada por Michael Cancel, director del Centro de Aplicaciones Robóticas de Science Application Inc, en 1984, la cuál divide a los robots en cinco generaciones:

- 1ª generación: Manipuladores

Se trata de robots manipuladores mecánicos de sencillo control. Se basan, generalmente, en sistemas de levas (Figura 2.1 Leva), las cuales son unas piezas capaces de transformar un movimiento rotativo en una traslación, de acuerdo al perfil de dicha leva:

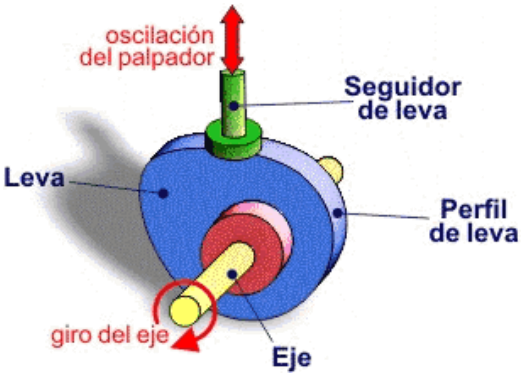


Figura 2.1 Leva

- 2ª generación: Repetitivos

Robots capaces de memorizar y repetir un mismo proceso. La primera "memoria programada" fueron las tarjetas perforadas para el telar automático de Jacquard (Figura 2.2). Se

puede tomar éste como el nacimiento de la programación. Según la distribución de los orificios sobre la placa perforada, la máquina realizaba distintas operaciones sobre la materia prima.

Figura 2.2 Telar automático de Jacquard

- 3ª generación: Sensorializados

Contienen sensores, de manera que procesando la señal de éstos en un computador, se obtienen a unas órdenes de control a aplicar sobre los actuadores. El desarrollo de la instrumentación electrónica (tecnología en sensores) ha impulsado el crecimiento de la robótica. Gracias a éstos los robots pueden captar información de su entorno y procesarla así como nosotros hacemos gracias al olfato, gusto, oído, vista o tacto.

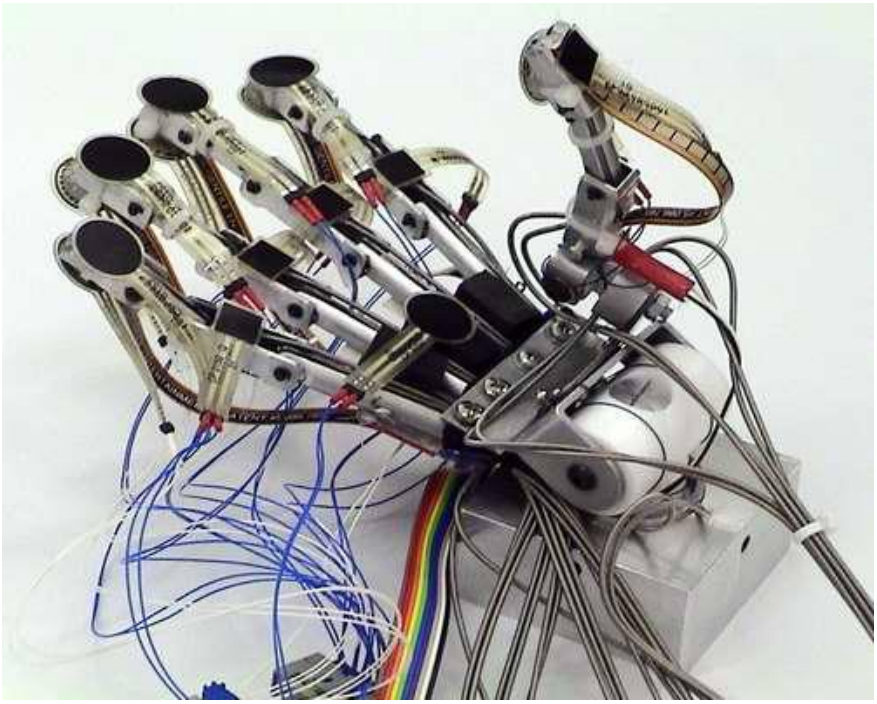


Figura 2.3 Articulación sensible

- 4ª generación: Visión artificial

Se añaden cámaras a los robots de manera que aumentan sus posibilidades tanto como se desarrolle el procesado digital de la imagen. Con una cámara es posible la detección de objetos

en un plano 2D, sin embargo en la actualidad se está desarrollando la tecnología 3D que se basa en la adquisición de una tercera dimensión (la profundidad) gracias a una o más cámaras extra. La visión estereoscópica (Figura 2.4) en los robots, con dos cámaras, intentan emular la de los humanos, no obstante no es un proceso sencillo ya que intervienen muchos factores complejos como convergencia y enfoque.

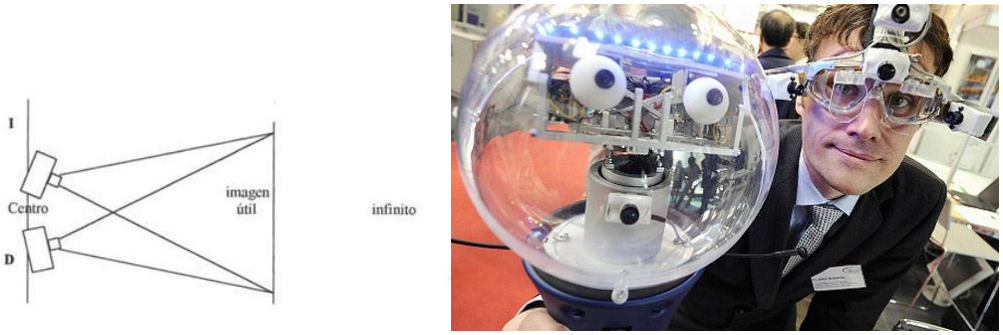


Figura 2.4 Visión estereoscópica

- 5ª generación: Inteligencia artificial

Esta es la generación más joven y la que más camino tiene por delante. Se trata de proporcionar los medios a un computador para que posea la capacidad de los humanos de analizar la situación y responder de la mejor manera posible ante situaciones no previstas, además de aprender de éstas. El principal propósito de técnicos, investigadores y expertos en el campo de la inteligencia artificial es llegar a desarrollar un robot que aprenda por sí sólo, con el tiempo, así como hace un niño. Sin embargo, es obvio que no es tarea sencilla, pues nosotros estamos preprogramados en la genética para movernos por instintos y necesidades intrínsecas al ser humano. Sin embargo un robot no tiene necesidades, sólo la programación que lo dirige, éste es uno de los escollos al que se enfrentan los investigadores en esta rama.

Figura 2.5 Inteligencia artificial

2.3.2 Clasificación según su estructura

Otra posible clasificación se puede hacer según la estructura y configuración de sus elementos:

❖ Poliarticulados:

Están estructurados para mover sus elementos en un determinado espacio de trabajo, según uno o varios sistemas de referencia y con un número limitado de grados de libertad, y son generalmente para uso sedentario. Ej: Manipuladores, robots para soldadura, etc. Son los más usados en muchos tipos de industria, debido a su amplia gama de posibilidades según la herramienta usada. Se suelen usar para tareas repetitivas, pues, aunque sea necesaria una inversión inicial fuerte, debido al costo de los equipos y del personal cualificado para su programación y puesta en marcha, se rentabiliza al reducir considerablemente los costes de producción y por lo tanto, la eficiencia de la instalación. También son capaces de soportar condiciones en las que un trabajador humano corre riesgos. (Figura 3.1 Robot articulado)

Figura 3.1 Robot articulado

❖ Móviles:

Capacidad de desplazamiento, gracias a su sistema locomotor, que pueden ser ruedas, cadenas o "patas" articuladas. Siguen su camino por telemando o guiándose por la información recibida de su entorno a través de sus sensores. Estos Robots son capaces de transportar piezas de un punto a otro de una cadena de fabricación. Guiados mediante pistas materializadas a través de la radiación electromagnética de circuitos empotrados en el suelo, o a través de bandas detectadas fotoeléctricamente, pueden incluso llegar a sortear obstáculos y están dotados de un nivel relativamente elevado de inteligencia. En este grupo encontramos nuestro pequeño robot KJunior. (Figura 3.2 Robot móvil aspiradora)

Figura 3.2 Robot móvil aspiradora

- Androides:

Emulan en forma y/o comportamiento al ser humano. No se encuentran extremadamente desarrollados debido al problema de la locomoción bípeda y la conservación del equilibrio.

Debo de mencionar los robot P3 y ASIMO (Figura 3.3 Robot ASIMO) de Honda, pasado y presente de la robótica bípeda, capaces de caminar, subir y bajar escaleras, correr, etc. [5]

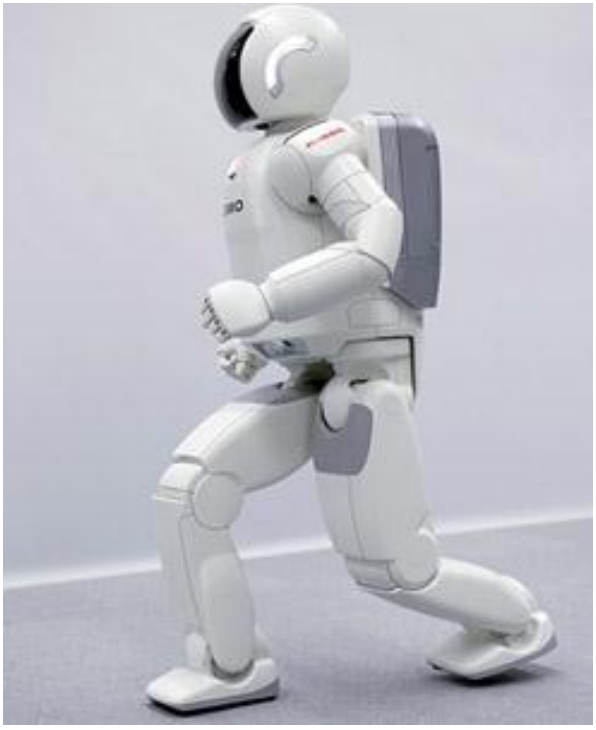


Figura 3.3 Robot ASIMO

❖ Zoomórficos:

Emulan a diversos seres vivos en la estructura de su sistemas de locomoción, es decir, usan, para el movimiento, o bien sistemas de patas articuladas de manera que el robot puede avanzar en caminos accidentados o poco homogéneos, o bien otras configuraciones basadas en la evolución natural de las especies, como peces, pájaros o reptiles. (Figura 3.4 Robot zoomórfico)

Figura 3.4 Robot zoomórfico

❖ Híbridos:

Los robots con características mixtas pueden ser denominados híbridos. Los robots denominados metamórficos, son capaces de cambiar su forma/estructura de acuerdo con los requerimientos de la situación, del terreno, etc. Éstos robots están en la punta de la lanza de la flexibilidad y capacidades para la investigación y desarrollo, debido a la capacidad de moverse en muchos tipos distintos de terreno. (Figura 3.5 Robot híbrido)

Figura 3.5 Robot híbrido

2.3.3 Clasificación según el nivel del lenguaje de programación

Otras posible clasificacion las encontramos en cuanto al nivel del lenguaje de programación:

- Sistemas guiados:

El usuario conduce al robot a través de los movimientos a ser realizados. Un simple ejemplo sería un coche radio-control de juguete.

- Sistemas de programación de nivel robot:

En los que el usuario escribe un programa de computador al especificar el sensado y el movimiento. Por ejemplo, nuestro querido robot K-Junior.

- Sistemas de programación de nivel tarea:

En este caso, el usuario especifica la operación a realizar sobre los objetos que el robot manipula. Un ejemplo de este caso puede ser un autómata en modo manual: girar la plataforma 90º, elevarla 90 centímetros, etc.

2.4 Sistemas de un robot móvil

Un robot móvil consta principalmente de un sistema de locomoción para el movimiento, un sistema de sensores para la interactuación con el exterior, y uno o varios sistemas de control que determinan el comportamiento del robot en función a la información provista por los sensores.

2.4.1 Sistemas de locomoción

Existe una gran variedad de maneras de moverse sobre una superficie; entre los robots móviles, las más comunes son las ruedas, las cadenas y las patas.

En su elección, los factores más importantes son: el uso al cual está destinado el robot, el terreno en el que debe de moverse y el nivel de prestaciones que deseemos relacionadas con la inversión que se pretenda hacer.

2.4.1.1 Configuraciones a ruedas

El sistema de locomoción más utilizado es el basado en ruedas. Los robots con ruedas son más sencillos y más fáciles de construir, la carga que pueden transportar es mayor, relativamente. Los sistemas basados en cadenas o en patas, son más complejos de diseñar y, generalmente, más

pesados por utilizar más cantidad de piezas. Sin embargo, en un terreno irregular, las ruedas pueden no resultar tan eficientes como los sistemas de patas o cadenas. En estos casos de terreno irregular, el control de posición basado en sensores “simples” se hace muy complicado debido a la dificultad para medir correctamente el avance, se hace necesaria la incorporación de sistemas fiables tales como GPS.

Existen varios diseños de ruedas para elegir cuando se quiere construir una plataforma móvil sobre ruedas: diferencial, sincronizada, triciclo y la Ackerman.

2.4.1.1.1 Configuración diferencial

El sistema más simple en muchos aspectos. Dos ruedas motrices colocadas de manera simétrica permiten al robot avanzar o retroceder, girar sobre sí mismo, o seguir una trayectoria parabólica. Tiene una gran ventaja respecto a otros sistemas como triciclo o coche: su posición es independiente de su orientación, lo que supone una gran ventaja en determinadas aplicaciones.

El principal problema es el equilibrio del robot, y para resolverlo, se utilizan uno o dos puntos de apoyo adicionales, en un diseño triangular o romboidal. En el caso de KJunior, tenemos un sistema romboidal con diseño diferencial.

Otro aspecto a tener en cuenta es que la superficie ha de ser homogénea, de lo contrario, la misma tensión aplicada a ambos motores, puede no suponer iguales velocidades de rotación y provocar el giro indeseado, es por ello que debe existir un lazo de control de velocidad para solventar este inconveniente. En nuestro robot KJunior no tenemos dicha posibilidad, debido a que no está equipado de encoders o similares.

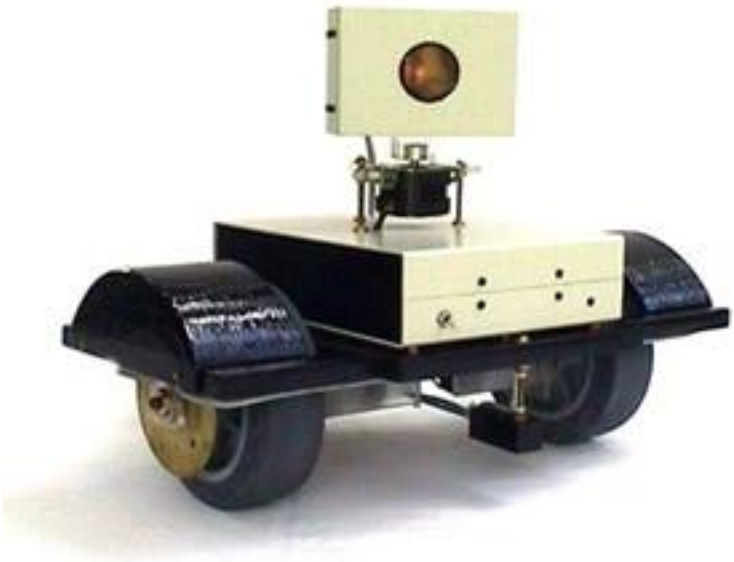


Figura 4.1 Configuración diferencial

2.4.1.1.2 Configuración sincronizada

Complejo mecánicamente pero eficiente en las prestaciones. Este sistema se basa en que el chasis del robot no cambia de dirección al girar. Las ruedas, motrices y directrices simultáneamente, giran en concordancia sobre un mismo eje vertical para el giro del robot, manteniendo la dirección del chasis. Este hecho en robots asimétricos puede suponer un problema en cuánto a la distribución de los sensores, problema solventable con un sistema de control del ángulo formado entre las direcciones de chasis y ruedas. Al igual que el diseño diferencial, su orientación es independiente de la posición.

Figura 4.2 Configuración sincronizada

2.4.1.1.3 Configuración de moto

No se suele usar en robots, debido a que en parado no mantiene el equilibrio, pero creía oportuno nombrar este sistema. Se trata de dos ruedas coplanares, una motriz y otra directriz. En movimiento se mantiene el equilibrio debido al efecto giroscópico de las ruedas: la fuerza centrípeta (inercia) de cada uno de los puntos de la rueda hacen despreciable la fuerza que se ejerce sobre el centro de gravedad hacia uno u otro lado, según la distribución de pesos horizontal.

Sin embargo, tiene el problema de que en parado no es estable, por lo que se precisa de un sistema que evite este problema si se quiere construir un robot con esta forma y diseño. Es por esto que no se suelen ver vehículos automatizados en forma de motocicleta.

2.4.1.1.4 Configuración de triciclo

Relativamente simple. Este sistema se basa en un par de ruedas motrices y una única rueda directriz. El diseño generalmente es triangular. No es necesario el control de la velocidad debido a que la rueda directriz es la responsable de la dirección del movimiento y será la dirección de ésta y no la diferencia entre las velocidades de rotación de las ruedas motrices la que marque la trayectoria.

Otro diseño en triciclo que últimamente está llegando a las “moto-triciclo” de calle es mucho más complejo pero más estable que el típico en los vehículos de dos ruedas. Se trata de la inclusión

de una segunda rueda directriz, acoplada de tal manera que la inclinación que se produce en el giro es coherente en las dos ruedas. Es decir, se tiene una rueda motriz trasera y dos ruedas directrices delanteras. En los robots este sistema apenas llama la atención, pues no supone ventaja alguna respecto de otros sistemas más simples, sin embargo y al igual que el diseño de dos ruedas, me parecía conveniente, por lo menos, citarlo.

Figura 4.3 Configuración de triciclo

2.4.1.1.5 Configuración Ackerman

Complejo y estable. Es el sistema más ampliamente utilizado en automoción. Se basa en un diseño con cuatro ruedas, dos traseras y dos delanteras. Los diseños más extendidos son:

a) con las ruedas traseras motrices y las delanteras directrices:

b) con las ruedas delanteras motrices y directrices.

c) ambos pares de ruedas directrices y tracción delantera, trasera o 4x4

Existe la complejidad de la unión entre ambas ruedas directrices. En este sistema es frecuente el uso de suspensiones para mayor estabilidad y evitar daños por desperfectos del terreno, sin embargo se pueden usar éstas en cualquier diseño.

Este diseño, al igual que el de triciclo, tiene la desventaja de que su orientación está ligada a la posición, es decir, no puede orientarse en una determinada dirección sin avanzar o retroceder, es decir, sin variar su posición, lo que supone un inconveniente en según qué situaciones.

Figura 4.4 Configuración Ackerman

2.4.1.2 Configuraciones articuladas

El diseño de un sistema de locomoción a "patas" supone un auténtico reto de ingenio, debido al elevado número de grados de libertad y a la mayor complejidad mecánica y de los algoritmos de control. El desarrollo de estos sistemas es muy atractivo para la exploración espacial, o de lugares de difícil acceso para el humano tales como volcanes, rifts (grietas entre dos placas terrestres), etc

2.4.1.2.1 Configuración bípeda

Uno de los diseños más populares son los denominados "humanoides" con dos patas con varias articulaciones, de manera que es capaz de emular el caminar del humano. Tiene el inconveniente del equilibrio. En el ser humano, en el oído interno, existen unos "receptores" del equilibrio que son capaces de detectar cambios de orientación de la cabeza en las 3 dimensiones. Se puede decir que el ser humano está provisto de acelerómetros biológicos. Este inconveniente ha sido resuelto de distintas maneras, sin embargo, no se ha conseguido que un robot realice la actividad de correr como un ser humano, debido a que nosotros somos capaces de mantener el equilibrio sin

ninguno de los pies en contacto con el suelo. Un robot bípedo ha de tener al menos uno de los dos pies en el suelo, así como un corredor de marcha.

Figura 4.5 Configuración bípeda

2.4.1.2.2 Configuración de insecto

Se emplea un número par de patas, generalmente con diseño simétrico, es capaz de avanzar perfectamente en terrenos muy irregulares. Por lo general suelen tener 6 u 8 patas.

El avance del robot se basa en el movimiento de parejas de servos: el servo A levanta la pata, el servo B adelanta la posición relativa de la pata respecto del cuerpo, el servo A baja la pata y se reafirma en el suelo, en este momento el servo B vuelve a su posición inicial, haciendo avanzar el cuerpo. Para que el movimiento sea efectivo no debe haber incongruencias en la coordinación de los distintos movimientos de las diferentes patas.

Figura 4.6 Configuración de Insecto

2.4.1.2.3 Configuración zoomórfica

Se diseñan con la forma de algún animal superior. Son muchos los ingenieros/científicos que piensan: "si algo ya ha sido pensado y desarrollado durante millones de años por la naturaleza, usemos ese conocimiento para el diseño de nuestros sistemas de locomoción artificiales". Como ejemplo podemos citar el famoso pato de Vaucanson (Figura 4.7), o robots más modernos como perros (Figura 4.8), tiburones, pájaros, etc

Figura 4.7 Pato de Vaucanson

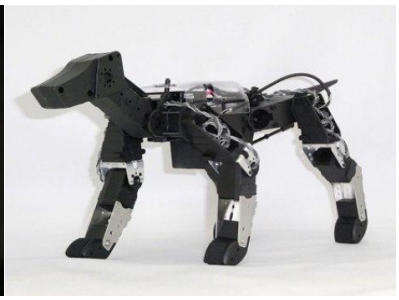


Figura 4.8 Configuración zoomórfica

2.4.1.3 Otras configuraciones

2.4.1.3.1 Configuración a cadenas

Es pesada, costosa y compleja. Sin embargo en condiciones adversas de irregularidad del terreno son más eficientes que las configuraciones a ruedas. Se le llama comúnmente configuración de "oruga". Se basa en rodear todas y cada una de las llantas con una misma cadena por cada lado.

Figura 4.9 Configuración a cadenas

2.4.1.3.2 Robots subacuáticos

Son robots cuya estructura les permite nadar o moverse en el agua. Son muchas las ventajas de estos robots en su ámbito. Sitios hasta ahora inaccesibles ahora lo son, además permiten realizar tareas peligrosas para los submarinistas.

Figura 4.10 Robot subacuático

2.4.1.3.3 Robots aéreos

Ante sitios inaccesibles por tierra o mar, y para muchos propósitos distintos, un robot teleoperado o inteligente , capaz de volar, es la mejor o única solución. En casos de emergencia: incendios o catástrofes de cualquier índole, son el recurso usado cada vez más frecuentemente.

Figura 4.11 Robot aéreo

2.4.1.3.4 Robots espaciales

Para la exploración espacial de otros planetas o satélites. Son empleados estos robots.

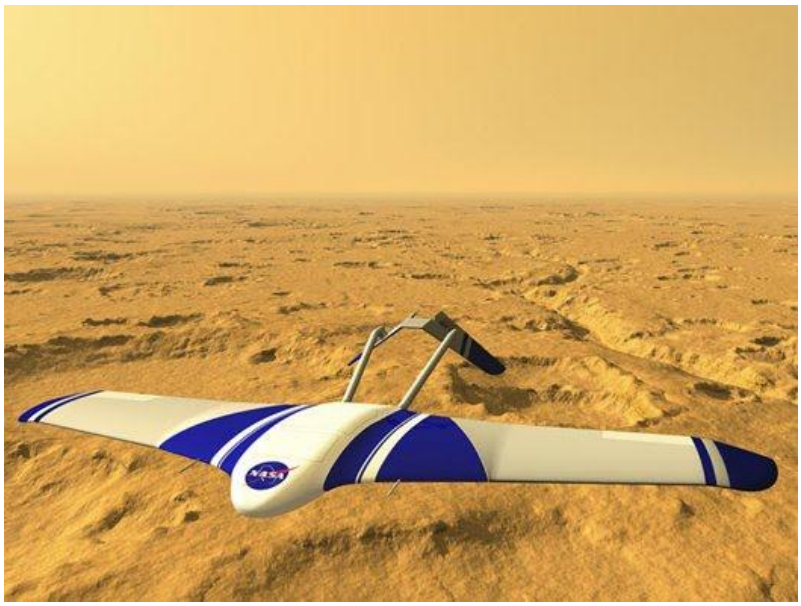


Figura 4.12 Robot espacial

2.4.2 Sistemas sensoriales

Son muchos y variados los tipos de sensores utilizados en la industria robótica. Se pueden clasificar de muchas maneras, pero una de las más comunes es:

• Sensores propioceptivos y exteroceptivos: es la clasificación más utilizada en

robótica móvil. La propiocepción se refiere a la percepción del estado interno del robot; por ejemplo, medidas de carga de baterías, posición del robot, etc. La exterocepción se refiere a la percepción de aspectos externos al robot; por ejemplo, temperatura, presión, localización de objetos. [1]

2.4.2.1 Sensores de ultrasonido

Los sensores de ultrasonido son una tecnología de medida activa en donde se emite una señal ultrasónica en forma de pulso, para posteriormente recibir el reflejo de la misma o eco. Se pueden explotar diferentes aspectos de la señal reflejada: el tiempo de vuelo o la atenuación. Su principal aplicación es la medida de distancias.

El ultrasonido es producido por una tensión sobre un material piezoeléctrico. El diseño del emisor es tal que las ondas sonoras son propagadas en una forma cónica de ángulo estrecho, de tal manera que se obtiene un margen de medida, con una mínima y máxima distancia de detección.

Los sensores de ultrasonido están formados por una cápsula emisora y otra receptora situada al lado de la emisora o bien por un transductor que actúa de emisor y receptor (Figura 5.1). En los robots móviles se suelen montar en la periferia, de forma que los sensores se encuentren separados a intervalos uniformes a lo largo del contorno del robot. Esto se hace así porque estos sensores son baratos. Una estrategia alternativa es colocar un sensor montado en una plataforma rotatoria, obteniendo así omnidireccionalidad.

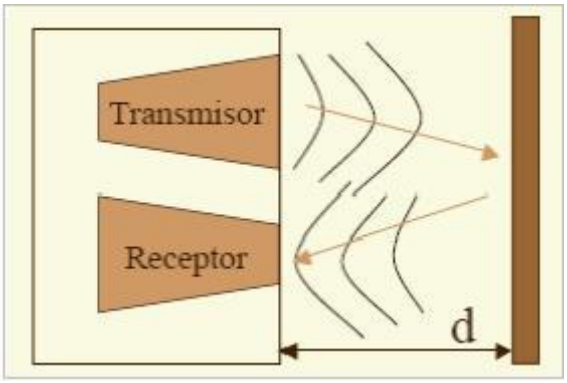


Figura 5.1 Transmisor- Receptor

La forma estándar de usar un sensor ultrasónico es dar un impulso corto, pero de gran voltaje y a alta frecuencia, a la cápsula emisora para producir una onda ultrasónica. Si la onda ultrasónica viaja directamente contra un obstáculo, rebota, y vuelve directamente hasta el receptor. La distancia que hay entre el sensor y el objeto es la mitad de la distancia que ha recorrido la señal y se calcula:

$$d = c*t/2$$

Donde d es la distancia al objeto, c es la velocidad del sonido en el aire, y t es el tiempo que tarda la señal desde que se emite hasta que se recibe. La velocidad del sonido depende de la temperatura y se calcula aproximadamente:

E.1 $C = C0 + 0.6*T$ (m/s)

Donde T viene dado en grados Celsius y C0 es 331 m/s.

Este tipo de sensores tiene unas limitaciones que son necesarias conocer a la hora de poderlos aplicarlos en un robot móvil:

• La velocidad del sonido es variable: la ecuación E.1 es una aproximación, ya que la velocidad del sonido también viene afectada por la densidad del aire, la humedad y la concentración de polvo en el aire. En entornos típicos la velocidad de la luz puede variar un 2% de un lugar a otro.

• El tiempo en blanco (blanking time): es necesario guardar un pequeño tiempo desde que se emite hasta que se prepara al receptor para recibir, con el objetivo de que no se vea influenciado por la onda que sale del emisor. Este tiempo se denomina tiempo en blanco, y generalmente limita la medida mínima de distancia de 3 a 15 cm dependiendo de la potencia de la señal emitida.

• La atenuación: la onda ultrasónica que sale del emisor se va dispersando y atenuando según avanza en el medio, de manera que cuanto más tarda el eco en llegar al receptor, más débil se espera que sea. Otro factor que influye en la amplitud del eco es la superficie del objeto. Si se calcula la distancia a un objeto por el tiempo de vuelo de la señal, esta limitación no influye más que en la distancia máxima que se quiere llegar a medir.

• El ángulo de medida (field of view). La señal emitida tiene un perfil de amplitudes complejo. Por ello, un eco que vuelve no necesariamente permite calcular de forma precisa la localización del objeto usando simplemente el tiempo de vuelo de la señal, ya que el eco puede provenir de cualquier lugar del espacio que se encuentre en el frente de onda ultrasónico. La frecuencia de la onda que se emite suele ser de unos 40-50kHz. Cuanto mayor sea la frecuencia más direccional es la onda, pero se atenúa más.

• Las reflexiones: son el impedimento más serio para poder detectar la posición de un objeto a partir del eco que se recibe. A frecuencias ultrasónicas, la superficie de la mayoría de los objetos reflejan la onda, de manera que una onda acústica que incida de forma muy oblicua en un objeto, se refleja en una dirección diferente a la de retorno hacia el receptor. Si se recibe un eco, no se puede asegurar que el eco sea un resultado de una serie de complejas reflexiones por el entorno.

• Cross-Talk: se produce si el eco generado por un sensor es recibido por otro que en ese momento está midiendo. Este problema se evita dejando un tiempo entre las medidas de dos sensores de ultrasonido diferentes.

Si se analiza la potencia de la onda emitida, cuando mayor sea mayores distancias se pueden medir, más problemas existen con las reflexiones y la distancia mínima de medida 10 es mayor. Si la potencia de la onda emitida es pequeña, se reducen las reflexiones, la distancia máxima de medida y la distancia mínima de medida.

De todas las restricciones comentadas, las más importantes son la variación de la velocidad del sonido, las reflexiones y el ángulo de medida. Por ello a lo largo de los años los investigadores han intentado dar solución a estos problemas. Las reflexiones así como el ángulo de medida del sensor generalmente se pueden tratar como ruido; por tanto son propiedades de un sensor que se pueden modelar.

Cuando se trata de modelar el ángulo de medida del sensor, suponiendo que no hay reflexiones, se puede usar un filtro de Kalman o mínimos cuadrados. El problema es que en un entorno convencional existen muchas reflexiones ya que está formado por varias superficies. [1]

2.4.2.2 Sensores infrarrojos.

Se incluyen en esta sección los sensores de infrarrojo y láser. A través de estos sensores se pueden estimar las distancias a las que se encuentran los objetos en el entorno. Hay diferentes métodos para medir la distancia a un objeto:

• Triangulación: usa relaciones geométricas entre el rayo de salida, el de entrada y la posición del sensor.

• Tiempo de vuelo: Mide el tiempo que transcurre desde que sale el rayo de luz hasta que se recibe, después de haber rebotado en un objeto. La precisión que se obtiene con estos sensores es muy elevada, debido a que son muy direccionales al ser muy pequeña su longitud de onda. La distancia máxima de medida depende de la potencia que se aplica al rayo de salida.

Se suele montar un solo láser en una plataforma móvil (pan-tilt), o con un espejo móvil que permita direccionar la señal a diferentes zonas del entorno, debido a que son caros. Mientras que los sensores de infrarrojo se suelen montar de forma similar a los sensores ultrasónicos.

Las ventajas de este tipo de sensores se pueden resumir en:

• El láser puede generar un millón de medidas en un segundo, con una precisión de milímetros en medidas de 30 metros.

• Son sensores ideales para medidas de profundidad, ya que el ángulo de medida es infinitesimal en un láser y muy pequeño en los sensores de infrarrojo.

Por otro lado existen un conjunto de desventajas muy importantes:

• En el caso del láser el precio es muy elevado (3000-10000 euros).

• El consumo del láser es elevado para llegar a obtener medidas de 30m.

• Al ser tan direccionales no detectan obstáculos ni por encima, ni por debajo del plano horizontal de medida, por tanto no son buenos para evitar obstáculos.

• En el caso de los sensores de infrarrojo las medidas de profundidad son muy limitadas (< 80cm).

2.4.2.3 Cámaras de visión artificial

Se refiere al procesamiento de datos de cualquier tipo que use el espectro electromagnético que produce la imagen. Una imagen implica múltiples lecturas dispuestas en una rejilla o matriz de dos dimensiones.

Se pueden distinguir al menos dos tipos de cámaras, las que trabajan en el espectro visible y las que trabajan en el espectro infrarrojo.

Cada vez se usan más ambos tipos de cámaras en robótica debido a que los precios bajan y el tamaño disminuye, y porque a veces es necesario conocer la textura y el color de algunos objetos.

Se puede considerar que la visión artificial es un campo de estudio separado de los demás en robótica. En este campo se han desarrollado algoritmos para filtrar ruido, compensar problemas de iluminación, encontrar líneas, emparejar líneas con modelos, extraer formas y construir representaciones tridimensionales. Los robots reactivos no suelen utilizar este tipo de algoritmos porque requieren de gran cantidad de cálculos.

Las cámaras de vídeo tienen como ventajas:

• Ofrecen mucha información en cada imagen.

• Son el sensor ideal para detectar colores, texturas y para reconocer de objetos.

Por otro lado existen desventajas como:

• Alto coste computacional. Debido a esto, los robots suelen tener varios procesadores, algunos de ellos asignados exclusivamente al procesado de imágenes.

• El precio es elevado. Un sistema de visión artificial, suele constar de una o dos cámaras de vídeo, un procesador dedicado a cálculos y una plataforma pan-tilt.

Este conjunto tiene un coste elevado (mayor que 600 euros).

• Las imágenes dependen de la iluminación, aunque ya se están explorando técnicas que disminuyen este problema.

• No es el sensor adecuado para medir profundidad, aunque se pueden hacer mapas de profundidad a partir de dos cámaras de vídeo.

2.4.2.4 Sensores inductivos

Los sensores basados en un cambio de inductancia debido a la presencia de un objeto metálico están entre los sensores de proximidad industriales de uso más frecuente. El principio de funcionamiento de estos sensores consiste fundamentalmente en una bobina arrollada, situada junto a un imán permanente empaquetado en un receptáculo simple y robusto. El efecto de llevar el sensor a la proximidad de un material ferromagnético produce un cambio en la posición de las líneas de flujo del imán permanente. En condiciones estáticas no hay ningún movimiento en las líneas de flujo y, por consiguiente, no se induce ninguna corriente en la bobina. Sin embargo, cuando un objeto ferromagnético penetra en el campo del imán o lo abandona, el cambio resultante en las líneas de flujo induce un impulso de corriente, cuya amplitud y forma son proporcionales a la velocidad de cambio de flujo. La tensión medida a través de la bobina varía como una función de la velocidad a la que un material ferromagnético se introduce en el campo del imán. La polaridad de la tensión, fuera del sensor, depende de que el objeto este penetrando en el campo abandonándolo. Existe una relación

entre la amplitud de la tensión y la distancia sensor-objeto. La sensibilidad cae rápidamente al aumentar la distancia, y el sensor sólo es eficaz para fracciones de un milímetro. Puesto que el sensor requiere movimiento para generar una forma de onda de salida, un método para producir una señal binaria es integrar esta forma de onda. La salida binaria se mantiene a nivel bajo en tanto que el valor integral permanezca por debajo de un umbral especificado, y luego se conmuta a nivel alto (indicando la proximidad de un objeto) cuando se supera el umbral.

2.4.2.5 Sensores de efecto Hall

El efecto Hall relaciona la tensión entre dos puntos de un material conductor o semiconductor con un campo magnético a través del material. Cuando se utilizan por sí mismos, los sensores de efecto Hall sólo pueden detectar objetos magnetizados. Sin embargo, cuando se emplean en conjunción con un imán permanente en la configuración tal como la indicada en la figura, son capaces de detectar todos los materiales ferromagnéticos. Cuando se utilizan de dicha manera, un dispositivo de efecto Hall detecta un campo magnético intenso en ausencia de un material ferromagnético en el campo cercano.

Cuando dicho material se lleva a la proximidad del dispositivo, el campo magnético se debilita en el sensor debido a la curvatura de las líneas del campo a través del material. Los sensores de efecto Hall están basados en el principio de una fuerza de Lorentz que actúa sobre una partícula cargada que se desplaza a través de un campo magnético. Esta fuerza actúa sobre un eje perpendicular al plano establecido por la dirección de movimiento de la partícula cargada y la dirección del campo. Es decir, la fuerza de Lorentz viene dada por $F = q(v \times B)$, en donde q es la carga, v es el vector de velocidad, B es el vector del campo magnético.

Al llevar un material ferromagnético cerca del dispositivo de imán semiconductor disminuirá la intensidad del campo magnético, con la consiguiente reducción de la fuerza de Lorentz y, finalmente, la tensión a través del semiconductor. Esta caída en la tensión es la clave para detectar la proximidad con sensores de efecto Hall. Las decisiones binarias con respecto a la presencia de un objeto se realizan estableciendo un umbral de la tensión fuera del sensor.

Además, la utilización de materiales semiconductores permite la construcción de circuitos electrónicos para amplificación y detección directamente en el propio sensor, con lo que se reduce el tamaño y el coste del mismo.

2.4.2.6 Sensores capacitivos

A diferencia con los sensores inductivos y de efecto Hall que detectan solamente materiales ferromagnéticos, los sensores capacitivos son potencialmente capaces (con diversos grados de sensibilidad) de detectar todos los materiales sólidos y líquidos. Como su nombre indica, estos sensores están basados en la detección de un cambio en la capacidad inducido por una superficie que se lleva cerca del elemento sensor.

El elemento sensor es un condensador constituido por un electrodo sensible y un electrodo de referencia. Estos electrodos pueden ser, por ejemplo, un disco y un anillo metálicos separados por un material dieléctrico. Una cavidad de aire seco se suele colocar detrás del elemento capacitivo para proporcionar aislamiento. El resto del sensor está constituido por circuitos electrónicos.

Hay varios métodos electrónicos para detectar la proximidad basados en cambios de la capacidad. Uno de los más simples incluye el condensador como parte de un circuito oscilador diseñado de modo que la oscilación se inicie solamente cuando la capacidad del sensor sea superior a un valor umbral preestablecido. La iniciación de la oscilación se traduce luego en una tensión de salida, que indica la presencia de un objeto. Este método proporciona una salida binaria, cuya sensibilidad de disparo dependerá del valor umbral. Se suelen usar disparadores Schmitt-Trigger.

La capacidad varía como una función de la distancia para un sensor de proximidad basado en los conceptos anteriores. Es de interés destacar que la sensibilidad disminuye mucho cuando la distancia es superior a unos pocos milímetros y que la forma de la curva de respuesta depende del material objeto de detección. En condiciones normales, estos sensores son accionados en un modo binario, de modo que un cambio en la capacidad mayor que en un umbral preestablecido T indica la presencia de un objeto, mientras que los cambios por debajo del umbral indican la ausencia de un objeto con respecto a los límites de detección establecidos por el valor de T.

2.4.2.7 Sensores de iluminación

Los sensores de luz visible y de infrarrojos cubren un amplio espectro de complejidad. Las fotocélulas se encuentran entre los más sencillos de todos los sensores para hacer su interfaz con el microprocesador, y la interpretación de la salida de una fotocélula es directa. Las cámaras de vídeo, por el contrario, requieren una buena cantidad de circuitería especializada para hacer que sus salidas sean compatibles con un microprocesador, además las complejas imágenes que las cámaras graban son todo menos fáciles de interpretar.

Los sensores de luz posibilitan comportamientos de un robot tales como esconderse en la oscuridad, jugar con un flash, y moverse hacia una señal luminosa. Los sensores de luz simples son fotorresistencias, fotodiodos o fototransistores. Las fotorresistencias son simplemente resistencias variables con la luz en muchos aspectos parecidos a los potenciómetros, excepto en que estos últimos varían girando un botón.

Los fototransistores dan mayor sensibilidad a la luz que las fotorresistencias. El fototransistor es básicamente un transistor con la corriente de base generada por la iluminación de la unión base-colector. La operación normal del transistor amplifica la pequeña corriente de base. Un fototransistor tiene una interfaz con un microprocesador casi tan fácil como el de una fotorresistencia.

Los fotodiodos tienen una gran sensibilidad, producen una salida lineal en un amplio rango de niveles de luz, y responden con rapidez a los cambios de iluminación. Esto les hace útiles en los sistemas de comunicación para detectar luces moduladas; el mando a distancia de casi todos los TV, equipos estéreos y reproductores de CD los emplean. La salida de un fotodiodo requiere, no obstante, amplificación antes de poder ser empleada por un microprocesador.

2.4.2.8 GPS

Este sistema proporciona una medida de la posición del robot, y por lo tanto se considera un sensor de medida absoluta. Da información de la situación geográfica en la que se encuentra el robot

móvil, gracias a un conjunto de más de 20 satélites que tiene el departamento de defensa de los EEUU.

El GPS tiene la siguiente ventaja:

• Es el único sensor que de una medida de la posición absoluta del robot que funciona en cualquier entorno exterior.

Por otro lado tiene varios inconvenientes:

• En el mejor de los casos se consiguen precisiones que rondan los 100 m. Para poder conseguir mejor precisión se puede usar el DGPS, reduciendo así el error a centímetros, pero es mucho más caro (9000 euros) y necesita de una estación base en un lugar conocido. El próximo sistema Galileo europeo será mejor.

• No se puede usar en edificios, ya que bloquean la señal de los satélites.

• Es caro, aunque los precios están bajando de forma espectacular.

2.4.2.9 Brújulas e inclinómetros

Un inclinómetro es un dispositivo muy simple y barato (unos pocos euros) que mide la orientación del vector gravitacional. Los más comunes usan mercurio. Para poder medir la inclinación necesita estar en una plataforma que no esté sometida a aceleración, ya que sino la medida es errónea. Es muy sensible a las vibraciones. Pero es fundamental si el robot va a trabajar en un entorno que no sea llano, ya que puede evitar que el robot vuelque o que se dañe la carga que lleva.

La brújula usa el campo magnético de la tierra para conocer la orientación del robot. La ventaja fundamental es que:

• Es el único sensor de medida absoluta, que mide la orientación del robot en prácticamente cualquier lugar del mundo. Un pequeño error en la orientación supone cometer constantemente errores en la posición según avanza el robot, por eso tener un sensor que ayude a corregir estos pequeños errores de forma inmediata, permite al robot moverse de forma más precisa.

Tiene varias limitaciones muy importantes:

• Es sensible a los campos magnéticos externos. Si provienen del robot y son constantes se pueden corregir. Esto se denomina Hard Iron Distortion. Para evitarlos basta con separarse de la fuente magnética una pequeña distancia, ya que el campo se atenúa con la distancia al cuadrado.

• Es sensible a los elementos metálicos que estén muy cerca del robot, ya que distorsionan el campo magnético de la tierra. Para que suceda esto el elemento metálico tiene que estar muy cerca del robot. Si la distorsión viene causada por el propio robot se puede corregir. Esto es lo que denomina Soft Iron Distortion.

Las precisiones que se pueden obtener oscilan desde cinco grados los más baratos (15 euros), hasta décimas de grado los más caros (900 euros). Existen brújulas de dos dimensiones, ya que miden el campo magnético en dos direcciones ortogonales, que solamente se pueden usar en entornos llanos. Si el entorno no es llano, es necesario usar una brújula de tres dimensiones y dos inclinómetros, para poder saber la orientación del robot. La brújula medirá el campo en tres ejes ortogonales y los inclinómetros medirán la inclinación del robot en los dos ejes horizontales.

Las brújulas más caras son capaces de corregir los errores que provienen del robot y además detectan posibles medidas espúreas cuando existe un campo magnético externo o un elemento metálico que pueda influir en la medida.

2.4.2.10 Giroscopio

Son brújulas de medida incremental; es decir, miden cambios en la orientación del robot. Se basan en medir la aceleración usando las leyes de newton.

Tienen varias limitaciones:

• Acumulan el error con el paso del tiempo

• Si se quieren tener precisiones aceptables (de décima de grado por segundo) son muy caros (6000-60000 euros).

• Los giroscopios baratos (80-300 euros) tienen errores mayores de un grado por minuto.

Existen unos giroscopios ópticos basados en láser, que permiten medir con mucha precisión la orientación del robot (errores que van desde varios grados por hora, hasta 0.0001 grado por hora, según el precio)

2.4.2.11 Shaft Encoder incremental

Los codificadores ópticos o encoders incrementales constan, en su forma más simple, de un disco transparente con una serie de marcas opacas colocadas radialmente y equidistantes entre sí, de un sistema de iluminación en el que la luz es colimada de forma adecuada, y de un elemento fotorreceptor. El eje cuya posición se quiere medir va acoplado al disco transparente. Con esta disposición, a medida que el eje gire se irán generando pulsos en el receptor cada vez que la luz atraviese cada marca, y llevando una cuenta de estos pulsos es posible conocer la posición del eje y la velocidad de rotación.

Este tipo de encoders se emplean en un gran número de aplicaciones dada su simplicidad y "economía". Por otro lado las principales limitaciones con las que cuenta son:

• La información acerca de la posición se pierde cuando la alimentación al sistema falla ó cuando es desconectado y cuando hay fuertes perturbaciones.

• Siempre es necesario un circuito contador para obtener una salida digital compatible con el puerto de entrada/salida de un microcontrolador. Otra posible forma de hacerlo se basaría en software especial según sea la aplicación específica, como por ejemplo, alguna interrupción o programación de alta velocidad, tiempo real, para obtener el tiempo de cambio entre un sector y otro.

• Los encoders pueden presentar problemas mecánicos debido a la gran precisión que se debe tener en su fabricación. La contaminación ambiental puede ser una fuente de interferencias en la transmisión óptica. Son dispositivos particularmente sensibles a golpes y vibraciones, estando su margen de temperatura de trabajo limitado por la presencia de componentes electrónicos.

• Existe desconocimiento en un momento dado de si se está realizando un giro en un sentido o en el opuesto, con el peligro que supone no estar contando adecuadamente.

Una solución a este último problema consiste en disponer de otra franja de marcas, desplazada de la anterior de manera que el tren de pulsos B que con ella se genere esté desplazado 90° eléctricos con respecto al generado por la primera franja. De esta manera, con un circuito relativamente sencillo, es posible obtener una señal adicional que indique cuál es el sentido de giro y que actúe sobre el contador correspondiente indicándole que incremente o reduzca la cuenta que se está realizando. Es necesario además disponer de una marca de referencia Z sobre el disco que indique que se ha dado una vuelta completa y que, por tanto, se ha de empezar la cuenta de nuevo. Esta marca sirve también para poder comenzar a contar tras recuperarse de una caída de tensión.

La resolución de este tipo de sensores depende directamente del número de marcas que se pueden poner físicamente en el disco. Un método relativamente sencillo para aumentar esta resolución es, no solamente contabilizar los flancos de subida de los trenes de pulsos, sino contabilizar también los de bajada, incrementando así la resolución del captador, pudiéndose llegar, con ayuda de circuitos adicionales, hasta 100.000 pulsos por vuelta.

El modelo de encoders más utilizado son los ópticos. Estos constan de diferentes sectores que pueden ser opacos y transparentes, reflejantes y no reflejantes. Otro tipo también muy usados son los magnéticos. Están equipados con un sistema de detección magnética sin contacto. La variación de campo magnético provocada por los dientes de una rueda de medida produce una onda senoidal de tensión. La electrónica del encoder transforma esta señal senoidal en una señal de onda cuadrada, triangular, etc. A pesar de no ser los más usados presentan ventajas sobre los ópticos:

• Altamente resistentes al polvo, los golpes o las vibraciones.

• Funcionamiento en un amplio rango de temperaturas: -20ºC a 85ºC.

El funcionamiento para cualquiera de estos casos es el mismo. Se implementa mediante un disco delgado metálico con una serie de ranuras equidistantes y homogéneas que se acopla directamente al eje del motor. Éste gira dentro de un switch óptico, que deja pasar e interrumpe un haz de luz infrarroja. Este haz de luz está suministrado por el led infrarrojo de la cabeza lectora fija. El sistema se completa con un fotodetector (LDR, celda fotoeléctrica o fototransistor). El switch optoelectrónico está formado, pues, por un par de dispositivos, un emisor y un receptor de luz infrarroja, colocados en un paquete plástico uno frente a otro con una ranura entre ellos por donde gira libremente el disco dentado montado en el eje de la rueda.

La precisión de un encoder incremental depende de factores mecánicos y eléctricos entre los cuales se encuentran el error de división del retículo, la excentricidad del disco, la de los rodamientos, el error introducido por la electrónica de lectura, imprecisiones de tipo óptico, etc. La unidad de medida para definir la precisión de un encoder es el grado eléctrico. Éste determina la división de un impulso generado por el encoder: en efecto, los 360º eléctricos corresponden a la rotación mecánica del eje, necesaria para hacer que se realice un ciclo o impulso completo de la señal de salida. Para saber a cuántos grados mecánicos corresponden 360º eléctricos es suficiente aplicar la fórmula siguiente:

360º eléctricos = 360º mecánicos / nºimpulsos por giro

El error de división en un encoder está dado por el máximo desplazamiento expresado en grados eléctricos, de dos frentes de onda consecutivos. Este error existe en cualquier encoder y se debe a los factores antes citados.

2.4.2.12 Encoders absolutos

El funcionamiento básico de los codificadores o encoders absolutos es similar al de los incrementales. Se tiene una fuente de luz con las lentes de adaptación correspondientes, un disco graduado y unos fotorreceptores. En este caso, el disco transparente se divide en un número determinado de sectores (potencia de 2), codificándose cada uno de ellos según un código binario cíclico (normalmente código de Gray) que queda representado por zonas transparentes y opacas dispuestas radialmente. No es necesario ahora ningún contador o electrónica adicional para detectar el sentido del giro, pues cada posición (sector) es codificado de forma absoluta. Su resolución es fija, y vendrá dada por el número de anillos que posea el disco graduado. Las resoluciones habituales van desde 2^8 a 2^19 bits (desde 256 a 524288 posiciones distintas).

2.4.2.13 Sensores de velocidad

La captación de la velocidad se hace necesaria para mejorar el comportamiento dinámico de los actuadores del robot. La información de la velocidad de movimiento de cada actuador se realimenta normalmente a un bucle de control analógico implementado en el propio accionador del elemento motor.

Normalmente, y puesto que el bucle de control de velocidad es analógico, el sensor usado es una taco generatriz que proporciona una tensión proporcional a la velocidad de giro de su eje.

Otra posibilidad, usada para el caso de que la unidad de control del robot precise conocer la velocidad de giro de las articulaciones, consiste en derivar la información de posición que ésta posee.

3. Capítulo III: Robot K-Junior

3.1 Introducción.

El proyecto de final de carrera está enfocado en la rama de la robótica, o mecatrónica (ya que une los conocimientos de electrónica y mecánica); concretamente se basa en el análisis de un robot móvil, K-Junior.

K-Junior es un robot comercial enfocado principalmente a la docencia, dotado de 10 sensores IR que le permiten análizar su entorno, dos ruedas en diseño diferencial que permiten al robot el giro sobre su propio eje, un pequeño altavoz, diferentes diodos leds indicadores, emisor / receptor infrarrojos y bluetooth, lo que le da versatilidad en su comunicación.

El control del robot se realiza por comandos, enviados por puerto serie, o por programación en c. El proveedor nos proporciona muchas funciones preprogramadas, simplificadas para controlar el microcontrolador, "cerebro" del robot.

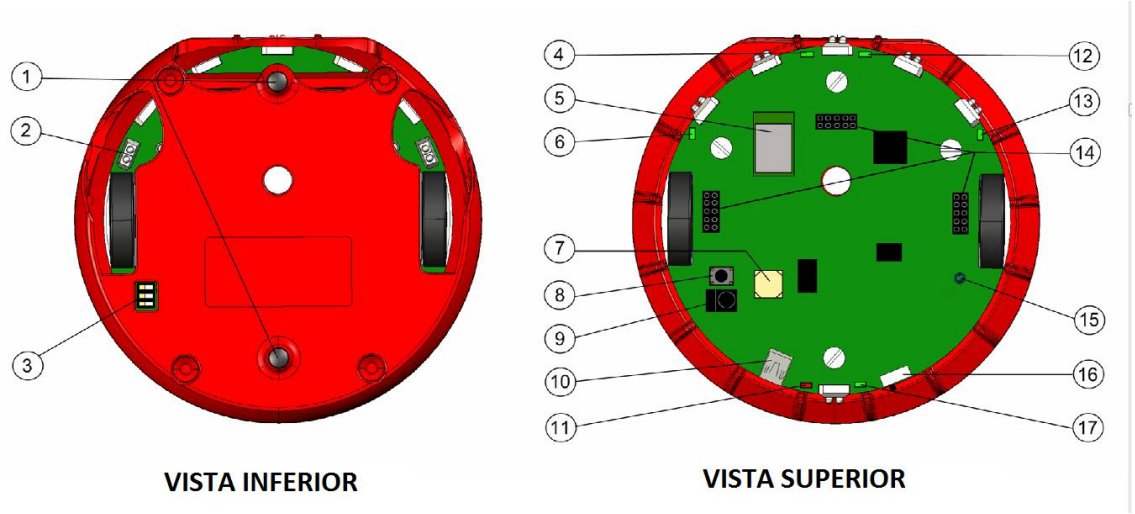
3.2 Descripción general

3.2.1 Especificaciones técnicas de KJunior.

Elementos	**Información técnica**
Movimiento	2 ruedas motrices movidas por motores de CC. Velocidad máxima: 160mm/s (0.16m/s)
Procesador	Microcontrolador PIC16F887, 8MHz, 3.3V, 35 instrucciones • 8192 palabras de 14 bits de memoria Flash • 368 Bytes de RAM • 256 Bytes de EEPROM
Batería	Batería recargable Li-Pol 3.7V 1200mAh que permite una autonomía de 4 horas(sin extensiones conectadas)
Modo de carga	Cable USB a PC, o bien, un adaptador de corriente externo (opcional)
Sensores IR	6 de proximidad y de detección de luz ambiental. 4 sensores inferiores para seguimiento de lineas y evitar caídas. Rango de hasta 20cm para obstáculos blancos con mínima absorbancia.
Receptor IR	Transmisión de comandos por medio del control remoto TV estandar o desde otros robots K-Junior.
Emisor IR	Comunicación con otros robots K-Junior con un rango de hasta 1m.
Bluetooth	Comunicación sin cables con un rango de hasta 20 m (sin obstáculos).
Conector USB	Para recarga de la batería o comunicación robot-PC.
LEDs	5 leds programables + 1 led indicador del estado de carga.
Altavoz	Capaz de producir sonidos desde 20 Hz hasta 2 kHz x=(0-100); 0 = off ; 1 = 20Hz; 100 = 2000Hz Frecuencia del sonido = 2000/(101- x)
Conmutadores	3 Conmutadores permiten seleccionar los diferentes modos de funcionamiento.
Hueco para rotulador	Agujero de fijación en el centro del robot permite a K-Junior dibujar en una hoja de papel

Sistema operativo	El código fuente es proporcionado y permite la modificación de las funciones del robot así como programar sus propias funciones	
Tamaño	Diámetro: Altura: 40mm	125mm

3.2.2 Vistas de KJunior.



1: Contactos con la superficie
2: Sensor de proximidad IR
3: Conmutadores de selección de modo
4: Led 1 (Verde)
5: Módulo bluetooth
6: Led 0 (Verde)
7: Altavoz
8: Botón reset
9: Receptor remoto IR
10: Conector mini-USB
11: led de indicador de carga (rojo)
12: Led 2 (Verde)
13: Led 3 (verde)
14: Conectores de extensiones
15: Led emisor IR
16: Conmutador ON/OFF
17: Led de encendido(Led 4, verde)

3.2.3 Similitudes entre el sistema robótico KJunior y un ser vivo

Así como un ser vivo puede ser visto como un conjunto de subsistemas que se intercomunican entre sí, permitiendo captar estímulos externos o internos, y procesando éstos, conseguir información útil, produciendo una respuesta acorde a las necesidades del momento. Es posible hacer un símil entre estos subsistemas propios de un ser vivo de alta complejidad y los del robot:

a) Sistema nervioso:

Microcontrolador: El cerebro, es el director de la orquesta, todos los datos pasan por él y son procesados, produciendo la información necesaria para responder acorde a lo preprogramado en la memoria.

Pistas y cables de cobre: actúan como neurotransmisores, enviando datos de uno a otro a subsistema.

b) Sistema circulatorio:

En lugar de transmitir energía a través de sangre, en nuestro robot, distribuiremos la energía multiplexando la utilidad de las pistas y cables de cobre, a través de ellos y provenientes de la batería de Li-Pol, llegará la energía requerida por cada parte de nuestro robot.

c) Sistema locomotor: Desde nuestro microcontrolador le llegan las órdenes a nuestro sistema dedicado al movimiento, dos ruedas movidas por dos motores de corriente contínua sobre los cuales controlamos el voltaje mediante PWM (pulse width modulation o modulación por ancho de pulso). El PWM (Figura 6.1) es un tipo de control ampliamente utilizado en sistemas electrónicos por la eficacia y sencillez. Consiste en variar el tiempo que un interruptor se encuentra cerrado durante un período constante. Se verá mejor en la figura siguiente:

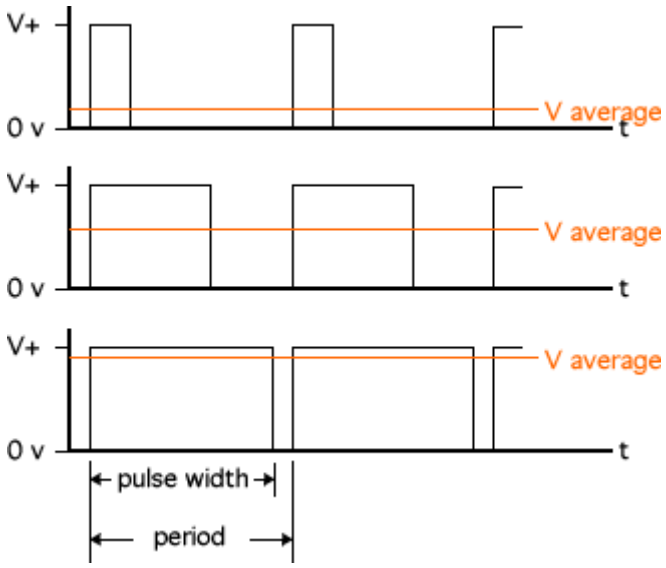


Figura 6.1 PWM

Así, tenemos programada una función que nos varía el voltaje medio sobre el motor: KJunior_set_speed(x,y) ; donde x e y varían entre 5 y 20, por debajo de 5, el voltaje medio sobre el motor no es suficiente para hacerlo girar, y por encima de 20 existe un saturador digital (programación) que limita el valor a 20.

d) Sistema sensorial: Nuestros cinco sentidos tienen un equivalente electrónico.

Vista: cámaras, basadas en sensores de radiación visible. En nuestro robot tenemos una cámara en escala de grises de 104 pixeles de resolución

Oido y habla: micrófonos, basados en el efecto piezoeléctrico, teniendo en cuenta las características mecánicas de las ondas de sonido.

La facultad del habla es la capacidad de producir perturbaciones de frecuencia variable en el medio (aire) para que sean recibidos por otros sujetos, los cuáles han de ser capaces de procesar esas perturbaciones y transformarlas a información para que se produzca la comunicación. K-junior tiene un altavoz capaz de vibrar a en un amplio rango de frecuencias, recordar no está de más que el rango audible oscila entre los 20 Hz y los 20 kHz., no permaneciendo dichos umbrales constantes con la edad. Un efecto que me parece interesante de comentar es que las fuentes conmutadas de alta frecuencia están diseñadas para trabajar en rangos para los cuales la sensibilidad a esas frecuencias en nuestro oído es muy baja, y no nos causan molestia alguna. Sin embargo, cuando una fuente de estas características tiene algún fallo de diseño o desgaste por uso y su frecuencia de trabajo es cercana al rango audible, provoca sonidos molestos para algunas personas.

Gusto y olfato: van unidos porque ambos son receptores químicos, activados bajo condiciones específicas (mecanismo llave-cerradura): este mecanismo biológico "inspecciona" tanto las características geométricas como las químico-físicas de la molécula en contacto con los receptores.

Tacto: tanto sensores capacitivos, inductivos, de ultrasonidos u ópticos constituyen una serie de elementos que nos posibilitan la detección de objetos en el espacio, y conocer su forma, sin embargo, el tacto como tal, es más complejo que esto.

La estereognosia es la facultad de conocer la silueta o forma de un objeto sólo mediante la información proveniente del tacto, así como cuando cerramos los ojos y cojemos una pelota, sabemos que es esférica; concretamente nuestro robot K-Junior realiza una tarea estereognósica cuando analiza la información proveniente de sus diferentemente colocados sensores IR y es capaz de saber si hay un borde, un objeto, e incluso un cambio de color brusco(un cambio en su absorbancia lumínica) en su entorno.

e) Sistema respiratorio y digestivo: es un tanto extraño pensar que nuestro robot realiza la respiración o la digestión, pero no lo es tanto pensar en colocar una placa solar y un convertidor CC-CC o colocar un transformador a la red y rectificar la señal para alimentar a nuestro robot, pues esto es precisamente lo que hacemos nosotros cuando respiramos o comemos, ambos casos van ligados y se basan en la obtención de energía química mediante la oxidación de moléculas complejas para poder usarla en nuestras necesidades (ya sea crecer, correr o incluso pensar). Nuestro transformador nos convierte la tensión alterna y de alto voltaje de la red a un valor pequeño y contínuo para usarlo en nuestro robot, a su vez esta energía de red proviene de una transformación energética, ya sea eólica, solar, hidráulica, nuclear, térmica, etc.

Otra característica de KJunior es que puede plasmar su trayectoria colocándole un rotulador en su hueco central para dibujar en el suelo.

Resumiendo, tenemos a nuestra disposición, una cámara en escala de grises de baja resolución, 10 sensores IR para detectar bordes, líneas u objetos, un altavoz para reproducir ciertas

frecuencias audibles, capacidad de movimiento autónomo y un sensor de ultrasonidos, para detectar distancias en un rango más elevado que con los IR.

3.2.4 Accesorios

Como ya hemos dicho, tenemos una cámara y un sensor de ultrasonidos como accesorios conectables al robot:

3.2.4.1 Cámara digital

Cámara lineal en escala de grises de 104 pixeles: cuando le damos la órden de que capte los valores instantáneos, nos devuelve una matriz cuyos valores representan el brillo en cada punto, en una escala de 0 a 255 (256 bits).

El funcionamiento está basado en la obtención de un mapa de valores digitales y el posterior proceso y almacenado. El haz de luz visible atraviesa un sistema óptico, gracias al cual, llegan los haces al sensor CCD (Charge coupled device), el cual es un chip sensible a la luz. En dicho proceso, es mencionable la función del obturador, pues es el encargado de controlar el tiempo de exposición, que es el tiempo durante el cual incide luz en el CCD. Un tiempo relativamente bajo, puede causar borrosidad y/o falta de información, en cambio si el tiempo es demasiado elevado, algunos pixeles pueden sufrir saturación y provocar la deformación de las relaciones entre pixeles vecinos.

Se puede controlar por puerto serie: con las tramas de datos que suministra el fabricante, o bien con programación, con las respectivas funciones preprogramadas.

Especificación técnica:

- Dimensiones [mm]: 42x 35 x 26
- Alimentación : 5 [V]
- Corriente [mA]: 10 (durante la adquisición de la imagen), 1 (en reposo)
- Máxima frecuencia de bus I^2C: 400 [kHz]
- Número de píxeles: 102
- Niveles de gris: 8 bits (0-255)
- Objetivo: M12 x 0.5
- Máxima tasa de refresco: 100 [Hz]

Figura 7.1 Cámara de K-Junior

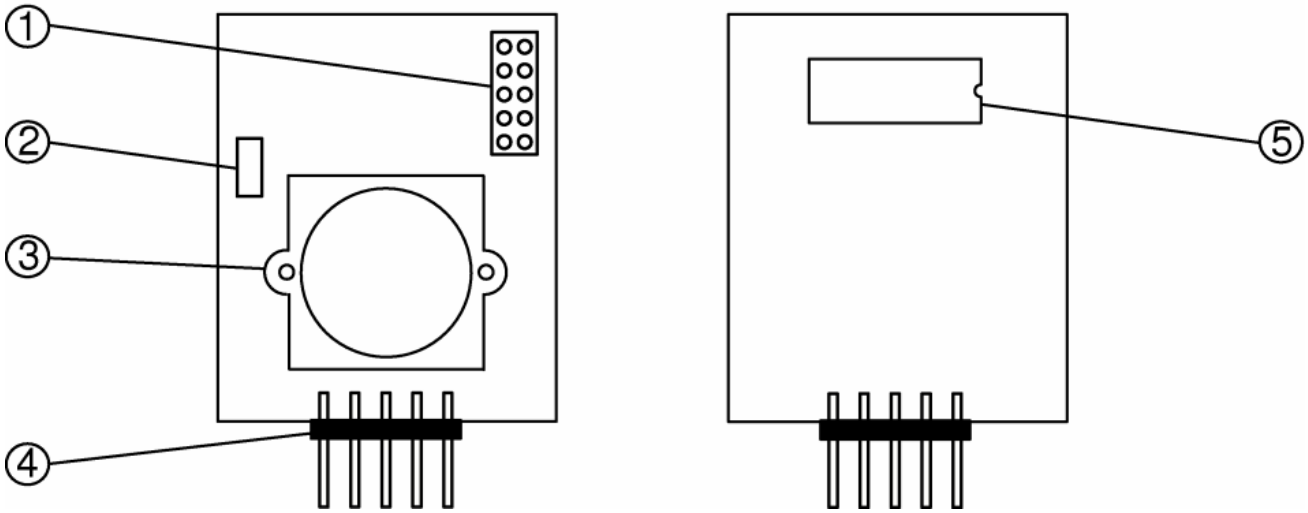


Figura 7.2 Conexiones de la cámara de K-Junior

1. Conector de reprogramación
2. Led
3. M12x0.5 Objetivo
4. Conector principal
5. Microcontrolador (PIC16F876)

3.2.4.2 Sensor de ultrasonidos

Al igual que los sensores infrarrojos son llamados sensores de proximidad, los módulos ultrasónicos permiten la medida de distancias. El principio es transmitir una onda ultrasónica y medir el tiempo hasta que se recibe el eco en el receptor. Conocida la velocidad del sonido en el aire y dicho tiempo de vuelo de la onda, es posible realizar una aproximación a la distancia a la que se ha producido el reflejo de la onda sonora.

Su control es parecido al de la cámara, todo muy simplificado por el proveedor, por programación o puerto serie.

Especificación técnica:

- Alimentación : 5 [V]
- Distancia mínima medible: 3 [cm]
- Distancia máxima medible: 600 [cm]
- El consumo varía dependiendo del modo de uso

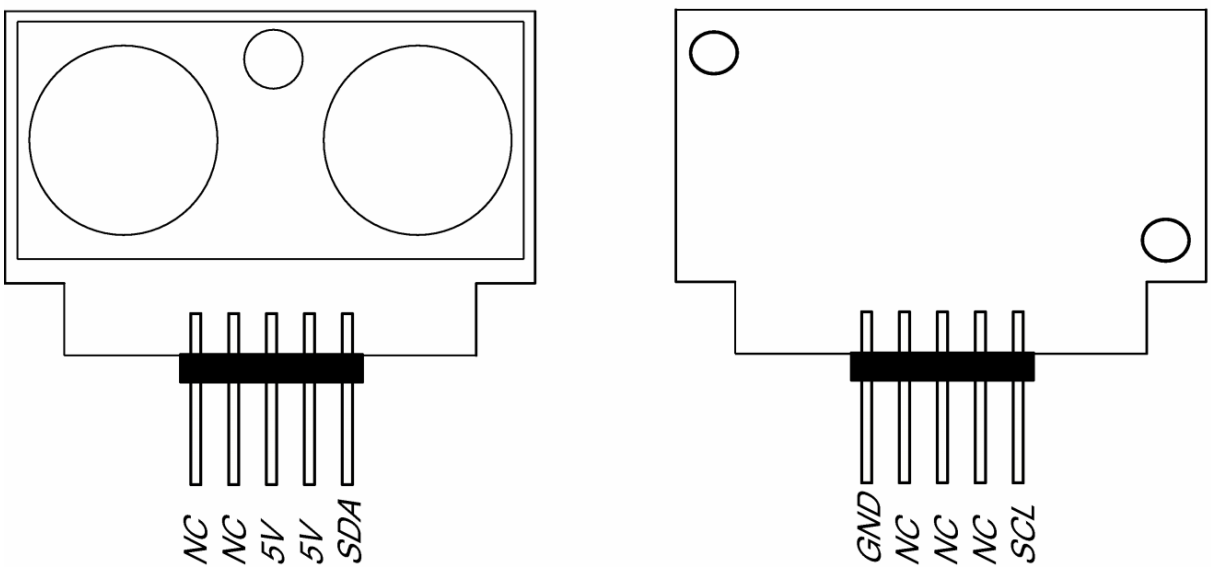


Figura 7.3 Pines del sensor de ultrasonidos

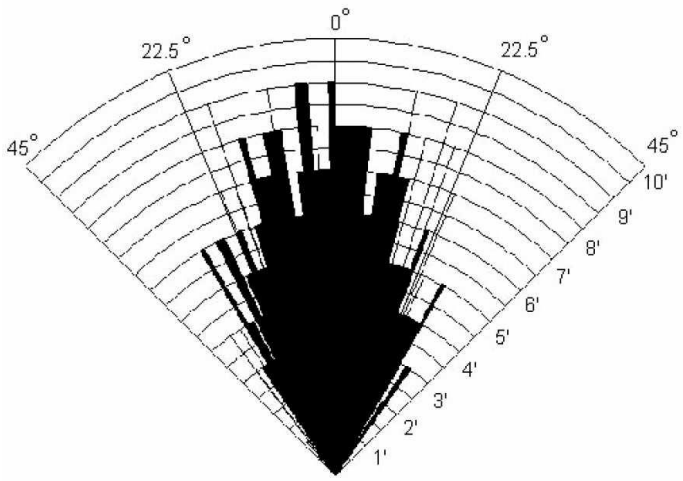


Figura 7.4 Diagrama polar de la magnitud del pulso ultrasónico

3.2.5 Accesorios no incorporados.

El proveedor dispone de accesorios extras para el robot K-Junior:

HemGenIO

Se trata de un módulo que permite conectar sencillamente circuitería propia. Contiene 12 entradas y salidas digitales, 5 entradas analógicas y bus I2C.

HemTextToSpeech

Permite hacer hablar a KJunior en inglés. Pronuncia palabras descargadas en código ASCII, a través del bus I2C.

HemGripper

Son unas tenazas para agarrar objetos de diferente tamaño, forma y peso. La fuerza es ejercida por dos servomotores DC, que son controlados por un microcontrolador dedicado (PIC16F690). Debido a su alto consumo, existen dos versiones, con distinta autonomía en función de la batería.

HemWirelessCam

Una cámara a color que permte ver en un dispositivo de salida de video lo que K-Junior ve. Libre de cables, usa una comunicación bluetooth para su control remoto.

3.3. Puesta en marcha

Cuando K-Junior se puso en mis manos, lo primero que hice fue probar las funciones predefinidas. En un apartado posterior las describo más a fondo. Tras realizar esta primera toma de contacto, me puse manos a la obra a buscar códigos ya escritos para dicho robot e información acerca de éste.

A la hora de programar el pic del robot, no surgen demasiados problemas. Con el programa PIC C compiler se escribe el código, ya sea en lenguaje c o ensamblador, y se inserta donde indica la imagen siguiente (Figura 8.1). Una vez depurado el código. Éste se vuelca a la memoria del PIC16F887 a través del cable mini-usb y con el programa TinyBootloader adjuntado con el cd-rom del fabricante. La conexión por USB crea un puerto serie virtual, el cual habrá que comprobar su número de puerto COM para introducirlo en el programa. Además, hay un botón para comprobar si la conexión está bien realizada.

El lenguaje usado es c, por lo que los estudiantes de ingeniería técnica industrial tenemos la base suficiente para afrontar una programación a nivel básico-medio del robot.

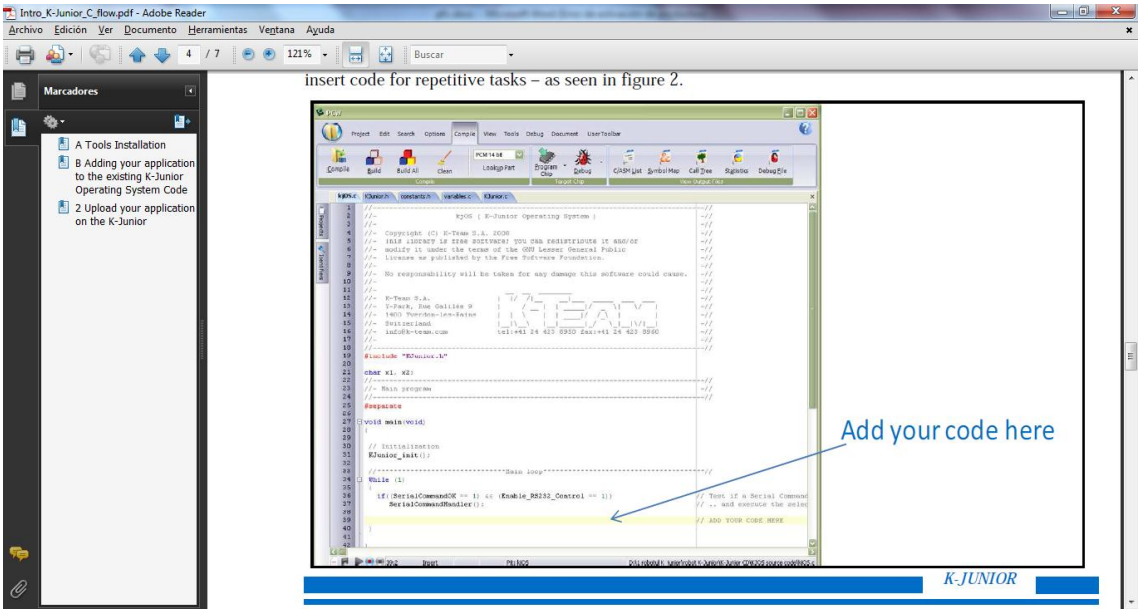


Figura 8.1 Inserción de código en el OS de K-Junior

El proveedor dispone en su web una serie de aplicaciones para K-Junior bajo el programa Labview de National Instruments, la que más útil me ha sido es la de conocer en tiempo real la información proveniente de los sensores IR del robot a través de un puerto serie, en mi caso, una transmisión inalámbrica bluetooth. Además de posibilitar el control del movimiento del robot, simplificado con las flechas del teclado y la tecla espaciadora. Este código se llama "KJ_navigator_v1.1.vi" (Figura 8.2).

Las flechas hacia delante y hacia atrás, sirven para variar la velocidad a ambos motores por igual. En cambio las flechas izquierda y derecha, sirven para provocar el giro hacia uno u otro lado, variando la diferencia de las velocidades de las ruedas izquierda y derecha. Existen, para el ajuste de la brusquedad del control, dos cajetines: "speed increment y "steer increment", cuyos valores determinan la variación, equitativa o diferencial, respectivamente, del giro de los motores, según el tiempo de pulsación de la flecha correspondiente.

El programa muestra la información de todos y cada uno de los sensores, los cuales tienen un valor digital de 0 a 1024 en la teoría, pero en la práctica no se alcanzan dichos valores. Del S1 al S6 son los sensores IR de la periferia, y del S7 al S10, los sensores del terreno .

Es necesario, además, introducir el puerto serie de comunicación con el robot. Es recomendable parar el programa, dándole a stop, para que se cierre la conexión serie, si se va a cerrar el programa y no surjan problemas a la hora de retomar la conexión.

Por comodidad, lo mejor es realizar la comunicación serie por bluetooth, por razones obvias de libertad del movimiento del robot. Con el cable se carga y programa el robot, aunque puede ser útil para realizar determinadas pruebas, como programar y probar casos determinados rápidamente.

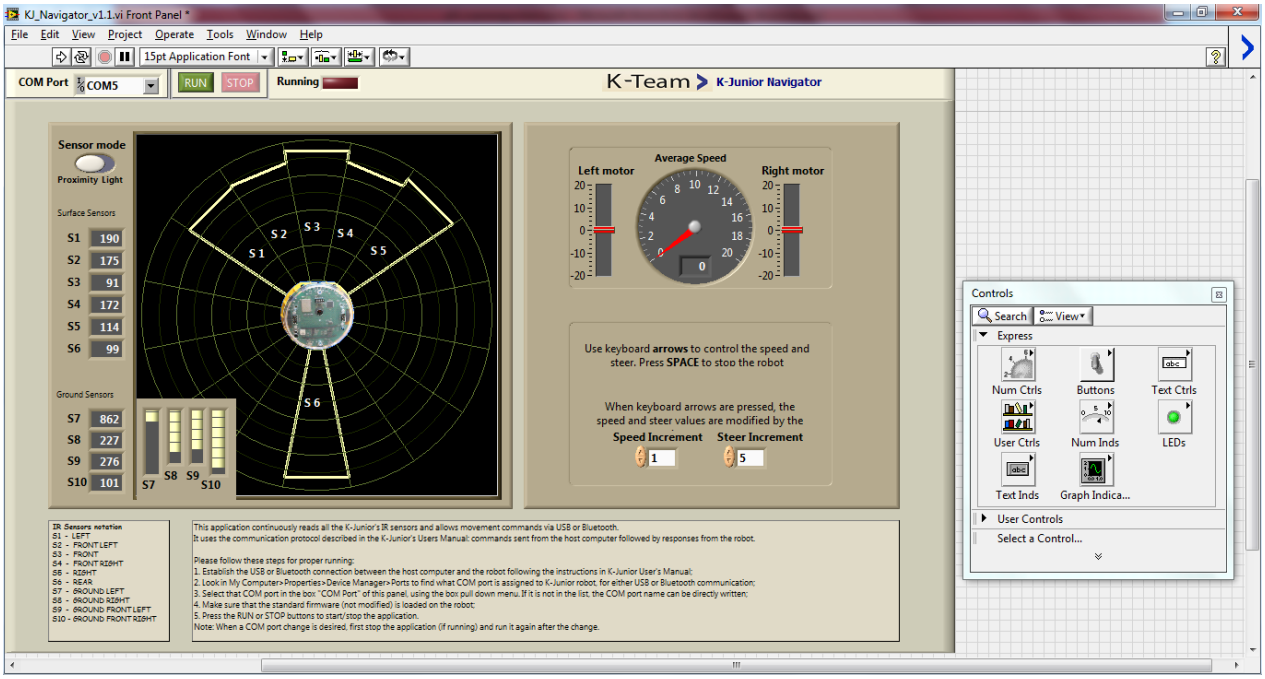


Figura 8.2 KJ_Navigator. Interface

La otra posibilidad: la comunicación serie, con los comandos que provee el fabricante, pueden ser enviados al robot con el programa Hyperterminal, muy sencillo de usar, Labview, o con Matlab, un tanto más complejo, pero con todas las posibilidades que otorga el programa en cuanto a programación.. Para pruebas aisladas es recomendable usar el Hyperterminal por facilidad, pero si las pruebas van a repetirse y pueden prolongarse, puede ser más cómo el Matlab, una vez se le dedique tiempo a realizarse los scripts correspondientes para abrir o cerrar la comunicación, o el envío de determinadas tramas en según qué situaciones, etc.

3.4 Sistema Operativo (OS)

El sistema operativo de K-Junior es el responsable de gestionar los recursos del robot y permite el acceso a los elementos de hardware externos conectados.

Está compuesto de varios archivos:

- « *KJOs.c* »: Archivo principal. Aquí se declaran variables, funciones y se escribe el código "casero".
- « *KJunior.c* »: Contiene el código de las funciones de control y acceso al hardware de K-junior.
- « KJunior.*h* »: Contiene las funciones prototipo y los parámetros de k-junior(frecuencia del cuarzo, velocidad de modulación, etc).
- « *16f887.h* »: Archivo que contiene las direcciones de los registros del procesador.
- « *variables.c* »: Contiene las variables internas.

- « *constantes.h* », Contiene la definición de constantes.

- « *versions.txt* », Describe las modificaciones hechas en cada version.

Hay otros archivos disponibles en el directorio « *Libs* » para usar los módulos de extensión (ultrasonidos, cámara lineal, etc.)

3.5 Control del robot

El robot se puede controlar de diferentes maneras, aunque principalmente se pueden dividir en dos:

1. **Por programación**: Es posible desarrollar un código en c basado en unas funciones preestablecidas que facilitan el uso del robot, así como de sus diferentes accesorios

2. **Por puerto serie**: Este tipo de control, a su vez, se divide en otros tres:

a) USB: una conexión por cable que, restringiendo el rango de operación del robot, permite la carga de la batería y la programación del robot. Usando el protocolo RS232 es posible enviarle comandos al robot y ejecutar acciones en tiempo real. Es necesario el uso de algún programa como matlab, labview, o sencillamente hyperterminal.

b) Bluetooth: conexión inalámbrica que permite al robot total libertad de movimiento hasta unos 20 metros. Usando los mismos comandos que por USB podemos controlar al robot igual que por cable. Al igual que para la conexión USB, para realizar intercambio de datos via puerto serie es necesario algún software para ello.

C) IR: Tanto el emisor como el receptor IR permite a dos o más robots intercambiar datos entre ellos o darse órdenes.

A través de este receptor IR, un demodulador de 36 kHz basado en el código rc5, y con un mando de televisión estándar o universal, es posible controlar el movimiento del robot.

Podríamos decir que la diferencia de este control con la programación, es que es ‘just in time’, es decir, el robot ejecuta una acción predeterminada previa recepción de unos comandos previamente establecidos, que se le mandan via puerto serie e instantáneamente obtenemos la respuesta del robot, no obstante, podemos realizar un código que implemente las acciones en formato comando serie y que las realice, para este propósito una herramienta muy útil puede ser el programa Matlab

3.6 Funciones predefinidas:

El robot tiene tres funciones predefinidas que son:

1. bailar (el robot oscila y se mueve algorítmicamente).
2. evitar obstáculos y caídas (el robot avanza de frente mientras no haya un obstáculo o un cambio de nivel, en cuyo caso, cambia la trayectoria para salvar el obstáculo.
3. seguimiento de líneas (el robot sigue una trayectoria definida por una línea oscura en un suelo claro).

La primera es una sucesión de órdenes a los motores, mientras que en las dos siguientes los motores se controlan según los parámetros recibidos desde los sensores IR horizontales que indican la proximidad de un objeto, o los verticales que alertan de una caída inminente o la presencia de una línea de diferente absorbancia que el fondo (blanca-negro, negra-blanco).

Cuando reprogramas el robot con el OS del CD del fabricante, o bien, le añades algún código a este OS, el robot pierde estas funciones predefinidas. Sin embargo, podemos descargar dichas funciones de la página del proveedor y volver a implementarlas sin problema alguno.

4. Capitulo IV: Resolución de laberintos y generación de trayectorias

4.1 Introducción.

Una de las aplicaciones que más atractiva me resultó fue la de que el robot fuera capaz de salir de un laberinto diseñado al azar. Se podía plantear de varias maneras, con sensor ultrasónico para laberintos con gran distancia entre paredes, con la cámara vía visión artificial, o con los sensores IR que lleva por defecto el robot. Ésta última fue mi decisión. Empecé a analizar el comportamiento del robot en las más variadas situaciones, y tras diferentes pruebas, concluí un código con un comportamiento aceptable y suave.

4.2 Aplicación para K-Junior: robot de laberinto

El "KJ_navigator_v1.1.vi", adjuntado en el PFC, fue de vital importancia, para caracterizar los sensores, de manera simplificada, pudiendo después adoptar rangos de medidas, para la programación del código para salir del laberinto. En cada una de las posibles situaciones, analicé los valores de los sensores, y comencé a realizar un código simple que permitiera salir de un laberinto simple hecho con envases de 200 ml rectangulares, típicos de los zumos de fruta.

Utilizando los valores medidos de los sensores en las distintas situaciones posibles en el laberinto, comencé a programar el programa que llevaría al robot a salir de cualquier laberinto.

El programa es simple y su bucle principal (Figura 9.1):

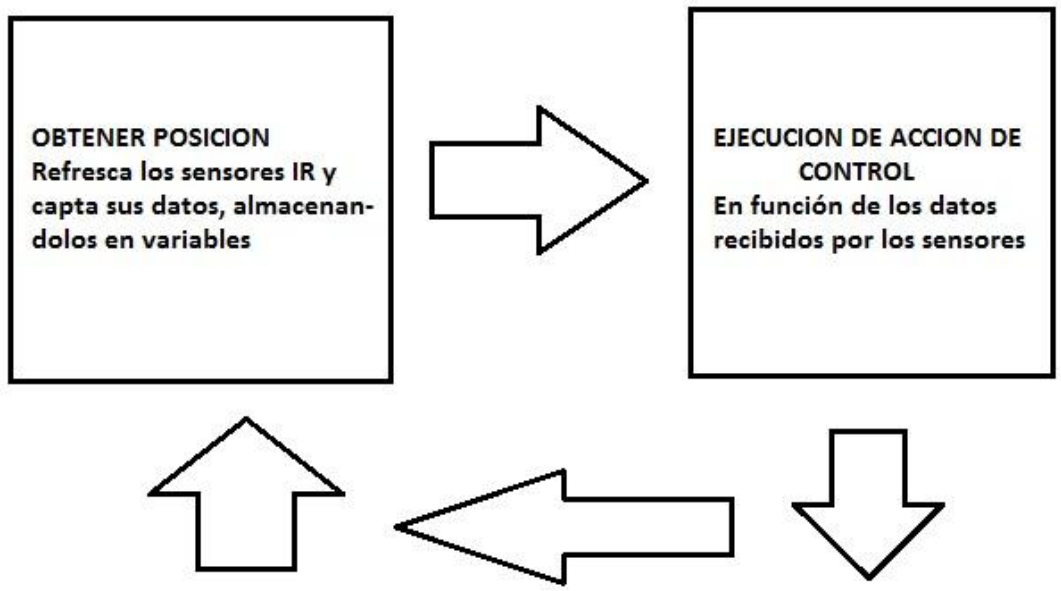


Figura 9.1 Bucle principal del código

La premisa principal del programa es, siempre que sea posible, girar a la izquierda, con este algoritmo, no aseguramos la trayectoria óptima, pero sí la salida del laberinto, siempre y cuando no tenga "puentes" o pasos elevados.

Como segunda prioridad tenemos el seguir de frente, como tercera el giro a derecha, y como última la calle sin salida, giro 180º.

1. prioridad primera: giro a izquierda
2. prioridad segunda: seguir de frente
3. prioridad tercera: girar a la derecha
4. prioridad cuarta: girar 180º

Lo primero a optimizar, era el giro. Debido a la disposición de los sensores, no resultó sencillo, de primeras, que realizara el giro correcto. La primera idea fue usar retardos para contrarrestar el tiempo que dejan de detectar los sensores el muro hasta que el robot sea capaz de girar 90 grados sin chocar con el muro (Figura 9.2). El resultado era bastante mediocre e inextrapolable a otros casos.

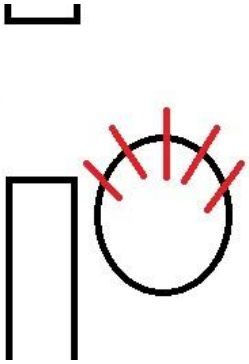


Figura 9.2 Croquis de situación problemática 1

Solucioné este problema del giro sin chocar con una segunda idea: giro a pasos, es decir, giro durante 100 ms y durante otros 100 ms sigo recto, cíclicamente, hasta que complete el giro y vuelva a detectar muro a su izquierda. Esta solución daba un aspecto más suave a la trayectoria, y además, sí se podía usar este recurso para el giro a derecha, lo cual me resultó muy positivo.

Solucionado este problema, me centré en la solución del siguiente, que consistía en que al detectar la necesidad de giro a derecha , comenzaba a girar pero a mitad del bucle, el sensor de la izquierda detectaba un hueco e intentaba girar a la izquierda, produciendo conflictos, esto se debía a la configuración de los muros de zumos, en L (Figura 9.3). Se puede apreciar en el siguiente esquema de posición, la distancia en verde era la causante del conflicto. En este caso la solución consistió en usar una variable que se ponía a 1 cuando se estaba girando a derecha e inhibía la prioridad del giro a izquierda, para dársela a seguir de frente.

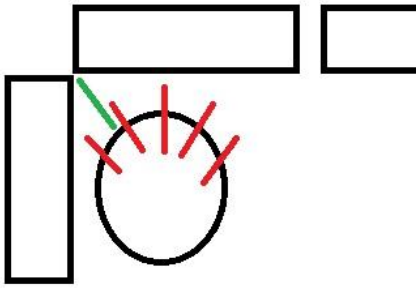


Figura 9.3 Croquis de situación problemática 2

Ahora ya sí, el robot salía del laberinto. Me sentía satisfecho por el momento. Pero era momento de comenzar a realizar pequeñas optimizaciones. Además de depurar el código con algunas mejoras en las estructuras repetitivas y condicionantes. Comencé a vislumbrar la aplicación del concepto de Lógica Borrosa o "Fuzzy". Para intentar asemejar un poco más al comportamiento humano.

El programa también lo adjunto para que pueda servir de base a quien lo desee, pues está abierta a muchas y variadas mejoras.

El siguiente diagrama de bloques (Figura 9.4) profundiza en el funcionamiento del programa:

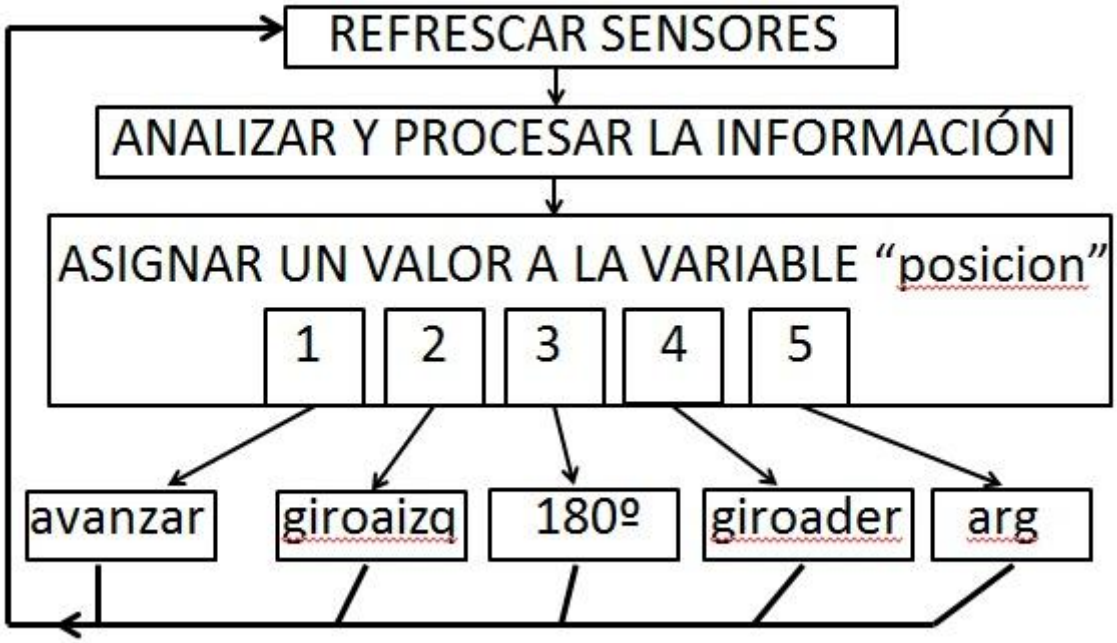


Figura 9.4 Diagrama de bloques del código

4.3 Métodos de resolución de laberintos

Los algoritmos de resolución de laberintos pueden ser clasificados según los siguientes criterios: [2]

- La velocidad que necesita el algoritmo para encontrar la solución, que es siempre proporcional al tamaño del laberinto y/o a las condiciones del computador.

- La memoria externa y/o de pila que programa necesita para su ejecución

A continuación se caracterizarán algunos:

- Llenar los caminos sin salida: es un algoritmo simple, muy rápido y que no utiliza memoria extra. Solamente es necesario examinar el laberinto y llenar los caminos sin

salida. Al final sólo quedará la solución o las soluciones si hay más de una. Este algoritmo buscará la única solución en los laberintos perfectos, pero no hará gran cosa en un laberinto-trenza y, evidentemente, será totalmente ineficaz en un laberinto que no tenga caminos sin salida.

- Recorrer la pared: es otro método de resolución de laberintos simple, muy rápido y no necesita memoria suplementaria. Comienza recorriendo el camino y, cada vez que hay un cruce, gira a la derecha (o a la izquierda). Si se quiere, se pueden marcar los sitios por donde se ha pasado y los que se han visitado dos veces de manera que al final se pueda resolver el laberinto pasando por los caminos utilizados una sola vez. Este método no encontrará el camino más corto y no funcionará mejor en los laberintos que tengan el final en el centro y un circuito cerrado a su alrededor.

- Rehacer caminos: Es un método rápido para todo tipo de laberintos, utiliza memoria en función de las dimensiones del laberintos y la solución que encuentra no es necesariamente la más corta. La idea es intentar moverse en cuatro direcciones; se traza una línea cuando se prueba una nueva dirección, se borra si lo que hay es un camino sin salida y, finalmente, obtenemos una solución simple. Este algoritmo encontrará siempre una solución, pero no necesariamente la más eficaz.

- Eliminar colisiones: este método buscará las soluciones más cortas. Es rápido para todo tipo de laberintos y requiere como mínimo una copia de éste en la memoria. Básicamente inunda el laberinto con "agua", de manera que todas las distancias desde el inicio se llenan al mismo tiempo y cada vez que dos "columnas" de agua unen dos caminos por sus respectivos finales (indicando una vuelta), se añade una pared al original. Cuando todas las partes del laberinto están llenadas, debemos repetir el proceso hasta que no haya más colisiones.

- Azar: en contraposición a los otros métodos, aquí encontramos un algoritmo de resolución de laberintos que se basa en moverse al azar. Esto significa desplazarse en una dirección y seguir el pasadizo hasta la siguiente unión; no debemos hacer ningún giro de 180º hasta que no sea necesario. Podríamos decir que este programa simula un humano sin memoria (no recuerda por donde ha pasado), por lo que es lento y no garantiza la llegada al final. Encontrada la solución del laberinto, será casi imposible rehacer los pasos; no obstante, es muy simple y no requiere memoria extra.

4.4 Posibilidades del robot K-Junior en la resolución de laberintos

El método para la resolución de laberintos adoptado en el código adjunto se basa en el método descrito anteriormente como: "recorrer la pared". Se podrían haber usado en su lugar algún otro método, como el método de "rehacer caminos", complejo pero con muchas ventajas, el método de "llenar los caminos sin salida", o bien el método del "azar", algo no recomendable, salvo en algún

caso particular. Para la ejecución de éstos métodos sería suficiente con el sensado IR o ultrasónico y una programación adecuada para tal fin.

En cambio, el método de "eliminar colisiones" requiere del conocimiento previo del laberinto. Una solución a este problema podría ser la conexión de la cámara del robot, de manera inalámbrica, y que ésta fuera "el ojo desde arriba". Sería necesario un sistema diseñado para procesar la imagen del mapa y diferenciara el robot del suelo y muros.

4.5 Lógica borrosa. Potencial artificial.

El concepto de lógica borrosa en el control de trayectorias tiene una importancia especial debido a que el comportamiento definido es más parecido al del humano.

"Quizá sea necesario un replanteamiento radical de nuestros conceptos clásicos de verdad y falsedad, sustituyéndolos por el concepto de vaguedad o borrosidad, dentro de los cuales la verdad y/o falsedad no son más que casos extremos. Por borrosidad entendemos el hecho de que una proposición pueda ser parcialmente verdadera y parcialmente falsa de forma simultánea. Una persona no será simplemente alta o baja, sino que participará de ambas características parcialmente, de tal forma que sólo por encima y debajo de determinadas alturas la calificaremos de forzosamente alta o baja, mientras que en la zona intermedia de ambas alturas existirá una graduación por la que va dejando de ser alta. Parece que intuitivamente, el concepto de borrosidad está enraizado en la mayor parte de nuestros modos de pensar y hablar. Otra cuestión distinta es la valoración que cada individuo otorgue a tal borrosidad (el vaso medio lleno o medio vacío) que dependerá de cuestiones psicológicas de difícil evaluación.

El principio borroso afirma que todo es cuestión de grado. Todas las proposiciones adquieren un valor veritativo comprendido entre el uno (verdad) y el cero (falsedad), ambos incluidos (Figura 10 Lógica borrosa). La asignación de estos valores extremos sólo se dará en el caso de verdades o falsedades lógicas o de inducciones fuertes :"Todos los hombres son mortales" puede ser un ejemplo de inducción fuerte (si las modernas biotecnologías no lo cambian) puesto que no existe ningún contraejemplo." [3]

Así pues, en el caso del control de la trayectoria de nuestro robot, podríamos aplicarle predicados de "cerca" y "lejos" al sensado IR. Y con ello ajustarse más al comportamiento humano: por ejemplo, cuando estamos caminando, una pared que está lejos no te hace cambiar la trayectoria hasta que la etiqueta de "cerca" llama la atención para no colisionar. De manera que mantenemos un rumbo definido por nuestra meta a largo plazo, pero siempre salvando los obstáculos cercanos.

Un concepto interesante que puede ser aplicado también es el de crear campos de potencial artificial. Un punto, nuestra meta, ejerce unas fuerzas atractivas sobre nuestro robot, cuanto más cerca esté, mayor será esta fuerza imaginaria. En cambio, los obstáculos ejercen una fuerza de repulsión, forzándolo a esquivar a éstos.

Figura 10 Lógica borrosa

4.6 Generación de trayectorias: splines

El desarrollo de los splines (NURBS) empezó en 1950 por ingenieros que necesitaban la representación matemática precisa de superficies de forma libre como las usadas en carrocerías de automóviles, superfices de exteriores aeroespaciales y cascos de barcos, que pudieran ser reproducidos exacta y técnicamente en cualquier momento. Las anteriores representaciones de este tipo de diseños sólo podían hacerse con modelos físicos o maquetas realizadas por el diseñador o ingeniero.

Los pioneros en esta investigación fueron Pierre Bézier quien trabajaba como ingeniero en Renault, y Paul de Casteljau quien trabajaba en Citroën, ambos en Francia. Bézier y Casteljau trabajaron casi en paralelo [4]

Se trata de un método de interpolación matemática por el cual se puede aproximar una trayectoria que pase por determinados puntos, y con pendiente y curvatura también definidas.(Figura 11 Método de interpolación: Splines)

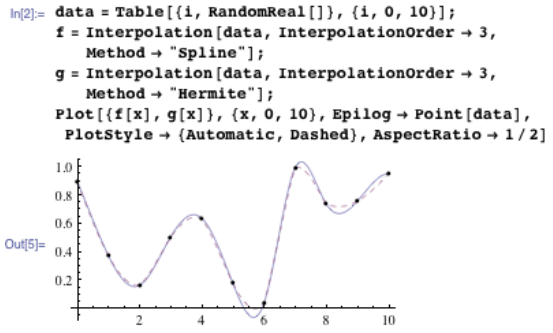


Figura 11 Método de interpolación: Splines

5 Conclusiones, futuras ampliaciones y recomendaciones

Con el manejo del robot se ponen en práctica conocimientos previamente adquiridos sobre programación y lógica. Además, estos conocimientos se asientan en un contexto práctico, facilitando así su aprehensión.

En estas situaciones prácticas surgen problemas propios de ésta. Del mismo modo hay que actuar, pensar e introducirte de lleno en dicho problema, de una manera práctica. Pensar en algo complicado suele llevar mucho tiempo, en ocasiones, más del que disponemos. Por ello, una solución sencilla, eficaz y rápida, puede ser en muchos casos el mejor comienzo en la solución del problema surgido.

Además de lo aprendido en el manejo del robot, existen otros conceptos vistos durante la carrera, que se refuerzan con el estudio de éste en profundidad. Tal es el caso del control PWM, funcionamiento de sensores y motores, establecimiento de comunicaciones inalámbricas, convertidores A/D, funcionamiento de un microcontrolador, etc

Por otro lado, se podrían realizar innumerables trabajos futuros en este campo. Algunas de las ideas surgidas a lo largo del camino son:

- Realizar un código que utilice el concepto de lógica borrosa y potencial artificial
- Realizar la inspección del laberinto desde una cámara que capture todo el laberinto, así como el movimiento del robot.
- Utilizar los splines para el dibujo de curvas sobre el papel con el rotulador que se le puede colocar en el centro a K-Junior.
- Convertir un coche R/C en un robot. Es sencillo,didáctico y de bajo costo.
- Realizar un sistema de comunicación verbal con el robot.
- Realizar un sistema de vocalización del texto
- Realizar un sistema de generación de mapas apartir de la información de los sensores IR o ultrasónicos.
- Sistema rotatorio para sensor ultrasónico

6 Bibliografía y referencias

6.1 Bibliografía referenciada:

[1] http://www.iit.upcomillas.es/~alvaro/teaching/Clases/Robots/teoria/Sensores%20y%20actuadores.pdfhttp://laberintos.weebly.com/tipos-de-laberintos.html

[2] http://www.aula-ee.com/webs/laberints/castella/res.htm

[3] http://personal.telefonica.terra.es/web/mir/ferran/kosko.htm

[4] http://es.wikipedia.org/wiki/NURBS

[5] http://asimo.honda.com/asimo-history/

[6] http://www.elotrolado.net/hilo_tutorial-programando-en-c_1546639

6.2 Bibliografía consultada:

Documental: “Beyond human. Living machines.” ó “Más allá de lo humano. Robots”

Documental: “Frontline of construction. Robots” ó “Construcciones vanguardistas. Robots”

Documental: “Extreme machines” ó “Máquinas extremas”

Libro: Robótica. John J. Craig.

Sitios Web:

http://www.forosdeelectronica.com/f19/apuntes-robotica-2740/
http://www.forosdeelectronica.com/f19/carrito-control-remoto-42229/
http://www.dtic.upf.edu/~jlozano/interfaces/interfaces5.html
http://cfievalladolid2.net/tecno/cyr_01/robotica/movil.htm
http://indianabot.tripod.com/
http://www.caminos.upm.es/matematicas/Fdistancia/MAIC/CONGRESOS/SEGUNDO/014%20Determinaci%C3%B3n2.pdf
http://www.tecnalia.com/es/divisiones/salud/tecnologias-salud/noticias/robotica-movil-cooperativa.htm
http://www.profesormolina.com.ar/circuitos/circuitos.php?codigo=169
http://www.conscious-robots.com/
http://www.robotsperu.org/foros/forum.php
http://www.jmnlab.com/sumo1/sumo1.html
http://es.wikipedia.org/wiki/L%C3%B3gica_difusa

http://es.wikipedia.org/wiki/Espacio_de_configuraci%C3%B3n
http://sensorultrasonico.blogspot.com/2008/05/diseo-de-un-sistema-de-medicion-basado.html
http://www.nortecnica.com.ar/pdf/teoria_opticos_2_2.pdf
http://www2.ing.puc.cl/~iic11021/materia/ejemplos/maze.htm
http://neoparaiso.com/logo/actividad-laberintos.html
http://www.taringa.net/posts/info/4765871/El-origen-de-los-Laberintos.html
http://www.librosmaravillosos.com/inventos/capitulo14.html
http://www.taringa.net/posts/apuntes-y-monografias/10453552/las-primeras-computadoras-del-mundo.html
http://khenya30informaticaeducativa2010.blogspot.com/2010_08_01_archive.html
http://visionartificialparatodos.wordpress.com/author/agutierrezp/
http://skiras.blogspot.com/2008/05/3ms-iii-configuraciones-mecnicas.html
http://es.wikipedia.org/wiki/Inteligencia_artificial
http://cfievalladolid2.net/tecno/cyr_01/robotica/movil.htm#locomocion
http://robotilandia.blogspot.com/2007/08/componentes-de-un-robot.html
http://aprender20.es/periodico/node/429
http://www.slideshare.net/felix.rivas/robotica2-presentation
http://serpientes_llorente.galeon.com/locomocion.htm
http://www.inf-cr.uclm.es/www/cglez/downloads/docencia/AC/splines.pdf

ANEXO : PROGRAMACIÓN EN C

Este tutorial me resultó muy productivo para recordar conceptos de programación: estructuras repetitivas y condicionantes. Además, no resulta muy incómodo de leer, pues es bastante ameno. Espero que pueda resultar igual de útil que a mí.

[6] http://www.elotrolado.net/hilo_tutorial-programando-en-c_1546639

1. Introducción
C, como cualquier otro lenguaje de programación no requiere más que tiempo, una cabeza y mucha paciencia. No pretendáis que leyendo este tutorial aprendáis C a nivel experto en un día, y empecéis a programar homebrew como quien no quiere la cosa.
Empezar a programar en C puede ser algo muy bueno o muy malo. C es un lenguaje que, si no te lo tomas con calma y te lo relees todo hasta entenderlo, no puedes continuar avanzado, este tutorial está pensado para ir progresando poco a poco. En cambio, una vez sepas o domines C, cualquier otro lenguaje de programación te parecerá igual, sino muy fácil. Es decir, empezar con este tutorial sin ánimos, a desganas, o con la creencia de que no aprenderéis nada, será contraproducente.
No me mal interpretéis, pues yo soy el primero que quiero que aprendáis, pero también soy el primero que no quiero ver gente desanimándose por el camino.

Nota: para programar en la SDK de sony o el psl1ght, hacen falta unos buenos conocimientos de C, asi que, ¡toca empaparse de C primero!

Hasta el capítulo 5 no veremos un primer programa en ejecución ni haremos ninguno, pues primero es necesario entender un par de conceptos.

Si queréis, para entrar en calor y ver que pretendo que sepáis hacer, como mínimo, cuando acabe este tutorial, os dejo un video mio también, donde explicaba como crear una DLL para hackear juegos (de ordenador). Aun que el lenguaje en que se explica en el video es C++, se ven un par de conceptos interesantes.
Si no sabéis programar nada, el vídeo os aburrirá, así que, pasad al tutorial xd
[C++] [VideoTutorial] Hack DLL [Nivel: Medio]
Mas adelante, cuando creemos nuestro primer programa, utilizaré este vídeo como referencia, pero sois libres de usar el programa que queráis para programar.

2. Variables y tipos de variables
Primero de todo, ¿que es una variable? Una aproximación teórica podría ser, por ejemplo y para que sea fácil de entender, un "contenedor" donde se puede guardar un valor. Asi por ejemplo,

CÓDIGO: SELECCIONAR TODO

```
int variable1;
```

Seria una variable.

CÓDIGO: SELECCIONAR TODO

```
variable1 = 10
variable1 = 5
```

Aquí vemos como en el primero caso, la variable "variable1" le guardamos un valor de 10, y luego le guardamos un 5.
Esto es básicamente una variable, algo que nos permite guardan en ella valores.
Para dar un nombre a una variable, en el anterior ejemplo "variable1" se deben seguir unas normas muy simples.

1. Una variable debe empezar con una letra, es decir, una variable como "1variable" no es válida y provocaría un error.
2. Una variable no puede contener caracteres especiales (como "#¢∞¬" etc.), a excepción de "_" pues "variable_1" si sería correcto
3. Una variable no puede contener espacios, pues "variable 1" daría error.
4. Hay nombres reservados que no pueden ser usados para nombrar variables (por ejemplo: int, char, public, private, class, void, y algunos otro que ya iremos viendo).
5. El lenguaje C distingue entre mayúsculas y minúsculas, no siendo lo mismo, pues "VARIABLE1" que "variable1" ni "VaRiAbLe1"
6. No podemos tener 2 variables con el mismo nombre

Con esto ya podemos dar nombre a las variables, pero, en C existen tipos de variable, es decir, si queremos que una variable contenga un número no la declararemos (declarar: acto de escribir o programar una variable) del mismo modo que haríamos con una que fuera a contener, por ejemplo, letras. Hagamos una breve referencia a los diferentes tipos de variable, y luego pondremos ejemplos y explicaciones más detallas de su funcionamiento. Normalmente nos referimos a los tipos de variable por su nombre ingles, "Data Type".
Veremos a continuación los nombres "signed" y "unsigned", luego explicaremos que significa cada uno

```
char -> caracteres o números pequeños enteros

signed: -128 to 127
unsigned: 0 to 255

short int (short) -> variable numéricas enteros

signed: -32768 to 32767
unsigned: 0 to 65535

int -> variables numéricas

signed: -2147483648 to 2147483647
unsigned: 0 to 4294967295

long int (long) -> variables numéricas enteros

signed: -2147483648 to 2147483647
unsigned: 0 to 4294967295
Notese que en máquinas UNIX a 64 bits, esto equivale a long long, cuyo rango es (gracias
```

waninkoko):
-9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807

float, double, long double -> variables numéricas con decimales (de punto flotante)
Como las int se diferencia en el tamaño que pueden contener.

Algunos de vosotros os preguntaréis que significan los valores que hay debajo de los diferentes tipos de variable. Bien, eso es el tamaño máximo del número que pueden contener. Pongamos un ejemplo práctico

```
int variable1 = 1; //CORRECTO
int variable2 = 2147483649; //INCORRECTO!
long variable3 = 2147483649; //CORRECTO
```

Expliquemos algunas de las características de los siguientes ejemplos. Primero de todo y el más importante, ¿veis ese punto y coma (;)? En C, asi como en otros lenguajes como C++ o PHP, toda instrucción (instrucción: "pedazo" o función de código, aquí por ejemplo la instrucción es declarar la variable), antes de la siguiente instrucción debemos poner un ";", para indicar el fin de la anterior instrucción, de lo contrario nos daría error.
Luego, supongo que habéis visto esos "//", eso significa un comentario. Los comentarios son simples textos que son ignorados por el compilador (compilador: programa que "convierte" el código en un archivo "ejecutable".). Los comentarios pueden contener todo lo que quieras y se pueden poner donde quieras, pero existe una única limitación, todo lo que vaya a continuación de los // formará parte del comentario **hasta fin de linea**. Para evitar esta limitación existe otro tipo de comentario

CÓDIGO: SELECCIONAR TODO

```
/*
TODO ESTO QUE HAY AQUI
es un comentario, incluido si
saltamos de linea
*/
```

Es el que sigue se encuentra entre /* aquí */. Nótese que mediante este tipo de comentarios podemos hasta hacer lo siguiente.

CÓDIGO: SELECCIONAR TODO

```
int /*esto es una variable numérica*/ variable1;
```

ya que sería lo mismo que

CÓDIGO: SELECCIONAR TODO

```
int variable1;
```

NOTA: Los comentarios con // son propios de C++ mientras que los de /* */ son propios de C. Aunque hoy en día un buen compilador te aceptará ambos sin ningún

problema.

Pero, volvamos al ejemplo que hemos hecho hace un momento. ¿Porqué el segundo ejemplo pone INCORRECTO? Mirad otra vez el cuadro de los Data Type, ¿veis el limite de las variables int? ¡Lo estamos superando!
La solución es declararlo como "long".
Ese ejemplo no tiene mucho más a decir, asi que, proseguimos a explicar que es eso de "signed" y "unsigned" que ponía allí. Para aquellos que no lo hagáis deducido ya, la diferencia es muy simple, las variables "unsigned" no pueden contener valores negativos, y los "signed" pueden contener valores negativos y positivos.
Por defecto, si no se pone de que tipo es, se declaran como signed. Veamos ahora un ejemplo de como poner el signed y el unsigned.

CÓDIGO: SELECCIONAR TODO

```
signed int variable1;
unsigned int variable2;
```

NOTA: Para aquellos que hagáis programado ya anteriormente, en C no existe el tipo de variable "bool". En posteriores capítulos veremos soluciones más o menos complicadas a este "problema"

Para declarar una variable y darle un valor lo podemos hacer asi:

```
int variable1 = 2; //donde lo guardamos directamente
//o asi, que es lo mismo
int variable2;
variable2 = 2;
```

Notese, que el la segunda parte, como ya hemos declarado la variable en la primera línea, en la segunda ya no volvemos a poner de que tipo es (en este caso int)

Explicaremos ahora con más detalle los otros tipos de variables, puesto que los int no tienen demasiado secreto.

2.1 - Tipo char

Bueno, para hacerlo fácil, primero de todo tenemos que entender que cada letra del abecedario tiene un equivalente numérico. Podemos ver una lista de referencia en esta web:http://www.asciitable.com/
Pongamos un ejemplo usando la letra 'a'.

```
char variable1 = 'a';
char variable2 = 97;
```

Si miramos la tabla anterior, veremos que a == 97, por lo que variable1 == variable2;

(En posteriores capítulos explicaremos porque aquí ponemos ==, y no =, puesto que no es un error).
Daros cuenta que al asignar el valor a variable1, hemos usado comillas simples (') para delimitar la letra 'a'. Por lo general en C, como en cualquier lenguaje de programación, las cadenas de letras se delimitan por comillas dobles ("), pero el caso de char es especial, ya que lo que queremos indicar es el valor númerico de la letra, no la letra en sí misma. Por lo demás, las variables char se comportan de la misma forma que una variable númerica como "int".

2.2 - Tipo float
Esta variable tiene un comportamiento muy similar, sino igual, a las variables int. La única diferencia existente con estas, es que una variable float admite decimales, cosa que no hace una int.
Veamos un ejemplo que lo demuestre:

CÓDIGO: SELECCIONAR TODO

```
int variable_1 = 1.5;
int variable_2 = 1.6;
float variable_3 = 1.653;
```

Primero de todo y lo más importante, ya que puede conducir a errores tontos. En C los decimales siguen a un punto (.), puesto que se usa el sistema inglés.
Veamos la variable_1, hemos dicho que no podemos ponerle decimales, ¿verdad?, entonces, os preguntaréis, ¿que pasa aquí? La respuesta es simple, el compilador va a coger ese numero y, simplemente, quitarle los decimales. Es decir, tanto variable_1 como variable_2 se convertirán en, 1.
En cambio, variable_3, al aceptar decimales por ser del tipo float, permenecerá intacta.
Aun que no lo haya puesto en la tabla de arriba, las variables float también tienen un limite máximo. Para más información, buscad en google Data Type, y obtendréis la respuesta.

3. Operaciones con variables numéricas
Primero de todo introduciremos un concepto que recibe el nombre de "Type Casting". ¿Suena a chino no?, pero en realidad, es algo muy común y necesario. (NOTA: los ejemplos siguientes, para gente que ya ha programado, me dirá que es innecesario hacerlo, pero, van bien para explicar). Imaginaros que tenemos una variable float, con sus respectivos decimales, y decidimos que queremos pasarla a int. ¿Y como hago eso? En un principio, si dos variables son de tipo diferente, no podemos igualarlas por las buenas. Pongamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
float variable1 = 16.5;
int variable2;
variable2 = variable1;
```

(Repito, gente que ya haya programado en C o C++, absteneros de comentarios no constructivos aquí).

Bien, esto en un principio no seria posible, ya que las variables son de tipo diferente, y por lo tanto, no podemos hacer esto, que consiste en darle a la variable1 el valor que tiene la variable2, es decir, seria lo mismo que escribir variable2 = 16.5, puesto que variable1 hemos dicho que valia 16.5
Pero, existe un método mediante el cuál si es posible hacerlo:

```
float variable1 = 16.5
int variable2;
variable2 = (int)variable1;
```

Nótese ese (int), importante que esté entre paréntesis. Eso significa que queremos que primero convierta el 16.5 ha int, es decir, quedaria 16, y luego, ese 16 lo guarda en variable2. Algunos compiladores modernos, quizás se quejen de que esto puede provocar perdidas de datos, pero si lo pensamos bien, es normal, puesto que perdemos 5 decimales (.5) por el camino.
Quizás ahora esto suene raro, pero a base de practicar, es muy fácil y intuitivo.
Cabe decir que la variable sigue siendo de su tipo original, el type casting solo se aplica allí donde se pone, por lo demás, la variable se comporta igual.

Prosigamos pues, a explicar como hacemos operaciones con variables numéricas. En este caso, pondré ejemplos, y explicaré solo en detalle aquellos que me parezcan un poco más complicados de explicar. Por ahora solo pondré operaciones sencillas, echas con números de base 10 (los de toda la vida), y no tocaré nada del tema bits, eso irá más adelante.

```
//EJEMPLO 1
int variable1 = 10;
int variable2 = 2;
int resultado;
resultado = variable1 + variable2; //resultado = 10 + 2 = 12
resultado = variable1 - variable2; //resultado = 10 - 2 = 8
resultado = variable1 * variable2; //resultado = 10 * 2 = 20
resultado = variable1 / variable2; //resultado = 10 / 2 = 5
//Para poder hacer el siguiente ejemplo, hace falta incluir un archivo. En el capítulo 5 veremos como se incluye
//#include <math.h>
resultado = pow(variable1, variable2); //resultado = 10 ^ 2 = 100, potencia de 10 a la 2

//EJEMPLO 2
int variable1 = 10;
int variable2 = 2;
int resultado = variable1 * variable2; //Lo mismo que antes, pero directamente

//EJEMPLO 3
int variable1 = 10;
int variable2 = 5;
int variable3 = 2;
int resultado1 = variable1-variable2*variable3; //resultado1 = 0 = 10 - 5 * 2
/* En C, como en cualquier lenguaje de programación, a no ser que nosotros pongamos paréntesis, se respecta el orden de operaciones según el signo */
int resultado2 = (variable1-variable2)*variable3; //resultado = (10-5)*2 = 5*2 = 10
```

```
//EJEMPLO 4 - UTILIZANDO DIFERENTES TIPOS DE VARIABLE
int variable1 = 2.5; //Sabemos que esto acabará siendo un simple 2
int variable2 = 2.5; //Sabemos que esto acabará siendo un simple 2
int resultado = variable1 * variable2; //resultado = 2 * 2 = 4;

//Probemos ahora con una simple modificación
float f_variable1 = 2.5;
float f_variable2 = 2.5
//Sabemos que en ambas se guarda el resultado
int resultado = (int)(f_variable1 * f_variable2); //resultado = (int)(2.5 * 2.5) = (int)6.25 = 6;

/*A diferencia del caso anterior, con float se ha guardado el resultado, y aunque lo convertimos a int, tal como vemos eso es el último paso, y en ese último paso, ya tenemos el 6, simplemente, le quitamos los decimales.
Vemos pues, como puede variar mucho entre un caso y entro, ¡son 2 enteros de diferencia!*/
```

Más cositas, quizás ahora no tiene mucho sentido, pero lo tendrá en el capitulo 10:

```
variable < numero (menor que)
variable > numero (mayor que)
variable <= numero (menor o igual que)
variable >= numero (mayor o igual que)
```

Ejemplos

CÓDIGO: SELECCIONAR TODO

```
int var = 8;
//var < 9 (puesto que 8 menor que 9)
//var > 1 (puesto que 8 mayor que 1)
//var <= 10 (puesto que 8 es menor ([b]o[/b] igual) que 10
//var <= 8 (puesto que 8 es menor ([b]o[/b] igual) que 8
//var >= 2 (puesto que 8  es mayor ([b]o[/b] igual) que 2
//var >= 8 (puesto que 8 es mayor ([b]o[/b] igual) que 8
```

4. Array

No se si lo habéis pensado ya, pero, con los tipos de variables anteriores, ¿como podríamos, por ejemplo, poner una sola cadena de texto, como "hola mundo" en una sola variable? Bien, ahora veremos la respuesta.

Imaginaros por un momento una cajonera. En sí misma es una sola cosa, pero en "detalle" puede tener 2, 3, 4... cajones. Esto es una array.
Vamos a hacerlo prácticamente. Recordáis la variable tipo char? Pues la vamos a utilizar para ilustrar el ejemplo. (Esto que viene a continuación es solo para que se entienda, no es necesario para formar una array). Si quisiéramos poner el "hola mundo" en variables tipo char, haríamos esto:

```
char letra1 = 'h';
char letra2 = 'o';
char letra3 = 'l';
char letra4 = 'a';
char letra5 = ' ';
etc.
```

Pero como comprenderéis, el día que tengáis que hacer lo mismo con un texto de 1000 letras, os vais a morir en el intento. Ahora imaginaros que estas variables, eran los cajones de la dicha cajonera. ¿No podríamos unirlos, y hacer de ello una sola variable? Sí:

```
char texto[] = "Hola Mundo";
```

Fijaros, ¿que ha cambiado aquí? En primer lugar, el texto está delimitado por comillas dobles, muy importante. En segundo lugar, detrás del nombre de la variable, texto, encontramos unos corchetes []. En realidad, allí en medio habría un número. Ese número equivaldría al número de letras más 1, contando desde 0. Pero C, y C++, tienen la ventaja, de que si inicializamos la variable (le guardamos un valor) al mismo tiempo que la declaramos, podemos omitir ese número.
Fijaros en la afirmación anterior: "[...]**contando desde 0**[...]". Las arrays no empiezan en el número 1 como sería de pensar, sino que empiezan por el 0. Si descomponemos la anterior array, tendríamos esto:

CÓDIGO: SELECCIONAR TODO

```
texto[0] = 'H'; //Pensad que cada uno de los "cajones" es una variable
char, en conjunto, un char array.
texto[1] = 'o'; //Y así hasta llegar a:
texto[10] = 'o';
```

Pero, revisemos otra vez la afirmación "[...] al número de letras **más 1**[...]". ¿Y ese más 1? Resulta que en C, y C++, es necesario indicar que la cadena de texto se ha acabado. En el caso anterior, donde lo inicializamos al momento, no es necesario añadir nada más, pero si siguiéramos descomponiendo esa array, encontraríamos esto:

CÓDIGO: SELECCIONAR TODO

```
texto[11] = '\0'; //Esto, \0, indica final de una cadena
```

Podríamos entonces decir que lo siguiente es lo mismo:

CÓDIGO: SELECCIONAR TODO

```
char texto1[] = "Hola Mundo";
char texto1[11] = "Hola Mundo";
```

¿Y que pasaría si en vez de 11, pongo 21? Absolutamente nada (aparte de que ocuparás más memoria), porque recuerdas ese '\0', seguirá estando en 11, no en

21.
¿Y si pongo 10? Error como una casa xd

De la misma forma que podemos hacer arrays con variables char, las podemos hacer con cualquier otro tipo.

CÓDIGO: SELECCIONAR TODO

```
int arraydeint[10];
arraydeint[0] = 1;
arraydeint[1] = 2;
etc.
```

¿Y no puedo poner en esta algo del tipo "111111"? No, porque recuerda que las comillas dobles (") indican cadena de texto, y esto deben ser números. ¿Y entonces, si quiero inicializarlo directamente?

CÓDIGO: SELECCIONAR TODO

```
int arraydeint[] = {0,1,2,3,4,5,6};
//Lo cual es lo mismo que hacer:
//arraydeint[0] = 0;
//arraydeint[1] = 1;
//etc.
```

5. Nuestro primer programa - Hola Mundo
Para realizar este capítulo es necesario de disponer de alguna plataforma en que programar, o al menos, un compilador. Una fácil de usar y muy útil, porque corrige los errores de sintaxis (sin taxis quiere decir, por ejemplo, las ; al final de instrucción) usaremos el Visual C++ Express 2010. Lo podemos encontrar aquí:
http://www.microsoft.com/express/Downloads/
Buscamos Visual C++ 2010 Express, escogemos idioma, descargamos e instalamos.
Quizás os preguntaréis porqué usamos uno en cuyo nombre pone C++ y no C.
Resulta que un compilador que compila C++, también va a compilar C.
Una vez instalado, le damos a (algo parecido a lo que diré, ya que yo lo tengo en inglés):
Inicio->archivo->nuevo
Le damos a "Proyecto Vacío" en la nueva ventana que saldrá y le damos un nombre
Una vez ya en el entorno de programación, es decir, la pantalla principal, le damos a:
Proyecto->Añadir nuevo elemento
Seleccionamos en la nueva ventana "Archivo C++ (.cpp)" le damos un nombre, normalmente recibe el nombre de "main" ya que es el archivo principal (main, inglés = principal) y le damos a "Añadir".
Notese que normalmente los archivos en C no acaban en .cpp, ya que dicha terminación es propia de los archivos C++, tal y como dice allí, sino que acaban en .c. Pero, para no complicarnos la vida, lo dejaremos en .cpp, que tampoco pasa nada, va a compilar igual.

Bien, todo esta listo. Si no se ha abierto ya el archivo "main.cpp" lo abrimos desde el explorador de la izquierda.

Empecemos a programar. Primero de todo, tenemos que plantearnos una pregunta, ¿como haremos para enseñar un texto en una pantalla? Por suerte, esto ya viene hecho, y todo lo que tenemos que hacer, es incluir un archivo que tiene dicho código ya programado.
Para este programa, bastará que incluyamos el archivo "stdio.h", lo cuál haremos de la siguiente forma:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>
```

Bien, ahora ya tenemos lo esencial para que el programa pueda funcionar. Pero seguimos teniendo un problema, el código no lo puedes escribir por ahí libremente, debe estar dentro de una **función**. De momento no entraremos en detalle sobre que es una función, simplemente debemos saber que en C, siempre debe existir la función "main". Esta es la primera función que se ejecutará de nuestro programa. De no existir, tendríamos un grave error y problema. Insisto en que por ahora no explicaré como funcionan las funciones, así que, escribiremos esto (en una nueva linea):

CÓDIGO: SELECCIONAR TODO

```
int main(){
//El código va aquí, entre las llaves {}.
return 0;
}
```

Ya que he dicho lo de una nueva línea, en C los espacios en blanco, tanto antes de una instrucción como después, como los saltos de línea, no influyen en nada, es una mera cuestión de estilo y entendimiento. Pues, es mucho mas fácil de entender:

CÓDIGO: SELECCIONAR TODO

```
int main(){
     //El código va aquí
     return 0;
}
```

Que no el anterior ejemplo, ya que podemos ver la jerarquía, es decir, que ese comentario está dentro de una función.
¿Veis ese return 0; del final de todo? Eso simplemente indica que el programa ha terminado, y que se cierra bien (en este caso, más adelante veremos para que sirve realmente la palabra return).
Vale, ¿quién se acuerda de como poníamos en una variable un texto entero? Esto es lo que vamos a hacer ahora, simplemente tenemos que escribirlo dentro de la función main:

CÓDIGO: SELECCIONAR TODO

```
char texto[] = "Hola Mundo";
```

Perfecto, ahora, como nos lo montamos para que este texto salga en la consola (cuando ejecutéis el programa, ya veréis que es una consola). Pues utilizando la siguiente función:

CÓDIGO: SELECCIONAR TODO

```
printf(/*variable o texto*/);
```

Aplicado quedaría

CÓDIGO: SELECCIONAR TODO

```
printf(texto);
```

Si probáramos de ejecutar ahora el programa compilado, no llegaríamos a leer el texto, por el simple hecho de que printf se limita a mostrar el texto, pero recordemos que a continuación tenemos el return 0; que para el programa.
Una bonita forma de evitar esto, es usando lo siguiente:

CÓDIGO: SELECCIONAR TODO

```
getchar();
```

Esto permite que el usuario (tú) escribas algo en la consola, y hasta que no pulses [ENTER] no continua. En este caso, continuar equivale a terminar el programa.

Para probar el programa, en el visual c++ podemos compilar pulsando F5, y automáticamente, si no hay errores, se abrirá el programa, o con Depurar->Empezar a depurar

El programa acabado quedaría de la siguiente forma:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>

int main(){
   char texto[] = "Hola Mundo";
   printf(texto);

   getchar();

   return 0;
}
```

Otra solución alternativa al tema de pausar la consola, con getchars() es el uso de system("pause"), para ello debemos incluir un nuevo archivo justo debajo del anterior

CÓDIGO: SELECCIONAR TODO

```
#include <stdlib.h>
```

Y substituir el getchar(); por: system("pause");

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>
#include <stdlib.h>

int main(){
   char texto[] = "Hola Mundo";
   printf(texto);

   system("pause");

   return 0;
}
```

Os animo a que experimentéis un poco vosotros con lo ya dado, y que hagáis pequeñas modificaciones a este "Hola Mundo".
Os quiero dar un regalito, antes de irme a la cama

CÓDIGO: SELECCIONAR TODO

```
char resultado[100];
int numero = 5;
sprintf(resultado, "Esto es una cadena de texto donde le añadimos un numero: %d", numero);
//ahora resultado es: Esto es una cadena de texto donde le añadimos un numero: 5
Si hacemos printf(resultado), veremos en pantalla dicho texto.
¡Probad cosas ahora, antes de pasar al siguiente capítulo. (No temáis que en el siguiente os explico esto con más detalle, ¡pero siempre va muy bien experimentar por uno mismo las cosas primero!)
```

6. Punteros I. Introducción

En este capítulo veremos un "tipo de variable" llamados puntero, del inglés "pointer". Os tengo que pedir varias cosas en este capítulo:
Es un capítulo denso, aunque intentaré hacerlo lo más sencillo y entendedor posible. Si no lo entendierais, releerlo, pero hacedlo a conciencia. Si aun así no conseguís entenderlo, enviad un post con la duda.

Empecemos pues, primero de todo debemos entender el concepto dirección de memoria. La memoria de un ordenador es el "sitio" donde nuestro código se ejecuta. Esta memoria va des de un punto 0, hasta un punto X que es el máximo, no entro en detalles en esto. Entonces, una dirección de memoria es la posición en la que se encuentra una orden (orden: "pedazo" de código, instrucción). Estas direcciones no están en el sistema numérico habitual, sino que se encuentran en sistema hexadecimal. ¿Y que significa esto? Esto significa que "no hay 10 números" sino 16:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
Para poder explicar el ejemplo que pondré, aprovecho para deciros que, tanto en C como en C++ siempre que vayamos a escribir un número en hexadecimal, este debe ir precedido de "0x" (sin las comillas).
Para poder entender el siguiente ejemplo, debo hablaros sobre el tipo de variable "BYTE", "WORD" y "DWORD" (entre otros). En realidad, estes solo existe en C++, pero, podemos "usarlo" tambien en C.
En C, un BYTE equivale a un "unisgned char", lo cual si conocemos.
En C, una WORD equivale a un "unsigned short", y esto si sabes que es, ¿verdad?
En C, una DWORD equivale a un "unsigned long".
Los que hayáis trabajado ya en C++, acostumbraros a los unsigned short y unsigned long.

Vamos a ver, ¿y porqué exite algo llamado BYTE, WORD y algo llamado DWORD, no bastaría con solo 1? El caso, es que estas variables también tienen un rango, mirad la tabla superior. Normalmente usaremos los unsigned long, asi que, no nos preocuparemos más por este tema.
¿Todos sabemos que es un BYTE? Bueno, para no complicarnos, un diremos que un byte es dos numero en hexadecimal. Sabiendo esto, decimos que una "unsigned short" esta formada por 2 bytes. O_o ¿Y que me estás contando, no? xd

```
0xAABB

0xAA es un byte
0xBB es otro byte
0xAABB es un "unsigned short"
```

Un "unsigned long" es el doble que un "unsigned short", es decir, 4 bytes:

```
0xAABBCCDD

0xAA es un byte
0xBB es otro byte
0xCC es otro byte
0xDD es otro byte
0xAABBCCDD es un "unsigned long"
```

Vale, para entender esta empanada mental que os acabo de soltar, vamos poner un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
unsigned long hexa_1 = 0x0000A; //realmente, esto es lo mismo que poner 0x0A, RECORDAT, que un BYTE son 2 numericos, asi que, poner 0xA estaria mal!
unsigned long hexa_2 = 0x00006; //0x06
//vamos a sumarlo, puesto que son numeros, podemos hacerlo
unsigned long resultado = hexa_1 + hexa_2;
/*
resultado = 0x06 + 0x0A;
resultado = 0x10
Que acaba de pasar aquí, ¿no?
Pensemos que esto es una suma decimal de toda la vida, (los números no son equivalentes)
suma = 8 + 2, que es lo que pasa aquí, cuando llegamos a 9, si le sumamos otro, pasa a ser 10, no? allí es lo mismo, tenemos
0x0A + 0x01 = 0x0B
0x0B + 0x01 = 0x0C
0x0C + 0x01 = 0x0D
0x0D + 0x01 = 0x0E
0x0E + 0x01 = 0x0F
0x0F + 0x01 = 0x10
cuando llegamos a F, lo que hacemos es pasar a 1 y dejar un 0, lo mismo que con decimal.
*/
```

Esto no ha acabado aun... la siguiente parte explicaré las variables puntero. Lo haré con imagenes si puedo, para que sea más fácil de entender, de momento me voy que he quedado. Espero que esta parte sea entendible, y os animo a practicar con ella. Por cierto, si queréis probarlo en un programa real, os dejo esto: =P

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>

int main(){
   unsigned long hexa_1 = 0xAABBCCDD;

   char texto[100];
   sprintf(texto, "0x%x", hexa_1); //Copiamos en la variable texto el hexa_1
   //NOTA: el %x se sustituye por hexa_1, pero no pone el 0x, por eso lo ponemos nosotros delante
   //es simple estética, ¡no influye en nada!
   printf(texto); //va a mostrar 0xAABBCCDD

   getchar();

   return 0;
}
```

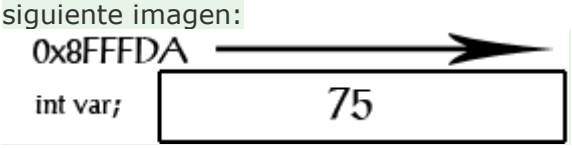
Nota: ¡La función sprintf está explicada a en capítulo 7!, realmente, su uso es guardar algo en una variable, por lo que en el anterior ejemplo también podríamos hacer esto:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>

int main(){
   unsigned long hexa_1 = 0xAABBCCDD;

   printf("0x%x", hexa_1); //va a mostrar 0xAABBCCDD

   getchar();

   return 0;
}
```

Esto se aplica a cualquier ejemplo donde haya usado o use sprintf y posteriormente haga un printf(...); Mi intención era simplemente mostraros como guardar un texto dentro de una variable, aunque ocupe más espacio en la memoria. El 2o ejemplo hace lo mismo en menos espacio y gastando menos memoria.

Vamos a empezar con lo que realmente pretende enseñar este capítulo. Veamos la siguiente imagen:



¿Que vemos en esta imagen? Primero de todo deberías poder identificar una variable de tipo int con nombre "var". El número en hexadecimal de arriba es la dirección (imaginaria) de memoria donde se encuentra esta variable, y lo que hay dentro del recuadro es el contenido de dicha dirección, es decir, de la variable. En este caso es una simple variable, y no tiene mas misterio que eso.
Si nosotros quisiéramos ver el contenido de esa variable, simplemente utilizaríamos métodos anteriores como:

CÓDIGO: SELECCIONAR TODO

```
char texto[100];
int var = 75;
sprintf(texto, "%d", var);
printf(texto);
```

Pero, y si lo que queremos ver es la dirección de memoria donde se encuentra esta variable? La respuesta es el signo &. Veamoslo con el ejemplo anterior.

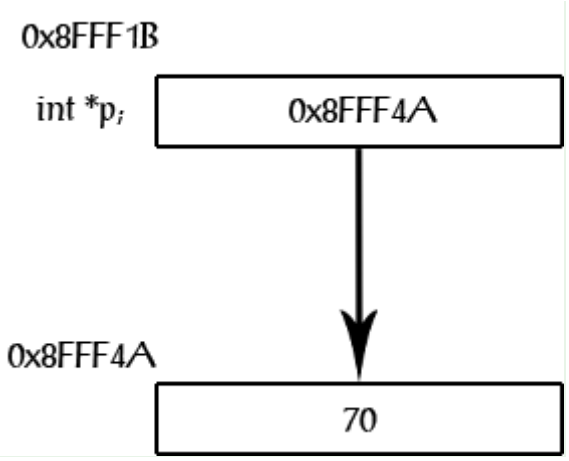
CÓDIGO: SELECCIONAR TODO

```
char texto[100];
int var = 75;
```

```
sprintf(texto, "La variable tiene un valor de: %d ; y está en la
dirección de memoria: 0x%x", var, &var);
printf(texto);
```

Vamos a ver, la función sprintf, que en el siguiente capitulo veremos con más detalle, sustituye los %d por variables numéricas, y las %x por variables numéricas en hexadecimal por el orden en que las encuentra, aquí lo que hace es (%d = var) , (%x = &var).
Veis el & que os decía? simplemente, en vez de devolver el valor de var, nos devolverá la dirección.

El caso de los punteros (punteros de memoria es su nombre concreto) es similar al anterior. Un puntero, tal y como dice su nombre, no contiene un valor, sino que contiene una dirección de memoria.
Veamos otra imagen:



Lo primero que os debería llamar la atención es como declaramos la variable, ¿veis que antes del nombre hay un asterisco (*)? Ese asterisco indica que la variable en cuestión es un puntero. Luego, el contenido de la variable, tal y como podéis ver, es una dirección de memoria. Si vamos a esa dirección, encontramos un valor.
Vamos a adaptar el anterior ejemplo:

CÓDIGO: SELECCIONAR TODO

```
int var = 70;

//Hemos dicho que un puntero se declaraba con *
int *p;

/*Debemos hacer que el puntero p, tenga un contenido, de lo contrario el
programa final daría error (atención, el compilador no nos diría nada,
pero obtendríamos un bonito crash de no darle un valor.
El valor debe ser una dirección de memoria que contenga algo, y que
casualidad, justo arriba tenemos una variable con un contenido, ¿como
podríamos obtener su dirección? Exactamente, con &
p = &var;
/*Fijaros, el * solo lo utilizamos cuando declaramos el puntero, poner
un asterisco aquí haría otra cosa que explicaremos a continuación*/
```

```
//Vamos a volver a mostrar un texto
char texto[200];
sprintf(texto, "Contenido var: %d, dirección de var: 0x%x, contenido del puntero: 0x%x", var, &var, p); /*Otra vez p sin *, ya que queremos ver su contenido*/
//Vereis que ahora, la dirección de var y el contenido de p, es lo mismo.
printf(texto);
```

Vale, supongo que ahora mismo no le veis ninguna aplicación a esto, tranquilos, es normal. En posteriores capítulos empezaremos a ver aplicaciones de los punteros. Para terminar este capítulo, ¿qué haríamos si quisiéramos leer el contenido de la dirección donde apunta (un puntero apunta a una dirección, es decir, el contenido del puntero es la dirección a la que apunta)? Bien, ¿os acordáis de que en un ejemplo he puesto varias veces "aquí no ponemos * porqué tendría otra utilidad? Pues esta es, si nosotros tenemos un puntero, y queremos ver que contiene la dirección donde apunta, debemos ponerle un * delante:

CÓDIGO: SELECCIONAR TODO

```
int var = 70;
int *p;
p = &var;

char texto[300];
sprintf(texto, "Contenido var: %d, contenido de la dirección en que apunta el puntero p: %d, dirección de var: 0x%x, contenido del puntero: 0x%x", var, *p, &var, p);
/*Fijaros en la segunda variable que se sustituye, pone *p, justo lo que os explicaba.*/

printf(texto);
```

7. Operando con las cadenas de texto

Para poder entender este capítulo debemos haber entendido el anterior, en caso contrario pueden pasar dos cosas:
A. Que este capítulo te ayude a entenderlo y al final se te quede
B. Que no entiendas nada y te rayes.

¿Os acordáis de que dije que había dos formas de declarar una array de char?

CÓDIGO: SELECCIONAR TODO

```
char texto[] = "Hola Mundo";
//y
char texto[11] = "Hola Mundo";
```

Bueno, en realidad hay otra que ahora ya os puedo explicar gracias al capítulo

anterior.

CÓDIGO: SELECCIONAR TODO

```
char *texto = (char*)malloc(11);
```

Primero de todo y lo más importante, para poder usar la función malloc, que ahora os explicaré que hace, debemos incluir el archivo

CÓDIGO: SELECCIONAR TODO

```
#include <malloc.h>
```

¿Qué es malloc(tamaño);? Se trata de una función que reserva un espacio en la memoria y devuelve la dirección donde empieza dicha reserva. Analizemos esto. Malloc nos devuelve una dirección donde podemos guardar un contenido, y la única cosa que nos sirve para almacenar direcciones con contenido son los punteros ¿no? Es mas, queremos que sea un texto, así que, ¿porqué no hacemos un puntero de tipo char?
Luego, os fijáis que malloc pide un tamaño, (11), ¿de donde sale ese 11? Quizás aquí parezca obvio porqué el texto tiene 11 caracteres (recordad, empezamos por 0 y le sumamos 1 debido al \0), pero, no es tan obvio.
En realidad, allí debería poner

CÓDIGO: SELECCIONAR TODO

```
char *texto = (char*)malloc(11 * sizeof(char))
```

¿... sizeof(variable o tipo variable)? Sí, la función sizeof nos devuelve el espacio de memoria que ocupa un tipo de variable en concreto. Lo que pasa que un "char" ocupa 1 espacio de memoria, es decir, queda como: malloc(11*1), lo cual es: malloc(11);
Pensadlo, es una array de char, por lo que está formada por variables tipo char, y cuantas de ellas tenemos, 11. ¿Y si tuviéramos un puntero de int?

CÓDIGO: SELECCIONAR TODO

```
//Esto es solo para explicar el sizeof
int *punteroaint = (int*)malloc(1 * sizeof(int));
*punteroaint = 10; //Recordad, este * aquí implica que estamos editando el valor de la dirección donde apunta, ¡no el puntero!
```

Sigamos, ¿porqué hay un (char*) antes de malloc? Si recordáis lo del "Type Casting", eso lo explica todo. La función malloc no devuelve algo del tipo (char*) porque, imaginaros que quisiéramos guardar espacio para un puntero int (int*), no habría forma de convertir ese (int*) a (char*) por más typecasting que hiciéramos. En cambio, nos lo devuelve en un tipo llamado (void*). Para entendernos, cuando hablamos de variables y no de funciones, void* es un tipo "universal".

Vale, ahora tenemos una variable llamada texto, que es un puntero char, con un espacio de memoria. Pero, no tiene texto. ¿Como podemos ponerle texto? Primero

de todo debemos incluir otro archivo:

CÓDIGO: SELECCIONAR TODO

```
#include <string.h>
```

Nota: normalmente una cadena de texto es referida por string, pero no confundáis esto con un tipo de variable exclusivo de C++ que tiene el mismo nombre.
Luego, podríamos darle un valor a texto mediante:

CÓDIGO: SELECCIONAR TODO

```
strcpy(texto, "Hola Mundo");
```

Esto simplemente borra todo lo que haya en texto, y le guarda "Hola Mundo".

¿Es realmente necesario que para usar la función strcpy() inicialicemos las variables de ese modo? No, realmente no lo es, también podríamos hacerlo con "char texto[11];" y luego usar strcpy.

Hagámoslo "más complicado". Imaginad que primero queremos ponerle solo "Hola ", y luego ponerle "Mundo". Si usamos strcpy() no podremos, ya que como ya he dicho, primero borra cualquier cosa que haya en la variable, en cambio haremos esto:

CÓDIGO: SELECCIONAR TODO

```
strcpy(texto, "Hola ");
strcat(texto, "Mundo");
```

Otra función más a la lista de funciones! strcat() simplemente coge una variable de tipo char* y le añade otro texto a continuación, con lo que nuestra variable "texto" ya quedaría con "Hola Mundo".

Aquí vemos nuestro Hola Mundo complicado al completo:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>

int main(){
   char *texto = (char*)malloc(11);
   strcpy(texto, "Hola ");
   strcat(texto, "Mundo");
   printf(texto);

   getchar();
```

```
   return 0;
}
```

Quizás alguien se ha preguntado una cosa, porque cuando usamos strcpy, strcat o printf no usamos "*texto" y usamos "texto", puesto que es un puntero, y lo que queremos es editar el contenido de la dirección donde apunta, no el contenido del puntero, ¿no? La cuestión reside en que esas funciones nos piden explícitamente una variable de tipo (char *) y no de tipo (char) (ya que *(char *) = char), es decir que tenemos que usar el puntero, "texto". ¿Pero entonces, porqué tambien podemos utilizar las array de char, tipo texto[11]? Digamos que una array se comporta como un puntero, pero ahora no voy a entrar en detalles sobre este tema.

Para finalizar este capítulo y cumplir una promesa que he hecho en anteriores ejemplos, os explico la función sprintf.
Aun y parecerse a printf(), esta hace una cosa distinta. Mientras que printf nos enseña algo en la pantalla directamente, sprintf nos guarda un texto en una variable. ¡Como lo hace?, veamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
char texto[100];
char mundo[] = "Mundo"
sprintf(texto, "Hola %s, este es mi %d ejemplo", mundo, 5);
```

Veis todas esas letras que están a continuación del %, bien, sprintf se dedica a substituir esos "códigos" por el valor de variables. Lo hace siguiendo un orden, que es básicamente, lo que sigue al texto, el 2o parámetro (parámetro: cada uno de los valores o variables que ponemos en una función, aquí el 2o parámetro es la cadena "Hola %s, este....") hasta que se acaba la función.
Es decir, el primer %s se sustituye por el 3r parámetro, en este caso, mundo; y el %d se sustituye por el 4o parámetro, 5 (no hace falta que sea una variable, puede ser un valor directamente). Si hubiera otro %(algo) al final, ahora explico este algo, se substituiría por el 5o parámetro. Siempre pensad que va en orden, el 1r %(algo) con el 1r valor o variable (entendemos que el 1o es en realidad el 3o, tal y como he dicho arriba), el 2o con el 2o, etc.
El primer parámetro de todos de la función, en este caso "texto", es la variable tipo (char*) donde se guarda el resultado.

¿Porqué he puesto en un lado %s y en otro %d? Resulta que debemos especificar el tipo de valor por el cual vamos a substituir, veamos esta tabla:
(Pensad que la primera letra, iría después de un %; es decir, en la primera linea, sería %c)

c Carácter (char) a
d o i signed int 392
e Notación científica, X*10^Y 3.9265e+2
E Notación científica, X*10^Y 3.9265E+2
f float 392.65
g Use the shorter of %e or %f 392.65

G Use the shorter of %E or %f 392.65
o Signed octal (numero en sistema octal, base 8) 610
s string (cadena de texto) sample
u Unsigned int 7235
x Unsigned hexadecimal integer (lo que vendría siendo todos los hexadecimales) 0x7fa
X Unsigned hexadecimal integer (lo que vendría siendo todos los hexadecimales) 0x7FA
p Puntero de memoria B800:0000
n No se muestra nada. Debe ser un puntero a un signed int, donde se guarda el número de caracteres a escribir.
% Si escribimos %%, simplemente se subtituirá por un %, puesto que poner un solo % no es valido.

NOTA: No hay diferencia destacable entre aquellos iguales pero uno en mayúsculas y otro en minúsculas.
NOTA2: Haber si consigo subirlo en una imagen, porque aquí no se ve muy claro, ¿no?

Un Hola Mundo con sprintf podría ser:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>

int main(){
   char texto[11];
   char hola[] = "Hola";

   sprintf(texto, "%s %s", hola, "Mundo");

   printf(texto);

   getchar();

   return 0;
}
```

8. Funciones

Ya hemos visto, muy brevemente, en que consiste una función, pero ahora profundizaremos un poco más. Cada vez que nosotros hacemos algo del estilo "printf(texto);" estamos llamando a una función (del inglés "call a function"). Llamar una función no significa nada más que utilizar-la, en este caso la función es printf. Lo que escribimos entre los paréntesis () son los parámetros. Simplemente son valores que la función nos pide para poder funcionar.
También hemos visto ya como podemos nosotros crear una función, como es el caso de "main". ¿Os acordáis? Vamos a ver con mucho más detalle esto.

Este capítulo es vital para programar, ya sea en C o en cualquier otro lenguaje. Es muy importante que lo entendáis y que lo practiquéis mil veces.

La estructura de una función es como sigue:

CÓDIGO: SELECCIONAR TODO

```
/*
tipo_de_variable_que_devuelve   nombre_funcion ( parametros_funcion ){
      ...
      return variable_o_valor_acorde_al_tipo_de_variable_que_devuelve
}
```

Mmmm... apliquemos esto a la función main, para que quede más claro:

CÓDIGO: SELECCIONAR TODO

```
int main(){
      ...
      return 0;
}
```

¿De que tipo es esta función? int
¿Nombre de la función? main
¿Parámetros de la función? Ninguno, pues entre los () no hay nada. No es obligatorio que una función tenga parámetros
¿Qué es el return 0;? De momento dejo la idea en el aire, y luego veremos con más detalle que es esto del return. Como he dicho en el ejemplo anterior no, el otro, la función debe "devolver" un valor o variable acorde al tipo de función. Esta función es "int", asi que si devuelve un 0 está bien, puesto que 0 es un numero.

Vamos a declarar nuestra primera función. Es más, quiero que primero lo intentéis solos, y luego miréis como se hace. Imaginarios que os viene un empresario y os dice que quiere una función tipo "int", con 2 parámetros int, que la función sume esos parámetros, y devuelva el resultado. No os agobiéis, se que no es fácil. Si no sale, es normal, pues aun hay muchas cosas que no he explicado.

¿Lo habéis intentado? Vamos a ver como sería el código:

CÓDIGO: SELECCIONAR TODO

```
int sumar(int variable1, int variable2){
      int resultado;
      resultado = variable1 + variable2;
      return resultado;
}
```

Los parámetros, cuando declaramos una función, también deben ser declarados, por eso pone "int variable1". Luego, si queremos declarar otro parámetro, debemos poner una coma ",", y declarar normalmente el parámetro.
Tened en cuenta que estos parámetros solo son válidos dentro de la misma función, es decir, así como en la función luego hace "variable1 + variable2", si intentamos hacer lo mismo en otra función, sin declarar estas variables, nos daría error. Los parámetros de una función solo son válidos en la misma función.

"return resultado;": esto viene a significar que cuando se llega a este punto, ya no se hace nada más. Si nosotros hubiéramos puesto, "resultado = 0;" justo después del "return", eso no llegaría a pasar. En vez de esto, a partir de esto momento, la función "sumar" adquiere un valor, resultado. pongamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
int me_da_palo_hacerlo_a_mano;
me_da_palo_hacerlo_a_mano = sumar(2, 3);
//la función sumar hará, 2+3 = 5
//y el return quedará como, return 5;
//es decir, al final tendremos esto:
//me_da_palo_hacerlo_a_mano = 5;
```

En conjunto quedaría asi:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>

int sumar(int variable1, int variable2){
     int resultado;
     resultado = variable1 + variable2;
     return resultado;
}

int main(){
   int resultado = sumar(2, 3);

   printf("%d", resultado);
   //NOTA: printf utiliza el mismo sistema que sprintf, pero lo muestra
directamente en pantalla, tal y como ya se ha comentado

   getchar();

   return 0;
}
```

Fijaros que la función sumar(...) esta antes que la función main(), si lo hicierais al revés, daría error, ahora explico porqué y como solucionarlo.
Vale, imaginaros que hemos puesto la función sumar después de la función main. Esto en C es un error (a no ser que hagamos una cosa que luego explico) ya que en C, y C++, para utilizar algo primero tenemos que declararlo. En este hipotético caso, primero lo utilizamos, y luego lo declaramos (ERROR!).
Una perfecta solución sería cambiarlo de orden, pero, ¿y si no podemos o no queremos? La respuesta son los prototipos (del inglés "prototype"). Un prototipo es algo muy muy simple, pues consiste en coger la primera linea de la función, donde la declaramos, en este caso "int main(int variable1, variable2);" y ponerlo al principio, después de los include.
Fijaros en una cosa, después de los parámetros, no ponemos los {}, sino que

ponemos directamente un ;
Veamoslo en conjunto:

CÓDIGO: SELECCIONAR TODO

```
#include <stdio.h>

int sumar(int variable1, int variable2);

int main(){
   int resultado = sumar(2, 3);

   printf("%d", resultado);
   //NOTA: printf utiliza el mismo sistema que sprintf, pero lo muestra
directamente en pantalla, tal y como ya se ha comentado

   getchar();

   return 0;
}

int sumar(int variable1, int variable2){
     int resultado;
     resultado = variable1 + variable2;
     return resultado;
}
```

Vale, luego, se pueden hacer tantos tipos de funciones como tipos de variable. Eso experimentarlo vosotros. Yo solo os voy a hablar de un tipo en concreto, "void".

CÓDIGO: SELECCIONAR TODO

```
void funcion(){
     return;
}
```

Hay una cosa que, mirando lo que os he explicado antes, no debería cuadraros. Los parámetros están bien, pues hemos dicho que no tenían porque haber, pero, ¿y el "return;"? ¿Como es que solo pone return y nada más? Sencillamente, las funciones void no devuelven nada. Estas funciones están pensadas para reducir el tamaño del código, entre cosas. Imaginaros que en vuestro programa hacéis 50 veces esto:

CÓDIGO: SELECCIONAR TODO

```
char texto[] = "Hola Mundo";
printf(texto);
```

Quizás no es el mejor ejemplo, pero que es más cómodo, poner 50 veces todo eso, o 50 veces "funcion()" (o el nombre que le deis). Pues podemos poner todo lo

anterior en un "void" y llamar al void 50 veces.

CÓDIGO: SELECCIONAR TODO

```
void funcion(){
      char texto[] = "Hola Mundo";
      printf(texto);
}

//dentro de alguna otra función
funcion();
```

Habéis notado que aquí, en este ejemplo, no he puesto el "return;". Puesto que la función no va a devolver nada, la función de "return" es simplemente la de parar la función, es decir, que todo lo que haya después del "return;" no se ejecute.

Del mismo modo que podemos hacer funciones de cualquier tipo, también podemos hacer que una función sea un puntero, por ejemplo:

CÓDIGO: SELECCIONAR TODO

```
char *dame_un_texto(){
     char *texto = (char*)malloc(11);
     strcpy(texto, "Hola Mundo");
     return texto;
}
```

Y, quizás alguno os preguntéis, porqué no ha echo esto?:

CÓDIGO: SELECCIONAR TODO

```
char *dame_un_texto2(){
   char texto[] = "Hola Mundo";
   return texto;
}
```

Esto me lleva a explicaros la "area de efecto" de una variable, puesto que no se me ocurre mejor manera de decirlo xd.
Hay 2 formas de declarar una variable, de forma global, o de forma local.
Cuando declaramos una variable, lo podemos hacer fuera de toda función. Si hacemos esto, la variable sera accesible (se podrá usar) en cualquier función (recordad que, primero declaramos, luego usamos, vigilad el orden). Veamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
int a;
void funcion1(){
     a = 1;
}
```

```
void funcion2(){
     a = 2;
}
```

En cambio, si hacemos esto:

CÓDIGO: SELECCIONAR TODO

```
void funcion1(){
     int a;
     a = 1;
}
void funcion2(){
     a = 2;  //ERROR!
}
```

En esto 2o caso, la variable a es local, puesto que solo sirve para la funcion1, si la utilizamos fuera, nos dará error el compilador.
Hay más formas de declarar variables, peeero, las veremos mas tarde.

Retomemos el tema de las funciones que devuelven punteros. ¿Os acordáis de los ejemplos aun? Si no los recordáis, echadles una ojeada antes. En el segundo, la variable es local, puesto que simplemente hacemos una array de texto. Devolver un puntero, una dirección, a una variable local es un error, puesto que la variable, en cuando se termine la función dejará de existir, así que, la dirección esa, apuntará a la nada.
En cambio, en el primero, usamos la función malloc() para guardar una memoria. Guardar memoria es un acto global, así que, cuando termine la función esa memoria seguirá existiendo, por lo que no hay problema alguno en devolver esa dirección.
Esto es denso, aseguraros de entenderlo, y practicad, practicad mucho. Si lo entendéis **y lo habéis practicado** continuad.

Del mismo modo, los parámetros también pueden ser de cualquier tipo, incluidos punteros. Si habéis entendido todos los temas anteriores, también entenderéis lo que hace esta función, pero de todas formas, añado comentarios:

CÓDIGO: SELECCIONAR TODO

```
void cambiar_valor(int *variable, int valor){
     *variable = valor;
/*
La función pide un puntero, es decir una dirección de memoria como
parámetro 1.
Luego, va a la dirección de ese puntero (*variable) y le cambia el valor
por el de la variable "valor".
*/
}
int main(){
     int var = 5;
     printf("%d\n", var); //Nota, "\n" hace que salte de linea en la
```

```
consola. Linea nueva. Saldrá un 5 en la pantalla
     cambiar_valor(&var, 10);
/*
Como ya sabemos, la función pide un puntero int, una dirección. Pues le
damos la dirección de nuestra variable var.
Luego nos pide un valor para guardarle, 10.
Si simplificáramos todo el proceso, haríamos esto:
*(&var) = 10;
Pues, primero pasamos el puntero, &var, y luego obtenemos el valor de la
dirección con *, *(&var)
y esto, aun más simplificado, es:
var = 10;
*/
     printf(%d\n", var); //Saldrá un 10 en la pantalla
     getchar();
     return 0;
}
```

Para terminar este capítulo, solo me queda deciros una cosa, ¡practicad hasta aburriros!

9. Profundizando en las arrays

Bueno, pequeño cambio en el planning. Antes de explicaros el tema de las estructuras y "sus hermanos", vamos a profundizar en el tema arrays.
Vamos a dar primero una definición un poco más, "real", de lo que es una array: es una secuencia de objetos del mismo tipo. Es decir, valores o objetos de un mismo tipo de variable. Sabemos ya, además, como se declara una array:

CÓDIGO: SELECCIONAR TODO

```
tipo nombreArray[numeroDeElementos];
```

Un tema que he dado por entendido, pero que no he explicado, es que, a cada posición de la array(0, 1, 2, .., numeroDeElementos-1) podemos guardar un valor y acceder a el:

CÓDIGO: SELECCIONAR TODO

```
nombreArray[0] = X; //X es un valor acorde al tipo de variable
```

Este 0, o lo que ponemos entre los corchetes [] será llamado a partir de ahora como "indice".
Los elementos de una array se almacenan de forma contigua en la memoria, veamos una imagen que lo ilustra:

Tengamos en cuenta que, la separación entre cada una de estas casillas en memoria, equivale al tamaño del tipo de variable. Recordad "sizeof(tipo)" nos dirá el tamaño de x tipo en memoria. Así, si en esta foto podríamos hablar de una array de char, puesto que entre cada casilla solo hay 1 espacio de memoria, y un char

ocupa 1 espacio de memoria.

Imaginaros que hacemos una array de 604 elementos, y más tarde, en otra función, queremos comprobar el tamaño de dicha array y no nos acordamos del número. Entonces recurriremos otra vez a sizeof(), en este caso "sizeof(nombreArray);" nos devolvería el numero de elementos que existen en una array. Veamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
char char_array[1024];
printf("%d", sizeof(char_array) / sizeof(char)); //Mostraría 1024.
```

Varías cosas a tener presente:
¿Porqué dividimos por sizeof(char)? Recordad que he dicho que cada casilla, de esa imagen para tener una referencia, estaba separada de la otra por el tamaño del tipo de variable, es decir, que lo que realmente devuelve sizeof(nombreArray) es el número de elementos multiplicado por lo que ocupa cada elemento en memoria. Si queremos obtener el número de elementos real, debemos dividirlo por el espacio que ocupa en memoria, de otra forma obtendríamos el espacio que ocupa, no el número de elementos. En el caso de una array de char no hay problema, puesto que un char ocupa 1 espacio, por lo que estamos dividendo por 1, y el resultado será el mismo, pero si fuera otro tipo, como int, estaríamos dividiendo por 4. Probarlo vosotros, haced un programa simple basado en el ejemplo anterior y mirad que pasa si ponéis la división o no la ponéis.
Otra cosa tener en cuenta, si que es verdad que la anterior array tiene 1024 elementos, pero, tiene 1024 porqué empezamos desde 0 hasta 1023, y el 1024 se reserva para el '\0', es decir, intentar escribir algo en la posición 1024 probablemente provocaría un error de corrupción de datos. Digo probablemente porque todo dependería de en que posición de memoria se encuentra y de que tiene detrás.

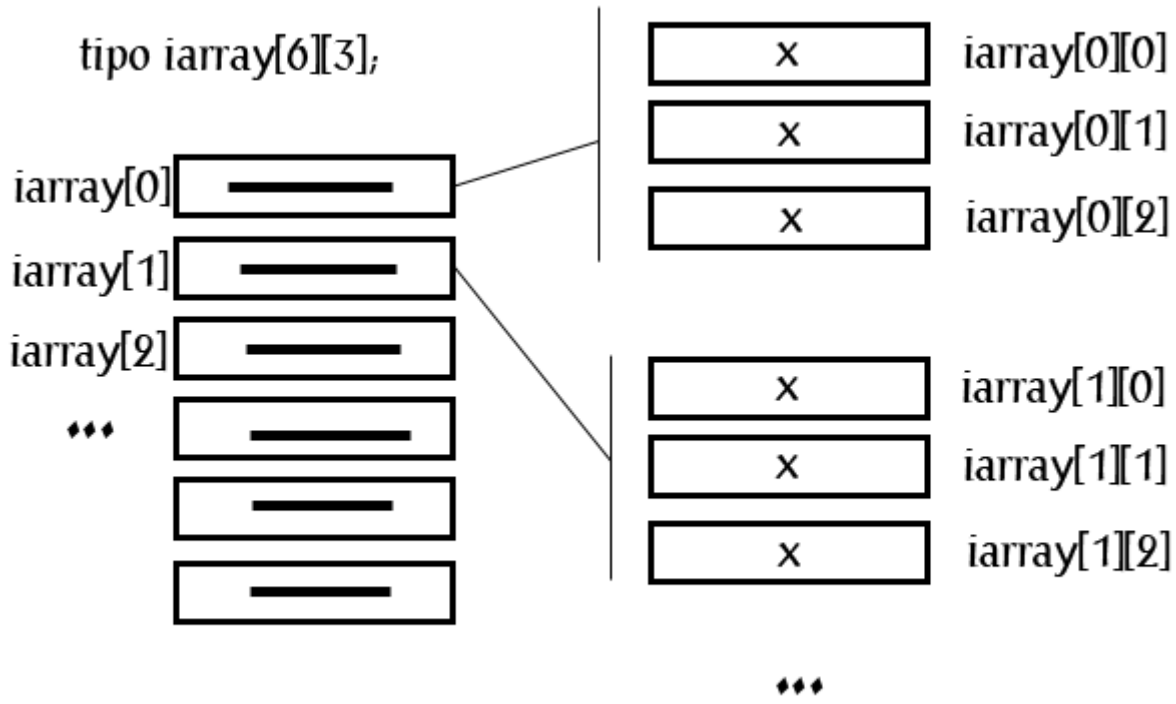
Más cosas, como podemos comprobar si podemos guardar algo en una posición de una array, me explico imaginad que la array tiene 10 elementos, y por error intentamos escribir en el 12. Deberíamos primero de comprobar el tamaño de la array para asegurarnos de que nuestro programa va a funcionar bien y no intentará escribir en el 12 cuando el máximo seria 10.

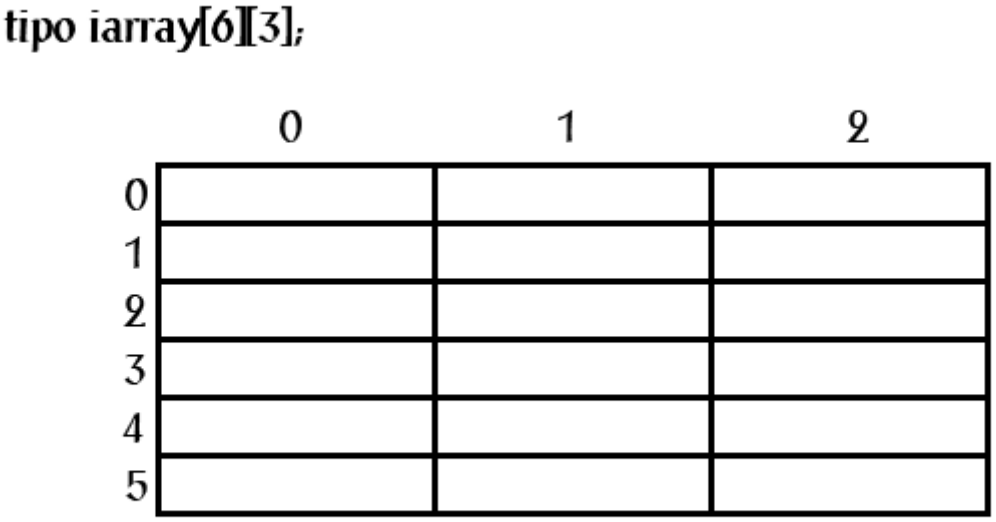
Vamos a otro tema dentro de las arrays, las arrays multidimensionales. Las arrays anteriores eran unidimensionales, puesto que solo admitían 1 indice, en cambio, nosotros podemos hacer que nuestra array acepte más de un indice, veamos un prototipo de como sería:

CÓDIGO: SELECCIONAR TODO

```
tipo nombreArray[indice1][indice2]
```

Nota: Aveces nos referimos a las arrays unidimensionales como listas (no confundir con el termino de C++ std::list) y a las arrays multimensionales como tablas o matrices.
Explicaré esto a partir de dos imágenes y ejemplos, pues es mucho mas sencillo:





Yo creo que se ve bastante claro, pero por si acaso: Tenemos un primer índice donde no podemos guardar valores, sin embargo, dentro de este indice si podemos guardar cosas.
Ejemplo:

CÓDIGO: SELECCIONAR TODO

```
int i_array[6][3];
i_array[0][1] = 1;
i_array[5][2] = 0;
i_array[2][0] = 2;
//etc.
```

Si queremos, también podemos inicializar directamente esta array a un número:

CÓDIGO: SELECCIONAR TODO

```
int i_array[6][3] = {/*estamos en el primer indide, 0 */ { /*accedemos
al segundo indice*/ 0 /*posicion 0*/, 1, 2 }, {/*2o indice 1*/ 3,4,5},
etc.}
//Veamoslo bien hecho y sin comentario de por medio:
int i_array[6][3] =
{{0,1,2},{3,4,5},{6,7,8},{9,10,11},{12,13,14},{15,16,17}};
//Esto es equivalente a esto:
int i_array[6][3] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};
/*Puesto que automaticamente, cada 3 números (el máximo del 2o indice)
aumentará el primer indice en 1, cosa que hacíamos manualmente en el
primer ejemplo mediante el uso de comas "," y llaves {}. */
```

Del mismo modo que podemos hacer arrays bidimensionales, podemos hacer de 3, 4, 5 e infinitas dimensiones:

CÓDIGO: SELECCIONAR TODO

```
int array1[1][1][10];
int array2[10][12][100][1000];
```

En el siguiente capítulo, puesto que aun no hemos visto bucles ni condicionales, trataremos estos y explicamos como ordenar los elementos (numéricos) de una array de menor a mayor.

10. Condicionales (if, switch)

Como todo buen lenguaje de programación, C es capaz de ejecutar código solo si se confirma una condición. Para ello, usaremos el llamado "if", que traducido a español significa "Si". Veamos su estructura y funcionamiento:

CÓDIGO: SELECCIONAR TODO

```
if( condicion ){
     //Codigo a ejecutar si la condición es verdadera
}
```

Me gustaría hacer una alusión a un lenguaje de programación llamado VB (Visual Basic), en el cual los if se declaran de forma diferente, ahora entenderéis porqué

hago esta referencia:

CÓDIGO: SELECCIONAR TODO

```
if condicion then
        //codigo
endif
```

Bien, el hecho esta en que, todas ellas son palabras inglesas:

if -> Si
then -> entonces
endif -> fin de la condicion

En C, este then se sustituye por un "{" y el endif por "}". Es decir, la traducción de lo que hace el if es

Si se cumple esta **condicion entonces**
ejecutar este codigo
fin de la condicion

Veamos un ejemplo práctico:

CÓDIGO: SELECCIONAR TODO

```
int var = 1;
if(var == 1){
        printf("La condicion del if, var == 1, es verdadera");
}
```

Recordad el uso de "==", puesto que estamos comparando, no guardándole un valor.
Probad a cambiar el 1 por cualquier otro número, y veréis como no se cumple la condición, y en consecuencia, no aparece nada en pantalla.

NOTA: No podemos comparar variables del tipo char*. Así pues,
sería **incorrecto** hacer lo siguiente:

CÓDIGO: SELECCIONAR TODO

```
char h[] = "hola";
if(h == "hola")..... // ERROOOR COMO UNA CASA
```

Más tarde ya trataremos como comparar, entre otras cosas, variables del tipo char*.
Notese que si podemos comparar variables de tipo char.

CÓDIGO: SELECCIONAR TODO

```
char c = 'a';
if(c == 'a')...
```

Puesto que si lo pensáis bien, simplemente estamos comparando números, recordad la definición de char.

Es importante destacar que, igual que pasaba con los bucles for y while, podemos omitir los { }, en caso de que el cuerpo sea una sola sentencia, asi pues, el anterior ejercicio podría quedar como:

CÓDIGO: SELECCIONAR TODO

```
int var = 1;
if(var == 1)
	printf("La condicion del if, var == 1, es verdadera");
```

Introduzcamos ahora una palabrita más, "else". Su traducción aproximada sería "sino". Con esto ya podríamos deducir que lo que esta palabra hace, es ejecutar un código cuando la condición no se cumple:

CÓDIGO: SELECCIONAR TODO

```
if( condicion ){
	//La condición es verdadera
	//Ejecutar código
}else{
	//La condición es falsa
	//Ejecutar otro código
}
```

Notese que aquí también podemos omitir los { } tanto en el if como en el else, de forma independiente, asi pues, todas las siguientes estructuras serían validas (además de la anterior)

CÓDIGO: SELECCIONAR TODO

```
if( condicion ){
	//1
	//2
}else
	//1
```

CÓDIGO: SELECCIONAR TODO

```
if( condicion )
	//1
else{
	//1
```

```
	//2
}
```

CÓDIGO: SELECCIONAR TODO

```
if( condicion )
	//1
else
	//1
```

Cuando usamos las llaves "{ }" podemos no poner ningún código en su interior sin hacer ningún cambio especial, pero si los omitimos, se dan condiciones especiales.

CÓDIGO: SELECCIONAR TODO

```
if( condicion );
```

Ponemos un ; al final, cosa que no tiene mucho sentido, puesto que poner una condición y no hacer nada, es un poco, raro. Lo mismo haríamos con el else, "else;", pero este si que no tiene sentido alguno hacerlo, si no vas a hacer nada en caso de que no se cumpla la condición, simplemente, no pongáis el else.

Ahora os recomendaría, para variar, que practicarais, puesto que para entender lo siguiente es necesario haber entendido y practicado lo anterior.
Si ya habéis practicado, continuad leyendo:

Que pasa si nosotros queremos hacer algo del tipo, "Si la variable NO es 1".
Una solución chapucera sería la siguiente:

CÓDIGO: SELECCIONAR TODO

```
if(var == 1);
else{
	//Puesto que ELSE solo se ejecuta si no se cumple la condición, y por tanto solo llegamos aquí si var NO es 1.
}
```

Pero, resulta que C, y C++ y muchos otros lenguajes, disponen de un carácter especial, "!", el signo de exclamación. Este significa NEGAR.
Así pues, podemos hacer una comparación negativa, usando esto: !=

CÓDIGO: SELECCIONAR TODO

```
if(var != 1){
	//Si llegamos aquí, significa que var NO (!=) es igual a 1
}
```

Hay otra forma, pero debemos entender muy bien su funcionamiento.
Tenemos var == 1, en el primer caso. Si esto fuera verdad, nos devolvería que es

verdad (que redundante pensaréis), pero resulta que decir que algo es verdad, siempre que hablemos de programación, equivale a un 1. Pensemos un momento en binario, eso que solo hay 000111010101. ¿Cuál seria el contrario de 1 en binario? Pues 0, tampoco hay más elección posible. Dejo esto al aire, puesto que me interesa y mucho para más tarde.
Dicho de otra forma más fácil, ¿cuál es el contrario de verdadero? FALSO.
Así pues, podríamos decir que si se da que, !VERDADERO, es decir, FALSO, se ejecute algo.
VERDADERO, ¿a que equivale en nuestro ejemplo anterior? A "var == 1", verdad? Entonces quedaría así:

CÓDIGO: SELECCIONAR TODO

```
!(var == 1); //!VERDADERO
```

Fijaros en los paréntesis, queremos negar la condición entera, hacer "!var == 1" no serviría de nada.

Tenemos pues que,

CÓDIGO: SELECCIONAR TODO

```
if( var != 1 ){}
```

Es lo mismo que:

CÓDIGO: SELECCIONAR TODO

```
if( !(var == 1) ) {}
```

Nota: El uso de != es válido, pero no lo es el de !> o cualquier parecido, por lo que, si nuestra condición es:

CÓDIGO: SELECCIONAR TODO

```
if(var > 1){} // Siempre que var sea mayor estricto a 1 será verdadero
```

Y queremos negarlo, debemos hacer lo siguiente:

CÓDIGO: SELECCIONAR TODO

```
if( !(var > 1) ){} //No mayor ni igual a 1 = verdadero
```

Aunque si pensamos un poco, veremos que el contrario de > es <=, por lo que podemos simplemente escribir:

CÓDIGO: SELECCIONAR TODO

```
if ( var <= 1 ) //Menor o igual a 1 = verdadero
```

En fin, aprovecho para decir que probéis de hacer condicionales con las estructuras "==", "!=", ">", "<", ">=", "<=".

Y por hoy, creo que ya he escrito más que suficiente. Mañana más y mejor, puesto que el "if" aun no se ha acabado, tiene cuerda para rato.
Practicad todos estos conceptos nuevos del "if" por favor!

10.1 Entrando en detalle en las condiciones

Pensemos una cosa, que pasa si queremos comprobar mas de una condicion. Del plan:

Si pasa esto X, pero si no pasa esto y pasa Y, o no pasa ni X ni Y y pasa Z... etc

Podriamos hacer algo del tipo:

CÓDIGO: SELECCIONAR TODO

```
if(X){

}else{
     if(Y){

     }else{
          if(Z){

          }
     }
}
```

Claramente esto funcionaria, pero, ¿que pasa si queremos comprobar 20 condiciones de esta manera? Simplemente, que al final no sabremos ni lo que hacemos. Es por esto, que podemos utilizar un formato distinto pero que hace exactamente lo mismo:

CÓDIGO: SELECCIONAR TODO

```
if(X){

}else if(Y){

}else if(Z){

}else{
     //Si ninguna se ha cumplierto
}
```

¿Mucho mejor no? Código claro, estructurado y nos permite comprobar varias condiciones con el simple uso de "else if".

Ahora quiero que os imaginéis otra situación muy parecida a la anterior. También queremos comprobar distintas condiciones, pero todas ellas recaen sobre la misma variable de tipo numério (es decir, todos los tipos menos cadenas del tipo "abc", recordad que comparaciones de cadenas (char *), como cadena no puntero, lo trataremos más tarde).
Tenemos una variable que llamaremos "test", la cual queremos comprobar si vale 1, 2, 3, 4, 5, 6, 7, 8, 9 o 10. Efectivamente una solución bastante lógica, por lo que sabemos ahora, sería el uso de los multiples "else if", quedando asi:

CÓDIGO: SELECCIONAR TODO

```
if(test == 1);
else if(test == 2);
else if(test == 3);
else if(test == 4);
else if(test == 5);
else if(test == 6);
else if(test == 7);
else if(test == 8);
else if(test == 9);
else if(test == 10);
```

Mmmm... sí, vale, está bien, pero, realmente, ¿vamos a escribir 10 veces la variable "test"? ¿El día que queramos hacer 100 comprobaciones, lo escribiremos 100 veces? NO, la respuesta es NO.
Para esto existe otro tipo de condicional, el llamado "switch". ¡Abrimos nuevo subapartado!

10.2 Switch

La sintaxis de este condicional, que por cierto su nombre significar "cambiar" en castellano, es la que sigue:

```
switch(NOMBRE_VARIABLE){
case VALOR1:

break;
case VALOR2:

break;
}
```

Realmente solo hay una cosa que debería extrañarnos, que es la palabrita "break". Por lo demás, lo que hacemos es como un "else if". Cada "case X:" equivale a un "else if". Hagamoslo con el ejemplo anterior.

CÓDIGO: SELECCIONAR TODO

```
switch(test){
     case 1: break;
     case 2: break;
     case 3: break;
```

```
     case 4: break;
     case 5: break;
     case 6: break;
     case 7: break;
     case 8: break;
     case 9: break;
     case 10: break;
}
```

Mucho, pero que mucho, mejor. NOTA: El condicional switch solo sirve para IGUALDADES (==), en caso de querer otras cosas, podríamos recurrir a lo que explicaré a continuación.
La palabra break la veremos con más detalle en el capítulo 11, al final. Pero de todas formas, para poder entender si función, diremos que lo que hace es evitar que el codigo que haya después se ejecute. Veamos un ejemplo en que omitiremos el uso de break:

CÓDIGO: SELECCIONAR TODO

```
switch(test){
     case 1: printf("A"); break;
     case 2: printf("A"); break;
     case 3: printf("B"); break;
}
```

Fijaros, ¿no es verdad que tanto el caso 1 como el 2 hacen exactamente lo mismo? Hemos dicho que el break evitaba que se ejecutara lo que había a continuación, pero, ¿y si queremos simplificarlo?

CÓDIGO: SELECCIONAR TODO

```
switch(test){
     case 1:
     case 2: printf("A"); break;
     case 3: printf("B"); break;
}
```

Interesante, ¿no?. Esto podríamos interpretarlo de la siguiente forma: "Si test == 1 o test == 2 -> printf("A");"
Si test vale 1, se ejecutará lo mismo que si test vale 2, pero como luego hay un break, no se ejecutará nada más.

Ahora las cosas van a, no complicarse pero sí enredarse un poco, asi que, es mi consejo que ¡PRACTIQUÉIS MUCHO ESTO!

10.3. Más sobre condiciones

Anteriormente hemos hablado de que hacer cuando queremos evaluar 2 (o más) condiciones por separado, el else if, pero, y si lo que queremos es evaluar en un mismo if más de una condición a la vez, estilo si I < 10 i M > 8, etc. En nuestro idioma hablado, sea el que sea, dispones de las conjunciones "y", y "o". Así pues decimos si eso y aquello, si eso o lo otro. En C tenemos unos equivalentes llamados

&& (y) y || (o). Nótese que son 2 carácteres, no me escribáis uno solo porque entonces lo que hacemos son operaciones con bits (que más adelante explicaremos).
Veamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
int a = 1;
int b = 2;

if(a == 1 && b == 2)
     printf("a = 1 y b = 2");
```

Probadlo, y luego cambiad el valor de cualquier de estas dos variables, veréis como no se muestra el resultado en pantalla. Una vez hecho, probemos cambiando el && por un ||

CÓDIGO: SELECCIONAR TODO

```
int a = 1;
int b = 2;

if(a == 1 || b == 2)
     printf("a = 1 y b = 2");
```

Basta que con una de las igualdades, o a == 1 o b == 2, para que se ejecute el condicional.

Esto es facilito, pero luego podemos hacer combinaciones de && y ||. Y nótese que podemos usar paréntesis para ello

CÓDIGO: SELECCIONAR TODO

```
if((a == 1 && b == 2) || c == 3)
```

NO es lo mismo que

CÓDIGO: SELECCIONAR TODO

```
if(a == 1 && (b == 2 || c == 3))
```

En el primer caso primero se comprueba si a == 1 Y b == 2, si eso se cumple, tenemos la primera condicion verdadera. Ahora bien, luego viene un ||, es decir, que o todo el primer bloque es cierto, o lo es la 2 condicion, con una basta.
Miremos el segundo caso. Primero tenemos a == 1, ¿es cierto? Si lo es, ya tenemos un bloque verdadero. Luego nos exige mediante un && que se cumpla que b sea 2 o, recalco o, que c sea 3.
Si habéis leído atentamente, las diferencias y usos están más claros.
Ahora, tanto si lo tenéis claro como si no, ¡PRACTICAD!

11. Bucles

Un bucle es cualquier construcción de programa (código) que repite una sentencia o conjunto de sentencias de forma repetida.
En todo bucle encontramos un cuerpo, las sentencias que se repiten, y iteraciones, que son cada una de las repeticiones del bucle.
Encontramos diferentes tipos de bucles, analizamos primero el while

11.1 - Bucle While
Un bucle while se ejecuta según una condición. Antes de cada iteración (repetición), se comprueba que la condición sea válida. Si lo es, se ejecuta el cuerpo, si no, no.
En otras palabras, un bucle while se ejecutará de 0 a más veces, según la condición siga siendo válida.
Veamos la sintaxis básica de un bucle while:

CÓDIGO: SELECCIONAR TODO

```
//Caso 1:
while(condicion)
	sentencia;

//Caso 2:
while(condicion){
	sentencia_1;
	sentencia_2;
	sentencia_3;
	...
	sentencia_n;
}

//También sería valido en el caso 1 poner {}, pero simplemente los omitimos:
while(condicion){
	sentencia;
}
```

Pongamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
int n = 0;
while (n < 10)
	n++;
```

Primero de todo, "n++". Siempre que veáis un ++ después de una variable numérica, significa lo mismo que, "n += 1", y esto significa lo mismo que "n = n+1". Por poner un ejemplo: "7++" = "7 += 1" = "resultado = 7 + 1 = 8".

Luego, la condición aquí es n < 10. Para todos aquellos que hagáis hecho un poco de matemáticas, sabréis que "<" significa "menor que", y si no, ahora ya lo sabéis.

Aprovecho para recordar:

```
variable < numero (menor que)
variable > numero (mayor que)
variable <= numero (menor o igual que)
variable >= numero (mayor o igual que)
```

Es decir, la condición es que "n", nuestra variable, sea menor que 10. Si es menor que 10, le sumamos 1. En caso de que nuestra variable llegue a 10, la condición ya no será verdadera, puesto que 10 no es menor que 10, es igual.

Veamos otro ejemplo. Os acordáis que en capítulos anteriores había usado un == en vez de = ? En C, y C++, cuando comparamos dos valores (o variables) para saber si son iguales debemos usar == , a diferencia de cuando le guardamos un valor que usamos =.
Este ejemplo es un poco tonto:

CÓDIGO: SELECCIONAR TODO

```
int n = 5;
while(n == 5)
     n++;
```

Si os fijáis solo se va a ejecutar una vez, pues, a la primera pasada 5 == 5, así que se ejecuta el cuerpo, n para a ser 6. En la segunda pasada 6 == 5, y eso no es verdad, así que, ya no se ejecuta.

11.2 - Bucle for

SI habéis entendido el anterior, este es muy parecido:

CÓDIGO: SELECCIONAR TODO

```
for( declaraciones_o_inicializaciones ; condicion ; accion){
     cuerpo
     //recordad que si el cuerpo es 1 sola instrucción podemos eludir
los {}
}
```

Primer, que son las declaraciones o inicializaciones. De la misma forma que con el while, poníamos justo antes "int n = 5", aquí también tenemos que hacerlo, solo que podemos hacerlo dentro.
A diferencia de el bucle while, el primer "apartado", declaraciones (me refiero a los 3 que se separan por  , solo se ejecuta a la primera pasada, o iteración. Es decir, solo se ejecutará 1 vez, la primera de todas y antes de todo.
Veamos un par de ejemplos:

CÓDIGO: SELECCIONAR TODO

```
for(int n = 0; ... ; ...);
```

Como n no existe, la declaramos en el primer "apartado".
Nota: los "..." los pongo para indicar que allí va algo.

CÓDIGO: SELECCIONAR TODO

```
int n;
for(n = 0; ... ; ...);
```

"n" ya está declarada, pero no tiene ningún valor, así que, la inicializamos a 0, "n=0".

CÓDIGO: SELECCIONAR TODO

```
int n = 0;
for( ; ... ; ...);
```

Puesto que "n" ya existe y ya tiene un valor (0) no hace falta hacer nada. No es obligatorio poner nada en el primer apartado, de hecho, en ninguno de ellos es obligatorio.

Siguiente apartado, condición. Este es exactamente lo mismo que en el while, una condición que se comprueba a cada iteración.
Y el último, acción. El bucle for, a diferencia del while, ejecuta una ultima acción antes de pasar a la siguiente ronda, veamos un esquema de lo que hace el for:

CÓDIGO: SELECCIONAR TODO

```
-- RONDA 0
--- declaraciones_inicializaciones
--- condición verdadera?
----- cuerpo
----- acción

-- RONDA 1
--- condición verdadera?
----- cuerpo
----- acción

... etc.
```

¿Entendéis pues lo que os quiero decir? Justo al acabar el cuerpo, ejecutará algo que tu hayas puesto. Veamos un ejemplo práctico:

CÓDIGO: SELECCIONAR TODO

```
for(int n = 0; n < 10; n++)
      printf("%d\n", n);
```

Esto hará lo siguiente:

CÓDIGO: SELECCIONAR TODO

```
-- RONDA 0
--- int n = 0;
--- (n < 10), (0 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 1

-- RONDA 1
--- (n < 10), (1 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 2

-- RONDA 2
--- (n < 10), (2 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 3

-- RONDA 3
--- (n < 10), (3 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 4

-- RONDA 4
--- (n < 10), (4 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 5

-- RONDA 5
--- (n < 10), (5 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 6

-- RONDA 6
--- (n < 10), (6 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 7

-- RONDA 8
--- (n < 10), (8 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 9

-- RONDA 9
--- (n < 10), (9 < 10), verdad
----- printf(...)
----- n++, n +=1, n = n + 1 => n = 10
```

```
-- RONDA 10
--- (n < 10), (10 < 10), falso
```

11.3 - Bucle do-while

Para entender este bucle, debemos tener claro el concepto del bucle while. El funcionamiento es prácticamente el mismo, solo que en while, si recordamos bien, primero se comprobaba la condición y si era verdadera se ejecutaba el cuerpo. En el do-while, en cambio, primero se ejecuta el cuerpo y luego se comprueba la condición, si esta es verdadera, se vuelve a ejecutar el código.
Ejemplo:

CÓDIGO: SELECCIONAR TODO

```
do{
     //Cuerpo
}while(condicion); //Notese el ;
```

Si sabemos un poco de inglés, la cosa se "facilita", traducido, dice esto:

CÓDIGO: SELECCIONAR TODO

```
HACER {
     //cuerpo
}MIENTRAS(condición);
```

Creo que es fácil de entender, y es por esto que no daré mas detalles sobre ello. SI tenéis dudas, PRACTICAD! Y si las tenéis siguiendo, PRACTICAD MÁS! Y si aun tenéis, pues preguntar aquí.

Eso si, os dejo un ejemplo, que no es mío, pero que ilustra perfectamente el uso de do-while:

CÓDIGO: SELECCIONAR TODO

```
char c;

do {
printf("\n\nIntroduzca una letra: ");
c=getche();     /*devuelve el carácter leido desde la consola, y lo
visualiza.*/
c=toupper(c); /* devuelve el carácter c en mayúscula, y si no lo hay,
devuelve el mismo*/
printf("\nSu mayúscula correspondiente es: %c",c);
} while (c!='X');
int j=100;   /* sentencia que ejecutamos cuando deje de cumplirse la
condición del do-while. */
```

Palabritas mágicas

¿Buen título eh? Resulta que mientras estamos en un bucle disponemos de dos

palabras reservadas, dicho de otra forma, que tienen una función especial. Se trata de "break" y de "continue".
Planteemos una primera situación. Que pasa si nosotros llegamos a un punto en que queremos salir del bucle. Podriamos pensar las mil y una para añadir una condición más al bucle, pero, ¿para que complicarnos tanto la vida? La palabra "break" (romper en castellano) nos permite salir de un bucle. Veamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
for(int i = 0; i < 10; i++){
if(i == 5) break;

printf("%d\n", i);
}
```

El resultado será este:

```
1
2
3
4
```

El 5 (y posteriores) nunca lo llegaremos a ver, puesto que al llegar a este número, salimos del bucle sin ejecutar el código restante ni las siguientes pasadas.

Luego existe otra situación, ¿que pasa si queremos que, llegado a una condición X, no se ejecute el código, pero que en vez de salir por completo, como en el caso del "break", continue con la siguiente pasada? La respuesta es "continue". Vamos un ejemplo:

CÓDIGO: SELECCIONAR TODO

```
for(int i = 0; i < 10; i++){
if(i == 5) continue;

printf("%d\n", i);
}
```

El resultado será este:

```
1
2
3
4
6
7
8
9
```

El 5 no lo veremos, puesto que cuando lleguemos al 5, se saltará directamente a la siguiente pasada sin ejecutar el código que haya a continuación.

PROGRAMACION

Funciones de configuración

void KJunior_init(void)

Objetivo : Inicializa KJunior
Parámetro : Ninguno
Devuelve : Nada
Ejemplo : KJunior_init();
Resumen : Esta función debe estar al principio del programa para inicializar la siguiente configuración por defecto:

- Lectura automática de los sensors IR activada
- Administración RS232 activado
- Lectura automática del receptor TV activada
- Control remoto TV activado

void KJunior_config_auto_refresh_sensors(int1 Bit)

Objetivo : Configura la lectura de los sensors IR
Parámetro : MANUAL ó REFRESH (por defecto)
Devuelve : Nada
Ejemplo : KJunior_config_auto_refresh_sensors(MANUAL);
Resumen : Configura si los sensores IR se leen automáticamente(REFRESH) o no(MANUAL)

void KJunior_config_auto_refresh_tv_remote(int1 Bit)

Objetivo : Configura la lectura automática del receptor TV
Parámetro : MANUAL ó REFRESH (por defecto)
Devuelve : Nada
Ejemplo : KJunior_config_auto_refresh_tv_remote(MANUAL);
Resumen : Configura si el receptor TV se refresca automáticamente(REFRESH) o no(MANUAL)

void KJunior_config_rs232_control(int1 Bit)

Objetivo : Activar o desactivar el control remoto por puerto serie
Parámetro : DISABLE o ENABLE (por defecto)
Devuelve : Nada
Ejemplo : KJunior_config_rs232_control(DISABLE);
Resumen: Esta función activa (enable) o desactiva (disable) el control remoto por puerto serie RS232.
Atención: Si este modo está activado, usted puede usar la función “printf” (por ejemplo, para ver algunos valores en un terminal) pero no puede leer nada del puerto serie(getc, gets, etc no están disponibles), todos los datos serán leídos por el proceso de control remoto serie.

void KJunior_config_tv_remote_control(int1 Bit)
Objetivo : Activar o desactivar el modo de control remoto TV
Parámetro : DISABLE o ENABLE (por defecto)
Devuelve : Nada
Ejemplo : KJunior_config_tv_remote_control(DISABLE);
Resumen *:* Esta función activa (enable) o desactiva (disable) el modo de control remoto TV(control remoto de los motores con un mando a distancia de televisión).

Funciones de lectura de las 'flags'(indicadores de estado)

int1 KJunior_flag_sensors_refreshed(void)
Objetivo : Saber si los sensores IR se han refrescado (re-muestreado)
Parámetro : Ninguno
Devuelve : El bit está a 1 si los sensores se acaban de refrescar
Ejemplo : i = KJunior_flag_sensors_refreshed();
Importante : Se debe resetear el indicador tras la función «KJunior_flag_sensors_reset»

void KJunior_flag_sensors_reset(void)
Objetivo : Resetear los indicadores de los sensores IR
Parámetro : Ninguno
Devuelve : Nada
Ejemplo : KJunior_flag_sensors_reset();

int1 KJunior_flag_rs232_filtering(void)
Objetivo : Saber si el control remoto por puerto serie está activado o no.
Parámetro : Ninguno
Devuelve : 1 si está activado, 0 si no.
Ejemplo : i = KJunior_flag_rs232_filtering();

int1 KJunior_flag_tv_data_refreshed(void)
Objetivo : Saber si el receptor TV se ha refrescado o no.
Parámetro : Ninguno
Devuelve : El bit está a 1 si algún dato ha sido recibido en el receptor TV.
Ejemplo : i = KJunior_flag_tv_data_refreshed();
Importante : Se debe resetear el indicador tras la función « KJunior_flag_tv_data_reset»

int1 KJunior_flag_tv_data_emitting(void)
Goal: To know if the emitting is complete or not.
Parameter : None
Return : 1 if some data are still emitting or 0 it's done
Example : i = KJunior_flag_tv_data_emitting();

void KJunior_flag_tv_data_reset(void)

Goal : Reset the TV receiver flag.
Parameter : None
Return : None
Example : KJunior_flag_tv_data_reset();

6.4 Input and Output functions

Signed int16 KJunior_get_proximity(char Sensor)

Goal : Get the value of the proximity sensor
Parameter : FRONT, FRONTLEFT, FRONTRIGHT, LEFT, RIGHT, REAR, GROUNDLEFT, GROUNDRIGHT, GROUNDFRONTLEFT, GROUNDFRONTRIGHT
Return : 10bits value (0(nothing) to 1024(an obstacle is very near))
Example : i = KJunior_get_proximity(FRONT); Return the proximity value of the "Front" IR sensor.

signed int16 KJunior_get_brightness(char Sensor)

Goal : Get the brightness value of a IR sensor.
Parameter : FRONT, FRONTLEFT, FRONTRIGHT, LEFT, RIGHT, REAR, GROUNDLEFT, GROUNDRIGHT, GROUNDFRONTLEFT, GROUNDFRONTRIGHT
Return : 10 bits value (0(big luminosity) to 1024(dark))
Example : i = KJunior_get_brightness(REAR); Return the brightness value of the "Rear" sensor.

unsigned char KJunior_get_switch_state( )

Goal : Read the state of the dip switch
Parameter : None
Return : 0 (all switch turn Off) to 7 (all switch On).
Example : i = KJunior_get_switch_state( ); Return the state of the Dip switch

unsigned char KJunior_get_tv_data(void)

Goal : Get the data received on the TV receiver.
Parameter : None
Return : 6 bits value (0 to 63)
Example : i = KJunior_get_tv_data(); Return the coded value of the pressed TV remote button or from another K-Junior. i.e. the value will be 2 if the button 2 of the numeric pad is pressed.

unsigned char KJunior_get_tv_addr(void)

Goal : Get the address received on the TV receiver.
Parameter : None

Return : 5 bits value (0 to 31)

Example : i = KJunior_get_tv_addr(); Return the address of the TV remote or of another K-Junior. i.e. the value will be 2 if another K-Junior send a command like "KJunior_send_tv_value(2.10)"

void KJunior_send_tv_value(unsigned char addr, unsigned char data)

Goal : send a data and an address value on the IR emitter using the RC5 code

Parameter : addr (0 to 31), data (0 to 63)

Return : Nothing

Example : KJunior_send_tv_value(2,10); Send a message on the IR emitter with an address of 2 and a Data of 10. Every closest K-Junior will received this message.

void KJunior_set_speed(signed int8 LeftSpeed, signed int8 RightSpeed)

Goal : Set the speed of the motor left (first value) and right (second value)

Parameter : -20 to 20 (0 = Stop)

Return : None

Example : KJunior_set_speed(5 , -5); set a speed of +5 on the left motor and -5 on the right motor. The robot will turn on itself in this example.

Notice : The PWM command has a fixed frequency of 240Hz and variable duty cycle from 0 to 100% fixed by the consigne, where 20 set a PWM ratio of 100%.

void KJunior_beep(unsigned char Freq)

Goal : Make a continuous sound with the « Buzzer » at the selected frequency sound

Parameter : Frequency (0 to 100) where 0 is Off and 100 is 2kHz

Return : None

Example : KJunior_beep(91); Make a sound of 200Hz on the « Buzzer » of the KJunior.

Please refer to section 3.2.3 in the K-Juniot user manual to calculate the corresponding frequency.

void KJunior_led_left(int1 State)

Goal : turn ON or OFF the LED "Left"

Parameter : ON (1) or OFF (0)

Return : None

Example : KJunior_led_left(ON); Turn ON the "Left" LED of the robot.

void KJunior_led_frontleft(int1 State)

Goal : turn ON or OFF the LED “Front Left”

Parameter : ON (1) or OFF (0)

Return : None

Example : KJunior_led_frontleft(ON); Turn ON the “Front Left” LED of the robot.

void KJunior_led_frontright(int1 State)

Goal : turn ON or OFF the LED “Front Right”

Parameter : ON (1) or OFF (0)

Return : None

Example : KJunior_led_frontright(ON); Turn ON the “Front Right” LED of the robot.

void KJunior_led_right(int1 State)

Goal : turn ON or OFF the LED “Right”

Parameter : ON (1) or OFF (0)

Return : None

Example : KJunior_led_right(ON); Turn ON the “Right” LED of the robot.

void KJunior_led_onoff(int1 State)

Goal : turn ON or OFF the LED « On/Off »

Parameter : ON (1) or OFF (0)

Return : None

Example : KJunior_led_onoff(ON); Turn ON the « On/Off » of the robot.

void KJunior_manual_refresh_sensors(char Zone)

Goal : Refresh manually the IR sensors.

Parameter : None

Return : None

Example : KJunior_manual_refresh_sensors(); refresh the value of the IR sensors.

Notice : This function is useful only after disabling the automatic refresh sensor mode with the function « KJunior_config_auto_refresh_sensors ».
Then use one of the two functions to read the IR sensors (« KJunior_get_proximity » or « KJunior_get_brightness »).

6.5 Delay and « Timer » functions

void KJunior_delay_s(unsigned int Delay)

Goal : Put a delay in second (temporization).
Parameter : Time to wait in seconds (0 to 65535)
Return : None
Example : KJunior_delay_s(120); Wait 2 minutes.

void KJunior_delay_ms(unsigned int Delay)

Goal : Put a delay in millisecond (temporization).
Parameter : Time to wait in ms (0 to 65535) no….DE 0 A 255
Return : None
Example : KJunior_delay_ms(5000); Wait 5s.

void KJunior_delay_us(unsigned int Delay)

Goal : Put a delay in microsecond (temporization).
Parameter : Time to wait in us (0 to 65535)
Return : None
Example : KJunior_delay_us(500); wait 500 us.

A 32 bits timer provides the time (in millisecond) elapsed since the robot is turned on.

unsigned int32 KJunior_get_time(void)

Goal : Get the value of the elapsed time in ms.
Parameter : None
Return : Unsigned int32 (0 to 4294967295)
Example : Timer = KJunior_get_time(); Timer = elapsed time in ms.

void KJunior_set_time(unsigned int32 Time)

Goal : Initialize or force the elapsed time of the timer for the « KJunior_get_time » function.
Parameter : Unsigned int32 (0 to 4294967295)
Return : None
Example : KJunior_set_time(0); Initialize the timer to 0 ms.

6.6 I/O functions of the external connectors

int1 KJunior_ext_read_PINB6(void)

Goal : Read the digital Input/Output RB6 of the microcontroller.
Parameter : None
Return : 0 or 1
Example : i = KJunior_ext_read_PINB6();Return the state (0 or 1) of RB6 pin.
Notice : This Input/Output is connected to the external connector “Power supply and I^2C Bus” (see KJunior schematics).

int1 KJunior_ext_read_PINB7(void)

Goal : Read the digital Input/Output RB7 of the microcontroller.
Parameter : None
Return : 0 or 1
Example : i = KJunior_ext_read_PINB7();Return the state (0 or 1) of RB7 pin.
Notice : This Input/Output is connected to the external connector "Power supply and I^2C Bus" (see KJunior schematics).

void KJunior_ext_write_PINB6(int1 Bit)

Goal : Set the value of the digital output RB6 of the microcontroller.
Parameter : 0 or 1
Return : None
Example : i = KJunior_ext_write_PINB6(1); set the RB6 pin to 1 (+5V)
Notice : This Input/Output is connected to the external connector "Power supply and I^2C Bus" (see KJunior schematics).

void KJunior_ext_write_PINB7(int1 Bit)

Goal : Set the value of the digital output RB7 of the microcontroller.
Parameter : 0 or 1
Return : None
Example : i = KJunior_ext_write_PINB7(1); set the RB7 pin to 1 (+5V)
Notice : This Input/Output is connected to the external connector "Power supply and I^2C Bus" (see KJunior schematics).

8.3 Functions of the « Ultrasonic detector» module

void HemUltraSon_Init(void)

Goal : Initialize the turret
Parameter : None
Return : None
Example : HemUltraSon_Init();
Notice : It's the first function to call at the beginning of your Code to initialize the USsensor turret

char HemUltraSon_Read_Version(void)

Goal : Read the firmware version (OS) of the USsensor turret connected on the I^2C Bus
Parameter : None
Return : Unsigned int8 (Firmware version number)
Example : NumVer = HemUltraSon_Read_Version();
Notice : This function can be used to verify the presence of the turret. If 0xFF is returned then no turret is plugged.

unsigned char HemUltrason_Read_Brightness(void)

Goal : read the value of the brightness sensor
Parameter : None
Return : Unsigned int8.
Example : Lum = HemUltraSon_Read_Brightness( ); Return the analog value of the brightness sensor.

void HemUltraSon_Start_Mesure(void)

Goal : Start a distance acquisition.
Parameter : None
Return : None
Example : HemUltraSon_Start_Mesure();
Notice : This function includes a delay of 65ms (emitting and receiving time of the Ultrasons).

unsigned int16 HemUltrason_Read_Value(char EchoNumber)

Goal : Read the distance value of the selected Echo (unit cm)
Parameter : 0 to 7 (Number of the Echo)
Return : Unsigned int16(0 to 65535)
Example : Dist1 = HemUltraSon_Read_Value(0); Return the centimetre distance of the first echo.

void HemUltraSon_Init_Range_Register(unsigned char Value)

Goal : Initialize the maximal range of the sensor.
Parameter : unsigned int16 (0 to 65535)
Return : None
Example : HemUltraSon_ Init_Range_Register (24); maximal range distance limited to 1075 mm (see the calculation below)
Notice : The maximum range in mm is defined by:

Dmax = (Parameter x 43mm) + 43 mm ;
1 075mm = (24 x 43mm) + 43mm for the example.

The echo beyond this value will return 0.

8.4 Functions of the « Linear camera » module

The sensor of the linear camera is composed of 3 detection areas which hold 34 pixels each with 256 grey levels.
The pixels value (grey levels 0 to 255) of each area is defined in different tables of 34 unsigned int8 values as follows:

```
unsigned char HemLinCam_Pixels_Zone1[34] ;
unsigned char HemLinCam_Pixels_Zone2[34] ;
unsigned char HemLinCam_Pixels_Zone3[34] ;
```

To use in your code to have access to the pixels value .

void HemLinCam_Init(void)

Goal : Initialize the turret
Parameter : None
Return : None
Example : `HemLinCam_Init();`
Notice : It's the first function to call at the beginning of your Code to initialize the LinCam turret.

char HemLinCam_Read_Version(void)

Goal : Read the firmware version (OS) of the LinCam turret connected on the I^2C Bus
Parameter : None
Return : Unsigned int8 (« Firmware » version number)
Example : `NumVer = HemLinCam_Read_Version();`
Notice : This function can be used to verify the presence of the turret. If 0xFF is returned then no turret is plugged..

void HemLinCam_Set_Threshold(unsigned char Value)

Goal : Define the threshold value for the pixels reading.
Parameter : Unsigned int8 (0 to 255)
Return : None
Example : `HemLinCam_Set_Threshold( 100 );` Define the Threshold to 100
Notice : When the pixels threshold reading is made, the pixels with a grey level >= than 100 will have a value of 255 and the other (< 100) will reset to 0.

unsigned char HemLinCam_Read_Threshold(void)

Goal : read the setting threshold value.
Parameter : None
Return : Unsigned int8 (0 to 255)
Example : `Seuil = HemLinCam_Read_Threshold();`

void HemLinCam_Set_Exposition_Time(unsigned char Value)

Goal : Define the exposure time in ms.
Parameter : 1 to 10
Return : None

Example : HemLinCam_ Set_Exposition_Time (2);
Notice : Warning with the too big value, the pixels will be saturate very fast.

unsigned char HemLinCam_Read_Exposition_Time(void)

Goal : read the exposure time.
Parameter : None
Return : Unsigned int8 (0 to 255)
Example : ExpoMs = HemLinCam_ Read_Exposition_Time (); Return the exposure time in ms.

void HemLinCam_Read_Pixels(void)

Goal : Read the value of every pixels.
Parameter : None
Return : None
Example : HemLinCam_ Read_Pixels() ;

```
// Reading ot the pixel n°17 of the central area
If (HemLinCam_Pixels_Zone2[17] > 20)
        { ... }
```

Notice : After the reading, the values are stocked in the different value tables: « HemLinCam_Pixels_Zone1 », « HemLinCam_Pixels_Zone2 » and « HemLinCam_Pixels_Zone3 ». Each table has a size of 34 pixels, which correspond respectively to the left, central and right of the picture.

void HemLinCam_Read_Pixels_Tresholded(void)

Goal : Read the value of every pixel after the threshold
Parameter : None
Return : None
Example : HemLinCam_ Read_Pixels_Tresholded () ;

```
// Reading of the thresholded pixel n°17 of the central area
If (HemLinCam_Pixels_Zone2[17] == 255)
        { ... }
```

Notice : After the reading, the values are stocked in the different value tables: « HemLinCam_Pixels_Zone1 », « HemLinCam_Pixels_Zone2 » and « HemLinCam_Pixels_Zone3 ». Each table has a size of 34 pixels, which correspond respectively to the left, central and right of the picture.
When the pixels threshold reading is made, the pixels with a grey level >= than the threshold will have a value of 255 and the other (< threshold) will reset to 0.

void HemLinCam_Set_Led_State(int1 State)

Goal : Turn On or Off the Led
Parameter : ON (1) or OFF (0)
Return : None
Example : HemLinCam_Set_Led_State (ON); Turn On the camera LED.

APLICACIONES

ZUMBIDO DE FRECUENCIA VARIABLE CON LA DISTANCIA

```
KJunior_get_proximity(Front);

  if((IR_Proximity[Front] > 700)&& (IR_Proximity[Front] < 1000))
  KJunior_beep(100);
  if((IR_Proximity[Front] > 600)&& (IR_Proximity[Front] < 700))
  KJunior_beep(80);
  if((IR_Proximity[Front] > 500)&& (IR_Proximity[Front] < 600))
  KJunior_beep(60);
  if((IR_Proximity[Front] > 400)&& (IR_Proximity[Front] < 500))
  KJunior_beep(40);
  if((IR_Proximity[Front] > 300)&& (IR_Proximity[Front] < 400))
  KJunior_beep(20);
  if((IR_Proximity[Front] > 200)&& (IR_Proximity[Front] < 300))
  KJunior_beep(10);
  else KJunior_beep (0);
```

VELOCIDAD VARIABLE CON LA DISTANCIA

```
if( (KJunior_get_proximity(Front)) > 900)
    KJunior_set_speed( 20 , 20 );
     Else if ( (KJunior_get_proximity(Front)) > 300)
     KJunior_set_speed( (((KJunior_get_proximity(Front))/50)-4) , ((KJunior_get_proximity(Front)/50)-4) );
```

SALIR DEL LABERINTO:

```
#include "KJunior.h"

int16 F;                                    //Variables de los sensores (0(muy
lejano)-1024(muy cerca))
int16 FL;
int16 L;
int16 FR;
int16 R;
int posicion;                               //Variable para efectuar una acción
de control
int flag = 0;                               //Variable usada para evitar
conflictos en los giros

int obtener_posicion (void);                //Los nombres de las
funciones no dejan mucho a la imaginacion, no?
 void acciondecontrol(int);
```

```
 void avanzar();
 void giroaizq();
 void giro180();
 void giroader();
 void arg();

//-------------------------------------------------------------------------//
//- Main program                                                          -//
//-------------------------------------------------------------------------//




int obtener_posicion (void)
{
   F=(KJunior_get_proximity(Front));                                  //refresco los sensores
   FL=(KJunior_get_proximity(FrontLeft));
   L=(KJunior_get_proximity(Left));
   FR=(KJunior_get_proximity(FrontRight));
   R=(KJunior_get_proximity(Right));


   if((L < 500)&&(L>0)) posicion=2;                                   //posicion con posibilidad
de giro a izquierda,la condicion L>0 sirve para
                                                                      // que al principio del código no
empiece girando
   else if((F > 500)&& (FR > 500)&&(FL>500)&&(L>500)&&(R>500)) posicion=3;
//posicion sin salida, giro 180º
   else if((F > 500)&&(FL>500)&&(L>500)&&(R<L)) posicion=4;            //unica posicion
con posibilidad de giro a derecha
   else if (F<500) posicion=1;                                        //seguiremos de frente
   else posicion=5;                                                   //posicion desconocida, intentará
encontrar salida


   if(L>900) {                                                        //evita el choque con las paredes
de la izquierda,reajustando la trayectoria
      KJunior_set_speed(5,-5);
      KJunior_delay_ms(100);}
      else if ((F>900)||(FR>900)){                                    //evita el choque de frente
      KJunior_set_speed(0,0);
      KJunior_delay_ms(50);
      KJunior_set_speed(-5,5);}
```

```
  if((R>900)&&(flag==0)) {                                    //evita el choque a derechas,
la flag es para, inhibir el movimiento si
     KJunior_set_speed(-5,5);                                 //estoy girando 180º o a
derechas, evita conflictos de control
     KJunior_delay_ms(100);}
     else if ((FL>900)){                                      //evita el choque de frente-
izquierda
     KJunior_set_speed(0,0);
     KJunior_delay_ms(50);
     KJunior_set_speed(5,-5);}

  return posicion;
}

void avanzar(void){

  do {                                                        //avanzo mientras pueda

  obtener_posicion();
     KJunior_set_speed(5,5);}
  while ((F<800)&&(FL<800)&&(FR<800)&&(R<800)&&(L>500));

}

void giroaizq(void) {

  do {KJunior_set_speed(5,5);                                 //giro a izquierda, y avanzo,
a intervalos, para evitar el choque
     KJunior_delay_ms(200);
     obtener_posicion();
     KJunior_set_speed(-5,5);
     KJunior_delay_ms(100);
  } while ((L<500)&&(flag==0));
  return;
}

void giroader(void){                                          //giro a derecha, hasta que
pueda seguir de frente
  do {
  flag=1;                                                     //activo la flag ya definida
     KJunior_set_speed(5,-5);
```

```
    KJunior_delay_ms(100);
    obtener_posicion();
  } while (F>300);
  flag=0;
}

void giro180(void){

  do {                                          //giro hasta que pueda seguir de
frente, seguramente sea 180º
    flag=1;                                     //esta función es igual que giro a
derecha, pero para futuros cambios
    obtener_posicion();                         //puede venir bien separarlas
    KJunior_set_speed(5,-5);
  } while((F>300)&&(FL>400)&&(FR>400));
  flag=0;
  return;
}

void arg(void){

  do {                                          //posición indeseada, si se da el
caso, hay que revisar el cuándo se da
    KJunior_beep(50);
    KJunior_delay_ms(250);
    KJunior_beep(0);
    obtener_posicion();
    KJunior_set_speed(5,-5);
  } while((F<500)&&(FR<500));
  return;
}



void acciondecontrol(posicion) {                //elección de la acción de
control en función de la posicion
  if (posicion==1) avanzar();
  if (posicion==2) giroaizq();
  if (posicion==3) giro180();
  if (posicion==4) giroader();
  if (posicion==5) arg();
 }
```

```
 #separate

void main(void) {
   KJunior_init();                                          //se inicializa el robot con los
parámetros por defecto

   While (1) {
      obtener_posicion();                                    //programa principal muy
simple, no?  Un saludo ^^
      acciondecontrol(posicion);


   }
}
```

PROTOCOLO DE COMUNICACIÓN SERIE

Notation:

↵ Stands for carriage return (Enter or Return key pressed)
\r stands for ASCII character 0x0A (line feed)
\n stands for ASCII character 0x0D (carriage return)

A Unused

B Read firmware version

Format of the command: **B ↵**
Format of the response: **b,version_KJOs \r\n**
Effect: Return the software version stored in the flash memory

C Unused

D Set Speed

Format of the command: **D,speed_motor_left, speed_motor_right ↵**
Format of the response: **d \r\n**
Effect: Set the speed of the both motors. 0 will stop the engine. Maximum forward speed is 20 and maximum backward -20
Example: **D,7,-7↵**
Note: If you set a value bigger than 20, the K-Junior will limit by it self to 20

E Read Speed

Format of the command: **E ↵**
Format of the response: **e, speed_motor_left, speed_motor_right \r\n**
Effect: Read the speed of the motors. As the motors is in open loop, this value will the same as the last Set Speed command

F Read Time Stamp

Format of the command: **F ↵**
Format of the response: **f, time_stamp \r\n**
Effect: Read the time stamp base. The unit is the millisecond, and the maximum time is based on a 32 bits variable (more than 1000 hours). This value is reset at each boot.

G Reset Time Stamp

Format of the command: **G↵**
Format of the response: **g \r\n**
Effect: Reset the Time Stamp base.

H Buzzer

Format of the command: **H,frequency↵**
Format of the response: **h \r\n**
Effect: Configure the frequency of the Buzzer. 0 will turn it Off. 1 is 20Hz and 100 2 kHz. Please refer to section 3.2.3 to select your desired frequency.
Example: **H,80↵**

I Read Switches

Format of the command: **I** ↵
Format of the response: **i, value \r\n**
Effect: Read the value of the dip switch setting.
Note: switch 1 is LSB and 3 MSB (see figure 2.2).

J Scan the I2C Bus

Format of the command: **J** ↵
Format of the response: **j,extension_address \r\n**
Effect: Scan the whole I2C Bus to detect which extension is plugged. The returned value is in Hexadecimal.

K Unused

L Change Led State

Format of the command: **L, led_number, led_value**↵
Format of the response: **l \r\n**
Effect: Configure the state of the selected led (0 to 4). Value 0 will turn it Off, 1 = On, 2 = toggle and 3 = blinks.
Example: **L, 2, 1**↵
Note: If you change the state of the Power ON Led (Led4), it will stop blinking and change to your desired state.

M Read ambient light sensors (only an area)

Format of the command: **M, area** ↵
Format of the response: **m, area_ambiant_light_values \r\n**
Effect: Read the 10 bits (0 to 1024) brightness values of the sensors contains in the selected area. Area 0 is for the Front_left, Front and Front_right sensors. Area 1 is for the Left, Right and the Rear sensors. And area 2 is for the four ground sensors (Ground_Left, Ground_Front_left, Ground_Front_right, Ground_Right respectively).
Example: **M, 1** ↵
Note: A value of 0 means that the sensors is saturate with IR light, and a big value means that there's no IR light source in front of the sensor.

N Read proximity sensors

Format of the command: **N** ↵
Format of the response: **n, Left, Front_Left, Front, Front_Right, Right, Rear, Ground_Left, Ground_Front_Left, Ground_Front_Right, Ground_Right \r\n**
Effect: Read the 10 bit (0 to 1024) proximity value of each infra-red sensors.
Note: The smaller the value, the further the object is from it. A value of 900 (i.e.) means that an obstacle is very close from the sensor.

O Read ambient light sensors

Format of the command: **O** ↵
Format of the response: **o, Left, Front_Left, Front, Front_Right, Right, Rear,**

Ground_Left, Ground_Front_Left,
Ground_Front_Right, Ground_Right \r\n
Effect: Read the 10 bit (0 to 1024) brightness value of each infra-red sensors.
Note: A value of 0 means that the sensors is saturate with IR light, and a big value means that there's no IR light source in front the sensor.

P Read proximity sensors (only an area)

Format of the command: **P, area** ↵
Format of the response: **p, area_proximity_values \r\n**
Effect: Read the 10 bits (0 to 1024) proximity value of the sensors contains in the selected area. Area 0 is for the Front_left, Front and Front_right sensors. Area 1 is for the Left, Right and the Rear sensors. And area 2 is for the four ground sensors (Ground_Left, Ground_Front_left, Ground_Front_right, Ground_Right respectively).
Example: **P, 2** ↵
Note: A value of 0 means that the sensors is saturate with IR light, and a big value (
(~800) means that there's no IR light source in front of the sensor.

Q Reserved for Webots-K-Junior

R Read I2C Extension Bus

Format of the command: **R, Slave_Write_Address, Register_Address, Number_of_Register** ↵
Format of the response: **r, Register_Data\r\n**
Effect: Read a given number of registers of an I2C peripheral. Please provide register write address, i.e. odd address.
Examples: **R,C0,20,03** ↵
Reads three registers at address 0x20, 0x21 and 0x22 of peripheral 0xC0
Note: All parameters must in hexadecimal format. The response will be in decimal format (0 to 255)

S Reserved for Webots-K-Junior

T Read TV remote buffer

Format of the command: **T** ↵
Format of the response: **t, TV_Addr, TV_Data\r\n**
Effect: Read the last address and data bytes received by the TV remote receiver.

U Reserved for Webots-K-Junior

V Transmit a RC5 Data on the IR emitter

Format of the command: **V, TV_Addr, TV_Data** ↵
Format of the response: **v \r\n**
Effect: Send an address and a data through the IR emitter. All K-Junior close to the emitter, will receive the command as a TV remote control.

Example: **V,8,21** ↵

Note: The address value can be configure from 0 to 31 (5 bits), and the data from 0 to 63 (6 bits).