ISBN: 978-607-9394-XX-X, Asociación Mexicana de Mecatrónica A.C., 2016

Sistema embebido de detección de obstáculos con freeRTOS para un robot móvil

Ocampo Álvarez Arturo[1], Moreno Espinoza Oscar G. [1], Fernández de Vega Francisco[2], Olague Caballero Gustavo[3]

[1] Facultad de Estudios Superiores Aragón, UNAM. Av. Rancho seco s/n, Impulsora. C.P. 57130, Nezahualcoyotl, Edo. Mex., México. aoa@unam.mx, gokaar@live.com.mx
[2] Departamento de Tecnología de los Computadores y de las Comunicaciones. Universidad de Extremadura. Centro Universitario de Mérida. C/Sta. Teresa de Jornet 38. 06800, Mérida-Badajoz, España. fcofdez@unex.es
[3] Departamento de Ciencias de la Computación, CICESE. Km. 107 carr. Tij.-Eda., 22860, Ensenada, México. olague@cicese.mx

Resumen

Este artículo trata del uso del sistema operativo en tiempo real, que se implementa en un microcontrolador de 8 bits en una plataforma de robot móvil. La principal tarea del sistema de control realizado es asegurar la comunicación con los diferentes tipos de sensores (ultrasónicos e infrarrojos) para la detección de obstáculos, usando la técnica marco de programación jerárquica, para lo cual se realizó una prueba que produce latencia y se comparó con la forma tradicional de programación de funciones. Los resultados obtenidos usando FreeRTOS son aceptables, logrando el control y activación de los actuadores a través de la comunicación con un sistema superior, integrando diversas tareas en un solo microcontrolador.

Palabras clave: Microcontroladores; Sistema operativo en Tiempo Real; FreeRTOS; Sensores.

1. Introducción

Un sistema embebido es una combinación de hardware y software que utiliza microprocesadores o microcontroladores, este tipo de sistemas está diseñado para realizar una función específica o varias al mismo tiempo, dentro de un sistema más grande [1]. Máquinas industriales, automóviles, equipos médicos, cámaras, electrodomésticos, aviones, máquinas expendedoras y juguetes, así como teléfonos móviles son algunos de los dispositivos posibles para un sistema embebido. Sin embargo, la mayoría de las aplicaciones requieren un sistema operativo o un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés Integred Development Enviroment) para ser programadas, particularmente donde los entornos operativos en tiempo real deben ser atendidos, los diseñadores han decidido cada vez más que los sistemas son generalmente lo suficientemente rápidos y las tareas tolerantes de ligeras variaciones en el tiempo de reacción que los enfoques "casi en tiempo real" son adecuados. En estos casos, las versiones simplificadas del sistema operativo Linux se implementan comúnmente [2], aunque también hay otros sistemas operativos que han sido reducidos para funcionar en sistemas embebidos, incluyendo Embedded Java y Windows IoT (antes Windows Embedded).

Un sistema operativo en tiempo real (RTOS por sus siglas en inglés real-time operating system) es un sistema operativo que está optimizado para su uso en aplicaciones incrustadas o embebidas en tiempo real. Su principal objetivo es asegurar una respuesta oportuna y determinista a los eventos. Un evento puede ser una interrupción externa, como un interruptor de límite que se está golpeando, o interno como un carácter que se está recibiendo por transmisión serial. No es necesario utilizar un

RTOS para escribir un buen software embebido. Sin embargo, en algún momento, a medida que nuestra aplicación crece en tamaño o complejidad, los servicios de un RTOS podrían ser benéficos como es el caso que se expone en este trabajo.

El objetivo general de este trabajo es diseñar un robot móvil autónomo y en particular se muestran la implementación de un RTOS en un microcontrolador de 8 bits, para la detección de obstáculos. Se realizaron pruebas para medir el rendimiento del microcontrolador, elaborando tres rutinas que involucran interrupciones por timer y un proceso de cálculo, con el fin de generar latencia, siendo probadas de manera tradicional y con RTOS. Obteniendo buenos resultados se diseñó con RTOS una segunda prueba para implementar un sistema de percepción, usando sensores ultrasónicos e infrarrojos para realizar una navegación autónoma en un robot móvil con chasis tipo buggy escala 1:8.

2. Sistema de detección de obstáculos

Una tarea particular, como el control de un robot móvil, se puede separar en una serie de tareas más simples. Las tareas pueden incluir: la percepción del medio ambiente utilizando una cámara o escáner láser, la elaboración de mapas, planificar una ruta, monitorear el nivel de batería del robot y controlar los motores que impulsan las ruedas del robot. Cada una de estas acciones puede ser un código para realizar las tareas específicas. Se puede ejecutar código independientemente para realizar una tarea, pero también existe la posibilidad de comunicarse con otras tareas enviando o recibiendo mensajes. Los mensajes pueden consistir en datos o comandos u otra información necesaria para la aplicación. El sistema de detección propuesto utiliza **FreeRTOS** que es un popular núcleo del sistema operativo en tiempo real [3], para dispositivos embebidos que han sido portados a varios microcontroladores y se distribuye bajo la licencia GPL. Una de las características más importantes de un robot móvil es la navegación. Una navegación ideal significa que un robot puede planificar su trayectoria desde su posición actual hasta el destino y puede moverse sin ningún obstáculo. Estamos usando sensores ultrasónicos en este robot para encontrar la distancia a un objeto, y para obtener los datos de odometría del robot; también usamos sensores de proximidad IR (rayos infrarrojos) para detectar los obstáculos y evitar colisiones.

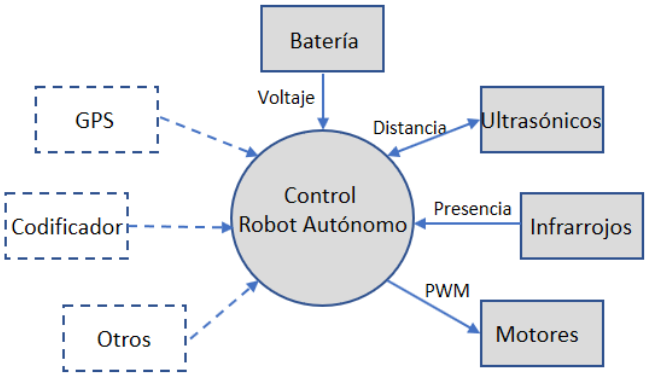

Figura 1. Diagrama de flujo de datos del sistema de control.

El primer paso del desarrollo de firmware es el diseño de la interacción entre componentes, que se puede observar en el diagrama de contexto de la figura 1, usando este diagrama se pueden observar las conexiones entre el programa de control y los otros componentes, así como la dirección del flujo de datos, los componentes en líneas punteadas serán considerados a futuro en el avance de este trabajo. Como se puede ver, la aplicación tiene que procesar el trabajo de muchos sensores, algunos necesitan comunicación bidireccional para hacer la inicialización, medición y lectura de datos; y en otros, sólo se trata de leer los valores. La siguiente tarea del control es activar los actuadores y la comunicación con un sistema superior en caso de ser requerida. Para este primer acercamiento al

objetivo general, se considera que el microcontrolador con FreeRTOS será el cliente, lo que significa que el sistema superior, es decir, el servidor tendrá la necesidad de enviar datos específicos o de configurar los actuadores y lanzar la aplicación en el microcontrolador, en otras palabras, el servidor deberá procesar los datos y enviar la respuesta si es necesario.

El siguiente diagrama, es considerado como el primer nivel de flujo de datos en la aplicación de control, y proporciona una mirada más cercana al objetivo específico de este trabajo (ver Figura. 2). Describe la división básica en las tres partes (procesos: Sensores, Actuadores y Comunicación con el servidor) y sus conexiones. Estos procesos estarán trabajando independientemente durante la mayor parte del tiempo. Por lo tanto, tiene que haber cuellos de botella entre ellos. Sin embargo, en este trabajo realizamos una prueba para determinar la eficiencia del micro de 8 bits propuesto usando la técnica llamada marco de programación jerárquica. La información de los sensores se guardó en un registro de memoria (Registro para sensores), que será compartida, y el proceso de comunicación con el servidor podrá leer estos datos. De igual forma se guardó en un registro de memoria para los codificadores y actuadores, logrando de esta forma triangular la información para un control más eficaz.

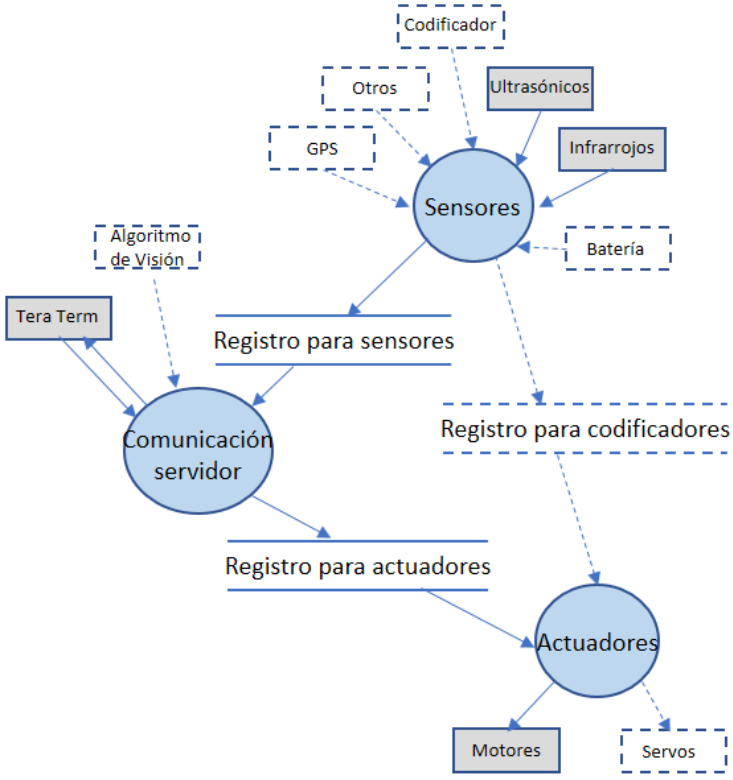

Figura 2. Diagrama de flujo del sistema de control.

2.1 *Sensores*

Los sensores ultrasónicos de distancia funcionan de la siguiente manera. El transmisor enviará un sonido ultrasónico que no es audible para los oídos humanos. Después de enviar una onda ultrasónica, se esperará un eco de la onda transmitida. Si no hay eco, significa no hay obstáculos delante del robot. Si el sensor receptor recibe un eco, se generará un pulso en el receptor, y se puede calcular el tiempo total de la onda, que se llevará al viajar del objeto y volver a los sensores del

receptor. Si conseguimos este tiempo, podemos calcular la distancia al obstáculo usando la siguiente fórmula:

$$Vel.delSonido * TiempPasado / 2 = DistObjeto \quad (1)$$

Aquí, la velocidad del sonido se puede tomar como 340 m/s. La mayoría de los sensores de rango ultrasónico tienen un rango de distancia de 2 cm a 400 cm. En este robot, utilizamos un módulo sensor llamado HC-SR04. La tensión de trabajo y la entrada/salida de este sensor ultrasónico es de 5V. El diagrama de tiempo de la forma de onda en cada pin se muestra en el diagrama de la figura 3. Necesitamos aplicar un pulso corto de 10 µs a la entrada del gatillo para iniciar el rango y luego el módulo enviará una ráfaga de ocho ciclos de ultrasonido a 40 KHz y elevará su eco. El eco es un objeto de distancia que es el ancho de pulso y el rango en proporción. Se calcular el rango a través del intervalo de tiempo entre enviar señales de disparo y recibir señales de eco utilizando la siguiente fórmula:

$$Rango = TiempoNivelAlto\_pinEcho * velocidad(340M / S) / 2 \quad (2)$$

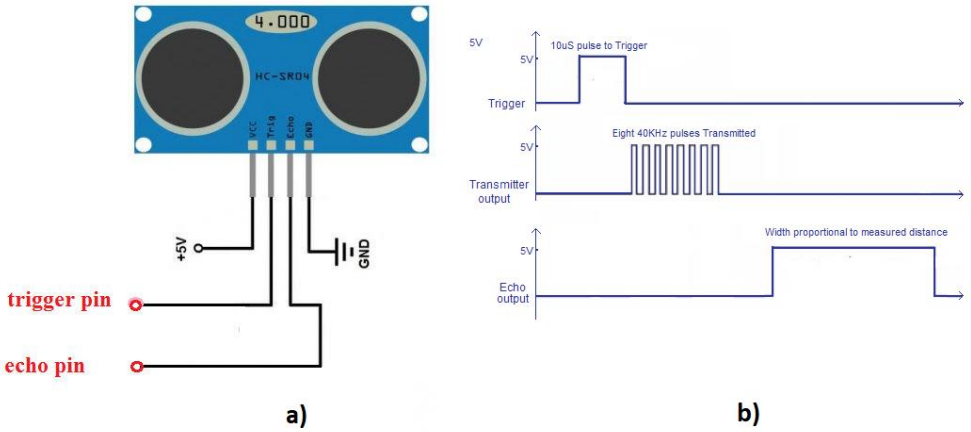

Figura 3. a) Configuración de pines para el HC-SR04. b) Diagrama de tiempos y forma de onda.

Se recomienda utilizar un retardo de 60 ms antes de cada disparo, para evitar la superposición entre el gatillo y el eco.

Los sensores infrarrojos son otro método para encontrar obstáculos, el principio de los sensores infrarrojos de distancia se basa en la luz infrarroja que se refleja desde una superficie al golpear un obstáculo. Un receptor IR capturará la luz reflejada y el voltaje se mide en base a la cantidad de luz recibida. El sensor envía un haz de luz IR y utiliza la triangulación para medir la distancia. El rango de detección del sensor GP2Y0A41SK0F utilizado para el diseño del robot es entre 4 a 30 cm. La transmisión y la reflexión de la luz IR se ilustra en la figura 4.

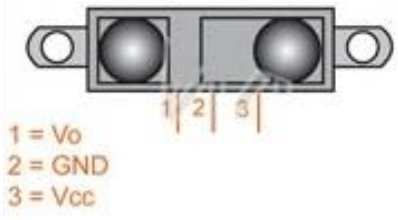

Figura 4. Sensor infrarrojo GP2Y0A41SK0F.

2.2 *Programación*

El sistema operativo en tiempo real FreeRTOS está diseñado para ser pequeño y simple y cada vez es más popular debido a que los proveedores de ambientes de desarrollo IDE con eclipse, agregan soporte de depuración dedicado para este sistema operativo. En este trabajo se utilizó CodeWarrior para MCU 10.7 en su versión espacial [4], de la compañía NXP (de Next eXPerience). Esta compañía fabrica semiconductores y fue creada en el 2006 a partir de la división de la empresa holandesa Phillips, pero actualmente hay una propuesta de compra de parte de la compañía Qualcomm's. Sin embargo, este sistema operativo puede ejecutarse en microcontroladores de varios fabricantes como: Altera, Atmel, NEC, Xilinx, Intel, entre otros.

Como ya se mencionó al principio de esta sección una de las principales razones por lo cual se desarrolló este trabajo es separar una serie de aplicaciones, relativamente simples de percepción a través de sensores ultrasónicos e infrarrojos, en un solo procesador. Esta técnica llamada marco de programación jerárquica (HSF por sus siglas en inglés Hierarchical Scheduling Framework) es una forma de integrar complejas aplicaciones en un solo procesador (ver figura 5).

Figura 5. Diagrama básico de división de tareas usando el marco de programación jerárquica.

La CPU es partida en varios subsistemas, cada subsistema contiene un conjunto de tareas que típicamente implementarían una aplicación o un conjunto de componentes. Cada tarea se asigna a un subsistema que contiene un planificador local para programar las tareas internas del subsistema. Cada subsistema puede utilizar una programación diferente y supervisada por una aplicación global. Para probar esta técnica y ver la eficiencia en nuestro microcontrolador de 8 bits, se programó un sistema modelo [5], en el cual tiene 3 tareas: LED 1 que se enciende y se apaga cada 100 ms, LED 2 que se enciende y apaga cada 500 ms y FIBONACCI que crea la serie Fibonnacci. Para realizar esta prueba se agregaron a CodeWarrior los componentes: FreeRTOS.PEupd, LED.PEupd y Utility.PEupd descargados de [6]. Para quienes no están familiarizados con CodeWarrior y Eclipse se puede consultar el Blog de E. Styger [7], que contiene mucha información de cómo implementar FreeRTOS en varias tarjetas de desarrollo.

La prueba para ver el desempeño del microcontrolador MC9S08JM60 de 8 bits, consistió básicamente en agregar los tres componentes que se muestran en la figura 6a, de tal manera que la ***tarea 1*** tiene como objetivo cambiar el estado del LED 1 cada 100 ms y la ***tarea 2*** cambiar el estado del LED 2 cada 500 ms (ver figura 6b). La cuestión interesante que evaluar con estas aplicaciones es que cada que se cumple el tiempo indicado para cada tarea, se envía una cadena de caracteres a través del puerto serial hacia una terminal del servidor, que para esta prueba consistió en una PC. Lo cual origina una latencia en el proceso. Y más aún la ***tarea* 3** genera la serie de Fibonacci, originando un gasto de recurso bastante fuerte para el microcontrolador (ver figura 7a).

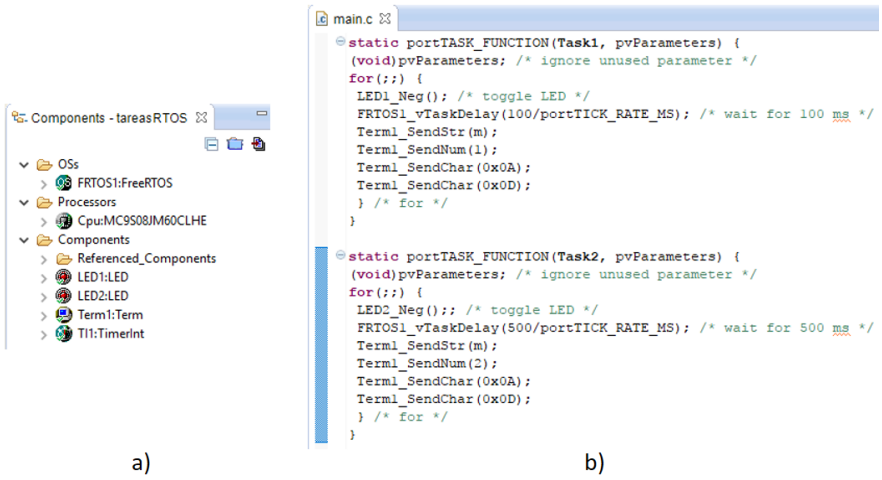

Figura 6. a) Componentes programados con CodeWarrior. b) Código con latencia para medir el rendimiento del microcontrolador MC9S08JM.

Se programó un código similar, usando la forma secuencial de programación en lenguaje C, con funciones e interrupciones. Tomando en ambos casos como frecuencia 60 Hz, la base de tiempo fue ***t****=1/f=16.66 ms,* de tal manera que cada vez que el ***Timer*** del microcontrolador llegaba al tiempo indicado se genera una interrupción por software (ver figura 7b). Esto nos permitió controlar la prueba y establecer un límite de 3 segundos como máximo, y contar del lado del servidor el número de veces que se ejecutó él envió de datos.

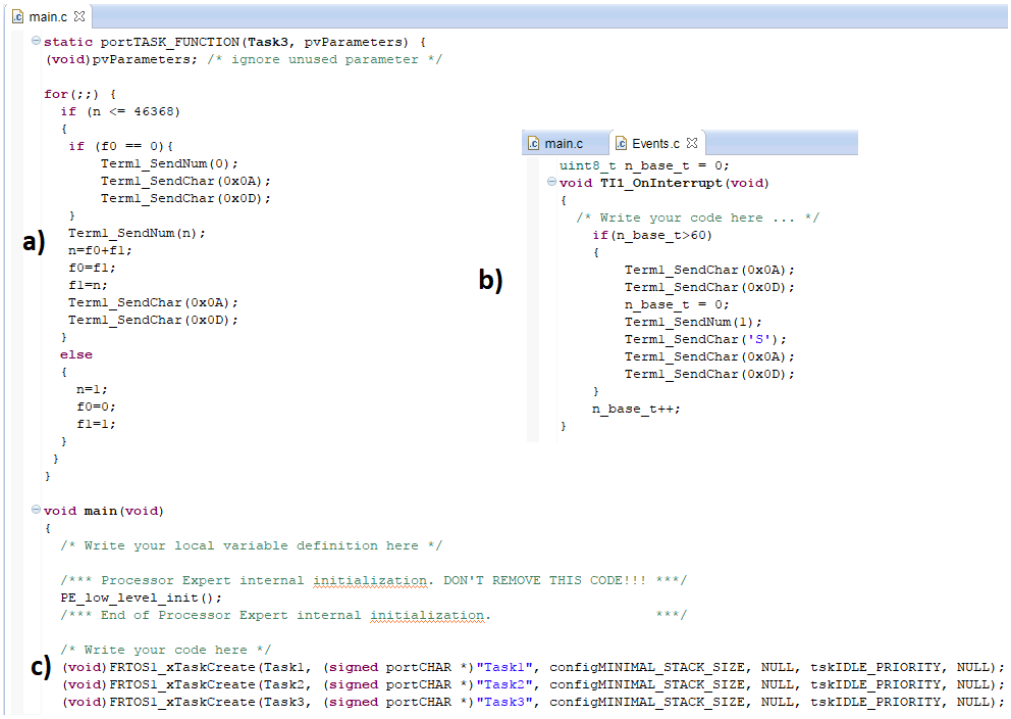

Figura 7. a) Código para generar consumo de recursos en él microcontrolador. b) Rutina de interrupción para monitorear él envió de datos. c) Definición de tareas en el programa principal.

3. Resultados

El sistema de detección de obstáculos es solo una parte del robot. La estructura de conexión entre capas se modificó para propósitos de prueba: el sistema superior fue reemplazado por un PC con ***Tera Term*** Pro versión 2.3, en Windows 10. Configurando un puerto virtual a 9600 baudios, 8 bits de datos, paridad non y un bit de paro.

Los sensores ultrasónicos e infrarrojos fueron montados en un chasis tipo buggy escala 1:8 (ver figura 8). El movimiento de avance y retroceso es proporcionado por dos motores DC conectados al circuito LB293 se utilizan dos baterías de 7.2V 3000 mAh NiMH y un regulador de voltaje LM7805. La dirección es manejada por un motor DC con un arreglo de engranes y se basa en la geometría de Ackerman. Los sensores utilizados ya fueron descritos en la sección 2.1 y el microcontrolador MC9S08JM60 de 8 bits fue seleccionado prácticamente por su portabilidad con la familia ColdFire MCF51JM128 de 32 bits para futuras ampliaciones [8].

3.1 *Primera prueba.*

Con el fin de comprobar si el microcontrolador seleccionado es capaz de soportar el sistema operativo FreeRTOS, se ejecutaron las tres tareas descritas en la sección anterior, para producir latencia a través de retardos, con la generación de la serie fibonacci hasta el número 46368. Y la transmisión de una cadena de caracteres cada 100 y 500 ms, durante 3 segundos, como se muestra en la figura 8.

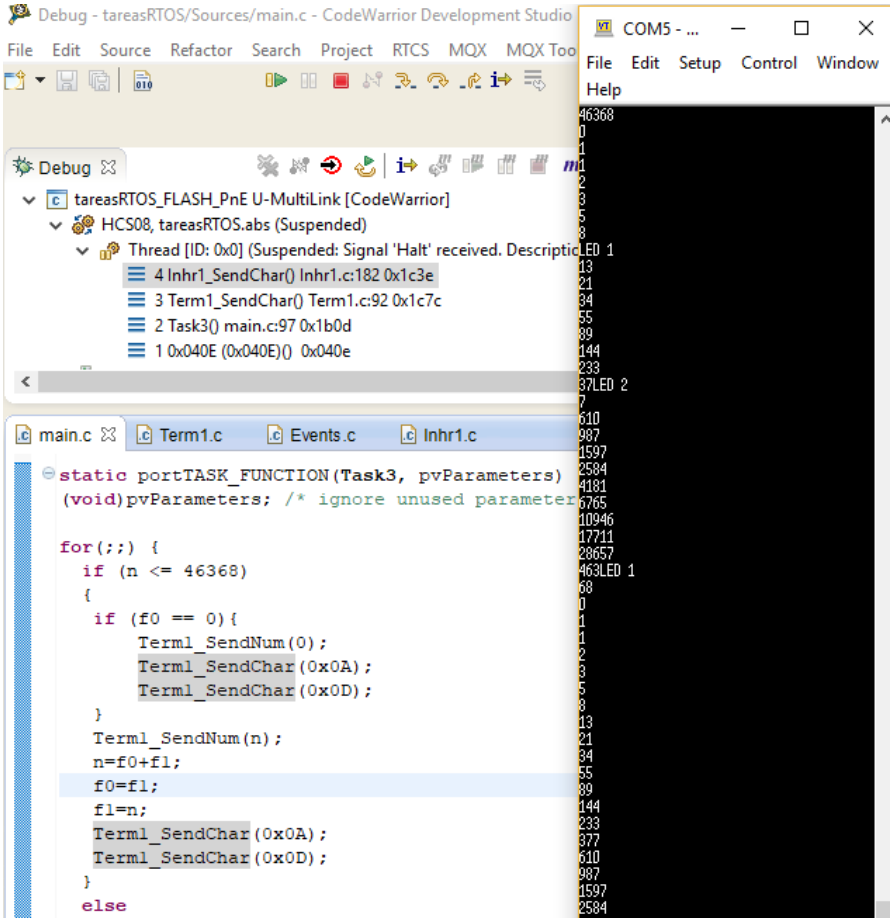

Figura 8. Resultados de la primera prueba ejecutando tres tareas al mismo tiempo.

Tabla 1. Prueba de latencia sin FreeRTOS.

MC9S08JM60 (Sin FreeRTOS)	LED 1	LED 2	# Fibonacci
1er segundo	10	2	160
2º segundo	1	0	176
3er segundo	0	0	168

Podemos observar en la tabla 1, que durante el primer segundo los resultados son como se esperaban, ya que las 10 veces que aparece la cadena de caracteres LED 1 concuerda con el tiempo de 10*100 ms = 1 segundo. De igual manera, la cadena de caracteres LED 2 concuerda con la duración de 2*500 ms = 1 segundo. Sin embargo, con forma aumenta el cálculo de la serie de Fibonacci la latencia se ve reflejada en los resultados de la tabla 1, desfasando él envió de las cadenas de caracteres.

Tabla 2. Prueba de latencia con FreeRTOS.

MC9S08JM60 (Con FreeRTOS)	LED 1	LED 2	# Fibonacci
1er segundo	10	2	194
2º segundo	10	3	187
3er segundo	10	3	187

Los resultados de la tabla 2, usando FreeRTOS en el microcontrolador son aceptables, observando que cumple prácticamente con él envió de las cadenas de caracteres en el tiempo asignado y calculando más números Fibonacci.

3.2 *Segunda prueba.*

Una vez realizada la prueba anterior se procedió a programar el sistema de detección de obstáculos como se muestra en la figura 9.

Recordando lo expuesto en la sección 2, la información de los sensores se guarda en uno o más registros de memoria, y para nuestro caso les llamamos: ***REGSensores***, ***REGReco*** y ***REGAuto***. que se encargan de almacenar en forma de bits, (0-sin obstáculo, 1-con obstáculo) la información de los sensores ultrasónicos: izquierda, derecha, central y reversa, así como infrarrojo y sus valores en x, y, z. como se muestra a continuación:

```
volatile static byte REGSensores=0x00;    //Sww Sxx Syy SIF SUr SUi SUc SUd
```

REGReco se usa para ejecutar a la acción proveniente del servidor: Alto, Derecha, Izquierda, Reversa, Avanza y Autónomo (ver figura 9a).

```
volatile static byte REGReco=0x00;        //www xxx yyy zzz CA1 CA0 MV1 MV0
```

De igual forma se asigna una codificación para el registro REGAuto, que se encarga de almacenar la información para accionar los actuadores (ver figura 9b).

```
volatile static byte REGAuto=0x00;        //AUw AUx MD1 MD0 MA1 MA0 AC1 AC0
```

Almacenando la información anterior en los registros de 8 bits, el servidor y las demás tareas pueden consultar el estado de los procesos para seleccionar la acción adecuada. Teniendo la seguridad que el sistema operativo FreeRTOS siempre le asignará un tiempo de procesamiento a cada tarea. También hay que utilizar eficiente y responsablemente el tiempo del microcontrolador, porque obviamente hay tareas que no requieren mucho tiempo de proceso ni tanta prioridad para completarlas y la desventaja que podemos observar con esta técnica es que una mala programación pueden producir estragos en el funcionamiento del sistema.

a)

```
*ProcessorExpert.c
  PTAD  = 0x00;
  PTADD = 0xFF; //Dir. PTA -> Output
  PTGD  = 0x00;
  PTGDD = 0x00; //Dir. PTG <- Input
  for(;;){
      if(!INPUT_1()) {
        //checa Dato del Servidor
          REGReco &= ~(1<<3);    //bit 0 a 0 --> Stop
      }
      else {
          REGReco |= (1<<3);     //bit 3 a 1 --> Auto
      }
      switch(REGReco & 0x0F){
          case 0  :
              accionAlto();
              tareas();
              break;
          case 1  :
          case 5  :
              accion60Derecha();
              tareas();
              break;
          case 2  :
          case 6  :
              accion60Izquierda();
              tareas();
              break;
          case 3  :
          case 7  :
              accionReversa();
              tareas();
              break;
          case 4  :
              accionAvanza();
              tareas();
              break;
          case 8  :
              autonomo();
              break;
      }
  }
```

b)

```
*ProcessorExpert.c
void tareas(void) {
byte Md,Ma,Accion;
byte VelH,VelL;

    Accion=REGAuto & 0x03;
    Ma=(REGAuto >> 2) & 0x03;
    Md=(REGAuto >> 4) & 0x03;
    switch(Accion){
        case 0  :
            VelH=0xCD;       // 262ms, Duty al 80% --> 210ms (52500)
            VelL=0x14;
            Avanza(Md,Ma,VelH,VelL);
            break;
        case 1  :
            VelH=0x7E;       // 262ms, Duty al 50% --> 130ms (32500)
            VelL=0xF4;
            Reversa(Md,VelH,VelL);
            break;
        case 2  :
            VelH=0x98;       // 262ms, Duty al 60% --> 156ms (39000)
            VelL=0x58;
            Avanza(Md,Ma,VelH,VelL);
            break;
        case 3  :
            VelH=0x65;       // 262ms, Duty al 40% --> 104ms (26000)
            VelL=0x90;
            Avanza(Md,Ma,VelH,VelL);
            break;
    }
}
```

c)

```
void Avanza(byte md,byte ma,byte velh,byte vell) {

    TPM2C0VH=velh;  //Velocidad
    TPM2C0VL=vell;
    PTAD_PTAD4=ma & 0x01;           //Motor-Traccion (Adelante)
    PTAD_PTAD5=(ma >> 1) & 0x01;
    LEDpin7_PutVal(md & 0x01);      //Motor Direccion (Izq.-Der.)
    LEDpin8_PutVal((md >> 1) & 0x01);
}
```

Figura 9. a) Tarea que selecciona la acción proveniente del servidor. b) Tarea que controla la velocidad de los actuadores. c) Tarea que mueve los motores según los argumentos enviados, hacia adelante o hacía atrás y con la velocidad indicada.

4. Conclusiones

Consideramos que la principal aportación de este trabajo ha sido demostrar que el uso del sistema operativo en tiempo real FreeRTOS tiene ventajas, frente al diseño clásico de funciones y se ha conseguido realizar un diseño de un sistema embebido para la detección de obstáculos, usando la técnica HSF para integrar diversas tareas en un solo procesador. El trabajo por realizar a corto plazo consistirá en desarrollar las tareas: GPS, codificador y nivel de batería. Así mismo, también se probará un microcontrolador de 32 bits con arquitectura ARM, con la posibilidad de utilizar las bibliotecas de visión. También creemos que la unidad robótica descrita se puede utilizar tanto en interiores como en exteriores debido a su construcción mecánica y sensores. Este robot se piensa utilizar para operar en pequeños grupos, para adquirir y distribuir datos de su entorno de tal forma que se pueda construir una plataforma de robots de enjambre.

5. Agradecimientos

Este trabajo es apoyado por el proyecto PAPIME PE109416 de la DGAPA-UNAM, “Taller de robótica para inteligencia de enjambres con código abierto” a quien se extiende el presente agradecimiento.

Referencias

[1] Zhou Jianjun, zhou Jianhong, “Research on Embedded Digital Image Recognition System Based on ARM-DSP” 2009 2nd IEEE International Conference on Computer Science and Information Technology.

[2] *“Move Innovation”*, [en línea]. [Fecha de consulta: 28 agosto 2017]. Disponible en: https://www.yoctoproject.org/organization/move-innovation

[3] *“What is An RTOS?”,* [en línea]. [Fecha de consulta: 06 septiembre 2017]. Disponible en: http://www.freertos.org/about-RTOS.html

[4] *“Special Edition Software”* [en línea]. [Fecha de consulta: 07 septiembre 2017]. Disponible en: https://www.nxp.com/products/developer-resources/software-development-tools/codewarrior-development-tools/downloads/special-edition-software:CW_SPECIALEDITIONS

[5] Rafia Inam, Jukka Maki-Turja, Mikael Sjodin, S. M. H. Ashjaei, Sara Afshar, “Support for Hierarchical Scheduling in FreeRTOS”, *ETFA2011 Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. :1-10 Sep, 2011

[6] *“FreeRTOS”*, [en línea]. [Fecha de consulta: 28 agosto 2017]. Disponible en: http://www.steinerberg.com/EmbeddedComponents/FreeRTOS/

[7] *“Tutorial: FreeRTOS on DEMOJM”*, [en línea]. [Fecha de consulta: 04 agosto 2017]. Disponible en: https://mcuoneclipse.com/2012/06/28/tutorial-freertos-on-demojm/

[8] Arturo Ocampo; Oscar M. Espinoza, “Diseño de sistemas mecatrónicos escalables usando flexis de freescale”, 1er Congreso Iberoamericano de Instrumentación y Ciencias Aplicadas. SOMI XXVIII. Sn Francisco Campeche, Campeche. México 2013.