Proyecto Final

Hardware de Control de Plataforma Robótica Móvil con Arquitectura ARM y RTOS. Caracterización.

Martín Baudino **Santiago Pérez**

Director
Ing. Carlos Luis Candiani

C.I.I.I.
Ing. Gonzalo F. Perez Paina

Agosto 2010

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA

CENTRO DE INVESTIGACIÓN EN
INFORMÁTICA PARA LA INGENIERÍA

Agradecimientos

Este Proyecto Final de Grado no hubiese sido posible sin la gran ayuda de muchas personas. Por esto queremos agradecer en primer lugar a nuestras familias, por habernos dado el apoyo incondicional desde el inicio de nuestras carreras, por sus esfuerzos, sacrificios, tristezas y alegrías que compartieron y siguen compartiendo con nosotros a través de los años.

A nuestros amigos, aquellos que estuvieron siempre, en las buenas y en las malas, que con su consejo o simple presencia fueron pilares fundamentales para nunca bajar los brazos.

A nuestros compañeros de la facultad, por darnos el ánimo y apoyo necesario para seguir adelante en los tramos más difíciles, por acompañarnos desde adentro con su amistad y conocimientos, por haber compartido con nosotros la vida universitaria.

Al Centro de Investigación en Informática para la Ingeniería (C.I.I.I.), por habernos permitido realizar este proyecto íntegramente en sus instalaciones, por brindarnos todos los elementos y conocimientos técnicos necesarios para el correcto desarrollo del mismo.

A todos los becarios que forman parte del Centro, por su compañía, amistad y conocimientos. En especial al señor Daniel Marchetti, por haber dedicado horas de trabajo incondicional y gratuito, sin el cuál el desarrollo del proyecto no hubiese sido posible.

Al ingeniero Gonzalo F. Perez Paina, por su tiempo y dedicación a dirigir nuestro Proyecto Final de Grado, por sus conocimientos técnicos, paciencia y amistad.

A todos ustedes, estaremos siempre enormemente agradecidos,

Martín S. Baudino - Santiago Pérez

Índice general

Índice de figuras

Capítulo 1

Introducción

El proyecto surge como respuesta a una necesidad concreta del Centro de Investigación en Informática para la Ingeniería (CIII), de la Universidad Tecnológica Nacional, Facultad Regional Córdoba; buscando no solo brindar solución al caso específico, sino también dar lugar a nuevas investigaciones y proyectos sobre el tema y/o temas afines.

La necesidad original era "disponer de una plataforma sobre la cual ensayar y validar las investigaciones que se llevan a cabo en el CIII, las cuales se enmarcan en las áreas de visión por computadora, control automático y robótica" [6]. Para ello se diseñó y construyó íntegramente la unidad llamada "Robot Móvil de Arquitectura Abierta" (RoMAA) , que se describe brevemente más adelante.

El proyecto final de grado presentado es una solución integral que contiene tanto componentes de hardware: diseño y construcción de circuito impreso (PCB), con fuente de alimentación conmutada y microcontrolador ARM7TDMI de 32bits, como de software: sistema operativo de tiempo real (RTOS) embebido; y es una evolución de soluciones adoptadas anteriormente. Esta solución será utilizada para realizar las tareas de control (colectar información de sensores y enviar señales de estímulo a actuadores) y de comunicación con interfaz para aplicaciones de alto nivel.

Originalmente el software aplicado al control de la plataforma RoMAA carecía de un RTOS, siendo puramente código secuencial en C (big loop). La modificación del control de dicha plataforma, como así también del hardware vinculado, tenía una cierta complejidad debido a esta estructuración. La migración de dicho código a un RTOS permitirá que futuras modificaciones se realicen de una manera mas sencilla, eficiente e independiente del resto del

programa.

Otro de los objetivos de esta migración o portación es poder caracterizar el RTOS en este hardware específico. De esta manera, antes de realizar cualquier aplicación en la plataforma robótica, se sabrá con exactitud de qué características y limitaciones se dispone, ahorrando esfuerzos y permitiendo un trabajo mucho mas preciso y eficiente.

1.1. Marco de desarrollo

A continuación se presenta una breve descripción del Robot Móvil de Arquitectura Abierta (RoMAA) como marco de desarrollo del presente trabajo de tesis, figura 1.1. El robot RoMAA presenta una estructura de tres ruedas, con dos ruedas de tracción controladas individualmente y una rueda giratoria (o rueda castor) detrás. Esta arquitectura conocida como de tracción diferencial es ampliamente utilizada en investigación de navegación autónoma en ambientes interiores por tener gran maniobrabilidad. En este tipo de robot los giros se realizan controlando la velocidades independientes de cada rueda, y permite lograr radios de giro desde cero o giro sobre su propio eje hasta infinito o trayectoria en línea recta. El sistema fue concebido para ser operado desde microcontroladores diseñados ad-hoc, palms, computadoras portátiles, etc. [6]. Actualmente su uso principal es mediante una computadora portátil lo que da mayor flexibilidad al momento de su elección, y permite tener siempre actualizado el equipo de cómputo a bordo.

La descripción del movimiento de este tipo de robots se representa en un sistema de coordenadas cartesianas mostrado en la figura 1.2 y se modela mediante las ecuaciones (1.1) y (1.2); donde w_L y w_R son las velocidades angulares de la rueda izquierda y derecha respectivamente; v y ω la velocidad lineal y angular del sistema de coordenadas fijo al robot, R es el radio de la rueda y b la distancia entre ruedas.

La figura 1.3 muestra los principales componentes de la plataforma robótica móvil como ser los motores de tracción de corriente continua, encoders ópticos incrementales y las baterías de suministro de energía eléctrica. La figura 1.4 presenta las distintas partes del proyecto RoMAA, tanto hardware como software, y la relación entre cada una de ellas. El software puede dividirse en una parte de bajo nivel o de control embebido del robot y otra de alto nivel que se ejecuta en el computador a bordo.

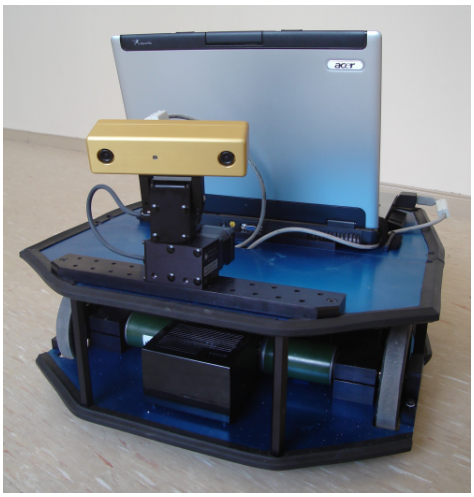


Figura 1.1: Robot móvil RoMAA

Para cerrar el lazo en velocidad de los motores y controlar así la tracción del robot para generar la trayectoria deseada [7] como se indica en las ecuaciones (1.1), cada motor lleva asociado un encoder óptico incremental. El control de la velocidad y el sentido de giro de los motores de corriente continua se realiza mediante modulación de ancho de pulso (PWM) a una frecuencia de 20KHz, a fin de evitar vibraciones audibles. Las señales PWM accionan drivers de potencia tipo puente "H", que tienen capacidad para manejar corrientes medias de hasta 10A sin necesidad de ventilación forzada.

El vehículo, además, esta equipado con 2 baterías de 12V y 26Ah conectadas en serie. Para la elección se supuso una autonomía de 7 horas, con un consumo promedio de alrededor de 35W para el sistema de tracción, y otros 35W para el resto del equipamiento.

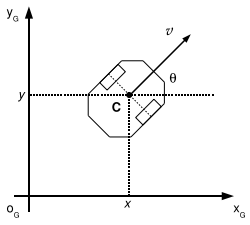


Figura 1.2: Robot móvil y sistema de coordenadas

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} R/2 & R/2 \\ -R/b & R/b \end{bmatrix} \begin{bmatrix} \omega_R \\ \omega_L \end{bmatrix} \quad (1.1)$$

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \omega \end{aligned} \quad (1.2)$$

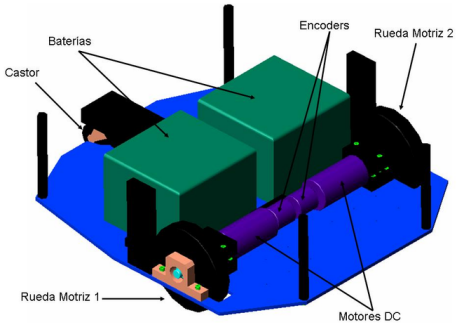


Figura 1.3: Representación 3D del robot RoMAA

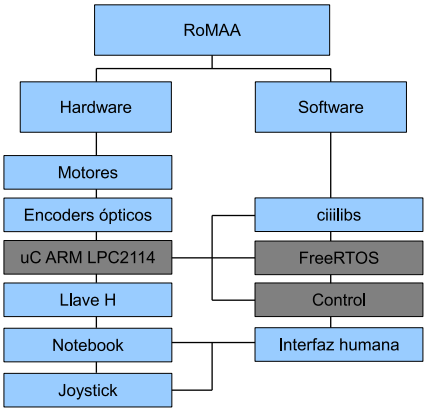


Figura 1.4: Proyecto RoMAA

1.2. Objetivos

Los objetivos generales del sistema de control están en sintonía con las necesidades que busca satisfacer. Es por eso que tanto hardware como software debieron ser diseñados de forma modular para que el producto final sea flexible, extensible y de fácil modificación. Cada módulo debe ser pensado para realizar una tarea específica y para que además de ser modificables, permitan ser reemplazados íntegramente por soluciones que pudieran desarrollarse en el futuro. También es necesario que el producto final cumpla con las especificaciones técnicas de las distintas interfaces que ya posee actualmente el robot RoMAA, para poder garantizar la compatibilidad con las soluciones anteriores. Finalmente se fijó como requerimiento tener en cuenta a la comunidad y al mercado local, tanto en la elección de los materiales disponibles, como en la utilización y generación de "know-how" que fomenta el progreso tecnológico.

Los objetivos particulares, que permiten llevar a cabo el proyecto se dividen en dos grandes áreas. Por un lado se encuentran los requisitos técnicos, que otorgan una orientación en cuanto a las tecnologías que se deberán utilizar, y por otro la metodología de trabajo, que delimita las herramientas, los costos, los tiempos y la forma en que se lleva a cabo el proyecto.

Para cumplir con la filosofía de desarrollo adoptada, es necesario

- que el software utilizado sea fácilmente modificable y extensible
- utilización de software de código abierto, que permita su libre modificación y distribución

- elección de componentes de hardware disponibles en el mercado local
- que el desarrollo en su totalidad sea una síntesis y evolución de desarrollos anteriores

Además, para poder realizar la tareas de control del robot móvil RoMAA es necesario contar con la capacidad de

- generar señales en Modulación de Ancho de Pulso (PWM) para controlar los motores de tracción
- sensar periódicamente las salidas de los encoders ópticos para el lazo cerrado de control de los servomotores
- mantener un registro de las señales de los encoders ópticos para el cálculo de la odometría
- realizar el cálculo de las variables de control en un intervalo fijo de tiempo
- comunicarse con otros dispositivos a través de alguna interfaz estándar
- cuidar el consumo de energía para lograr una mayor autonomía de las Baterías

Capítulo 2

Placa de Control

2.1. Versiones anteriores

Para cumplir con las tareas requeridas del sistema embebido de la plataforma robótica móvil es necesario contar con un controlador de alta gama, siendo de fundamental importancia la capacidad y velocidad de cálculo, la posibilidad de operar con punto flotante y los periféricos necesarios para la aplicación. Es por eso que se utilizó una placa de propósitos generales con microcontrolador ARM7TDMI de 32 bits desarrollada en el CIII [8](ver figura 2.1) y utilizada en diversos proyectos en el Centro. En concreto se utilizó el microcontrolador (μC) LPC2114 de NXP [3] para el que se generaron las librerías necesarias para su programación, las CIIIEmbLibs [9].

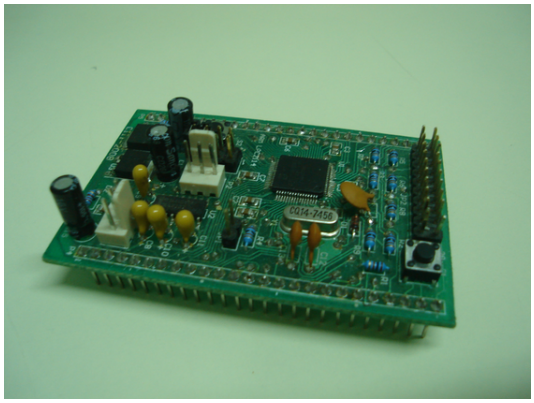


Figura 2.1: Placa multipropósito con μC ARM7.

La placa de control desarrollada, es una evolución de diseños anteriores utilizados en el CIII. En primera instancia, se creo una placa de control

"madre", donde solo poseía reguladores lineales de voltaje para descender el proveniente de las baterías a valores útiles tanto para la lógica de control como para el funcionamiento de los motores, y dos hileras de conectores hembras donde se encastraba la placa multipropósito. Esta placa de control, presenta varias deficiencias, como la utilización de reguladores lineales para alimentar los circuito digitales de bajo voltaje lo que genera un gran desperdicio de la energía de las baterías, la poca robustez de los contactos móviles por la utilización de la placa multipropósito, entre otros (ver figura 2.2).

La segunda versión de esta placa madre, fue también diseñada para funcionar con la placa multipropósito, pero ya sin los reguladores lineales, sino en cambio con dos fuentes conmutadas, una de 5V y otra de 12V. Esta versión, pese a hacer un uso mas eficiente de la energía de las baterías, sigue presentando el problema de los contactos de la placa multipropósito.
La figura 2.3 muestra la placa prototipo de control con las fuentes conmutadas y la figura 2.1 la placa de propósitos generales del microcontrolador LPC2114 utilizado en este proyecto.

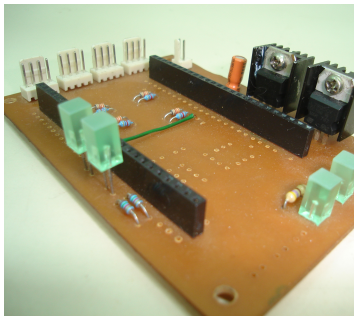


Figura 2.2: Placa madre con reguladores lineales.

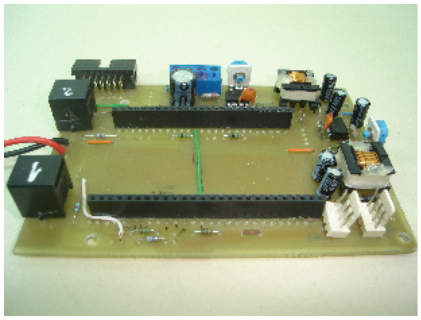


Figura 2.3: Placa madre con fuente conmutada.

Debido a estos motivos, se propuso la realización de una placa madre de dedicación exclusiva para esta aplicación en la que dichas falencias sean eliminadas luego de un minucioso estudio. El resultado de todo este proceso fue el diseño y construcción de la placa madre de control que a continuación se detalla.

2.2. Modificaciones en el nuevo diseño

La utilización del microcontrolador ya montado sobre la placa única de control, le da al sistema mayor robustez, evitando los contactos móviles,

posibles inducciones de ruido, etc. El cambio del tipo de conectividad que la placa madre brinda con la CPU externa, también es un cambio importante, ya que mientras el protocolo sigue siendo RS-232 estándar, la conexión es mediante USB, presente en todas las CPU actuales.

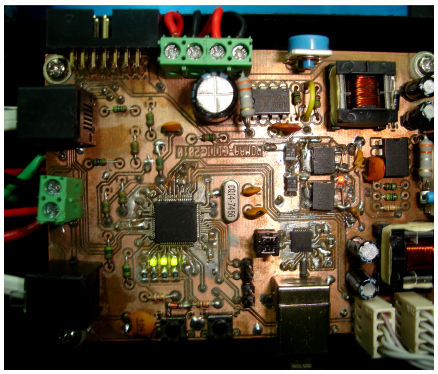


Figura 2.4: Placa de control desarrollada en este trabajo.

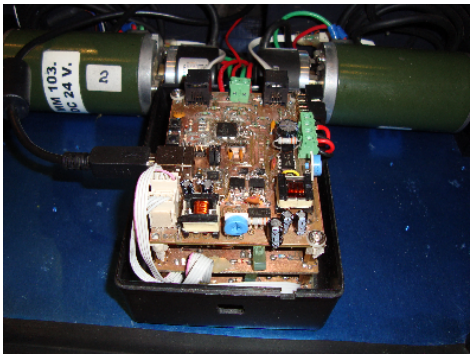


Figura 2.5: Montaje sobre el robot RoMAA.

En vez de poseer acceso a todos los pines del microcontrolador generando un uso intensivo e innecesario del espacio para una aplicación definitiva como es ésta, se realizó el diseño solo con los pines necesarios, teniendo en cuenta algunas variaciones, como por ejemplo algunos puertos accesibles para un posible debugg. La alimentación de los circuitos digitales, se llevo a cabo ahora mediante fuentes conmutadas, evitando el uso de reguladores lineales directamente conectados a la batería, con el ahorro de energía y aumento de eficiencia de la aplicación móvil que ello implica.

Los conectores de los encoders fueron reemplazados por conectores RJ-11, los que proveen una mayor robustez y facilidad de conexión/desconexión. Las conexiones desde la batería y hacia las llaves H, fueron también cambiadas por conectores mas robustos. Se agregó un conector tipo panel, para la conexión con un panel exterior, que brinda acceso a ciertos puertos del microcontrolador desde el exterior, haciendo mas práctico su uso.

Todo el control del RoMAA cuenta de tres placas:

- Dos llaves H para el control de los motores de C.C.
- Una placa madre de control general.

Estas tres placas están ubicadas una encima de otra, dentro de una caja cerrada de medidas acotadas ubicada en la base del RoMAA, junto a los motores

y la batería. Por lo tanto, el diseño las dimensiones de la placa madre de control es un requisito importante que fue tenido en cuenta en todo el desarrollo.

En la figura 2.4, se puede observar la placa de control implementada ya montada. En la figura 2.6 se puede ver en detalle el montaje de la placa de control sobre las dos placas de llaves H y sus conexiones.

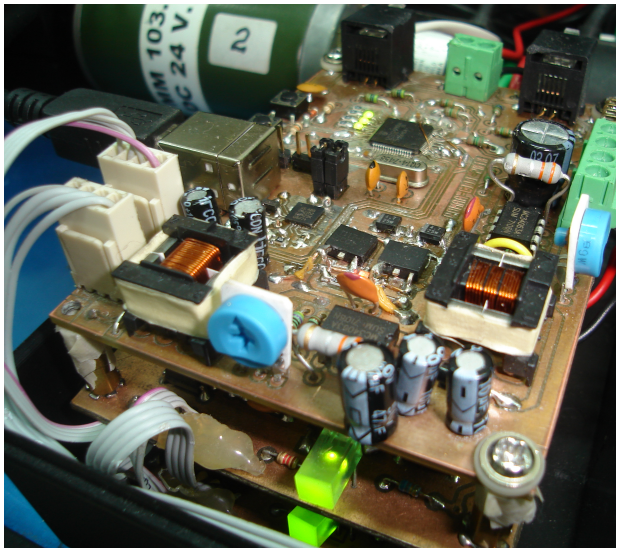


Figura 2.6: Placa de control montada sobre las dos placas de llave H.

2.2.1. Microcontrolador LPC2114

El microcontrolador LPC2114[3] de la empresa NXP, está basado en el núcleo ARM7TDMI-S de 32-bits. Además de la frecuencia de trabajo (hasta 60MHz) y la cantidad de periféricos con los que cuenta, se destaca la posibilidad de programarlo "in circuit". Otro factor determinante para su elección es su relación costo/performance, dado que tanto el costo del IC en si mismo, como de los componentes que le dan soporte, tienen un valor relativamente bajo y son fáciles de conseguir en el mercado local. También cabe destacar que dentro de esta gama media-alta para microcontroladores el encapsulado LQFP64 de este componente puede soldarse con mayor facilidad que otras opciones que se encuentran actualmente en el mercado (QFN, BGA).

2.2.2. LEDs y GPIO multipropósitos

A pesar de que es una aplicación definitiva y no todos los pines del microcontrolador fueron ruteados en el PCB, se consideró necesario y útil a futuro, poseer algunas formas de debug. Para ello, se diseño el acceso a cuatro entradas/salidas digitales GPIO mediante pines montados sobre el PCB, aptos para conectar un osciloscopio u conector hembra. Como se dijo, estas GPIO no tienen un fin específico, por lo que pueden ser usados como entrada/salida para la aplicación que este corriendo en el microcontrolador.

A su vez, se realizó el conexionado de cuatro LEDs de montaje superficial a otras cuatro GPIO, con el mismo fin. Dichos LEDs activos por bajo, ofrecen una herramienta visual para el debug de control de la aplicación. Cabe aclarar, que tanto los conectores de GPIO como los LEDs, están ubicados en la placa madre, siendo inaccesibles desde el exterior del gabinete de control. Por lo tanto, son útiles solo para el control de la aplicación en proceso de desarrollo y no para la aplicación final.

2.2.3. Conversor USB-RS232

Como es un requisito utilizar protocolos estándar, originalmente la comunicación con la PC se hacía a través de la UART del microcontrolador LPC2114 por RS232. En este desarrollo se ha interpuesto un conversor RS232-USB, teniendo en cuenta que la mayoría de las computadoras portátiles que se encuentran hoy en el mercado no disponen de puerto serie. Las opciones de chips comerciales son varias, uno de los mas comunes es el FT232BM, de la compañía FTDI. Este chip, pese a su conocido y confiable funcionamiento, tiene el inconveniente de utilizar varios componentes externos, como un cristal, memoria ROM, entre otros.

Sin embargo, debido a la disponibilidad en el mercado local, su bajo precio, tamaño y demostrado buen funcionamiento, se utilizó el chip CP2102[10], de la compañía Silicon Labs. Algunas características del mismo se detallan a continuación:

- Características del chip
    - Transmisor/receptor integrado, sin necesidad de resistores externos.
    - Clock integrado, no necesidad de cristal externo.
    - EEPROM de 1024 byte integrada para la identificación del vendedor, producto, número de serie, descriptor de energía, número de versión y cadena revisión del producto.

- Regulador de 3,3V incorporado. Alimentado directamente desde el bus de USB o de forma autónoma, utilizando el regulador incorporado o haciendo un by-pass del mismo.
- Encapsulado 28 pin QFN (5 x 5mm) (ver figura 2.8).

- Características del controlador de funciones del USB
  - Cumple con las especificaciones del USB 2.0, full-speed (12Mbps).
  - Soporta el estado de suspensión a través de los pines SUSPEND.

- BUS de datos seriales asíncronos (UART)
  - Todas las señales de módem y handshaking.
  - Baudrates desde 300bps a 1Mbits
  - Buffer de recepción de 576 Byte y buffer de transmisión de 640 Byte.

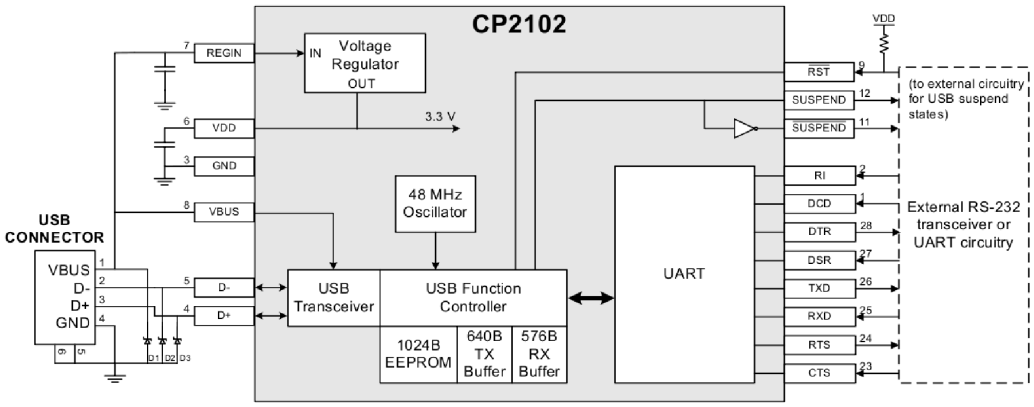


Figura 2.7: Circuito esquemático del conversor CP2102

La comunicación del CP2102 con el microcontrolador LPC2114, se realiza mediante la UART0 de éste. El LPC2114 posee dos UARTs, donde solamente la UART1 brinda todas las señales de handshaking. Este microcontrolador puede programarse de dos maneras: en forma serial a través de la UART0 o mediante el conector para debug JTAG. Debido a que la placa madre desarrollada fue pensada como producto final, la utilización de JTAG era innecesaria, quedando disponible solo una forma de programar al microcontrolador. Por estos motivos, es que se eligió a la UART0 para conectarse al conversor CP2102, usándose por lo tanto esta UART, tanto para la programación como

para la comunicación normal del flujo del software de aplicación. La UART1 queda inhabilitada.

Se decidió alimentar al CP2102 de la forma "self-powered", es decir, mediante una alimentación externa y no mediante el bus. Pese a que el conversor posee una regulador de 3,3V, no se lo utiliza, ingresando directamente con 3,3V desde el regulador usado también para el LPC2114. Esta determinación se debe a que la alimentación de los reguladores de la placa madre provienen de la fuente conmutada de 5V. Cualquier variación de ésta, debe ser soportada por los reguladores. El regulador del CP2102, es un regulador integrado, de baja corriente que soporta un máximo de 5,25V, por lo que cualquier variación que se produzca en la fuente conmutada, llevaría a la destrucción del mismo. Por lo tanto, al usar el regulador externo lineal de 3,3V utilizado también para el microcontrolador que soporta tensiones muchos mayores, se garantiza la protección del conversor con rangos de variaciones mucho mayores.

Como se dijo anteriormente, el encapsulado del conversor es del tipo QFN28 (ver figura 2.8). Este encapsulado de tan solo 5x5mm y teniendo en cuenta la mínima cantidad de componentes externos (solo capacitores para garantizar el nivel de continua de la alimentación), hacen que el CP2102 ocupe un mínimo espacio en la aplicación la que se desee utilizar. Vale aclarar que este menor tamaño agrega complejidad al montaje y soldado, aunque con un buen diseño del PCB y herramientas de soldado precisas, no produjo mayores inconvenientes y su funcionamiento ha sido exitoso en todos los diversos montajes realizados.

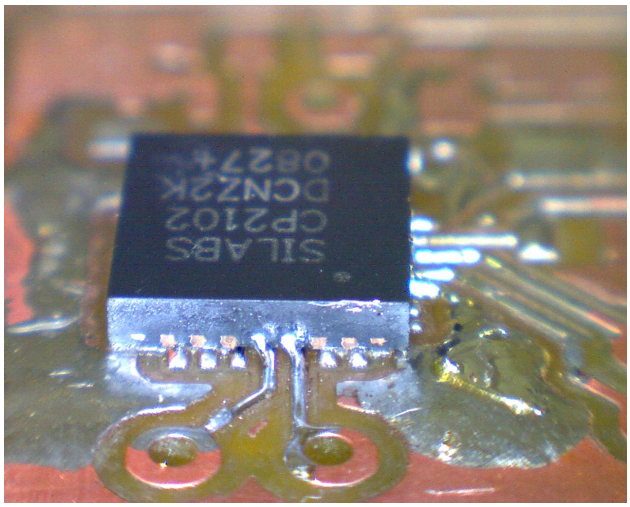


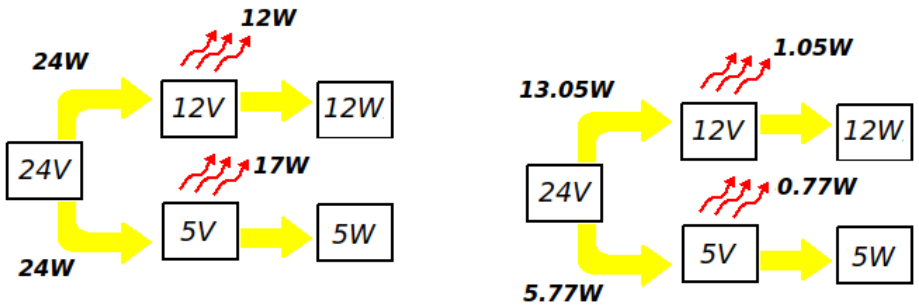
Figura 2.8: Encpasulado QFN-28 del conversor CP2102

2.2.4. Fuentes conmutadas

A la hora de considerar el diseño de una fuente de alimentación, capaz de proporcionar distintos voltajes a partir de una única tensión, se deben tener en cuenta cuestiones tales como el costo, la eficiencia y el tamaño. Las opciones para hacer esto eran dos con sus respectivas ventajas y desventajas. Por un lado utilizando los reguladores lineales y por el otro los reguladores tipo conmutados (switching) de topología step down. Los primeros tienen la ventaja de un costo menor, un tamaño reducido y poca necesidad de componentes externos; en el caso de los segundos, su tamaño y necesidad de componentes externos es mayor, sin embargo su eficiencia es mucho mayor, figure 2.9.

Como la mayoría de los robots móviles utilizan baterías como fuente de energía, la autonomía de las mismas resulta de vital importancia debiendo los sistemas de a bordo hacer un uso eficiente de la energía. Es por ello que se diseñaron y construyeron dos fuentes conmutadas para alimentar la electrónica de control y componentes asociados.

Para la presente aplicación en concreto se necesitaba obtener 5 y 12V a partir de una batería de 24V. Dada la escala de producción y la potencia requerida el factor costo se pudo dejar de lado; sin embargo, el factor eficiencia era clave. Por ello se eligió utilizar un circuito integrado (en concreto el MC34063 en modo buck) que posee todas las funciones básicas para implementar un conversor DC-DC, con el que se consigue un tamaño y eficiencia aceptable. En la figura 2.9 se puede ver una comparación entre los dos sistemas y que si bien el modelo de la fuente switching es muy simplificado, ayuda a dar una magnitud a la diferencia entre ambos tipos de sistemas. Las ecuaciones (2.2) y (2.3) muestra el rendimiento para cada tipo de fuente.



(a) fuente lineal

(b) fuente conmutada

Figura 2.9: Opciones de fuente de alimentación a utilizar

$$\eta(\,\%) = \frac{Potencia_{salida}}{Potencia_{suministrada}} \times 100 \tag{2.1}$$

$$\eta_{lineal}(\,\%) = \frac{(12W + 5W)}{(24W + 24W)} \times 100 = 35{,}4\,\% \tag{2.2}$$

$$\eta_{switching}(\,\%) = \frac{(12W + 5W)}{(13{,}05 + 5{,}77W)} \times 100 = 90{,}3\,\% \tag{2.3}$$

Circuito utilizado – MC34063[2] de ON Semiconductors en modo buck Básicamente el integrado, posee un latch S-R, que controla la salida de potencia. Este latch a su vez es seteado (activado) por el comparador de tensión (que se encarga de mantener la tensión de salida de la fuente) y por el oscilador, en su flanco de subida; y es reseteado por el oscilador, en su flanco de bajada.

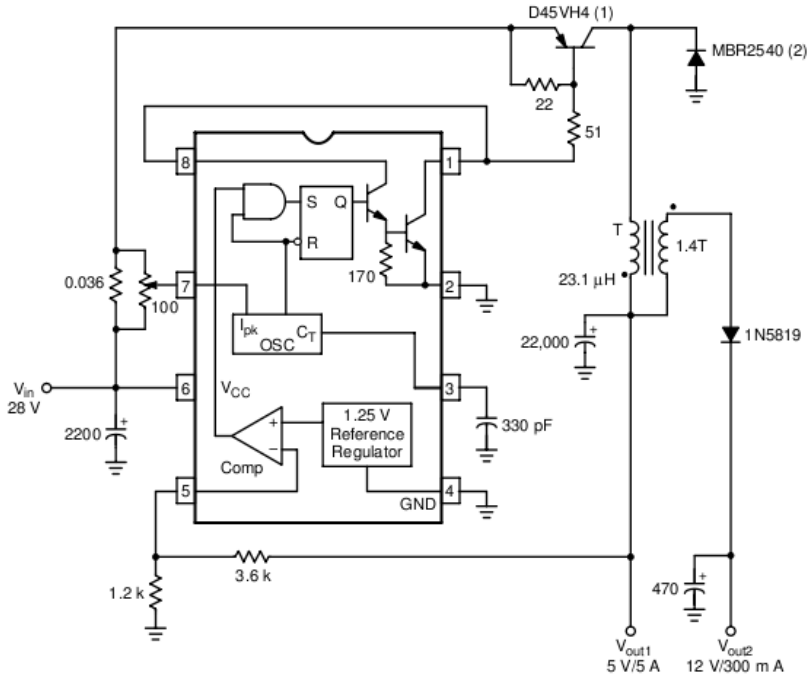


Figura 2.10: Circuito MC34063A en modo conversor Step Down[1]

Como se ve en el esquema, el comparador de tensión, solo puede activar al latch durante el flanco de subida de la señal generada por el oscilador. O sea que el ancho de pulso de salida del latch, puede variar entre cero y

Calculation	Step−Up	Step−Down	Voltage−Inverting
t_{on}/t_{off}	$\frac{V_{out} + V_F - V_{in(min)}}{V_{in(min)} - V_{sat}}$	$\frac{V_{out} + V_F}{V_{in(min)} - V_{sat} - V_{out}}$	$\frac{\lvert V_{out}\rvert + V_F}{V_{in} - V_{sat}}$
$(t_{on} + t_{off})$	$\frac{1}{f}$	$\frac{1}{f}$	$\frac{1}{f}$
t_{off}	$\frac{t_{on} + t_{off}}{\frac{t_{on}}{t_{off}} + 1}$	$\frac{t_{on} + t_{off}}{\frac{t_{on}}{t_{off}} + 1}$	$\frac{t_{on} + t_{off}}{\frac{t_{on}}{t_{off}} + 1}$
t_{on}	$(t_{on} + t_{off}) - t_{off}$	$(t_{on} + t_{off}) - t_{off}$	$(t_{on} + t_{off}) - t_{off}$
C_T	$4.0 \times 10^{-5}\, t_{on}$	$4.0 \times 10^{-5}\, t_{on}$	$4.0 \times 10^{-5}\, t_{on}$
$I_{pk(switch)}$	$2I_{out(max)}\left(\frac{t_{on}}{t_{off}} + 1\right)$	$2I_{out(max)}$	$2I_{out(max)}\left(\frac{t_{on}}{t_{off}} + 1\right)$
R_{sc}	$0.3/I_{pk(switch)}$	$0.3/I_{pk(switch)}$	$0.3/I_{pk(switch)}$
$L_{(min)}$	$\left(\frac{(V_{in(min)} - V_{sat})}{I_{pk(switch)}}\right) t_{on(max)}$	$\left(\frac{(V_{in(min)} - V_{sat} - V_{out})}{I_{pk(switch)}}\right) t_{on(max)}$	$\left(\frac{(V_{in(min)} - V_{sat})}{I_{pk(switch)}}\right) t_{on(max)}$
C_O	$9\,\frac{I_{out} t_{on}}{V_{ripple(pp)}}$	$\frac{I_{pk(switch)}(t_{on} + t_{off})}{8V_{ripple(pp)}}$	$9\,\frac{I_{out} t_{on}}{V_{ripple(pp)}}$

V_{sat} = Saturation voltage of the output switch.
V_F = Forward voltage drop of the output rectifier.

The following power supply characteristics must be chosen:

V_{in} − Nominal input voltage.
V_{out} − Desired output voltage, $\lvert V_{out}\rvert = 1.25\left(1 + \frac{R2}{R1}\right)$
I_{out} − Desired output current.
f_{min} − Minimum desired output switching frequency at the selected values of V_{in} and I_O.
$V_{ripple(pp)}$ − Desired peak−to−peak output ripple voltage. In practice, the calculated capacitor value will need to be increased due to its equivalent series resistance and board layout. The ripple voltage should be kept to a low value since it will directly affect the line and load regulation.

NOTE: For further information refer to Application Note AN920A/D and AN954/D.

Figura 2.11: Ecuaciones para fuente conmutada con MC34063A[2]

el máximo del flanco de subida. Esto se debe a que para setear el latch, se necesita que estén presentes, el flanco de subida del oscilador, que es algo obviamente periódico y que a su vez el comparador indique que la tensión de salida esta por debajo de lo fijado, lo cual puede ocurrir en cualquier momento. Para resetear el latch, la única forma, es la llegada del flanco de bajada del oscilador. Durante el mismo el comparador de tensión no puede activar el latch. Dentro de este punto, entra también el limitador de corriente, el cual evita que se suministren corrientes por encima de un valor prefijado. Este limitador, funciona sensando la caída de tensión a traversa de un resistor que conectado entre V_{CC} y el transistor que controla la bobina. Cuando se activa este sensor, aumenta notablemente la corriente de carga del capacitor C_T, con lo que "adelanta"la llegada del flanco de bajada del oscilador, y por lo tanto, el apagado del transistor que maneja a la bobina.

En este esquema se muestra el funcionamiento del integrado en distintas condiciones:

1. En esta parte,como se ve la pendiente de carga aumenta, esto se debe a una sobre-corriente en el sensor de corriente, esto puede ser por una carga excesiva en la salida o un transitorio por ejemplo. En este caso, el periodo de la señal se reduce por la disminución del tiempo que tarda

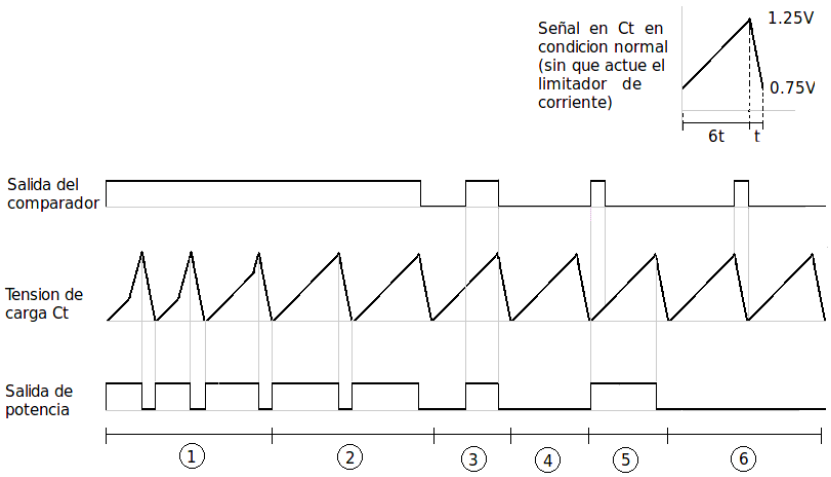


Figura 2.12: Detalle del funcionamiento de C_T

en llegar a su máximo punto la señal. En ningún caso el tiempo del flanco de bajada en C_T varia.

2. En esta región, la salida esta trabajando al máximo de su capacidad (sin exceder la corriente máxima). Este es el máximo tiempo de encendido continuo que puede generarse.

3. En este punto, al iniciarse el flanco de subida, la tensión era superior a la requerida, pero luego cayo, por lo que el comparador se activo, activando al mismo tiempo la salida de potencia.

4. Aquí, al ser la tensión de salida la requerida, no se activo la salida, pues si bien la rampa en C_T se sigue generando, no hay señal del comparador. Durante un funcionamiento normal pueden pasar mas de un ciclo sin que se requiera un nuevo encendido de la salida de potencia.

5. En este punto, se recibió la señal desde el comparador , justo al inicio de la rampa de subida, pero unos instantes después termina la señal del comparador. En este caso, la salida permanecerá encendida hasta el fin de la rampa de subida, ya que el comparador de tensión, puede iniciar un nuevo disparo en la salida de potencia, pero no puede detenerlo.

6. En este ultimo caso, la señal del comparador entro durante el flanco de bajada. En este caso, el mismo no activa la salida de potencia, aunque si esta señal continuara, activaría la salida durante la próxima rampa de subida.

Para un funcionamiento correcto, las activaciones de tipo 1 y 5 no deberían ocurrir, en cambio, las del tipo 2 y 3 son normales.

Frecuencia de funcionamiento La frecuencia de funcionamiento se fija mediante el capacitor C_T. Este genera la base de tiempo del sistema, pero no es la frecuencia a la que va a funcionar la bobina. Cuando se fija este capacitor, lo que se hace es fijar el tiempo máximo de encendido del transistor de potencia, pero la frecuencia a la que este se va a encender, depende del circuito RLC que esta a la salida de la fuente. Si no se almacena suficiente energía en un ciclo de carga, la frecuencia de encendido se incrementará, si por el contrario la energía almacenada es excesiva, la frecuencia de encendido debe bajar.

Circuito de base de tiempo Hay que tener especial precaución al realizar diseños basados en el integrado MC34063, ya que el circuito que genera la base de tiempo no cumple con las especificaciones dadas por el fabricante, con lo cual el calculo del mismo no dará a la fuente la frecuencia requerida. El valor requerido de C_T, el capacitor que se usa para generar la base de tiempo, se debe calcular con la siguiente formula:

$$C_T = I_{chg}(min) \times \frac{\Delta t}{\Delta v} \tag{2.4}$$

donde $I_{chg}(min)$ es la corriente de carga, en este caso usamos la mínima (el rango es de 24μA a 42μA, 35μA típico); Δv es la diferencia de tensión pico a pico, generada por el integrado. Este valor debe medirse para evitar inconvenientes, sin embargo la prueba de varios integrados dio un valor típico de 1V.

Diseño del inductor Para el diseño del inductor, se utilizó un núcleo toroidal de polvo de metal. Primeramente se hizo uso de la información obtenida de un catalogo, pero los resultados discrepaban con las mediciones hechas en el circuito. Se considera recomendable calcular los parámetros en base a pruebas hechas con el núcleo que se va a utilizar en la práctica. Para esto, se construye el inductor, basado en los datos del catalogo. Luego se lo coloca en el circuito y se miden la tensión y la corriente en la bobina, así como también la duración del encendido de la misma. Con estos valores se puede determinar el valor de Al correspondiente a ese núcleo en particular, con lo que podremos calcular el numero correcto de espiras a utilizar.

$$L = \frac{V_L \times t(on)}{I_L} \tag{2.5}$$

donde $t(on)$ es la duración del encendido de la bobina. Con esto se calcula la inductancia actual del inductor. Luego,

$$A_L = \frac{L}{N^2} \tag{2.6}$$

$$N = (\frac{L[nH]}{A_L[nN/N^2]})^{\frac{1}{2}} \tag{2.7}$$

donde N es el número de espiras requeridas.

Consideraciones y criterios para el diseño de la placa Para el diseño de la placa se tuvieron en una serie de consideraciones a fin de optimizar el funcionamiento de la misma.

1. Las pistas que llevan altas corrientes en conmutación, deben ser lo mas cortas y anchas que sea posible.
2. La masa del integrado se debe conectar en un solo punto con la masa de potencia.
3. Para los capacitores, tanto el de entrada de alimentación, como los que van a la salida de la fuente, deben ser implementados de ser posible en mas de un capacitor, a fin de reducir su resistencia en serie.
4. Al montar los componentes, estos deben soldarse con los terminales lo mas corto posible, a fin de reducir la resistencia térmica del sustrato del mismo con las pistas, a fin de usar las mismas como disipador.
5. Colocar un capacitor de 100nF lo mas cerca posible de los pines de alimentación del integrado para eliminar ruido.

2.2.5. Conectores

La robustez y versatilidad de las conexiones que brinda la placa madre, fue un requisito muy importante a tener en cuenta. Desde los tipos de protocolos y velocidad de trama, hasta el tipo de conector y disponibilidad en el mercado local fueron abordados. Todos han sido pensado para la aplicación específica, es decir, para brindar un uso efectivo y seguro a los desarrollos e investigaciones que se llevarán a cabos en el RoMAA controlados por esta placa madre.

Conector USB-B Como se describió anteriormente, uno de los grandes cambios fue el tipo de comunicación con el que interactúa una CPU externa con el controlador embebido. Para ello fue muy importante el tipo de conector utilizado en el PCB, ya que probablemente sea de entre todos los conectores el que mas uso y por lo tanto desgaste tenga. Como lo especifica la norma, el conector USB debe ser tipo B hembra, ya que el dispositivo embebido no funciona como Host sino como Slave. La forma de este conector da idea de robustez mecánica importante, por lo que la utilización de un tipo mini B fue descartada.
El tipo de cable USB a utilizar para la conexión con la PC también es un requisito importante. Debido a que el conversor USB-Serie funciona con el protocolo USB 2.0 (12Mbps), el cable utilizado debe soportar esta alta taza de transferencia. Es recomendable que posea también un filtro inductivo alrededor de los conductores en la cercanía del conector tipo B macho. Estas características, brindan seguridad y confiabilidad a la interfase CPU-Placa madre.

Conectores RJ-11 para encoders La conexión con ambos encoders, también necesita de un tipo flexible y robusto a la vez. Luego de analizar y experimentar con varias alternativas, se decidió la utilización del tipo RJ-11. Este tipo de conector es muy popular en el mercado local, económico y brinda una forma fácil de conexión desconexión así como una confiabilidad en la conducción de los contractos.

Conectores PWM para las llaves H La parte de baja tensión o control de las llaves H, se interconectan con la placa madre de control mediante dos conectores de cuatro terminales. A través de ellos se provee de los PWM proveniente del microcontrolador y de los 12V de alimentación desde la fuente conmutada de dicho voltaje. Este tipo de conexión permite un fácil y rápido conexión/desconexión de los mismos, para realizar pruebas, intercambiarlos, etc. La alimentación a través del conector provee también una forma de aislación de la alimentación de baja potencia con la de alta potencia de las llaves H.

Conector placa externa El RoMAA ha sido diseñado para la investigación y el desarrollo, como se describió en capítulos anteriores. La instalación y utilización de cámaras y diversos tipos de sensores calibrados, hacen que la placa de control este prácticamente inaccesible durante el funcionamiento

del RoMAA. Es por ello que se pensó en la utilización de un panel externo, con el cual se pueda acceder a partes fundamentales de la placa de control, mediante un conector tipo panel. Mediante este conector y el panel externo, se pueden acceder a los puertos de Reset y Bootloader del microcontrolador, necesarios para la grabación de nuevos softwares en el mismo. También se brinda acceso a un número limitados de puertos del microcontrolador para ser utilizados como señalización de un correcto funcionamiento del software, agregándole versatilidad y aislamiento a la caja de control donde se ubican las tras placas en conjunto.

Terminales alimentación desde la batería La aplicación principal del RoMAA es del tipo móvil, por lo que se lo alimenta con baterías. Sin embargo, durante el desarrollo de aplicaciones y cierto tipos de testeos, se lo utiliza con fuente de alimentación externa. Es por este motivo, que la entrada principal de alimentación de todo el conjunto de placas de control, se realiza mediante un conector doble ajustable presente en la placa madre. De esta manera, es muy simple cambiar el tipo de alimentación sin molestias para el usuario.

Terminales alimentación a las llaves H De la misma manera que con la alimentación desde las baterías, las dos placas que poseen las llaves H, se alimentan desde la placa de control mediante conectores. Estos son del mismo tipo que el de las baterías y se encuentran directamente conectados a este. Separar los conectores brinda portabilidad al sistema pudiendo alimentar cualquiera de las placas de las llaves H individualmente sin interferir unas con otras ni con la placa madre de control.

Capítulo 3

Microcontrolador de 32 bits de arquitectura ARM

Para la elección de una unidad de procesamiento es necesario considerar diversos aspectos, detallados a continuación.

3.1. Aspectos para la elección de una unidad de procesamiento

3.1.1. Performance

Es una estimación de la capacidad de procesamiento y generalmente se mide en MIPS (Millones de Instrucciones por Segundo). Otros factores que afectan el rendimiento, pero de forma más compleja son:

- Velocidad y tipo de memorias utilizadas: pueden introducir çuellos de botella", es decir tiempos en que la unidad de proceso espera para leer las instrucciones de la memoria.
- Cantidad de ciclos por instrucción: depende del tipo de arquitectura utilizada. En general los procesadores CISC (Complex Instruction Set Computing) necesitan una cantidad variable de ciclos, dependiendo de la instrucción a ejecutar, mientras que los de arquitectura RISC pueden llegar a ejecutar una instrucción por ciclo.
- Cantidad de Registros de Propósito General (GPR): Permite realizar operaciones aritméticas entre múltiples registros, reduciendo la necesidad de acceder a memoria RAM para guardar los resultados intermedios. Esto disminuye en gran medida la latencia del sistema.

3.1.2. Feature Set

Dentro de las distintas arquitecturas pueden existir varios fabricantes, que buscan diferenciar sus productos agregando en sus ICs distintos periféricos, cantidades y tipos de memorias diferentes, e incluso hasta integrando microcontroladores con dispositivos de lógica programable (PLDs). Algunos de los periféricos incluidos en los microcontroladores modernos son:

- MACs Ethernet 10/100 Base-T
- Interfaces para buses CAN o USB
- Transmisores de RF
- Procesadores gráficos
- Puertos de Entrada/Salida de Propósitos Generales (GPIOs)
- Memoria Flash integrada
- Memoria RAM integrada Buses paralelos para memorias extern

Poder contar con algunos de estos bloques integrados dentro del IC, permite bajar tanto el costo total del sistema al disminuir la cantidad de componentes externos, como el costo final de desarrollo por presentar una solución integrada de antemano.

3.1.3. Lenguajes de programación soportados

La posibilidad de poder utilizar lenguajes de programación de mediano o alto nivel puede llegar a ser un factor decisivo a la hora de elegir una determinada familia de microcontroladores. Las posibilidades dependen de la arquitectura (8/16/32 bits) y de los recursos disponibles en un determinado sistema (RAM, Flash y frecuencia de trabajo).

Es fundamental poder realizar parte del desarrollo en lenguajes como C o C++, dado que se incrementa la portabilidad del código. Incluso, si la velocidad o la escasez de recursos no son un limitante, existen hoy en día diversos lenguajes de script que se utilizan en sistemas embebidos, como Python o LUA. Este último ya es el lenguaje standard para los sistemas de televisión digital. Además, es necesario prestar atención a la existencia previa de librerías que ya hayan sido portadas a la plataforma en estudio, dado que de esta forma se pueden disminuir notablemente los costos de desarrollo.

3.1.4. Herramientas de desarrollo

Cada familia de microcontroladores posee diferentes herramientas de desarrollo, de cuya calidad dependen no solo los costos finales de desarrollo, sino también la calidad final del código generado. Los factores a tener en cuenta son:

- Compiladores: mientras más compiladores haya para una determinada familia de microcontroladores, mayores serán las posibilidades de conseguir herramientas de bajo costo, destacándose en esta categoría el entorno GNU, que se provee de manera gratuita para todas la opciones soportadas.
- Emuladores: Permiten realizar simulaciones del funcionamiento del software sin contar con el hardware específico. De esta manera se facilita el proceso de "debug", para encontrar condiciones de prueba de fallas, que pueden ser generadas artificialmente dentro de estos entornos.
- Programadores/Debuggers: Generalmente en un mismo dispositivo se concentran tanto la capacidad de grabar en la memoria Flash de un microcontrolador el software desarrollado en una PC, como la posibilidad de manejar desde la PC el ciclo de ejecución de dicho software. Por ejemplo, deteniendo la ejecución de un programa en un punto previamente determinado (breakpoint), o cambiando arbitrariamente un valor almacenado en una variable o un registro.

3.2. Requerimientos del proyecto

Las características de nuestro proyecto imponen algunos requerimientos que deberán ser cumplimentados en exceso, dado que nuestra aplicación será la mínima a utilizar, y deberá contar con la suficiente capacidad de expansión para permitir desarrollos futuros. Las necesidades específicas son las siguientes.

3.2.1. Lenguaje C con librerías de punto flotante

Dado que la función principal del microcontrolador será realizar los cálculos para el control del robot RoMAA, resulta indispensable poder expresar las fórmulas matemáticas de una manera clara. Esto no es posible en lenguaje Assembly. Además se necesita utilizar variables de punto flotante, ya sea soportadas por el hardware o emuladas por software, para poder tener una

mayor precisión.

3.2.2. Soporte del entorno de desarrollo GNU

Ya sea utilizando un Entorno Integrado de Desarrollo (IDE) o a través sus partes por separado, se necesita generar código de "compilación cruzada" (cross-compiling). Esto significa que el código es escrito originalmente desde una PC, compilado en una PC, pero para ser ejecutado en otra arquitectura, diferente de x86. Las herramientas provistas por el entorno GNU se componen por:

- GNU Compiler Collection (GCC): compuesta por un compilador, un ensamblador, un linkeador y programas para tratamiento del código (objcopy, objdump, etc.).
- GNU Debugger (GDB): Utiliza la información de debug generada por GCC tanto para realizar simulaciones a través de un emulador interno, como para hacer debug en tiempo real, a través de algún dispositivo de debug específico (como JTAG).
- GNU C Library (GLibC): Implementación de la librería standard de C.

3.2.3. Sistema operativo en tiempo real FreeRTOS

El Sistema Operativo de Tiempo Real FreeRTOS[5] requiere alrededor de 10kB de memoria Flash para un funcionamiento mínimo, y el uso exclusivo de un TIMER para el reloj interno del sistema. Además de esto, se necesita una cantidad variable de memoria RAM, que depende principalmente de la cantidad de tareas a ejecutarse en paralelo, y de la habilitación de las funciones de asignación dinámica de memoria (malloc).

3.2.4. Generación de señales PWM por hardware

La Modulación por Ancho de Pulso (PWM) se utiliza para controlar los dos motores de los que dispone el RoMAA. La capacidad de generar estas señales automáticamente por hardware disminuye notablemente la latencia total del sistema, al no necesitar de interrupciones en el flujo del programa, ni recursos específicos para la medición del ancho de pulso (generalmente un TIMER).

3.2.5. Medición de tiempos por hardware

Se realiza mediante un cálculo que tiene en cuenta la frecuencia de trabajo (valor conocido), y la cuenta de ciclos entre dos instantes de tiempo. La forma básica para llevar a cabo esta cuenta es a través de un TIMER, que es un registro que se incrementa por cada ciclo de ejecución transcurrido. Dependiendo de las capacidades del TIMER del microcontrolador utilizado, hay varias formas de hacer estas mediciones:

- Guardando el valor de este registro en determinados momentos, y luego comparando dichos valores.
- Generando una Interrupción cuando el registro llega a un valor de cuenta determinado y ejecutando dentro de la interrupción alguna funcionalidad necesaria.
- Utilizando una función llamada CAPTURE, que es otro registro en el que se guarda automáticamente el valor que llevaba la cuenta cuando se produce algún evento externo. También es posible generar una Interrupción en ese instante.

3.2.6. UART para protocolo RS-232

Para realizar la comunicación con la PC a bordo del RoMAA se utiliza el protocolo RS-232. Sin embargo, se utiliza luego un chip conversor RS-232 a USB que se comporta como un puerto COM virtual.

3.2.7. Herramientas para debug de bajo costo

Actualmente el standard de la industria para las sesiones de debug es el protocolo JTAG (Joint Test Action Group), en el modo "boundary scan". La principal ventaja de este protocolo es que existen en el mercado muchas opciones, desde dispositivos USB que cuestan alrededor de USD 50 y permiten sesiones básicas de debug, hasta herramientas de "trace" que almacenan todo lo que sucede en el microcontrolador en un período determinado de tiempo para poder ser analizado desde una PC.

3.3. ARM7TDMI-LPC2114

En base a todo esto es que se decidió utilizar el IC LPC2114 de la empresa NXP Semiconductors (anteriormente Philips), que es un microcontrolador ARM de 32bits que posee las siguiente características:

- Microcontrolador de 16/32-bit ARM7TDMI-S en encapsulado LQFP64.
- 16 kB de RAM estática integrada.
- 256 kB de memoria Flash integrada. Interfase de memoria de 128-bit de ancho.
- Frecuencia máxima de trabajo de 60 MHz.
- Bootloader que permite grabación de firmware por RS-232 (In-Circuit Programming).
- Interfase EmbeddedICE permite insertar breakpoints por hardware a través de JTAG.
- 4 canales de ADC de 10 bits. Tiempo de conversión mínimo de 2.44 μs.
- 2 TIMERS de 32-bit con 4 canales de captura y 4 de comparación
- 6 salidas PWM.
- Real-Time Clock (RTC) y watchdog timer.
- Interfases serie: 2 UARTs (16C550), Fast I2C-bus (400 kbit/s) y 2 SPIs.
- Controlador Vectorizado de Interrupciones (VIC) con prioridades y direcciones de vectores configurables.
- 46 pines de E/S de propósito general (GPIO) de 3,3V de salida, y entradas tolerantes a 5V.
- 9 pines para generación de Interrupciones Externas, ya sea por nivel o por flanco.
- Oscilador a cristal interno con rango de operación desde 1 MHz hasta 30 MHz.
- 2 modos de ahorro de energía: Idle y Power-down.
- Wake-up desde el modo Power-down a través de interrupciones externas.

- Desactivación individual de periféricos para optimización del consumo de energía.
- Tensión de operación del núcleo (CPU) de 1.8 V (± 0.15 V).
- Tensión de operación de periféricos de E/S de 3.3 V (± 10 %).
- 300mW de consumo con todos los periféricos activos y funcionando a 60MHz.

De las especificaciones mostradas se deduce que esta opción cumple con creces las necesidades planteadas con anterioridad. Además, cabe destacar que el costo de estos dispositivos no es superior a los ya conocidos microcontroladores de 8 o 16 bits (alrededor de USD 10).

3.3.1. Introducción a los microcontroladores ARM

Los microcontroladores de arquitectura ARM poseen algunas particularidades que es necesario aclarar, a fin de poder realizar un aprovechamiento eficiente de las ventajas que presentan.

Registros

El núcleo ARM7TDMI tiene 37 registros de 32 bits cada uno, aunque no pueden ser todos accedidos al mismo tiempo (dependen del estado del procesador y del modo de operación):

- 31 registros de uso general
- 6 registros de Estado

Set de Registros en Estado ARM

En todos los modos se encuentran disponibles 16 registros generales y 1 o 2 registros de estado (el CPSR y a veces también el SPSR en los modos privilegiados). De R0 a R7 (llamados "registros bajos") son compartidos por todos los modos (y accedidos de forma directa en modo Thumb). Lo mismo sucede con el registro R15 (PC). Los demás registros (registros altos) están organizados en forma de "bancos". Dependiendo del modo de operación del procesador, las direcciones de éstos registros se mapean a distintos registros físicos. De esta forma se preservan sus valores a través de los cambios en los

modos de operación.
El Modo FIQ es el que tiene mayor cantidad de registros "banqueados" (R8 a R12), para tratar de evitar el salvado de registros y acelerar el procesamiento de las interrupciones rápidas.

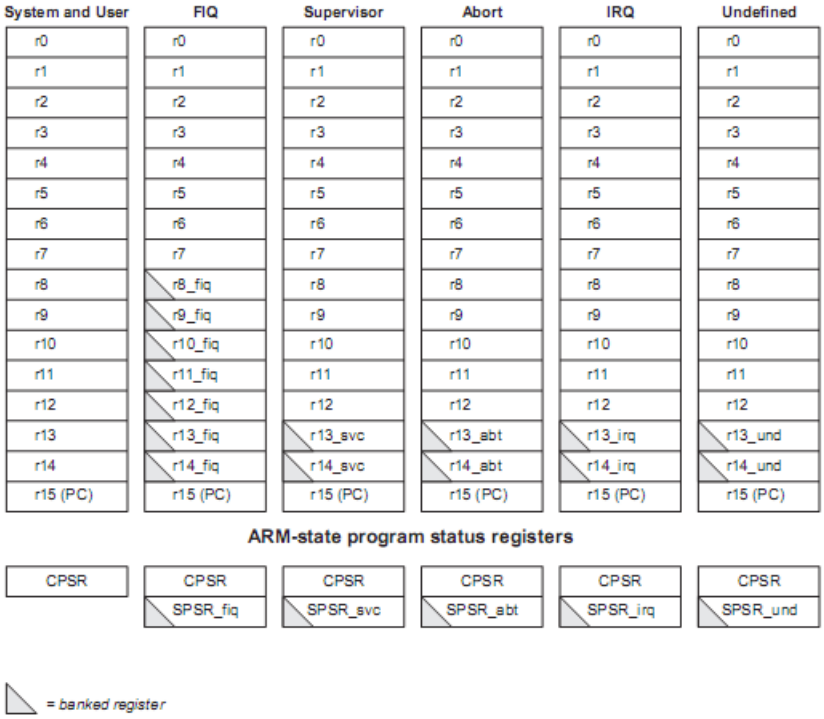


Figura 3.1: Organización de los registros en Estado ARM[3]

Los registros R0 a R13 son de propósito general que pueden contener datos o direcciones de memoria. Los registros R14 y R15 tienen funciones especiales:

- R14 es usado como registro de enlace (link register) en subrutinas porque recibe una copia del PC cuando se ejecuta la instrucción Branch with Link (BL). Esta dirección se utiliza para poder regresar al punto desde donde se produjo el salto.
- R15 contiene el Contador del Programa, PC (Program Counter)

Además, por convención, R13 es usado como Puntero de Pila (Stack Pointer).

Registros de Estado de Programa (PSR)

El Program Status Register sirve para:

- Guardar información de la operación más reciente de la ALU (Unidad Aritmética Lógica).
- Controlar la habilitación de las interrupciones.
- Establecer el Modo de Operación del procesador.

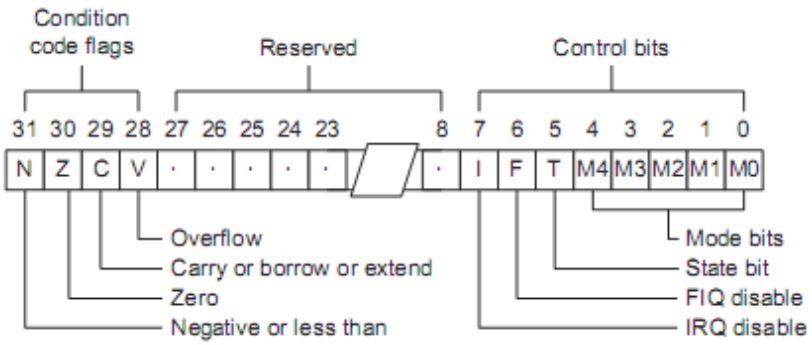


Figura 3.2: Formato del Registro de Estado del Programa (PSR)

El bit 5 en alto indica el Modo THUMB y los bits 6 y 7 deshabilitan las FIQ o IRQ cuando están en alto. Un valor no permitido programado en M[4:0] hace que el procesador entre en un estado irrecuperable, desde el que sólo se puede salir reiniciando el sistema. El Current Program Status Register (CPSR) contiene la información del estado actual del micro. El Saved Program Status Register (SPSR) es una copia del CPSR en el modo anterior de operación. Esto sirve para el poder regresar en forma transparente de los modos de excepción.

Ejecución condicional

Controla si el núcleo ejecuta o no una instrucción. Esto se puede hacer dado que a la mayoría de las instrucciones se les pueden agregar opciones para que se ejecuten dependiendo del estado de las banderas generadas por una instrucción ejecutada con anterioridad. Antes de ejecutar una instrucción, el procesador compara las banderas en CPSR y si cumplen con las condiciones de la instrucción, recién ahí la ejecuta.

Mnemónico	Nombre	Banderas de Condición
EQ	equal	Z
NE	not equal	z
CS HS	carry set/unsigned higher or same	C
CC LO	carry clear/unsigned lower	c
MI	minus/negative	N
PL	plus/positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater than or equal	NV or nv
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (unconditional)	ignored

Figura 3.3: Banderas de condición

La figura 3.4 muestra un algoritmo para encontrar el Máximo Común Divisor de un número. Para ello se muestra el algoritmo en C, y en Assembly con y sin ejecución condicional. Considerando que "a" está guardada en "r1" y "b" en "r2", se muestra cómo a la instrucción de resta (SUB) se le puede agregar un sufijo para volverla condicional:

Como se puede apreciar, la ejecución condicional genera código más compacto, más eficiente, y también más ordenado, reduciendo en gran medida la necesidad hacer "saltos" en el código.

Tubería de instrucciones

El núcleo ARM7TDMI que utilizan los microcontroladores LPC2xxx de NXP tiene un pipeline (tubería) de tres etapas:

- Lectura (fetch): Carga una instrucción de la memoria.
- Decodificación: Identifica la instrucción a ser ejecutada.
- Ejecución: Procesa la instrucción y escribe el resultado en un registro.

Esto significa que en cada ciclo, hay una instrucción que está siendo leída, otra que decodificada y una última ejecutada. La principal ventaja de este modo de funcionamiento es que permite la ejecución de una instrucción por ciclo, lo que incrementa significativamente la capacidad de proceso de la CPU trabajando a la misma frecuencia que en otras arquitecturas.

MCD en C	Assembly	Assembly Condicional

`while(a != b)`
`{`
`  if(a>b)`
`  a -= b;`
`  else`
`  b -= a;`
`}`
	`mcd`
	
	`  CMP  r1, r2`
	`  BEQ  finalizar`
	`  BLT  menorque`
	`  SUB  r1, r1, r2`
	`  B  mcd`
	
	`lessthan`
	`  SUB  r2, r2, r1`
	`  B  mcd`
	
	`finalizar`
	`...`
		`mcd`
		
		`  CMP  r1, r2`
		`  SUBGT  r1, r1, r2`
		`  SUBLT  r2, r2, r1`
		`  BNE  mcd`

Figura 3.4: Máximo Común Divisor

Como desventaja se debe mencionar que cada vez que se ejecuta un salto o una interrupción, tanto la instrucción leída como la decodificada se vuelven inservible. Esto significa que para poder ejecutar la primer instrucción después del salto, se deben esperar 3 ciclos, para que esta sea leída y decodificada.

Figura 3.5: Tubería de Instrucciones para núcleo ARM7TDMI

Como ya se dijo, el Program Counter (PC) se ubica en el registro R15. Es un puntero a la instrucción que está 2 instrucciones delante de la instrucción en ejecución, o sea que apunta a la instrucción que está siendo leída, no a la ejecutada, por lo que el valor del PC utilizado en la ejecución de una instrucción está siempre 2 instrucciones delante de la dirección.
La instrucción en ejecución es PC+8bytes. Una instrucción en la etapa de ejecución se completará aunque haya aparecido una interrupción. Otras instrucciones en el pipeline serán abandonadas y el procesador comenzará a rellenar el pipeline desde la entrada apropiada en la tabla de vectores.

Cuando la instrucción LDR (que se encuentra en 0x8000) llega a la etapa de ejecución, el PC ya se encuentra en la dirección 0x8008, o sea 8 bytes adelante. Esto es importante porque cuando se producen excepciones la instrucción en ejecución será completada, pero la que está siendo decodificada

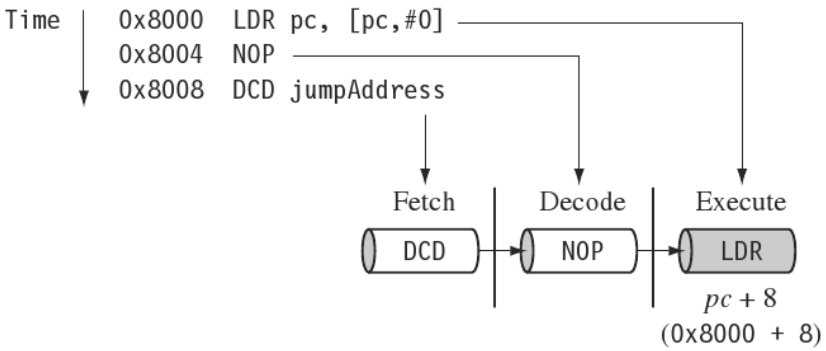


Figura 3.6: Ejemplo de funcionamiento del Pipeline de Instrucciones

no, y por eso puede ser necesario decrementar en 4 unidades el PC, para que vuelva a leerla e insertarla al pipeline al volver del modo de excepción.

Modos de operación

Excepto el Modo Usuario, los demás son llamados modos privilegiados y sirven para atender interrupciones, excepciones, o acceder a recursos protegidos.

Modo	Identificador	M[4:0] PSR
Usuario	USR	1 0000
Interrupción Rápida	FIQ	1 0001
Interrupción	IRQ	1 0010
Supervisor	SVC	1 0011
Abort	ABT	1 0111
Sistema	SYS	1 1111
Indefinido	UND	1 1011

Figura 3.7: Modos de operación

Excepciones

Una excepción es cualquier condición que necesita alterar el funcionamiento normal del programa. Son generadas por fuentes internas y externas que detienen el flujo del programa temporalmente para atender algún evento.
Antes de atender a la excepción el núcleo ARM7TDMI preserva el estado actual del procesador para que el programa original se pueda restablecer cuando la rutina de manejo de la excepción haya terminado.

Los microcontroladores ARM poseen 5 modos de excepción, a los que se puede ingresar desde 7 tipos de excepciones y cada una tiene algunas particularidades. Las interrupciones son un tipo especial de excepción.

Excepción o Entrada	Instrucción de Retorno	Estado Previo ARM R14_x	Estado Previo Thumb R14_x	Explicación
BL	`MOV PC, R14`	PC+4	PC+2	PC es la dirección del BL, SWI o UDEF que tuvo el Prefetch Abort
SWI	`MOVS PC, R14_svc`	PC+4	PC+2	
UDEF	`MOVS PC, R14_und`	PC+4	PC+2	
PABT	`SUBS PC, R14_abt, #4`	PC+4	PC+4	
FIQ	`SUBS PC, R14_fiq, #4`	PC+4	PC+4	PC es la dirección de la instrucción no ejecutada porque la interrupción tomó prioridad
IRQ	`SUBS PC, R14_irq, #4`	PC+4	PC+4	
DABT	`SUBS PC, R14_abt, #8`	PC+8	PC+8	PC es la dirección de la instrucción Load o Store que generó el Data Abort
RESET	`No se aplica`	-	-	El valor guardado en R14_svc es impredecible

Figura 3.8: Excepciones

Excepciones y modos de excepción

Cada excepción hace que el procesador entre en un modo específico. También se puede cambiar entre los distintos modos manualmente, modificando el CPSR y de hecho, esta es la única forma de cambiar a los modos Usuario y Sistema.

Entrada a una excepción

Cuando una excepción causa un cambio de modo, el procesador ARM7TDMI toma las siguientes acciones:

1. Preserva la dirección de la siguiente instrucción en el LR (R14) apropiado.

2. Copia el CPSR en el SPSR apropiado.

Excepción	Modo en Entrada	I en Entrada	F en Entrada	VE *	Dirección Normal	Dirección Alta
Reset	Supervisor	1	1		0x00000000	0xFFFF0000
Undefined Instruction	Undefined	1	Sin cambio		0x00000004	0xFFFF0004
Software Interrupt	Supervisor	1	Sin cambio		0x00000008	0xFFFF0008
Prefetch Abort	Abort	1	Sin cambio		0x0000000C	0xFFFF000C
Data Abort	Abort	1	Sin cambio		0x00000010	0xFFFF0010
Reserved	Reserved	-	-		0x00000014	0xFFFF0014
IRQ	IRQ	1	Sin cambio	0	0x00000018	0xFFFF0018
				1	Definido en Implementación	
FIQ	FIQ	1	1	0	0x0000001C	0xFFFF001C
				1	Definido en Implementación	
VE = Vectored Interrupt Enable (Control al Coprocesador 15)						

Figura 3.9: Excepciones y modos de exepción[3]

3. Impone en los bits de Modo del CPSR un valor que depende de la excepción.

4. Pone en el PC el valor de la instrucción del vector de excepciones relevante.

En pseudocódigo:

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = número de modo de excepción
CPSR[5] = 0                              /* Ejecuta en Estado ARM */
if <exception_mode> == Reset or FIQ then
  CPSR[6] = 1                            /* Desactiva FIQs */
else    CPSR[6] sin cambios
  CPSR[7] = 1                            /* Desactiva IRQ */
  CPSR[9] = CP15_reg1_EEbit              /* Endianness on entry */
  PC = dirección del vector de excepción
```

A los modos de excepción siempre se entra en estado ARM, automáticamente. También se debe salir en modo ARM, pero esto es responsabilidad del programa.

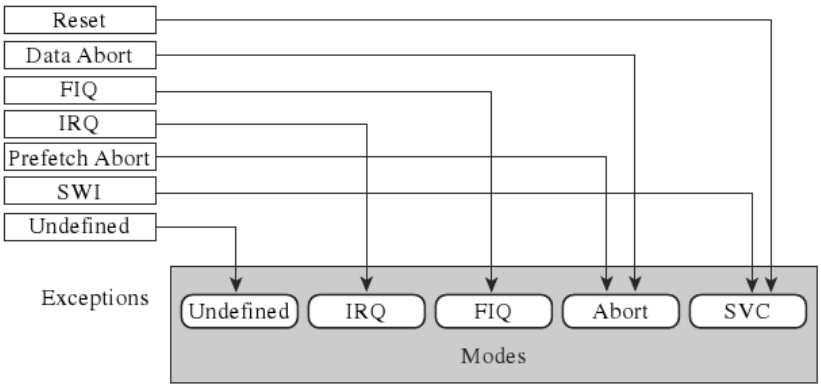


Figura 3.10: Excepciones y sus modos asociados

Salida de una excepción

Cuando la excepción ha concluido, el manejador de la misma debe:

1. Mover el LR, menos un desplazamiento (offset) al PC.
2. Restaurar el valor del SPSR al CPSR.
3. Limpiar las banderas de deshabilitación de interrupción que fueron establecidas a la entrada.

Esto se puede hacer de 2 maneras:

- Utilizando una instrucción de procesamiento de datos con el bit S establecido y el PC como destino.
- Utilizando la instrucción Load Multiple with Restore CPSR instruction, LDM, como se mostrará más adelante.

Fast Interrupt Request (FIQ)

En estado ARM, tiene un banco de 8 registros para eliminar la necesidad de salvar registros y así minimizar la sobrecarga del cambio de contexto. Se genera externamente llevando la señal nFIQ a BAJO, y la entrada pasa al núcleo a través de un sincronizador. El manejador de la FIQ retorna de una interrupción ejecutando:

```
SUBS PC,R14_fiq,#4
```

Las excepciones FIQ pueden deshabilitarse dentro de un modo privilegiado estableciendo la bandera F en el CPSR. Cuando esta bandera está en 0, el procesador chequea el estado del sincronizador de la FIQ al finalizar cada instrucción.

Interrupt Request (IRQ)

Se genera externamente llevando la señal nIRQ a BAJO, y la entrada pasa al núcleo a través de un sincronizador. El manejador de la FIQ retorna de una interrupción ejecutando:

```
SUBS PC,R14_irq,#4
```

Puede deshabilitarse estableciendo en bit I del CPSR en un modo privilegiado.

Abort

Indica que no se pudo completar un acceso a memoria. Se indica a través de la señal externa ABORT. El procesador ARM7TDMI chequea esta señal al final de cada ciclo de acceso a memoria. Hay 2 tipos: Aborto de Lectura de Instrucción y Aborto de Lectura de Datos. Para volver del manejador :

```
SUBS R14_abt, #4       → Recupera el PC y el CPSR y vuelve a ejecutar la
                         instrucción abortada.
SUBS PC,R14_abt, #8    → Idem para datos.
```

Software Interrupt (SWI)

Se utiliza para entrar en Modo Supervisor, generalmente requiriendo algún servicio (cuando hay un sistema operativo, generalmente).

```
R14_svc   = dirección de la instrucción posterior a la que causó la SWI
SPSR_svc  = CPSR
CPSR[4:0] = 0b10011              /* Entrar al Modo Supervisor */
CPSR[5]   = 0                    /* Execute in ARM state */
                                 /* CPSR[6] sin cambios */
CPSR[7]   = 1                    /* Deshabilita IRQ */
CPSR[9]   = CP15_reg1_EEbit      /* Endianness on exception entry */
if (vectores altos configurados) then
    PC    = 0xFFFF0008
else
    PC    = 0x00000008
```

Se regresa con:

```
MOVS PC,R14_svc → Recupera el PC y el CPSR y va hacia la instrucción
                  posterior a la que causó la SWI.
```

Undefined Istruction (UND)

Se utiliza para procesar instrucciones que no son entendidas ni por el núcleo, ni por algún coprocesador. De esta forma se puede expandir el set de instrucciones o emular por software funciones de un coprocesador.

```
R14_und   = dirección de la instrucción posterior a la indefinida
SPSR_und  = CPSR
CPSR[4:0] = 0b11011                    /* Entrar al Modo Undefined Instruction*/
CPSR[5]   = 0                          /* Ejecutar en Estado ARM */
                                       /* CPSR[6] din cambios */
CPSR[7]   = 1                          /* Deshabilita IRQ */
CPSR[9]   = CP15_reg1_EEbit            /* Endianness on exception entry */
if (vectores altos configurados) then
    PC   = 0xFFFF0004
else
    PC   = 0x00000004
```

Se regresa con:

```
MOVS PC, R14_und
```

Esta instrucción restaura el PC desde R14_und y el CPSR desde SPSR_und y retorna a la instrucción posterior a la que generó la interrupción.

Vectores de excepción

Dirección	Excepción	Modo en Entrada	I en Ent	F en Ent	Prioridad
0x00000000	Reset	Supervisor	1	1	1
0x00000004	Undefined Instruction	Undefined	1	Sin cambio	6
0x00000008	Software Interrupt	Supervisor	1	Sin cambio	6
0x0000000C	Prefetch Abort	Abort	1	Sin cambio	5
0x00000010	Data Abort	Abort	1	Sin cambio	2
0x00000014	Reserved	Reserved	-	-	
0x00000018	IRQ	IRQ	1	Sin cambio	4
0x0000001C	FIQ	FIQ	1	1	3

Figura 3.11: Vectores de excepción[3]

Tsyncmax	Tiempo más largo que puede tomarle al pedido atravesar el sincronizador es de 4 ciclos
Tldm	Tiempo que tarda en completarse la instrucción más larga. Ésta es LDM cuando carga todos los registros, incluido el PC. El tiempo es 20 ciclos cuando no hay configurados tiempos de espera en el acceso a memoria.
Texc	El tiempo de entrada a Data Abort (que tiene mayor prioridad que FIQ) es de 3 ciclos.
Tfiq	El tiempo de entrada a la FIQ es de 2 ciclos.

Figura 3.12: Latencias de las interrupciones[3]

Latencias de las interrupciones

Cuando las FIQs están habilitadas, el peor caso se produce por una combinación de causas:

La latencia mínima para la FIQ o una IRQ es el mínimo que puede tomarle pasar por el sincronizador, $T_{syncmin}$ más T_{FIQ}. En total son 5 ciclos de procesador.

Reset

Cuando la señal nRESET se pone en BAJO el procesador para la ejecución de la instrucción actual. Cuando vuelve a ALTO el núcleo ARM7TDMI:

1. Sobrescribe R14_svc y SPSR_svc copiando los valores que en ese momento tuvieran el PC y el CPSR. Dichos valores son indeterminados.

2. 2.Impone M[4:0] a b10011 (Modo Supervisor). Establece los Bits I y F y borra el bit T del CPSR.

3. 3.Impone al PC para leer la próxima instrucción en la dirección 0x00.

4. 4.Vuelve al Estado ARM si es necesario y restablece la ejecución.

Los valores de todos los demás registros quedan indeterminados. En pseudocódigo:

```
R14_svc   = valor IMPREDECIBLE
SPSR_svc = valor IMPREDECIBLE
CPSR[4:0] = 0b10011               /* Entra a Modo Supervisor */
CPSR[5]   = 0                     /* Cambia a Estado ARM */
CPSR[6]   = 1                     /* Deshabilita FIQ */
CPSR[7]   = 1                     /* Deshabilita IRQ */
CPSR[9]   = CP15_reg1_EEbit      /* Endianness on exception entry */
if (vectores altos configurados) then
    PC    = 0xFFFF0000
else
    PC    = 0x00000000
```

3.3.2. Utilización de los recursos del LPC2114

Para poder realizar las tareas de control del robot RoMAA, es necesario utilizar algunos periféricos específicos de la familia de microcontroladores LPC21xx.

Módulo Timer Los 2 módulos TIMER presentes en el LPC2114 son registros de 32-bits que se incrementan con cada ciclo del reloj de periféricos (Peripheral CLK). Cuentan con un registro de escalamiento previo (Prescaler) de 32-bits para aumentar la cuenta máxima posible, y permiten realizar acciones ante distintos eventos, e incluso generar distintas interrupciones para cada uno de ellos:

Registros de Captura (Capture) Toman automáticamente una muestra del registro TIMER en el flanco de una señal de entrada previamente definida.

Se utilizan 2 Registros de Captura del TIMER0 para generar una interrupción en cada transición de las señales que se reciben de los encoders ópticos de cada motor. Tomando la diferencia de dos pulsos consecutivos y teniendo en cuenta la cantidad de pulsos por revolución que generan los encoders y el diámetro de las ruedas, se puede calcular la velocidad instantánea del vehículo.

Registros de Comparación (MATCH) Permiten accionar automáticamente una señal de salida en intervalos de tiempo definidos.

Se utiliza 1 registro MATCH del TIMER1 para incrementar la variable xTickCount. Esta variable cumple una función fundamental del Sistema Operativo FreeRTOS, dado que se la utiliza para todos los manejos de los tiempos, el cambio de una tarea a otra de acuerdo a sus prioridades, y la cuenta del tiempo que una tarea debe estar "dormida".

PWM Es muy similar al módulo TIMER, y de hecho está basado en él, con la diferencia de que se puede configurar para que automáticamente genere pulsos cuadrados, pudiendo prescindir de la utilización de interrupciones.

Para enviar las señales de control a las Llaves H de los motores del robot RoMAA se utilizan 2 señales de Modulación de Ancho de Pulso Digital, que permiten variar la velocidad instantánea del vehículo.

GPIO Los pines de Entrada y Salida de Propósitos Generales producen señales digitales de 3.3V cuando están configurados como salida, y toleran

tensiones de hasta 5V configurados como entrada. Algunos pines tienen la capacidad de generar interrupciones tanto por nivel como por flanco, cuando están configurados como entrada.

La última versión del hardware de control utiliza varios pines para LEDs indicadores y para ayudar al módulo CAPTURE a detectar las direcciones de movimiento de las ruedas.

UART El microcontrolador LPC2124 cuenta con 2 UARTs, con capacidad para transmitir hasta 230400bps. Cada UART Colas de 16 bytes tanto para Transmisión (Tx) como para Recepción (Rx).

Se utiliza para la comunicación con una PC, una UART configurada para funcionar a 115200bps a la que se le agregó el desarrollo de dos buffers circulares por software para tener una capa de abstracción entre los datos de la aplicación, y la forma en que la información se codifica para ser enviada a la PC.

Embedded ICE Esta característica presente en el núcleo ARM7TDMI-S del microcontrolador LPC2114 permite utilizar el protocolo de comunicación JTAG para poder llevar a cabo sesiones de "debug", con la capacidad de insertar hasta 2 breakpoints por hardware. La principal ventaja es que todo esto se puede hacer con el chip soldado al circuito, pudiendo observar realmente cómo se comporta el código en la aplicación real.

Capítulo 4

FreeRTOS

Actualmente, los Sistemas Operativos en Tiempo Real (RTOS) son cada vez mas utilizados en aplicaciones embebidas, en particular en robótica. La necesidad de su uso se debe en gran medida, a la portabilidad que genera en el código, ya que al estar dividido en tareas sincronizadas por el RTOS, solo se deben agregar o modificar dichas tareas para realizar una acción determinada, independientemente del resto.

Con respecto a la portabilidad o independencia del hardware, la modularización permite que modificando solo una parte (módulo) del software, el mismo programa funcione en otra plataforma o dispositivo. Como resultado mejora la coordinación del mantenimiento del código fuente entre diferentes desarrolladores y también a través del tiempo. Mas allá de estas ventajas, el uso de RTOS posibilita el desarrollo de sistemas en tiempo real, en los cuales existen grandes exigencias en cuanto a la exactitud de la duración de determinados eventos y su periodicidad. El cumplimiento de estas exigencias favorece a la exactitud en trabajos de investigación de tiempo real, como son la utilización de sensores (sonares, láser, infrarrojo, etc.), cámaras de visión, navegación autónoma, creación de mapas, etc.

4.1. Sistema Operativo en Tiempo Real (RTOS) y el FreeRTOS

4.1.1. Sistemas en tiempo real

Los sistemas en tiempo real poseen la característica de que grandes consecuencias pueden resultar si las propiedades lógicas y temporales del sistema

no concuerdan con exactitud. Existen dos tipos de sistemas en tiempo real: los denominados SOFT, donde las tareas son llevadas a cabo por el sistema tan rápido como sea posible, pero no se les exige que finalicen en un tiempo específico; y los denominados HARD, donde las tareas son ejecutadas correctamente y en un tiempo específico. La mayoría de los sistemas en tiempo real son una combinación de ambos tipos y también de tipo embebida, es decir, donde la computadora esta construido dentro del sistema y no es visto por el usuario como una computadora. El control de la plataforma RoMAA es un ejemplo de sistema embebido de tiempo real de tipo HARD/SOFT.

4.1.2. Sistemas Foreground – Background

Sistemas pequeños de baja complejidad son generalmente diseñados de la forma "foreground/background" (la traducción literal sería "primer plano/-fondo") o también llamados super-loops (ver figura 4.1).

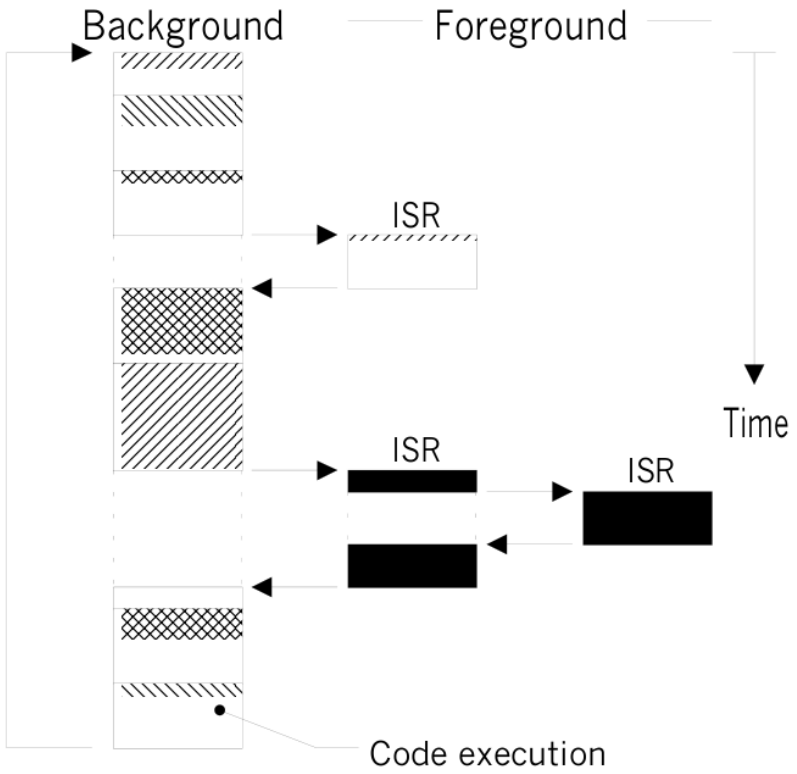


Figura 4.1: Sistemas en tiempo real foreground-background [4]

Una aplicación consiste de un loop infinito que llama a módulos (serían

funciones) para llevar a cabo las operaciones deseadas (background). Las Rutinas de Servicio a Interrupciones (ISRs, Interrupt Service Routines) manejan eventos asíncronos (foreground). Foreground es también llamado el nivel de interrupciones mientras que background es llamado el nivel de tarea.

Operaciones críticas deben ser llevadas a cabo por las ISRs para asegurar que sean tratadas oportunamente. Es por esto que las ISRs tienen una tendencia a tomarse mas tiempo del que deberían. Así también, información para un módulo de background que se haga disponible por una ISR, no es procesada hasta que se le permita ejecutarse a la rutina del background. A esto se le llama la respuesta a nivel de tarea. El peor caso de tiempo de respuesta del nivel de tarea depende de cuanto tiempo tarde el loop del background en ejecutarse. Como el tiempo en ejecutarse de un código típico no es constante, el tiempo para sucesivas pasadas por un trozo del loop no es determinístico. Además, si se produce algún cambio en el código, el tiempo del loop se ve afectado.

La mayoría de las aplicaciones de gran volumen basadas en microcontroladores, son diseñadas como sistemas foreground/background. Desde el punto de vista del consumo de energía, se podría decir también que sería mejor para al procesador y realizar todo el procesamiento en ISRs.

4.1.3. Sistema Operativo en Tiempo Real (RTOS)

Un Sistema Operativo en Tiempo Real (RTOS) es un sistema operativo optimizado para el uso en aplicaciones de tiempo real/embebidas. Su objetivo primario es asegurar una respuesta temporal y determinística a eventos, tanto externos como internos. Un plazo de tiempo real puede ser tan pequeño que el sistema aparenta reaccionar instantáneamente.

A continuación se listan una serie de razones por la cual usar un RTOS en aplicaciones embebidas, teniendo en cuenta que éstas no son absolutas, sino opiniones de diferentes autores:

- Resumen de información de tiempo.
  El scheduler de tiempo real (descrito mas adelante en el informe), es efectivamente una pieza de código que permite especificar las características temporales (timming) de la aplicación, produciendo códigos mas pequeños, simples y por lo tantos menos complejos de entender.

- Mantenibilidad/Extensibilidad.
  El no tener la información temporal dentro de código, permite una

mayor mantenibilidad y extensibilidad ya que habrá menos interdependencias entre los módulos del software. Éste, también será menos susceptible a los cambios en el hardware.

- Modularidad.
  Organizando la aplicación como un set de tareas independientes, permite una modularidad mas efectiva. Las tareas deberían ser unidades débilmente acopladas y funcionalmente cohesivas para que entre ellas se ejecuten de una manera secuencial.

- Interfaces mas limpias.
  Interfaces de comunicación entre tareas bien definidas, facilita el diseño y desarrollo en equipo.

- Testeo mas simple (en algunos casos).
  La interfase de las tareas pueden ser testeadas sin necesidad del agregado de instrumentación externa, que podría cambiar el comportamiento del módulo bajo testeo.

- Reuso del código.
  La mayor modularidad y la menor interdependencia entre módulos, facilita el reuso del código en diferentes proyectos. Las mismas tareas facilitan el reuso del código en el proyecto.

- Eficiencia mejorada.
  Usar un RTOS permite a una tarea bloquearse con eventos, ya sean temporales o externos al sistema. Esto significa que no hay tiempo desperdiciado haciendo polling o chequeando timers cuando no hay eventos que requieran procesamiento, pudiendo resultar en enormes ahorros de la utilización del procesador.

- Tiempo ocioso (Idle time).
  Generalmente es fácil medir la carga del procesador utilizando un RTOS. Siempre que la tarea "Idle" se este ejecutando, significa que el procesador no tiene otra cosa que hacer. Esto provee también un método simple y automático de llevar al procesador a un modo de bajo consumo.

- Manejo flexible de las interrupciones.
  Delegar el procesamiento de una interrupción al nivel de tarea, permite al manejador de la interrupción ser muy pequeño, y que las interrupciones se mantengan habilitadas mientras el procesamiento a nivel de

tarea se completa. Así también, el procesamiento a nivel de tarea permite una priorización mas flexible, mas de lo que permitiría usar solamente las prioridades asignadas a los periféricos por el mismo hardware (dependiendo de la arquitectura usada).

- Requerimientos de procesamiento mezclados.
  Simples patrones de diseño pueden ser usados para alcanzar una mezcla de procesamientos periódicos, continuos y manejados por eventos en la misma aplicación. A su vez, tanto requerimientos de tiempo real hard y soft pueden ser usados en conjunto mediante el uso de interrupciones y priorización de tareas.

- Control mas simple de los periféricos.
  Tareas para manejo de periféricos facilitan la serialización del acceso a los periféricos y provee un buen mecanismo de exclusión mutua.

4.1.4. FreeRTOS

El sistema operativo de tiempo real elegido es el FreeRTOS [5] por poseer las siguientes características:

- Posibilidad de realizar multitarea preemptiva o cooperativa
- Ejecución de tareas en orden de prioridad
- Creación y destrucción dinámica de tareas
- Comunicación entre tareas a través de "colas" coordinadas por el OS
- Sincronización de tareas a través de "semáforos"
- Protección de recursos compartidos utilizando "exclusiones mutuas" (Mutex)
- Gran cantidad de arquitecturas soportadas (23 al momento de esta redacción)
- Escrito casi en su totalidad en C, con secciones de configuración en assembly
- Libre disponibilidad del código fuente

Por definición oficial, se puede tomar la del sitio web de FreeRTOS :

> "FreeRTOS es un mini Kernel de tiempo real, portable, libre de regalías y de fuente abierta, libre para descargar y usar en aplicaciones comerciales sin requerimientos que expongan su código propietario.". "Descargado mas de 77.500 veces durante el 2008, es la plataforma cruzada de facto estándard para los microcontroladores embebidos".

Ha sido portado a 25 arquitecturas de hardware desde pequeños microcontroladores de 8 bit a procesadores completos de 32 bit, incluyendo ARM7, ARM9, Cortex M3, MSP430, AVR, PIC and 8051. FreeRTOS es portable. Su versatilidad en el port se debe a varios factores. Primero, el código base del FreeRTOS es pequeño. Esta compuesto por un total de tres archivos para el núcleo y un archivo adicional para el port necesitado por el mismo kernel. Segundo, FreeRTOS esta en su mayoría escrito en C estándar. Solo unas pocas líneas de código ensamblador son necesarias para adaptarlo a una plataforma en particular. Finalmente, FreeRTOS posee una extensa documentación en el código fuente como así también en el sitio web oficial [5] con benchmark a nivel de aplicación. FreeRTOS es de código abierto. Esta licenciado bajo GPL modificado y puede ser usado en aplicaciones comerciales bajo esta licencia. El código del FreeRTOS esta disponible de manera gratuita en su sitio web, lo que hace al kernel fácil de estudiar y entender.

4.1.5. Sistemas Multitareas (Multitasking)

Multitasking es el proceso de coordinar e intercambiar el CPU (Unidad de Procesador Central) entre varias tareas. Cada tarea es como un programa que se ejecuta. Un solo CPU cambia su atención entre varias tareas secuenciales. Multitasking es como los sistemas foreground/background pero con múltiples backgrounds. Un procesador convencional puede ejecutar solo una tarea a la vez, pero mediante una conmutación rápida entre tareas, el sistema operativo multitareas puede hacer parecer como si cada tarea se ejecutara concurrentemente. El diagrama 4.2 siguiente da una idea mas clara de lo anteriormente explicado:

El uso de sistemas operativos multitarea puede simplificar de gran manera el diseño de lo que sería de otra manera un aplicación de software compleja:

- Maximiza la utilización del CPU y permite una construcción modular de las aplicaciones.
- Permite a la aplicación del programador manejar la complejidad inherente a las aplicaciones en tiempo real.

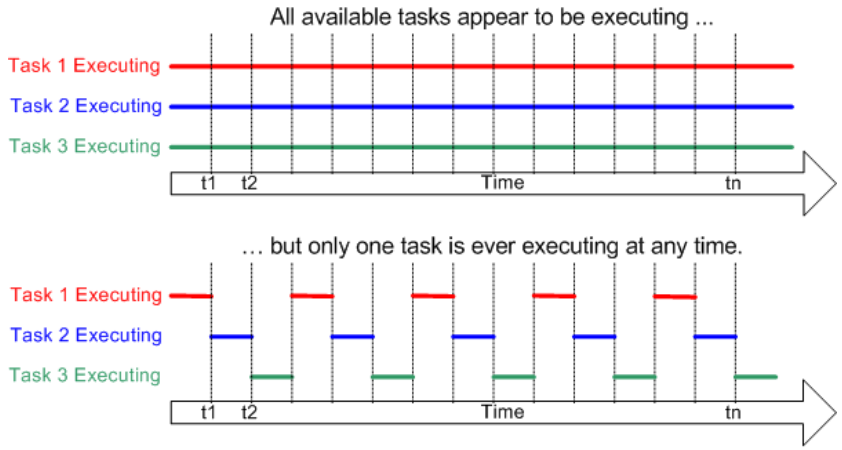


Figura 4.2: Sistemas multitareas [5]

- Las multitareas y la comunicación entre tareas ("inter-task communication") permiten al sistema operativo particionar una aplicación compleja en un set de tareas mas pequeñas y manipulables.
- El particionado puede resultar en un testeo mas simple del software, división del trabajo para el desarrollo en equipo, reusabilidad del código.
- Los programas de aplicación son usualmente mas fáciles de diseñar y mantener cuando se usa multitasking.
- Sincronizaciones complejas y detalles de secuencias pueden ser removidas del código de la aplicación y transformarse en responsabilidad del sistema operativo.

4.1.6. Kernel de tiempo real

El corazón de un RTOS y de cualquier Sistema Operativo (OS), es el kernel. El kernel es el núcleo central de un OS, y es el encargado de llevar a cabo todos los trabajos del OS, como el booteo, la administración de las tareas y las librerías estándares de funciones. Es un sistema embebido, generalmente el kernel bootea el sistema, inicializa los puertos y todos los datos globales. Luego, da inicio al scheduler e instancia cualquier timer de hardware que necesite ser iniciado. Después de todo esto, el kernel básicamente de descarga de la memoria (a excepción de las funciones de librería, si hubiese) y el scheduler comenzará a ejecutar las tareas propiamente dichas. Es decir que provee los servicios más básicos para el software de la aplicación que

corre en el procesador. El kernel de un RTOS provee una capa de abstracción ("abstraction layer") que esconde del software de la aplicación los detalles de hardware del procesador sobre el cual correrá la aplicación.

Como una definición mas específica para RTOS en general, podemos decir que el kernel es la parte de un sistema multitarea responsable del manejo de las tareas (es decir, el manejo del tiempo de CPU) y la comunicación entre tareas. El servicio fundamental provisto por el kernel es el cambio de contexto ("context switching"). Se debe tener en cuenta, que el kernel agrega sobrecarga al sistema, porque requiere espacio en la memoria ROM para su código, espacio adicional en la memoria RAM para su estructura de datos y su propio espacio de stack. Como valor típico, se puede decir que el kernel en general consume entre un 2 y 5 % de tiempo de CPU.

4.1.7. Scheduler

El scheduler es la parte del kernel responsable de decidir que tarea debe ser ejecutado en un determinado momento. El kernel puede suspender y luego reanudar la tarea varias veces durante el tiempo de vida de la tarea. La política del scheduler es el algoritmo usado por el mismo para decidir que tarea se ejecuta en cualquier punto determinado del tiempo.

En adición a ser suspendida involuntariamente por el kernel del RTOS, una tarea puede elegir por si misma ser suspendida. Lo puede hacer si desea demorarse (sleep) por un determinado lapso de tiempo o esperar (block) por recurso a que esté disponible (por ej.: un puerto serie) o a que ocurra un evento (por ej.: que una llave sea presionada). Una tarea en espera o durmiendo no puede ejecutarse, y no ocupará tiempo de procesamiento. La figura 4.3 aclara un poco lo explicado anteriormente:

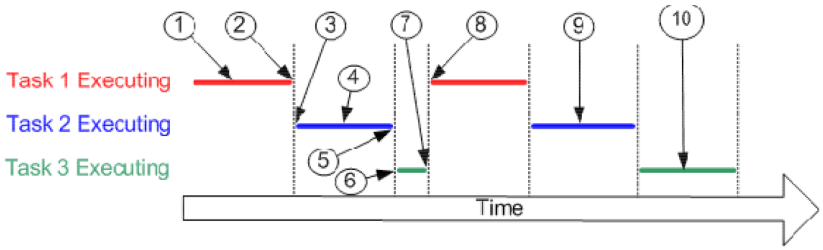


Figura 4.3: Funcionamiento del Scheduler [5]

En general, un scheduler puede funcionar de manera preemptiva o no

preemptiva, dependiendo del tipo de kernel que use, preemptivo o no preemptivo.

Scheduler en kernels no preemptivos En kernels no preemptivos, se requiere que cada tarea haga algo para explícitamente devolver el control del CPU. Para mantener la "ilusión" de la concurrencia, este proceso debe ser realizado frecuentemente. Al funcionamiento de schedulers no preemptivos, suele llamarse multitareas cooperativas. Los eventos asíncronos siguen siendo manejados por los ISRs. Por mas que un ISR lleve a una tarea de mayor prioridad al estado listo para correr, siempre retornará a la tarea interrumpida. La nueva tarea de mayor prioridad solo tomará control del CPU cuando la tarea en ejecución devuelva el control del mismo (ver figura 4.5).

Una de las ventajas de este tipo de kernel, es que la latencia de interrupción es típicamente baja, y funciones no reentrantes pueden ser utilizadas sin riesgo. El tiempo de respuesta a nivel de tarea puede ser mucho mas bajo que con el sistema foreground/background porque la respuesta a nivel de tarea es ahora dada por el tiempo de la tarea mas larga. Otra ventaja, es la menor necesidad de sincronización con otras tareas, porque no hay riesgo de que una tarea sea interrumpida por otra de mayor prioridad. La mayor desventaja es el tiempo de respuesta, ya que al igual que en los sistemas foreground/background, al no ser determinístico el tiempo de respuesta a nivel de tarea, no se sabe exactamente cuando la tarea de mayor prioridad tomará el control del CPU.

Scheduler en kernels preemptivos Kernel preemptivos son usados cuando la certeza de la respuesta temporal del sistema es importante. La tarea de mayor prioridad lista para correr tiene siempre el control del CPU. Si una tarea de mayor prioridad se vuelve lista para correr, se interrumpe inmediatamente la tarea en ejecución y se da el control a la tarea de mayor prioridad. Si una ISR lleva a una tarea de mayor prioridad al estado lista para correr, se suspende a la tare interrumpida y se restaura la tarea de mayor prioridad. La mayoría de los RTOS comerciales como lo es FreeRTOS, usan este tipo de kernel (ver figura 4.4).

Con un kernel preemptivo, la ejecución de la tarea de mayor prioridad es determinístico; se puede saber cuando la tarea de mayor prioridad tomará el control del CPU. Por lo tanto, el tiempo de respuesta a nivel de tarea es minimizado en este tipo de kernel. El código de aplicación en este tipo de kernel no debería usar funciones no reentrantes.

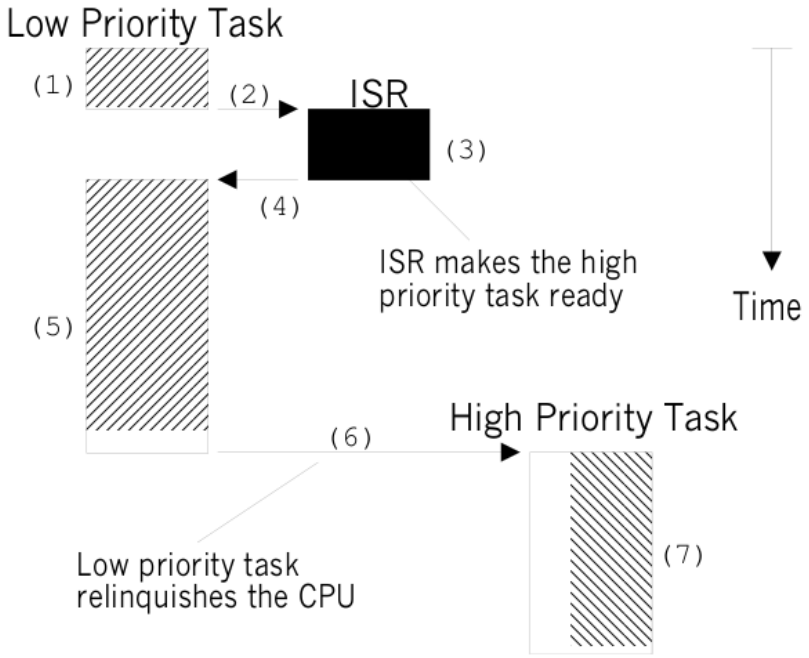


Figura 4.4: Funcionamiento del Scheduler en kernels preemptivos [4]

Scheduler de FreeRTOS FreeRTOS presenta un scheduler de tipo Round Robin basado en prioridades. A cada tarea se le asigna una prioridad. Las tareas con una misma prioridad, comparten el tiempo de CPU en forma Round Robin. Esta política de funcionamiento del scheduler, permite que si dos o mas tareas tienen asignada la misma prioridad, el kernel permitirá a una de las tareas que se ejecute por una predeterminada cantidad de tiempo, llamado "quantum", y luego seleccionará otra tarea para ser ejecutada. A esto se le llama división de tiempo ("time slicing"). El kernel le dará el control a la siguiente tarea si la actual tarea ya no tiene trabajo para hacer en su porción de tiempo (time slice) o si la actual tarea se ha completado antes de la finalización de su porción de tiempo. Este tipo de scheduling no es soportado por todo los RTOS comerciales, como lo es por ejemplo el μC/OS-II.

En el caso del FreeRTOS, el scheduler puede ser configurado además como preemptivo (preventivo) o colaborativo. El comportamiento en tiempo real del sistema requiere una planificación (scheduling) preemptiva. Para sistemas simples, el planeamiento colaborativo puede ser usado.

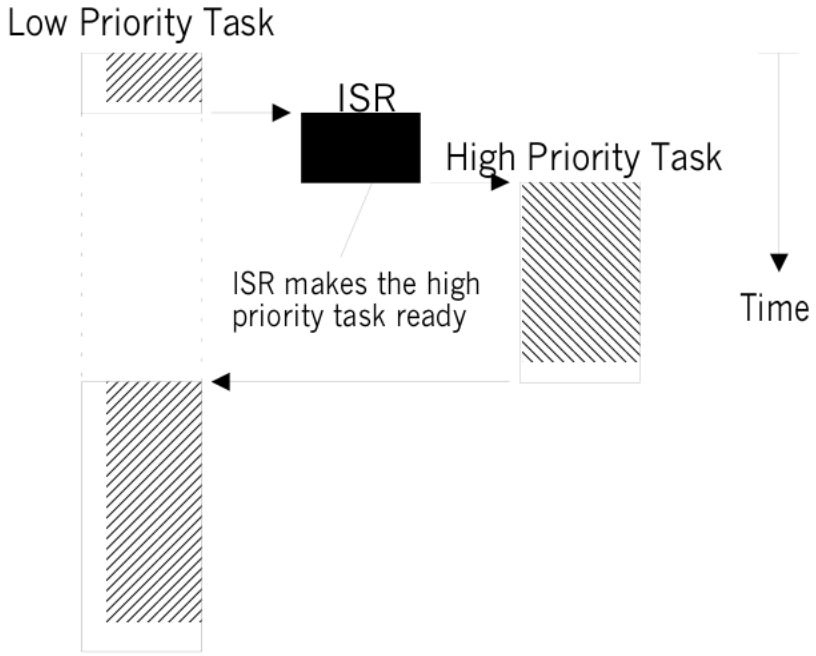


Figura 4.5: Funcionamiento del Scheduler en kernels no preemptivos [4]

4.1.8. Tareas (Tasks)

Una tarea ("task", llamada también "thread"), es un simple programa que funciona como si el CPU funcionará solo para ella. El proceso de diseño de una aplicación de tiempo real, implica dividir el trabajo a realizar en tareas que son responsables de una porción del problema. A cada tarea se le asigna una prioridad, su propio set de registros del CPU y su propia área de stack.

Cada tarea se ejecuta dentro de su propio contexto, independientemente de las demás tareas y del scheduler en si mismo. Solo una tarea de la aplicación puede estar siendo ejecutada en un determinado instante de tiempo, y es el scheduler de tiempo real el encargado de decidir cual tarea debe ser. Por lo tanto, el scheduler debe repetidamente detener y reanudar cada tarea (entrar y salir de la tarea) mientras la aplicación se ejecuta.

Como la tarea no tiene conocimiento de la actividad del scheduler, es responsabilidad del mismo el asegurar que el contexto del procesador (valores de registros, contenido del stack, etc.) cuando una tarea es restaurada, sea exactamente el mismo que cuando la tarea fue detenida. Para lograr esto, a cada tarea se le provee su propio stack. Cuando una tarea es detenida, el contexto de ejecución se guarda en el stack de esa tarea para que pueda ser

exactamente restaurado cuando se retome la misma.

En FreeRTOS, las tareas poseen las siguientes características:

- Simples;
- Ninguna restricción en el uso;
- Soportan "full preemption";
- Cada tarea mantiene su propio stack, resultando en un mayor uso de la memoria RAM;
- Si se usa la característica preemptiva, se debe considerar cuidadosamente las funciones reentrantes.

Una tarea consiste típicamente de un for infinito que puede estar en cualquiera de los cuatro estados posibles. Los estados en los que pueden existir las tareas son, en FreeRTOS y en general en todos los RTOS, los siguientes:

- Running
  Cuando una tarea se esta ejecutando, se dice que esta en el estado Running. Significa que es la tarea que actualmente esta usando el procesador.
- Ready
  Las tareas en estado Ready son aquellas que están disponibles para ser ejecutadas (no están bloqueadas ni suspendidas), pero no se están ejecutando porque una tarea diferente de igual o mayor prioridad se está ejecutando (estado Running).
- Blocked
  Se dice que una tarea esta en el está en este estado, cuando espera por un evento temporal o externo. Las tareas en estado Blocked, poseen un período de "time out", después del cual serán desbloqueadas. Cualquier tarea que este bloqueada, no esta disponible para ser utilizada por el scheduler.
- Suspended
  Las tareas en este estado tampoco están disponibles para ser utilizadas por el scheduler. Solo entrarán en este estado, las tareas en las que se use el comando explícito para tal fin. Un período de "time out" no puede ser especificado.

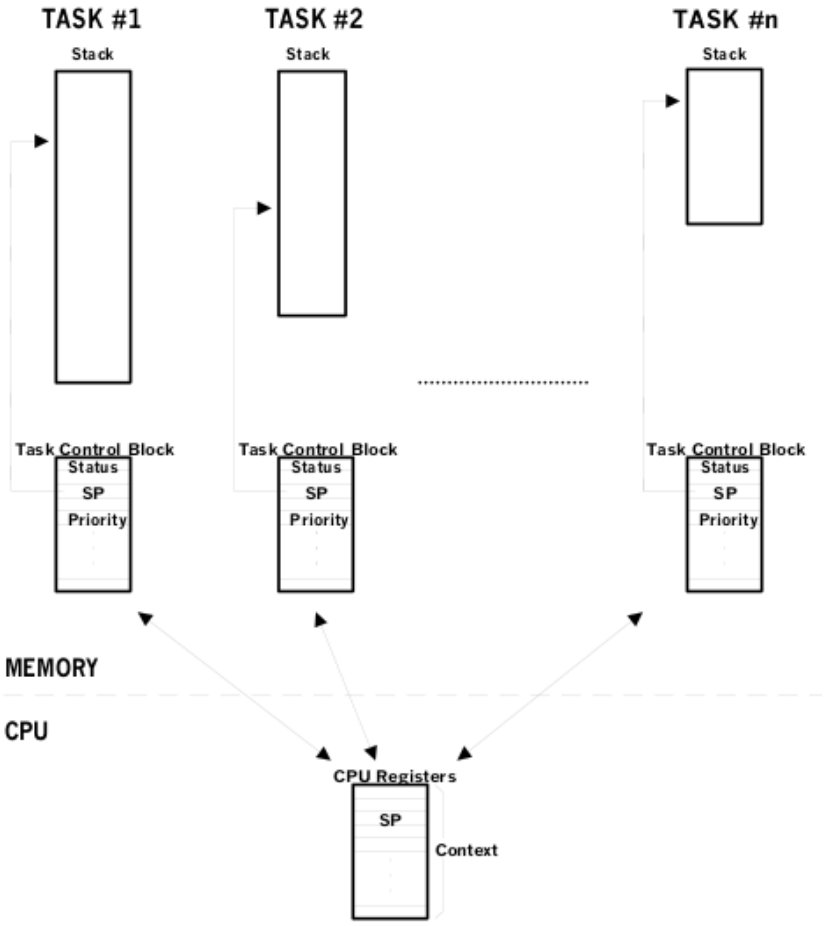


Figura 4.6: Funcionamiento de las tareas [4]

4.1.9. Context switching

Mientras una tarea se ejecuta, utiliza los registros, acceso a RAM y ROM del procesador/microcontrolador como cualquier programa. Estos recursos juntos (los registros del procesador, stack, etc.), componen el contexto de ejecución de la tarea. Una tarea es un pedazo secuencial de código – no sabe cuando será suspendida o reanudada por el kernel ni tampoco sabe cuando ha sucedido.

Para prevenir errores, es esencial que después de ser reanudada, la tarea disponga del mismo contexto que tenía antes de ser suspendida. El kernel del sistema operativo es el encargado de asegurarse que esto suceda, guardando el contexto de la tarea que esta siendo suspendida. Cuando una tarea es reanudada, su contexto salvado es restablecido por el núcleo del sistema

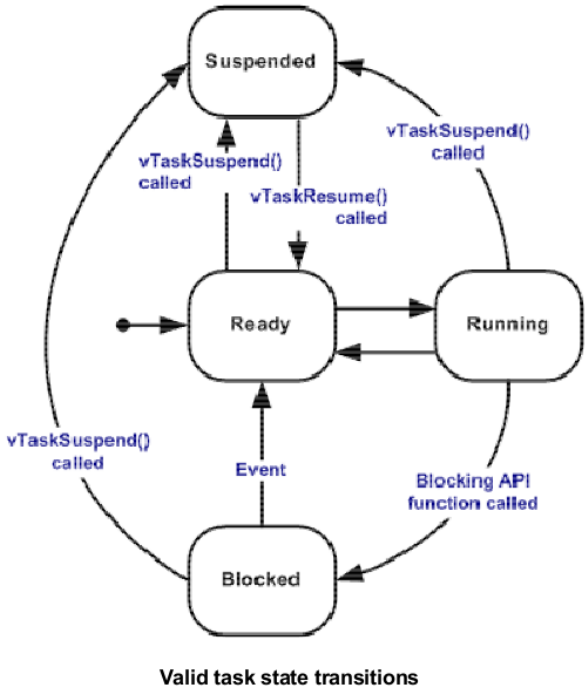


Figura 4.7: Estados de las tareas [4]

operativo antes de su ejecución. Este proceso de guardar el contexto de una tarea que está siendo suspendida y restablecer el contexto de una tarea que esta siendo reanudada, se lo denomina cambio de contexto.

El cambio de contexto, genera una sobrecarga la aplicación. Mientras mas registros tenga un CPU, mayor será la sobrecarga . El tiempo requerido para realizar un cambio de contexto está determinado por el número de registros que deben ser guardados y restaurados por el CPU. Sin embargo, la performance un kernel en tiempo real no debe ser juzgado por la cantidad de cambios de contextos que es capaz de realizar por segundo.

4.1.10. Prioridades de las tareas

Asignar prioridades a las tareas no es algo trivial, debido a la compleja naturaleza de los sistemas en tiempo real. En la mayoría de los sistemas, no todas las tareas son consideradas críticas. A las tareas no críticas deberían ser asignadas, obviamente, la menor prioridad. La mayoría de los sistemas en tiempo real tienen una combinación de requerimientos HARD y SOFT, como se mencionó anteriormente.

Una técnica interesante llamada Rate Monotonic Scheduling (RMS), ha sido establecida para asignar prioridades a las tareas basado en la periodicidad de ejecución de las tareas. Simplemente, las tareas con la mayor taza de ejecución son a las que se les asigna la mayor prioridad. RMS hace las siguientes suposiciones:

1. Todas las tareas son periódicas (ocurren a intervalos regulares).

2. Las tareas no se sincronizan una con otra, no comparten recursos ni intercambian datos.

3. El CPU debe ejecutar siempre la tarea de mayor prioridad que este lista para correr. Es decir, debe usarse planeamiento preemptivo ("preemtive scheduling").

Dado un set de tareas con prioridades asignadas mediante RMS, el teorema básico de RMS establece que todas las tareas con dead-lines de tiempo real tipo HARD, se producirán solo si la siguiente inecuación es verificada:

$$\sum_i \frac{E_i}{T_i} \leq n \times (2^{1/n} - 1) \tag{4.1}$$

donde, E_i corresponde al máximo tiempo de ejecución de la tarea i y T_i corresponde al período de ejecución de la tarea i. En otras palabras, E_i/T_i corresponde a la fracción del tiempo de CPU requerido para ejecutar la tarea i. La tabla muestra el valor para el tamaño $n \times (2^{1/n} - 1)$ basado en el número de tareas. El límite superior para un número infinito de tareas está dado por $ln(2)$ o 0,693. Esto significa para que todos los dead-lines de tiempo real tipo HARD basados en RMS se cumplan, la utilización del CPU de todas las tareas de tiempo crítico debe ser menor al 70 %. Este dato indica también, que todavía se pueden tener tareas de tiempo no críticos en un sistema y así usar el 100 % del tiempo del CPU. Sin embargo, nunca es deseada esta utilización del tiempo del CPU, porque no permitiría realizar cambios en el código o agregado de nuevas características. Como regla, generalmente se diseñan los sistemas en tiempo real para que utilice menos del 60 o 70 % de tiempo del CPU.

4.1.11. Comunicación entre tareas

FreeRTOS provee varios métodos para la comunicación entre tareas, incluyendo las colas de mensajes los semáforos binarios. El mecanismo de colas de FreeRTOS puede ser usado en la comunicación entre dos tareas y en la

Number of Tasks	$n(2^{1/n}-1)$
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
.	.
.	.
.	.
Infinity	0.693

Figura 4.8: Límite de número de tareas para scheduling tipo RMS [4]

comunicación entre una tarea y un ISR. Una cola es una estructura capaz de almacenar y recuperar datos.

Los semáforos en FreeRTOS están en realidad implementados como un caso especial de colas. Un semáforo es una cola de un solo elemento con tamaño cero. Los semáforos son utilizados para la sincronización entre tareas y para la exclusión mutua. La primer tarea que ejecute una sección de código mutuo, toma el semáforo. Si alguna otra tarea intenta ejecutar este código, deberá esperar hasta que la otra tarea lo libere.

Una tarea que intenta recibir un byte de una cola vacía, mandar un byte a una cola llena o tomar un semáforo que ya haya sido tomado, será bloqueada por el propio kernel. La tarea decide el máximo de tiempo que le permite al kernel bloquearla en la comunicación inter-tareas. En el caso que dicho tiempo haya expirado, el kernel reanudará la tarea y devolverá un error, el cual deberá ser procesado por la propia tarea.

4.1.12. Interrupciones

Una interrupción es un mecanismo de hardware usado para informar al CPU que un evento asíncrono ha ocurrido. Cuando una interrupción es reconocida, el CPU guarda parte o todo su contexto y salta a la subrutina especial llamada Rutina de Servicio a la Interrupción (ISR). La ISR procesa el evento y luego retorna a:

- El background, en un sistema de tipo foreground/background.
- La tarea interrumpida, en un kernel de tipo no preemptivo.
- La tarea de mayor prioridad que este en estado lista para correr, en un kernel preemptivo.

Las interrupciones le permite al microprocesador procesar los eventos en el momento que ocurren, sin necesidad de estar haciendo un polling continuo del posible evento. Los procesadores en general permiten la anidación de interrupciones ("nested interrupts"), lo que significa que mientras el procesador esta sirviendo a una interrupción, reconocerá y servirá otras interrupciones (siempre que sean de mayor prioridad).

4.1.13. Clock Tick

Un tick de reloj ("clock tick") es una interrupción especial que ocurre periódicamente. A esta interrupción se la puede ver como al latido de corazón del sistema. El tiempo entre interrupciones es específico de la aplicación y generalmente se encuentra entre 10 y 200ms. La interrupción del clock tick le permite al kernel demorar a las tareas por cantidades enteras de número de ticks de reloj y proveer timeout cuando una tarea está esperando que ocurra algún evento. Mientras más rápido sea la frecuencia del tick, mayor será la sobrecarga impuesta al sistema. Todos los kernels permiten a las tareas que sean demoradas por un número determinado de ticks de reloj. La resolución de la demora de las tareas es de 1 tick de reloj, sin embargo, no significa que la precisión sea de un tick de reloj.

El kernel de tiempo real del FreeRTOS mide el tiempo usando una variable de cuenta de ticks. Un "timer interrupt" (la interrupción de tick del FreeRTOS) incrementa la cuenta de ticks con una estricta precisión temporal, permitiendo al kernel de tiempo real medir tiempo con una resolución igual a la frecuencia de interrupción de tiempo elegida. Cada vez que la cuenta del tick es incrementada, el kernel debe verificar si es tiempo de desbloquear o despertar alguna tarea.

4.1.14. Administración de la memoria

El kernel del RTOS debe asignar RAM cada vez que una tarea, cola o semáforo es creado. En FreeRTOS hay tres esquemas de asignación de memoria. El primero reserva una gran tabla en la memoria llamada "the heap". Cada vez que la llamada al sistema de asignación de memoria es usada (malloc), el puntero por espacio libre es incrementado con el tamaño de la asignación y un puntero es retornado a la aplicación. La liberación de memoria simplemente no hace nada. FreeRTOS provee sin embargo, un esquema para alojamiento de memoria mas complejo. El segundo esquema utiliza el algoritmo que mejor se ajusta para re-asignar bloques de memoria que han sido liberados. Esta esquema de asignación no es determinístico y el

comportamiento en tiempo real del sistema puede ser afectado. Finalmente, el tercer esquema utiliza las funciones malloc() y free() de librerías estándar. Tampoco es determinístico, necesita soporte del compilador y puede llevar a una sobrecarga de uso de memoria.

En general, el primer esquema es el usado, porque por mas pequeño que sea el dispositivo donde se utilice, prácticamente nunca se usa liberación de memoria y teniendo en cuenta que afecta al comportamiento de tiempo real del sistema. Así, con el primer esquema se logra bajo consumo de energía y un comportamiento en tiempo real.

4.2. Portación al ARM7TDMI-LPC2114

Se decidió seguir los pasos propuestos por la pagina del FreeRTOS. Ellos son modificar el ejemplo que viene para ARM7 de NXP, con compilador GCC pero para la placa prototipo de OLIMEX con microcontrolador LPC2106.

4.2.1. Compilador

Como compilador se uso el ARM-ELF-GCCSuite, una versión que difundida en los trabajos anteriores del CIII, con:

- binutils-2.17
- gcc-4.2.0
- gnuarm-3.4.3
- gnuarm-4.0.2-x86_64
- newlib-1.15.0

En los primeros pasos, se modifico el script “rom_arm.bat” creado para Microsoft Windows que provista en la versión que del FreeRTOS provista de su sitio web, por el script para Linux “rom_arm.sh”. Este script setea las variables de entorno necesarias para ejecutar la compilación del FreeRTOS. Cabe aclarar, que en Linux dichas variables solo son seteadas para ejecutar la compilación, cuyo comando “make” se encuentra al final del script. Una vez que termina la ejecución del script, las variables son borradas. Es el mismo efecto que setear las variables individualmente y llamar al Makefile del proyecto mediante el comando “make”.

4.2.2. Interface serie

Para programar el microcontrolador, se utilizo el software "lpc21isp_161", mediante un adaptador USB-Serie, en principio externo y luego integrado a la placa madre de control, modificando el Makefile con el agregado de la siguiente linea:

```
lpc21isp -wipe -hex rtosdemo.hex /dev/ttyUSB0 115200 14745
```

La velocidad de grabación utilizada es de 115200bps, valor configurable.

4.2.3. Modificación del port existente

Los primeros pasos fueron los siguientes:

- Se creo una copia del ejemplo para el microcontrolador LPC2106 para ser adaptado al microcontrolador LPC2114.
- Se modificó el archivo lpc21XX-rom.ld, cambiando la memoria RAM por 16kB.
- Se modificó el archivo lpc21XX.h, comparándolo con el archivo ya usado en anteriores aplicaciones al LPC2114 en el CIII. En este archivo se describen las direcciones de memoria de todos los periféricos del microcontrolador, entre otros.

En la pagina del FreeRTOS, precisamente en la parte de "como modificar un Demo ya existente, ROM y RAM usados", aclara que si se modifica un Demo para un micro que tiene menos RAM, como es nuestro caso, en el que pasamos del Demo del LPC2106 con 128kB de RAM al LPC2114 con 16 kB de RAM, se debe modificar también el valor de "configTOTAL_HEAP_SIZE", ubicado en "FreeRTOSConfig.h", dentro de la carpeta del Demo. Este valor indica la cantidad de memoria que se le cede al RTOS para su funcionamiento de las tareas, denominada HEAP.

4.2.4. Testeos del port

Se implemento en primera instancia, una función principal "main" provista en la documentación del FreeRTOS, en el que no se utilizan tareas ni el scheduler del RTOS. Simplemente se utiliza para comprobar el correcto

funcionamiento del port, "toogleando" LEDs ubicados en la placa prototipo. El siguiente paso, fue correr el mismo programa, pero que se ejecute en una tarea guiada por el Scheduler del sistema operativo. De esta manera se comprobó la correcta configuración, seteo y funcionamiento del FreeRTOS.

Los Demos provistos por el FreeRTOS, llevan al sistema operativo al límite de sus prestaciones, probando todas sus características. Por este motivo, se fue creando de a una tarea por vez, de las provistas en el Demo. De esta manera, se fueron corroborando las configuraciones y seteos. Uno de los bugs que se encontró y que es un error típico en este tipo de aplicaciones, es el tamaño del stack de las tareas. Debido a que el port se baso en un microcontrolador con una cantidad de memoria RAM mucho mayor, el seteo por defecto del tamaño de stack de tarea mínimo, era demasiado grande al intentar que el programa Demo funcionara con todas las tareas. Modificando la configuración de este tamaño mínimo por uno menor, solucionó el problema.

4.3. Caracterización (Benchmarking)

Un kernel de tiempo real y su capa de abstracción de hardware tiene muchas ventajas. El código de aplicación desarrollado tiende a ser mucho más portable y flexible, porque el desarrollador solo debe preocuparse por codificar la aplicación sin tener tanto cuidado con los detalles de bajo nivel.

Sin embargo es necesario tener en cuenta que se necesitará mayor capacidad de proceso y recursos en general, porque todas las funciones del RTOS utilizan memoria y ocupan tiempo del procesador, incrementado la latencia del sistema. Por lo tanto, es necesario poder identificar la magnitud de estos fenómenos, a fin de tenerlos en cuenta para que no afecten el normal funcionamiento de la aplicación.

Dado que cada función particular del RTOS consume una cantidad específica de recursos, y debido a la naturaleza modular de FreeRTOS, se identificaron los módulos de mayor incidencia, se evaluó la importancia de dicha característica para nuestro proyecto, y en el caso en que fue necesario se quitaron las funciones no utilizadas.

Después de realizar diversos análisis, fue necesario efectuar modificaciones en el proyecto original:

4.3.1. Manejo de memoria

El esquema de manejo de memoria con las funciones de la librería standard de C malloc() y free() permite realizar asignación dinámica de memoria, con lo que se pueden crear y destruir tareas en tiempo de ejecución. Sin embargo, la implementación de dichas funciones requiere grandes cantidades de memoria, y además este esquema no es determinístico, por lo que afecta las capacidades de tiempo real.

Se optó por utilizar un esquema de asignación de memoria estática. Lo que se hace es reservar una gran tabla en memoria (heap). Cada vez que se realiza una llamada a las funciones malloc() y free() lo único que hace esta implementación es entregar un puntero a una porción de la tabla. Este esquema es mucho más eficiente en el manejo de los recursos, pero no permite la creación y destrucción de tareas en tiempo de ejecución.

4.3.2. Pila de memoria

Todas las llamadas a funciones que se producen en un programa escrito en C (tenga o no Sistema Operativo) utilizan memoria RAM, en la que se almacena como mínimo la dirección de retorno de la función y los parámetros que se le pasan a la misma. Dado que naturalmente en el código se presentan funciones que llaman a otras funciones, es necesario que toda la información de los sucesivos pases de parámetros quede almacenada. En los sistemas embebidos, que tienen recursos reducidos, hay un límite práctico en el que se produce lo que se llama "desbordamiento de pila", que genera una falla generalizada del sistema que lo vuelve inestable, siendo necesario reinicializarlo.

Para evitar este problema se realiza un estudio de la cantidad de memoria RAM disponible con todas las tareas activadas, y más especialmente cuando se ejecutan las que hacen uso más intensivo de memoria (tareas de comunicación y odometría). Para llevar a cabo este estudio se contó con el dispositivo JTAG ARM-USB-OCD, el servidor de Debug OpenOCD y el entorno de Debug Insight. Estas herramientas permiten detener la ejecución normal del programa introduciendo "breakpoints", e inspeccionar la memoria RAM. Como resultado se observó que de los 16Kb de RAM disponibles, casi 6Kb quedan libres en los momentos de mayor carga, y además que el mayor peso en memoria no lo tiene el Sistema Operativo en si mismo, sino las librerías de punto flotante, de las que no podemos prescindir por su importancia en el cálculo de la odometría y las funciones de control.

4.3.3. Tiempo de Procesamiento

Cada tarea tiene 3 parámetros relativos al tiempo que son su prioridad, su frecuencia de ejecución y el tiempo que necesitan para ejecutarse en su totalidad. Por ello es importante tener en cuenta que las tareas de mayor prioridad no utilicen demasiado tiempo de procesamiento, para que las de menor prioridad puedan ejecutarse normalmente. Si esto no sucede, se produce un fenómeno conocido como "Task Starvation", donde las tareas están listas para ser ejecutadas pero no se les asigna recursos.

Luego de realizar un análisis de las tareas en ejecución utilizando un osciloscopio y generando una señal dentro de la tarea "Idle", que se ejecuta cuando el RTOS no tiene ninguna otra tarea lista para ejecutar, se pudo observar que con toda la funcionalidad implementada aún queda casi un 40 % del tiempo disponible para poder agregar nuevas funcionalidades.

Capítulo 5

Aplicación

5.1. CiiiEmbLibs

Para la realización del proyecto, se hizo uso de las librerías CiiEmbLibs [9], desarrolladas íntegramente en el CIII, como un proyecto que tiene por objetivo la creación de módulos de programación o librerías para microcontroladores de la familia LPC21xx de NXP con el fin de simplificar el desarrollo de software embebido en las diferentes áreas de aplicación del centro. Estos módulos incluyen tanto librerías de acceso al hardware periférico como módulos de software o algoritmos.

Estas librerías (CiiiEmbLibs) disponen de funciones de interfaz sencillas para los principales periféricos del μC como por ejemplo el timer, los generadores de modulación de ancho de pulso PWM, el módulo de captura de entrada utilizado para la decodificación de los encoders ópticos, la UART (Universal Asynchronous Receiver-Transmitter) para la comunicación con la PC a bordo, entradas y salidas digitales generales, interrupciones, etc.

5.1.1. Módulo UART

Consiste en una librería que configura la UART0 y UART1 del microcontrolador LPC2114. Consta de un set de funciones que inicializan, controlan y manejan interrupciones de la UART.

5.1.2. Módulo GPIO

Consiste en una librería que configura, controla y setea los puertos de propósito general (GPIO) como tales. El microcontrolador LPC2114 posee dos puertos de propósito general: PORT0 y PORT1, de los cuales el primero

posee 30 pines disponibles y el segundo solo 16, siendo un total 46 pines de E/S.

5.1.3. Módulo Communication

Permite la transferencia de datos entre dos corrientes de diferente velocidad. La implementación provee rutinas para lectura y escritura de datos en forma de paquetes desde las aplicaciones; además implementa la rutina de los callbacks correspondientes al dispositivo físico o lógico utilizado en el otro extremo de la conexión de la librería.

La operación de la transmisión y recepción emplea la mecánica de productor-consumidor; actuando sobre un buffer intermedio implementado en el módulo y coordinado a través de una variable protegida incrementada por el productor y decrementada por el consumidor. En términos generales la rutina del productor carga los caracteres secuencialmente en el buffer de caracteres, arma los paquetes (index y length), chequea los desbordes, controla la circularidad e incrementa la cuenta de paquetes (semáforo). La rutina del consumidor extrae los caracteres secuencialmente, liberando el espacio en el buffer de caracteres, controla la circularidad y decrementa la cuenta de paquetes (semáforo).

5.1.4. Módulo Capture

El objetivo de esta librería consiste en brindar un módulo funcional para la detección de flancos, ya sea de subida, bajada o ambos, pudiendo utilizar hasta dos módulos captures, contenido uno en cada TIMER del LPC2114.

5.1.5. Módulo Encoders

El objetivo de esta librería consiste en brindar un módulo funcional para la lectura de encoder incremental. Esta librería está directamente relacionada con el módulo ARM para el manejo del capture, debido a que esta función comparte una funcionalidad aportada por el módulo Capture. Maneja un encoder incremental con Fase A y Fase B. Obtenemos cantidad de pulsos y sentido de avance. En esta librería no se configura ningún pin.

5.1.6. Módulo IRQ

Consiste en una librería que configura las interrupciones vectorizadas IRQ del microcontrolador LPC2114. Actualmente esta librería solo opera sobre las interrupciones vectorizadas IRQ.

5.1.7. Módulo PID

Consiste en un módulo que realiza el cálculo de un controlador PID. Se puede modificar el valor de los coeficientes a través de las APIs, recibe un dato como entrada al controlador (señal de error) y devuelve el valor de compensación de salida.

5.1.8. Módulo PWM

Consiste en una librería que configura y controla el PWM del microcontrolador LPC2114. Básicamente el PWM posee dos modos de funcionamiento: simple flanco y doble flanco, como se indica en la figura. Esta librería, actualmente, se centra solo en el control del PWM de simple flanco.

5.1.9. Módulo Timer

Consiste en una librería que configura y controla los dos TIMERS del microcontrolador LPC2114. Los dos TIMERS son idénticos, de 32 bits de Prescaler y TImer Counter. Esta librería se centra solo en modo de cuenta de intervalos de tiempo entre eventos (match).

5.2. División en tareas del FreeRTOS

Las tareas del sistema operativo para la aplicación del robot RoMAA se dividieron de la siguiente manera:

- Cierre de lazo de control PID
- Comunicación con la PC de abordo
- Cálculo de la odometría del robot
- Logging de datos hacia la PC de abordo

A continuación se describen en detalle dichas tareas y las funciones usadas para su correcto funcionamiento.

5.2.1. main(): función principal del programa

La función principal main, es la encargada configurar e inicializar todo el hardware y el software que el FreeRTOS utilizará para su funcionamiento normal. Es decir, que una vez que se ejecuta esta función main, cede el control al scheduler del RTOS y no vuelve a ejecutarse.

En primer lugar se utiliza una función llamada prvSetupHardware que configura e inicializa todo el hardware a utilizar. Una vez que el hardware ya ha sido inicializado, se procede a inicializar las variables de software mediante la función init_all(). Luego se procede a la creación de las tareas que utilizará el FreeRTOS para su funcionamiento. Como dijimos anteriormente, las tareas que se utilizan son cuatro y se describirán en detalle mas adelante en este capítulo:

- TaskPIDLoop();
- TaskCommunication();
- TaskOdometry();
- TaskLogging();

Para finalizar, solo resta cederle el control al sistema operativo, llamando a la función vTaskStartScheduler(). De esta manera, se inicializa el scheduler, que es quien controlará que tarea se ejecuta en que momento, administrando los cambios de contexto, prioridades, etc.

5.2.2. prvSetupHardware(): configuración del hardware

Esta función se encarga como se dijo anteriormente, de inicializar todo el hardware a utilizar. Entre ellos, se puede mencionar a los siguientes:

- Phase Locked Loop (PLL)
  El microcontrolador utilizado, funciona con un cristal de 14, 7456MHz, el cual es multiplicado por cuatro mediante un PLL interno para obtener una frecuencia de funcionamiento de 58,9824MHz. Estos valores de cristal, fueron elegidos de acuerdo a la hoja de datos del microcontrolador, ya que de esta manera se permite a la UART funcionar a una velocidad máxima de 115200bps.
- General Purpose Input Output (GPIO)
  Se configura todas las entradas/salidas digitales como salida por defecto.
- RS-232
  Se habilitan los pines de Tx y Rx de la UART0.
- Fases de los encoders
  Se configuran los pines 16, 23, 25 y 30 del puerto 0 como entrada, para su utilización como lectura de la fases A y B de los encoders,

- Capture
  Se configura e inicializa el módulo capture 0 y 3 en los pines 2 y 29 del puerto 0 respectivamente. Dichos puertos están conectados eléctricamente a las fases A de los encoders (pines 23 y 25 del puerto 0).

- Timer 0 – Interrupción Timer 0
  Se habilita el timer 0 y su respectiva interrupción para el funcionamiento con los módulos capture.

- Llaves H
  Se inicializan los pines de los puertos que habilitan a los motores como salida y se los setea a 0 por defecto.

- UART0
  Se configura e inicializa la UART0 y su respectiva interrupción.

5.2.3. init_all(): inicialización de variables

Esta función es similar a prvSetupHardware(), solo que aquí las configuraciones e inicializaciones son de software:

- Motores
  Se inicializan todas las variables de ambos motores, como la velocidad lineal, angular, velocidades de referencia, etc.

- PID
  Se inicializan todas los coeficientes de los controladores PID a los valores por defecto, tanto de los controladores de lazo de ambos motores como el controlador de Cross-Coupling.

- Constantes para cálculos de odometría
  Se inicializan las constantes velocidad angular de referencia, radio de las ruedas y base de las ruedas, como así también los valores de X, Y y Theta.

- Encoders
  Se inicializan todas las variables pertenecientes a la estructura de cada uno de los encoders.

- Tiempos de las tareas
  Se inicializan todos los tiempos de control de la periodicidad de las tareas.

- Banderas
  Se inicializan todas las banderas usadas para el logging de datos.
- Lazo
  Se inicializa el tipo de lazo con el que funciona el control por defecto.
- PWM
  Se inicializa los valores de los PWM con el que comienzan ambos motores.

5.2.4. irq_timer0: interrupción para decodificación de encoders

Los encoders incrementales generan dos señales (A y B) desfasadas 90º entre si conocido como señales en cuadratura, ver figura 5.2. La señal A está "adelantada" respecto de B cuando el motor gira en una dirección, y "atrasada" cuando gira en la dirección opuesta. El período de cualquiera de las dos señales permite calcular la velocidad de rotación. De esta manera es posible saber al mismo tiempo tanto la velocidad, como la dirección de giro de los motores.

En el momento de elegir el método de medición de la velocidad angular de las ruedas se presentaron dos opciones [11]. Una se denomina "medición por período" y consiste en medir el intervalo de tiempo entre dos pulsos de encoder. La otra se llama "medición por frecuencia" y se realiza midiendo la cantidad de pulsos de encoder recibidos en un intervalo de tiempo fijo.

Flanco A	**Flanco B**	**Sent. giro**
ascendente	bajo	horario
descendente	alto	horario
ascendente	alto	antihorario
descendente	bajo	antihorario

Figura 5.1: Decodificación de encoder

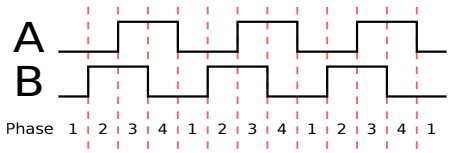


Figura 5.2: Señales en cuadratura de los encoders

El microcontrolador LPC2114 tiene una función de "captura" para sus Timers, que consiste en poder guardar en un registro el valor del registro contador (Timer Counter) en cada transición de una señal de entrada [3]. Por esta razón se eligió el método de medición por período, dado que solo utiliza la interrupción de captura. Dicha interrupción se genera cada vez que la señal "A" de alguno de los dos encoders cambia de estado. Allí se mide el período de la señal, comparando con el valor del Timer Counter en la interrupción anterior, y se obtiene la dirección de giro leyendo el estado de

la señal “B” del encoder correspondiente. Dado que dichas señales están en cuadratura, se pueden sintetizar los estados indicados en el cuadro 5.1.

La información obtenida se guarda en variables compartidas con la tarea de control de lazo cerrado, y el acceso a las misma se encuentra protegido mediante la utilización del servicio de “Exclusión Mutua” (Mutex) que ofrece FreeRTOS.

En la implementación del programa del microcontrolador para la decodificación de las señales de los encoders se utilizaron la estructura de datos mostrada en el listado 5.1

Listing 5.1: estructura de los encoders

```
struct encoder
{
  signed int count;
  signed int count_previous;
  signed int delta_count;
  signed int edge_current;
  signed int edge_previous;
  signed int period_pclk;
  signed int period_pclk_temp;
};
```

a partir de la cual se crean las siguientes variables

```
struct encoder encoder1;
struct encoder encoder2;
```

que mantienen un registro del estado de los encoders de cada motor.

5.2.5. TaskPIDLoop: tarea del lazo de control

Se implementa desde una tarea periódica, de intervalo ajustable a partir de un mínimo de $\Delta t = 20ms$, que se encarga de actualizar el ancho de pulso que generan los módulos PWM del microcontrolador que están conectados a cada uno de los drivers de los motores, a los valores que resultan de los algoritmos de control implementado. Además, en cada ciclo se actualizan los parámetros de velocidad y posición angular medidos a través de los codificadores ópticos de posición angular relativa accionados directamente por los ejes de cada uno de los motores de tracción.

El control implementado en el presente trabajo consta de sendos controladores PID [12] para cerrar el lazo de velocidad de cada motor de tracción y un tercer controlador PID para realizar el lazo de control de velocidad lineal y angular (v, ω) del robot de tracción diferencial (este lazo se conoce como “cross-coupling”), figura 5.3.

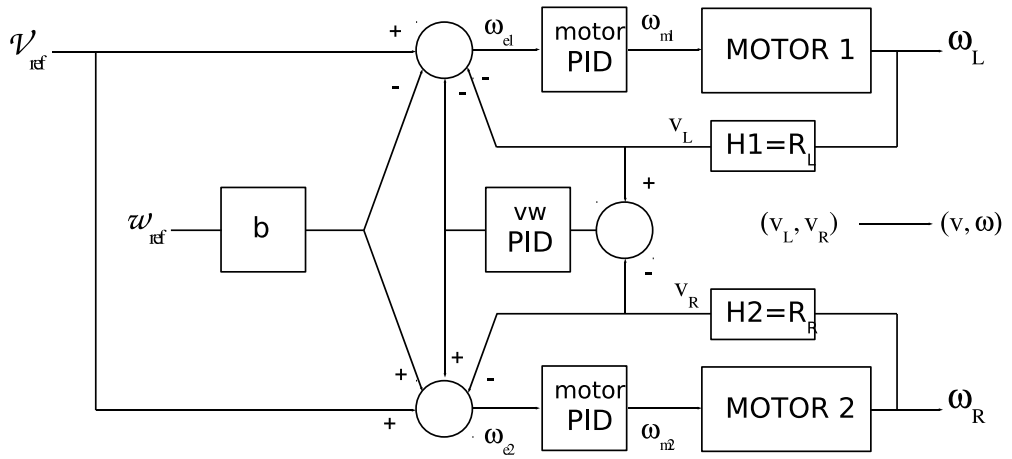


Figura 5.3: Lazo de control implementado en RoMAA. R_L y R_R son los radios de las ruedas y b la distancia entre ruedas

La señal de referencia salida de cada controlador PID esta dada por la siguiente ecuación

$$R_n = R_{n-1} + K_P(e_n - e_{n-1}) + K_I\frac{e_n + e_{n-1}}{2} + K_D(e_n - 2e_{n-1} + e_{n-2}) \quad (5.1)$$

donde e_n es el error actual, e_{n-1} el error en el período de muestreo anterior y e_{n-2} en dos períodos de muestreos anteriores. Y K_P, K_I y K_D las constantes del controlador proporcional, integral y derivativo, respectivamente.

Para la programación del controlador PID se generó la estructura de datos mostrada en el listado 5.2 y para el lazo del motor la estructura del listado 5.3.

Listing 5.2: estructura PID

```
struct PID {
  int KP;
  int KI;
  int KD;
  float proportional;
  float integral;
  float derivative;
  float in;
  float in1;
  float in2;
  float in3;
  float out;
};
```

Listing 5.3: estructura lazo motor

```
struct motor_loop {
  struct PID pid;
  float angular_speed;
  float linear_speed;
  float feedback_cte;
  int ZERO_OFFSET;
  float reference;
};
```

Y las siguientes variables para la aplicación del programa

```
struct motor_loop loop_motor1;
struct motor_loop loop_motor2;
struct PID vw_pid;
```

Así, el lazo de control mostrado en la figura 5.3 se implementa como se muestra en la siguiente sección de código (solo se muestra un motor)

```
//--------------------------------  vw  -------------------------------//
vw_pid.in = loop_motor1.linear_speed - loop_motor2.linear_speed;

vw_pid.proportional = vw_pid.KP * vw_pid.in;
vw_pid.integral = vw_pid.KI * (vw_pid.in1 + vw_pid.in) / 2;

vw_pid.out = vw_pid.proportional + vw_pid.integral;
vw_pid.in1 = vw_pid.in;

w_out = (float)NOM_WHEELBASE * w_ref;

//--------------------------------  Motor 1  -------------------------//
loop_motor1.pid.in = loop_motor1.reference - loop_motor1.linear_speed -
             vw_pid.out - w_out;

loop_motor1.pid.proportional = loop_motor1.pid.KP * loop_motor1.pid.in;
loop_motor1.pid.integral = loop_motor1.pid.KI * (loop_motor1.pid.in1 +
                  loop_motor1.pid.in) / 2;

loop_motor1.pid.out = loop_motor1.pid.proportional +
              loop_motor1.pid.integral;

loop_motor1.pid.in2 = loop_motor1.pid.in1;
loop_motor1.pid.in1 = loop_motor1.pid.in;
```

Esta tarea es en si una de las mas importantes, ya que realiza el control sobre cada uno de los motores según el tipo de lazo que se configure. Para poder actualizar los valores de los PWM de control de los motores a los valores correctos, debe calcular las velocidades angular y linear, los valores de los controladores PID según el tipo de lazo que se seleccione, entre otros.

La frecuencia de ejecución de la tarea, determina en si cada cuanta cantidad de tiempo se cierra el lazo de control, en el caso que se este utilizando el lazo cerrado. Este tiempo es ajustable y se puede setear a través de los parámetros recibidos por la tarea de comunicación, siendo el tiempo utilizado por defecto de 20ms. La tarea divide los procedimientos que realiza en tres grandes partes que se detallan a continuación.

Seteo de la velocidad angular y linear

Para el cálculo de las velocidades, la tarea utiliza la variable period_pckl_temp perteneciente a la estructura encoder_t, donde se almacenan las cuentas del timer 0 que se producen entre cada pulso de encoder (para mas detalle ver interrupción_capture). Es decir, que sabiendo la cantidad de pulsos de reloj que se produjeron entre cada pulso de movimiento de encoder, se puede determinar la velocidad lineal y angular de cada motor de la siguiente manera:

```
loop_motor1.angular_speed = (float)N_PCLK_TO_W / encoder1.period_pclk_temp;
        loop_motor1.linear_speed = loop_motor1.feedback_cte *
                            loop_motor1.angular_speed;
```

donde (float)N_PCLK_TO_W y loop_motor1.feedback_cte son parámetros de odometría de la plataforma RoMAA.

Actualización de los PWM según el tipo de lazo

A través del la tarea de comunicación, se reciben los comandos del tipo de lazo en el que se desea trabajar. Estos seteos se utilizan en esta parte de la tarea TaskPIDLoop para realizar los cálculos necesarios. El control de la plataforma RoMAA permite tres tipos de lazo de control:

- Lazo abierto
- Lazo Cerrado Completamente (Cross-Coupling))
- Lazo Cerrado de Motor

Lazo abierto En primera instancia, se chequea si el caso seleccionado es el Lazo abierto. Este es el caso mas simple, ya que ninguno de los controladores PID actúa. Simplemente se chequea si el valor deseado de PWM recibido de la tarea de comunicación no satura los valores posibles y se setea directamente el valor de PWM de cada motor. En el caso de que el valor deseado sature, es decir, iguale o sobrepase el valor máximo posible de PWM, se mantiene el valor anterior que poseían dichos PWM.

Lazo cerrado completamente (Cross-Coupling) En el caso que Lazo Abierto no fuese el seleccionado, se chequea si Lazo Cerrado Completamente lo es. Este caso es el mas complejo, ya que involucra tres lazos de realimentación y tres controladores PID implementados totalmente por software. Es decir que todos los cálculos de los tres PID se realizan en esta parte de la tarea TaskPIDLoop.

En primer lugar se calcula toda la rama del PID del Cross-Coupling. Para ello se usan los datos de velocidad linear calculados anteriormente al comienzo de la tarea TaskPIDLoop. Se obtiene así w_out, que se suma en los bucles de entrada del lazo de ambos motores. El lazo del motor 1 y del motor 2 son calculados independientemente pero de forma similar. Utilizando w_out anteriormente calculado, se determina la salida del controlador PID de cada uno de los motores, valores con los que se actualizan los PWM de los motores.

Lazo cerrado de motor únicamente Si el tipo de control seleccionado es el de Lazo Cerrado de Motor, los cálculos son idénticos al de Lazo Cerrado Completamente, con la diferencia de que el Cross-Coupling no se implementa y por lo tanto no se calcula ni interfiere w_out.

5.2.6. TaskOdometry: tarea de cálculo de odometría

Teniendo en cuenta el transcurso del tiempo, y correlacionando las señales recibidas desde los encoders ópticos, se lleva un registro de velocidad y posición del vehículo.

Las ecuaciones finales de odometría son [13]

$$d_C = \frac{d_L + d_R}{2} \tag{5.2}$$

$$\phi = \frac{d_R - d_L}{b} \tag{5.3}$$

$$\phi' = \phi + \theta \tag{5.4}$$

$$x' = x + d_C \cos\theta \tag{5.5}$$

$$y' = y + d_C \sin\theta \tag{5.6}$$

donde d_R y d_L son las distancias recorridas por la rueda derecha e izquierda respectivamente en un ciclo de muestreo y d_C es la distancia recorrida por el punto odométrico. ϕ es el cambio de orientación en un ciclo y (x, y, θ) es la pose (posición más orientación) del robot en un sistema de coordenadas global.

El microcontrolador obtiene los valores para realizar la odometría a partir de la lectura de los encoders y de los parámetros R y b, que son el diámetro de las ruedas y la distancia entre ruedas; y mantiene un registro de odometría con período de muestro ajustable controlado por el RTOS. La sección de código siguiente muestra el cálculo de odometría:

```
encoder1.delta_count = encoder1.count - encoder1.count_previous;
```

```
encoder1.count_previous = encoder1.count;
encoder2.delta_count = encoder2.count - encoder2.count_previous;
encoder2.count_previous = encoder2.count;

distance_wheel1 = encoder1.delta_count * (float)WHEEL_DIST_RESOLUTION;
distance_wheel2 = encoder2.delta_count * (float)WHEEL_DIST_RESOLUTION;

center_distance = (distance_wheel1 + distance_wheel2) / 2;
center_phi = (distance_wheel2 - distance_wheel1) / (float)NOM_WHEELBASE;

theta = theta + center_phi;
x = x + center_distance * cos(theta);
y = y + center_distance * sin(theta);
```

Esta tarea es similar a la tarea de logging de datos, solo que aquí no solo se cargan los datos a ser enviados por la tarea de comunicación al solicitarse la odometría, sino que también se calculan se realizan dichos cálculos. La frecuencia de repetición es por defecto de 20ms, seteable también a través de los comandos de comunicación.

5.2.7. TaskSerialCommunication: tarea de comunicación

Como el microcontrolador no cuenta con puerto USB, se utiliza una UART. La librería CIIILibs provee una estructura de buffers circulares y la posibilidad de separar en paquetes de cadenas de texto ASCII con indicadores de inicio y finalización de trama. A través de este canal actualmente se transmiten las referencias de velocidad lineal y angular, los valores de ajuste del controlador PID, e indicadores de estado del vehículo como posición y velocidad, junto con una referencia del momento de medición (timestamp).

Esta tarea es la encargada de la comunicación entre el FreeRTOS y la CPU externa utilizada en el RoMAA. A través de esta tarea, el software usado desde la CPU externa puede obtener información (por ejemplo, datos de la odometría de la plataforma, valores de los coeficientes de los controladores PID, lazo actualmente activo, velocidades angulares y lineales de los motores, valores de PWM de los motores, etc.) y dar información, es decir, comandos (por ejemplo, movimientos que se desee realice la plataforma, modificar los valores de los coeficientes de los controladores PID, setear velocidades, modificar tiempos de delay de las tareas, etc.). La frecuencia de repetición de la tarea es de 10ms, siendo este un parámetro seteable.

Todas las configuraciones de hardware de la comunicación son llevadas a cabo anteriormente a la ejecución de esta tarea, en prvSetupHardware. Desde allí se setea la utilización de la UART0 del LPC2114, la interrupción que generará dicha UART y será utilizada para la recepción y envío

de datos, etc. Desde la tarea TaskCommunication, se llama directamente a la función com_init(), perteneciente al módulo communication de la librería CiiiEmbLibs. Esta función inicializa los buffers circulares de transmisión y recepción, para ser usados en la comunicación con la CPU externa.

Una vez que el hardware esta configurado y los buffers inicializados, se utilizan solo dos funciones como APIs para la comunicación. Ellas son:

- `int send_packet(char* buf, int count);`
- `int receive_packet(char* buf);`

Para enviar paquetes, se utiliza la función send_packet(). Como parámetro se dan el buffer que contiene los datos a enviar y la cantidad de datos de ese buffer que se desean enviar. El módulo communication utiliza estos datos para crear el paquete a enviar, que se compone de:

- inicio de trama → por ejemplo, '*'
- la cantidad de datos en el paquete
- los datos
- final de trama → por ejemplo, '/'

Estos datos empaquetados, son almacenados en el buffer circular de transmisión, y serán enviados inmediatamente la UART este libre para realizar la acción. Los paquetes que el FreeRTOS recibe desde la CPU externa, deben estar compuestos de la misma manera, caso contrario serán descartados por el módulo communication. La función API receive_packet(), es la que se encarga de tomar datos ya ubicados en el buffer circular de recepción, almacenarlos en el buffer pasado como parámetro para su utilización dentro de la tarea TaskCommunication del FreeRTOS y devuelve el número de datos que posee el paquete extraído del buffer circular, valor que debe ser usado por la tarea para saber cuantos datos válidos posee el paquete.

Cuando la tarea de comunicación recibe paquetes desde la CPU externa, estos son siempre comandos que pueden ser como dijimos, de lectura o escritura de información. La tarea de comunicación usa el primer dato dentro del paquete para saber a que comando se refieren los datos que siguen a continuación. Estos comandos están representados por caracteres ASCII que para su mejor interpretación han sido definidos con etiquetas mas representativas. A continuación, se muestra una tabla con todos los comandos posibles y la acción que el FreeRTOS debe realizar cuando los reciba:

- CMD_RESET: llama a la función init_all() que resetea todas las variables de motores y sus mediciones. Es decir, funciona como un reset.

- CMD_SET_SPEED: se utiliza para setear un valor de velocidad linear y angular de la plataforma RoMAA.
  Se utiliza con el lazo cerrado completamente, por lo tanto lo setea. Asigna los datos leídos del buffer a los valores de velocidades lineares de cada motor de referencia y angular de referencia a usar en el cálculo del controlador PID de cada motor y del Cross-Coupling.

- CMD_SET_WHEEL_SPEED: se utiliza para setear las velocidades individuales de ambas ruedas.
  Se utiliza con lazo cerrado de motor únicamente, por lo tanto lo setea. Asigna los datos leídos del buffer a los valores de velocidades lineares de cada motor de referencia a usar en el cálculo del controlador PID de cada motor.

- CMD_GET_SPEED: se utiliza para solicitar los valores de velocidad linear y angular de la plataforma RoMAA.
  Se calculan ambas velocidades y se envía el paquete al buffer circular de transmisión mediante send_packet().

- CMD_SET_T_LOOP: se utiliza para setear el tiempo utilizado para cerrar el lazo.
  Se carga a la variable que controla la periodicidad de repetición de la tarea TaskPIDLoop con el valor recibido en el paquete.

- CMD_GET_T_LOOP: se utiliza para solicitar a través de una CPU externa el valor del tiempo de cierre de lazo seteado.
  Se envía a través de send_packet(), la variable que controla la periodicidad de repetición de la tarea TaskPIDLoop con el valor recibido en el paquete.

- CMD_SET_T_SAMPLING: se utiliza para setear el tiempo de muestreo o logging de datos.
  Se carga a la variable que controla la periodicidad de repetición de la tarea TaskLogging con el valor recibido en el paquete. Es decir, control la periodicidad con las que las variables de logging son actualizadas.

- CMD_GET_T_SAMPLING: se utiliza para solicitar a través de una CPU externa el tiempo de muestreo o logging de datos.
  Se envía a través de send_packet(), la variable que controla la periodicidad de repetición de la tarea TaskLogging con el valor recibido en el paquete.

- CMD_SET_T_ODOMETRY: se utiliza para setear la periodicidad de los cálculos de odometría. Se carga a la variable que controla la periodicidad de repetición de la tarea TaskOdometry con el valor recibido en el paquete.

- CMD_GET_T_ODOMETRY: se utiliza para solicitar a través de una CPU externa la periodicidad de repetición de los cálculos de odometría. Se envía a través de send_packet(), la variable que controla la periodicidad de repetición de la tarea TaskOdometry con el valor recibido en el paquete.

- CMD_SET_MOTOR_PID: seteo del kp, ki y kd de ambos motores.
  Es decir, se setean los coeficientes de los controladores PID de ambos motores.

- CMD_GET_MOTOR_PID: solicita el valor de kp, ki y kd de ambos motores.
  Se envía a través de send_packet() los valores de los coeficientes de los controladores PID de ambos motores

- CMD_SET_VW_PID: seteo del kp, ki y kd del controlador PID Cross-Coupling.

- CMD_GET_VW_PID: solicita el valor de kp, ki y kd del controlador PID Cross-Coupling

- CMD_SET_ODOMETRY: seteo de los valores x, y y theta para la odometría. Previamente cierra el lazo completamente.

- CMD_GET_ODOMETRY: solicita los valor x, y y theta de la odometría. Se envían dichas variables a través de send_packet().

- CMD_RESET_ODOMETRY: resetea los valores de x, y y theta a cero.

- CMD_SET_PWM: setea directamente los valores de los PWM de los motores para lazo abierto.
  Es decir, se asignan los valores de PWM recibidos directamente a los motores, funcionando estos en lazo abierto.

- CMD_GET_PWM: solicita los valores de PWM de ambos motores.
  Se envían a través de send_packet().

- CMD_GET_ENC_COUNTER: se solicitan las cuentas almacenadas en los encoders

- CMD_GET_WHEEL_V: se solicita las velocidades lineares de los motores.
- CMD_GET_WHEEL_W: se solicita las velocidades angulares de los motores.
- CMD_GET_WHEEL_D: se solicita las distancias recorridas por cada rueda.
- CMD_ENABLE_MOTOR: se habilita a cada uno de los motores a través de las llave H.
- CMD_DISABLE_MOTOR: se deshabilita a cada uno de los motores a través de las llave H.
- CMD_SET_OPEN_LOOP_MODE: se setea el control a lazo abierto.
- CMD_SET_CLOSE_LOOP_MODE: se setea el control a lazo cerrado completamente.
- CMD_SET_MOTOR_LOOP_MODE: se setea el control a lazo cerrado de motor únicamente.
- CMD_GET_LOOP_MODE: se solicita el tipo de lazo de control activo.
- CMD_SET_KINEMATIC: se utiliza para setear valores de wheel_base y el radio de las ruedas.
- CMD_GET_KINEMATIC: se utiliza para obtener los valores de wheel_base y el radio de las ruedas seteados.
- CMD_GET_BODY_GEOM: se solicitan los datos del largo, ancho y alto de la plataforma RoMAA cargados actualmente.
- CMD_LOG_STOP: se solicita la detención del logging.
- CMD_LOG_WHEEL_V: se solicita un logging de n cantidad de la velocidad lineal de cada rueda.
- CMD_LOG_WHEEL_W: se solicita un logging de n cantidad de la velocidad angular de cada rueda.
- CMD_LOG_WHEEL_D: se solicita un logging de n cantidad de la distancia recorrida por cada rueda.

- CMD_LOG_ENC_COUNTER: se solicita un logging de n cantidad de la cuentas de los encoders.
- CMD_LOG_PWM: se solicita un logging de n cantidad de los PWM de cada motor.
- CMD_LOG_VW: se solicita un logging de n cantidad de la velocidad angular y lineal del RoMAA.
- CMD_LOG_ODOM: se solicita un logging de n cantidad de los valores x, y y theta de la odometría.

5.2.8. TaskLogging: tarea de registro de datos (data logging)

El funcionamiento de esta tarea es muy simple. En la tarea de comunicación se activaron flags de acuerdo al comando pedido de logging de datos. Esas banderas, son utilizadas ahora para cargar en los datos a enviar por la tarea de comunicación, los datos solicitados. La frecuencia de repetición de la tarea es por defecto de 50ms, valor seteable a través de los comandos de comunicación como se describió anteriormente.

5.2.9. Prioridades de las tareas

Como se definió anteriormente, el método usado para la asignación de prioridades de tareas es el denominado RMS (Rate Monotonic Scheduler). Este método asigna a la tarea de mayor frecuencia de repetición, la mayor prioridad. Pese a que éste fue el método elegido, se decidió realizar algunas modificaciones para su mejor funcionamiento.

La tarea de mayor repetición no necesariamente es siempre la tarea mas importante, siendo este el caso que se presentó. Siguiendo al método RMS, debería ser la tarea de comunicación la de mayor prioridad, debido a que es la tarea con mayor frecuencia de repetición, quedando el orden de prioridades de la siguiente manera:

- TaskCommunication
- TaskPIDLoop
- TaskOdometry
- TaskLogging

Sin embargo, en nuestro caso la tarea mas importante es la de cierre de lazo de control, a pesar de ejecutarse con menor frecuencia. Cuando ésta se ejecuta, es necesario que no tenga demoras, para garantizar así la ejecución de la misma en tiempo y forma. De esta manera se garantiza que el lazo de control se cierre en el tiempo definido, para que el control sea lo mas similar posible al teórico calculado. De esta manera, el orden de prioridades utilizado es el siguiente:

- TaskPIDLoop
- TaskCommunication
- TaskOdometry
- TaskLogging

Capítulo 6

Conclusiones

La realización de este "proyecto final de grado" ha tenido como resultado principal un prototipo íntegramente desarrollado por estudiantes, con componentes disponibles en el mercado local, y con el asesoramiento de Ingenieros de la Universidad Tecnológica Nacional, Facultad Regional Córdoba.

La placa desarrollada cumple con las expectativas originales que fomentaron su construcción. Ahora es posible contar en un solo módulo de hardware lo que anteriormente se realizaba en varios, siendo las funcionalidades presentes: alimentación con ahorro de energía, adaptación de señales, conversión RS232-USB y procesamiento con microcontrolador de 32 bits. La utilización del Sistema Operativo de Tiempo Real FreeRTOS proveyó de la flexibilidad que se esperaba obtener en un principio, y ya están previstas varias modificaciones que obtendrán provecho de la modularidad alcanzada. Además, como agregar tareas resulta bastante sencillo, la posibilidad de incorporar nuevas funcionalidades también se ha visto mejorada.

El sistema ya ha dejado su estado prototipo y de hecho ya está siendo utilizado para tareas de experimentación en trabajos relacionados a la robótica y visión por computadoras. Se posee actualmente un producto totalmente terminado, que satisface en exceso las directivas que se propusieron en los objetivos de este trabajo.

Como trabajo futuro se encuentra en estudio la posibilidad de migrar a algún microcontrolador de la serie LPC de NXP que posea mayor cantidad de opciones de periféricos. Podría ser un controlador USB propio, o protocolos estándares dentro de la industria automotriz como CAN o LIN.

Bibliografía

[1] ON Semiconductors. *AN920/D. Theory and Applications of the MC34063 and uA78S40 Switching Regulator Control Circuits*, May 2006.

[2] ON Semiconductors. *MC34063A, MC33063A,SC34063A, SC33063A,NCV33063A. 1.5 A, Step-Up/Down/ Inverting Switching Regulators*, May 2010.

[3] NXP Semiconductors. *UM10114. LPC21xx and 22xx User Manual*, April 2007.

[4] Jean J. Labrosse. *Embedded Systems Building Blocks. Complete and Ready to use Modules in C.* Miller Freeman. A United News and Media publication, 2000.

[5] The freertos project. http://www.freertos.org.

[6] D. A. Gaydou, G. F. Pérez Paina, G. M. Steiner, and J. Salomone. Plataforma móvil de arquitectura abierta. In *Proceedings of the V Argentine Symposium of Robotics 2008*. Ediuns, 2008, November 2008.

[7] M. Nitulescu. Theoretical aspects in wheeled mobile robot control. In *AQTR '08: Proceedings of the 2008 IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 331–336, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Placa multipropósito para el desarrollo de aplicaciones con microcontrolador lpc2114 de arquitectura arm. http://ciii.frc.utn.edu.ar/LabElectronica/PlacaMadre100.

[9] Librerías desarrolladas por el ciii para sistemas embebidos basados en la familia lpc21xx de nxp. http://ciii.frc.utn.edu.ar/LabElectronica/ModulosLPCARMWeb.

[10] Silicon Labs. *Single-Chip, USB to UART bridge. Datasheet.*, August 2009.

[11] Stare Z. Mijat N. Stojkovic, N. Dual-mode digital revolution counter. volume 2, pages 950 –954 vol.2, 2001.

[12] Thomas Brunl. *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*. Springer Publishing Company, Incorporated, 2008.

[13] E. Olson. A primer on odometry and motor control. Technical report, Massachusetts Institute of Technology, 2007.