Estrategias de Diseño de Software para Sistemas Embebidos

Gustavo Monte†, Damian Marasco†, Pablo Liscosvky†, Hector Kessel†

Universidad Tecnológica Nacional
Unidad Académica Confluencia
P. Rotter s/n Campamento 1, Plaza Huincul, Neuquén.
gusmonte25@yahoo.com.ar

†Grupo **IDESIC UTN** Regional Académica Confluencia

***Resumen*— Cuando nos enfrentamos a un diseño de un sistema de mediana o alta complejidad basado en microcontroladores no alcanza con conocer perfectamente al microcontrolador y su juego de instrucciones, o de disponer de un compilador apropiado. Se debe sistematizar el desarrollo, sobre todo del software, para lograr un esquema estructurado, fácil de entender, y de modificar.**

El diseño del software es una de las actividades más creativas que enfrenta el desarrollador de sistemas embebidos. El diseñador logra profundidad cuando focaliza la energía sobre un aspecto perfectamente acotado de un problema. Esta creatividad debe enfocarse en los algoritmos que sintetizan al sistema y no a como estos algoritmos juntos pueden resolver un sistema.

Los sistemas embebidos controlan naturalmente múltiples eventos concurrentes. Es imperioso soportar el diseño sobre una estructura multitarea. Es prácticamente imposible realizar un diseño complejo sin el soporte de una estructura preconcebida. En el presente trabajo se plantea otra opción a la de soportar el desarrollo sobre un sistema operativo comercial. La propuesta es el diseño sobre una plataforma multitarea estado-cooperativa, en donde el sistema complejo es solamente la suma de subsistemas simples. Se desarrolla completamente un esquema de trabajo que presenta numerosas ventajas con respecto a planteos tradicionales que van desde una generación de auto documentación gráfica, hasta las técnicas de depuración de programas.

***Palabras Clave*— autodiagnóstico, desarrollo de software embebido, multitarea, cooperativo, detección de errores.**

I. INTRODUCCIÓN

En la actualidad existe una demanda creciente de sistemas embebidos cada vez más complejos. Los elementos de hardware, sobre todo los microcontroladores, son cada vez más económicos y completos, proporcionando la solución en un chip. Esta notable reducción de costos ha impulsado al diseño del software del sistema a ser el factor crítico en donde las herramientas de diseño juegan un papel preponderante.

(Marian, Guo, 2006). Por lo tanto destinar esfuerzos a lograr módulos de software que puedan reutilizarse de manera transparente es la clave del éxito en un diseño moderno basado en microcontroladores.

Es decir, ante un desarrollo nuevo debemos soportar nuestro pensamiento sobre una estructura que permita concentrar nuestra energía mental en un aspecto único y limitado del diseño. El ideal de esta estructura es cuando se permite desglosar el desarrollo en partes elementales ***sin pagar un alto precio por unir estas piezas para sintetizar el diseño total.***

Ante un desarrollo complejo como por ejemplo un sistema de tiempo real critico, si no se parte de una estructura de soporte, el resultado final será, en el mejor de los casos, un programa que cumple con los objetivos planteados pero tendrá las siguientes características:

PROBLEMAS MAS COMUNES QUE ENCONTRAREMOS SI NO PARTIMOS DE UNA ESTRUCTURA DE DESARROLLO	
PROBLEMA	RAZON
Difícil de modificar	Si se modifica es muy probable que deje de funcionar
Difícil de entender	No hay una estructura de soporte, un orden preestablecido
Difícil de diseñarlo para seguridad ante fallos	No existe un control global de ejecucuón
Difícil de depurar	No se encuentran fácilmente subsistemas para acotar fallas

Tabla 1. Principales problemas si se realiza un diseño sin una estructura de soporte.

Recordemos un concepto fundamental en el diseño de software: cuando el software de un sistema embebido no se modifica significa que ha dejado de usarse. En otras palabras, realizaremos cambios, actualizaciones, mejoras, depuraciones siempre durante la vida útil del

sistema. Por lo tanto el software de nuestro dispositivo debe estar preparado para soportar esos continuos cambios. Más aún, el diseño de sistemas embebidos complejos estará a cargo de un equipo, lo que hace imperioso utilizar una estructura particionable. Otra gran verdad del desarrollo de software embebido es que los tiempos dedicados a la depuración son órdenes de magnitud superiores a los tiempos de diseño. En el desarrollo de software embebido nos encontramos con más fuentes de error que en el desarrollo de software puro como ser el desarrollo de una pagina WEB. La razón de este incremento es la interacción con el hardware. Las fuentes de error más importantes en el desarrollo de sistemas embebidos son:

⇒ Errores de hardware

⇒ Errores de diseño de software

⇒ Errores de interacción software-hardware

⇒ Errores de tipeo de software

Los errores de hardware los podríamos clasificar en persistentes y esporádicos. Los persistentes son fácilmente detectables e incluyen por ejemplo, pistas cortadas o intercambiadas, componentes fallados, etc. Los errores esporádicos poseen una probabilidad de ocurrencia aleatoria o sistemática por lo tanto son más difíciles de corregir. Dentro de este tipo de error podríamos citar entre los más importantes; fallas por temperatura, humedad, vibraciones, falsos contactos, latch up e interacción electromagnética.

Los errores de diseño de software se producen cuando existe un error en la concepción del algoritmo que subyace en software. Los errores de interacción software-hardware son los más difíciles de detectar y corregir. Estos errores ocurren cuando resulta una simultaneidad o secuencia especifica de eventos que provocan que el binomio software-hardware falle, sin existir fallas en el software ni en el hardware. Como ejemplo podríamos citar casos como la interacción de una interrupción con una recepción de datos seriales o la aparición simultanea de señales que no son correctamente procesadas. Por último los errores de tipeo son generalmente detectables pero ocurren dentro de las primeras fases del desarrollo cuando hay desconfianza de todos los elementos del sistema, incluyendo a los diseñadores. Ejemplo típico de este error es escribir CONT1 en lugar de CONT10 y las dos son variables declaradas.

Con todos estos errores acechando, es imperioso trabajar sobre una estructura de diseño. Una posible solución es soportar nuestro desarrollo sobre una estructura multitarea comercial (Moore, 2001). En este caso se delega el control de ejecución de las tareas en una estructura que no conocemos exactamente, pero que nos garantiza que nuestras tareas llegaran a buen término en concordancia con su prioridad de ejecución. Existe una gran diversidad de sistemas operativos; de propósitos generales, para tiempo real y para tareas críticas.

A continuación desarrollamos la propuesta del presente trabajo comenzando por las estructuras de desarrollo basadas en esquemas de multitarea.

II. ESTRUCTURAS EN MULTITAREA

A. Necesidad de programación multitarea

¿Por qué programar con estructura multitarea? Porque *naturalmente* nuestro programa se encuentra compuesto de distintas tareas que compiten en forma concurrente para consumir recursos de la CPU.

Multitarea se refiere a la forma de escribir un programa de manera tal que atienda la evolución de eventos simultáneos siguiendo un esquema de prioridades dinámicas o estáticas. Las prioridades estáticas son fijadas off-line mientras que las dinámicas son seleccionadas en tiempo de ejecución.

El objetivo final de nuestro diseño lo podemos sintetizar con la siguiente frase: ***"Garantizar que cada tarea cumpla con sus objetivos".*** En general las tareas competirán por los recursos de CPU y de los periféricos. Por lo tanto se deberán administrar los recursos de hardware.

Las características deseables de un programa entre las más importantes son:

⇒ Desarrollo sistemático.

⇒ Adaptable a trabajar en equipo.

⇒ Módulos de software reutilizables.

⇒ Depuración simplificada.

⇒ Actualización y modificación sencilla.

⇒ Auto-documentado.

⇒ Independencia del hardware.

⇒ Independencia del lenguaje de programación.

⇒ Capacidad de autodiagnóstico.

El esquema tradicional de estructuras multitarea es el siguiente:

a) Diseñar las tareas como si se dispusiera de todos los recursos del microcontrolador.

b) El kernel o núcleo íntimo del sistema operativo se encargará de asignarle el tiempo de CPU adecuado según su prioridad y de garantizar la evolución exitosa de todas las tareas.

En este esquema de trabajo, el kernel del sistema operativo multitarea se encarga de superar los conflictos de hardware y de asignar los tiempos de CPU según si es de la filosofía *time slice* u orientado a eventos entre las más comunes.

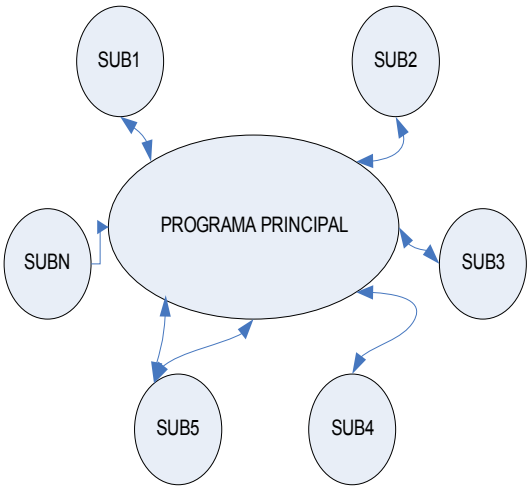

Fig. 1. Esquema tradicional: Un programa principal llama a distintas subrutinas. Él le da "vida" al sistema.

B. Multitarea estado-cooperativo

La propuesta es la de realizar uno mismo su propio sistema operativo. Un esquema apropiado para microcontroladores en sistemas embebidos, y la propuesta del presente trabajo, es la técnica de programación basada en una variante de multitarea cooperativo llamado estado-cooperativo. En este esquema ***cada tarea es una porción del sistema operativo*** y cada una de ellas administra su tiempo de CPU. Las propias tareas recuerdan su estado de evolución convirtiéndose en unidades de software independientes sin necesidad de intervención o control externo. El compromiso de diseño es el de emplear el tiempo de CPU en una fracción mínima e indispensable. No existe un programa principal que resuelva los algoritmos. El dialogo entre las tareas, a través de la habilitación de sus bits de estado, es la forma en la cual se sintetiza el diseño.

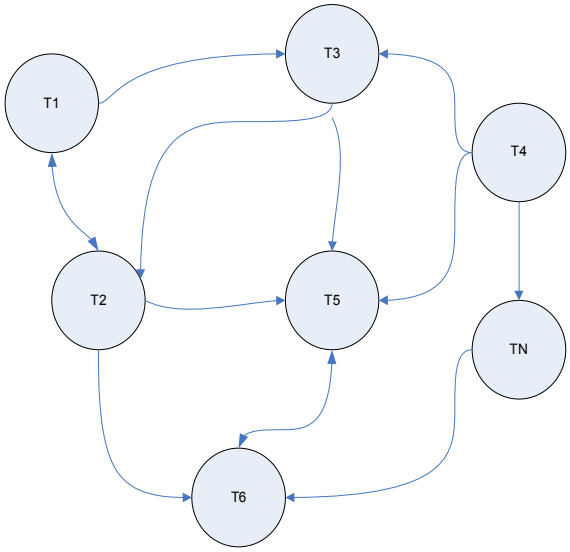

Fig. 2: Multitarea estado-cooperativo: el algoritmo reside en la comunicación de las distintas tareas. No existe un programa principal que controla.

Es conveniente comenzar por definir que significa una tarea en el contexto de multitarea cooperativo.

> Una tarea en un ambiente cooperativo es ***"una secuencia de instrucciones con una función especifica que esta caracterizada en cada instante por un estado de ejecución (STATUS)."***

La diferencia entre una tarea y una subrutina es que en la tarea se conoce el estado de evolución interno en todo momento. El status de una tarea captura el aspecto dinámico de la ejecución del programa a través del cual se determina exactamente su estado de evolución.

En el esquema más simple, el secuenciador principal (scheduler) es simplemente las llamadas a las tareas. No existe un algoritmo principal, la tarea necesita únicamente que sea ejecutada regularmente. Las tareas *se auto-organizan* y la comunicación entre tareas se realiza por bits de estados.

Existen variables locales solo disponibles en las tareas y variables globales. La administración del tiempo se realiza por interrupciones periódicas con contadores de resolución adecuada. Como el algoritmo principal reside en la comunicación entre tareas a través de los bits de estado, es posible controlar la ejecución de las tareas en forma muy sencilla. Es decir que las tareas son *observables* mediante sus bits de estado. Este aspecto es muy importante ya que el software desarrollado es totalmente transparente desde el exterior, lo que permite monitorear la evolución de las

tareas y controlar el normal desempeño de ellas. Esta transparencia permite, por ejemplo, que se diseñen tareas que controlen tareas, Logrando la capacidad de autodiagnóstico de manera muy directa.

El programa puede desarrollarse completamente en assembler o en algún lenguaje de alto nivel. El lenguaje C es el predilecto por su eficiencia en la compilación. No es necesario el uso de compiladores especiales que emplean costate o instrucciones especiales para multitarea.

Los pasos a seguir para desarrollar un programa con la filosofía multitarea cooperativo, en el esquema más simple, son los siguientes:

1. Identificar las tareas. Por ejemplo tarea comunicación, tarea CRC etc.
2. Identificar los estados de las tareas.
3. Asociar un bit a cada estado de la tarea. Un "1" implica estado en ejecución.
4. Escribir el código para cada estado de la tarea. Se prohíben los lazos ya que la tarea tiene el compromiso de devolver el control al scheduler en un tiempo breve. Si se necesitan acciones repetitivas se deberán incluir más estados.
5. Si un código asociado a un estado es excesivo, dividirlo en más estados.
6. Incluir la llamada de la tarea en el scheduler.

Con respecto al punto 4, los lazos deben ser evitados dentro de un mismo estado. Si se requiriese una acción repetitiva se deberá realizar empleando más de un estado. Por ejemplo, si tenemos que generar **N** pulsos rectangulares empleamos un estado para generar el nivel alto y controlando el tiempo con los contadores de la interrupción. Otro estado controla el tiempo de nivel bajo. Luego la tarea evoluciona hacia un tercer estado en donde se controla la cantidad de ciclos. Si no se ha alcanzado el límite N, se sitúa a la tarea, mediante la manipulación de los bits de estado, para que vuelva a generar el nivel alto y así sucesivamente. Lo que ocurre es que en cada estado se ejecutan 4 o 5 sentencias en promedio, por lo tanto la evolución de esta tarea es casi independiente del resto de las tareas.

El programa completo se diseña con una filosofía TOP-Down mientras que el desarrollo de las tareas se realiza en forma inversa. Es decir, primero identificamos las tareas sin resolverlas y comenzamos por resolver los estados de las tareas en un código visual que se describe en la sección siguiente. Dentro de las tareas trabajamos de lo particular hacia lo general. Ocho estados, un byte, para describir una tarea es más que suficiente. Si una tarea necesita más estados es conveniente desglosar los estados en más de una tarea.

Nuestro sistema se diseminó en tareas y además dentro de las tareas se identificaron estados operativos asociados a bits de status y las tareas no interfieren unas con otras. Por lo tanto se concluye que, sin importar de la complejidad del diseño, nuestra energía mental esta focalizada en un aspecto único y separable del resto. Lo que permite una alta eficiencia en la codificación, la generación de código reutilizable y la facilidad de trabajar en equipo.

B1. Auto documentación

La documentación de los programas se facilita con el esquema organizado en multitarea estado-cooperativo. Se propone una forma de documentarlo independiente del código que resulta muy descriptiva que se observa en la Fig. 3. En ella se describe en forma visual el código asociado a cada estado de la tarea. La auto documentación se logra ya que la descripción visual es traducida fácilmente a código en forma directa. A modo de ejemplo se observa una descripción de los dos primeros estados de la tarea de recepción de datos en pseudo código visual y el código equivalente en lenguaje C y en assembler de Microchip. En esta tarea, en el bit cero, espera la llegada del carácter "MI_ID" y en el bit 1 la llegada del carácter "ID_CPU". La ocurrencia de estos dos caracteres permite evolucionar la tarea hacia el bit 2.

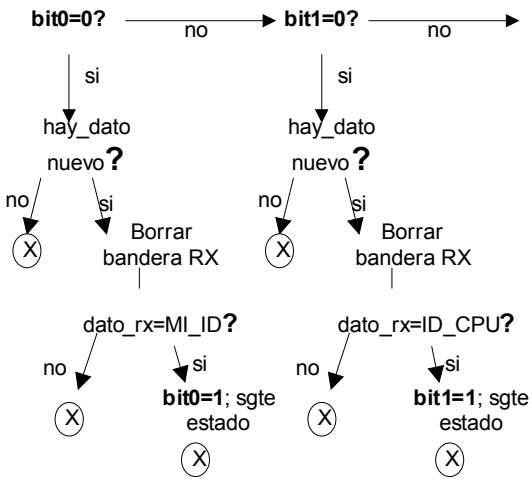

Fig. 3: pseudo código visual de los dos primeros estados de una tarea de recepción de datos.

```
tarea_comunicacion
        btfss
        st_comunicacion,0
        goto bit_0
        btfss
        st_comunicacion,1
        goto bit_1

bit_0   btfss hay_dato,,0
        return
        bcf hay_dato,0   ; borro bandera RX
        movlw MI_ID
        xorwf  dato_rx
        btfss STATUS,Z
        return
        bsf st_comunicacion,0
        return

bit_1   btfss hay_dato,,1
        return
        bcf hay_dato,0   ; borro bandera RX
        movlw ID_CPU
        xorwf  dato_rx
        btfss STATUS,Z
        return
        bsf st_comunicacion,1
        return
```

```
void tarea_comunicacion()
    if (!bit_test(st_comunicacion,0))  // bit0=0?
     {
       if (bit_test(hay_dato,0)
        return ;
        bit_clear(hay_dato,0);
        if (dato_rx==MI_ID)
        {
            bit_set(st_comunicacion,0);
          return ;
        }
        else
            return;

     }

if (!bit_test(st_comunicacion,1))  //bit1=0?
 {
   if (bit_test(hay_dato,0)
    return ;
    bit_clear(hay_dato,0);
    if (dato_rx==ID_CPU)
    {
        bit_set(st_comunicacion,1);
       return ;
    }
    else
        return;

 }
```

Fig. 4: La porción de pseudo-código visual presentada en la Fig. 3, traducido a assembler de Microchip, izquierda y en lenguaje C, derecha.

B2. Administración del tiempo

El manejo del tiempo en el esquema propuesto es llevado en una interrupción periódica donde solamente se permite incrementar contadores de resolución de milisegundos, segundos y minutos. Debe tratarse de evitar la llamadas a tareas dentro de la interrupción periódica por más que se deban ejecutar cada un tiempo prefijado. Es conveniente asociarle un evento y que se ejecute cuando le corresponda en el scheduler. Por ejemplo, si una tarea muestreo debe ejecutarse cada 10 milisegundos, se realiza de la siguiente forma: en el estado cero de la tarea borramos el contador de milisegundos que se incrementa en la interrupción. En el estado 1 se compara el contador con 10 y cuando se detecta la igualdad se avanza el estado 3 en donde se comienza a ejecutar la tarea de muestreo. Una vez concluida la tarea se borra el status volviendo al estado cero. Es importante destacar que la tarea muestreo no interfiere con otras tareas ya que necesita 3 ó 4 instrucciones para ubicarse en el estado correspondiente y realizar una comparación. Tampoco tiene importancia el orden en la que la tarea es llamada solo necesita ser ejecutada.

III. TIPOS DE SECUENCIADORES EN MULTITAREA ESTADO-COOPERATIVO

A. Secuenciador lineal

Es el esquema más simple, el scheduler es simplemente las llamadas a las tareas como se observa en la Fig. 5. No hay un esquema de prioridades. Las tareas que requieran tiempo real se pueden incluir N veces en el lazo principal o incluirla en una interrupción disparada por eventos. Esta organización incluye la interrupción periódica que se describió en el párrafo anterior.

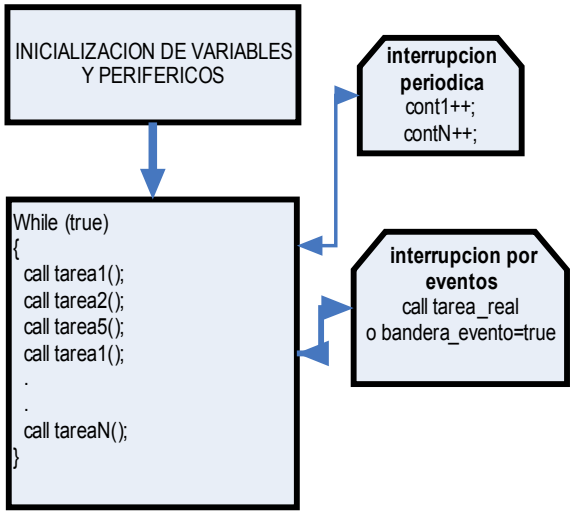

Fig. 5: Secuenciador lineal con tareas auto-organizadas.

Este esquema organizativo tiene la belleza de la simpleza y es muy sencillo de depurar incluyendo o no tareas en la fase de prueba. En base a resultados experimentales, hasta aproximadamente 10 ó 20 tareas el secuenciador es recomendable. Para mayor cantidad de tareas, posee el inconveniente que se realizan llamadas innecesarias a tareas inactivas aumentando el tiempo de latencia.

B. Secuenciador de macroestados

Cuando el número de tareas se incrementa, del orden de las decenas, el secuenciador lineal no es eficiente en el sentido que una gran cantidad de tareas son llamadas innecesariamente. Además la auto-organización de las tareas hace difícil la tarea de control de la ejecución. En este contexto se propone agrupar las tareas identificando macroestados operativos del sistema y dentro de cada macroestado organizar las tareas bajo un secuenciador lineal. Si bien las tareas son

las mismas que en el caso del secuenciador lineal, éstas se encuentran organizadas por función. Se logra una organización más eficiente al recorrer los secuenciadores lineales de macroestados activos.

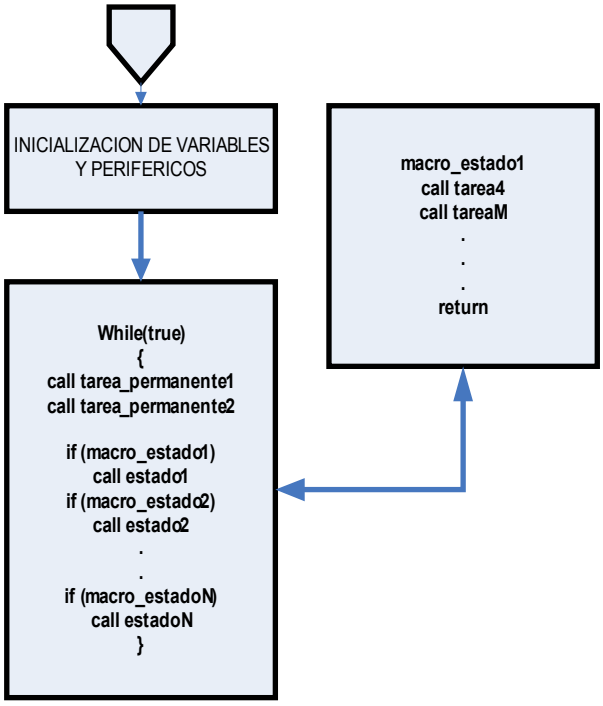

Fig. 6: Secuenciador de macro-estados organizados en secuenciadores lineales.

Este esquema también posee una estructura muy simple. Obsérvese en la Fig. 6 que hay tareas que son siempre ejecutadas como por ejemplo tarea_permanente1 y otras tareas se ejecutan solo dentro de macroestados activos. Una tarea puede pertenecer a unos o más macroestados. Esta estructura presenta grandes ventajas en la fase de la depuración y control de ejecución de los programas. En la etapa de la depuración los errores se presentarán identificados en estados operativos. Por ejemplo el sistema es inestable en el estado de control manual o en el estado de adquisición de señales. Rápidamente se visualizan cuales son las tareas involucradas en esas fases. Recordemos que los tiempos de depuración son generalmente de órdenes de magnitud superior a los de diseño del software. Además, este esquema organizativo es más simple de modificar o actualizar que el secuenciador lineal.

C. Confiabilidad del diseño

Diversos aspectos hacen que los diseños de multitarea estado-cooperativo sean confiables. Primero, la instrucción del watchdog es una sola y se encontrara solo en el scheduler. Esto es posible porque no existe ningún lazo fuera del lazo del scheduler. Ante un reset inesperado, es posible analizar los estados de las tareas y decidir acciones. Segundo, la conmutación de tareas (context switching) ocurre naturalmente y no es asincrónica con respecto a la ejecución de tareas (Curtis, 2006). Por lo tanto no existe la posibilidad de dejar operaciones multibytes u operaciones complejas truncas que generar resultados nos deseados en variables y tareas.

Tercero, es posible diseñar una tarea que controle a un grupo de tareas. Mediante simples operaciones lógicas AND y OR es posible controlar la ejecución de tareas no deseadas. Por ejemplo, si la tarea N no puede estar activada junto con la tarea M, con la simple AND de los bits de estado de las tareas obtengo la información de infracción. También se puede controlar la normal evolución de los bits de estado de las tareas buscando anormalidades, como por ejemplo que la secuencia de bits no es la correcta.

IV. DISEÑO ASISTIDO

La complejidad de los sistemas embebidos actuales requieren de herramientas de diseño y simulación (Belarbi, 2001).

La propia estructura auto-organizativa del multitarea estado-cooperativo posibilita automatizar el diseño del software. Es simple diseñar un programa que forma gráfica y mediante preguntas clave entregue como resultado el programa traducido al lenguaje C. Este programa facilita el diseño del software así como también la documentación y la depuración.

A. Pantalla principal

El programa se comienza con la acción de **nuevo proyecto**, luego nos pregunta el nombre y que tipo de secuenciador emplearemos: **lineal o de macroestados**. A continuación comenzamos por crear **nueva tarea**, se especifica un **nombre** y se comienza a describir las acciones en cada **estado**. En la Fig. 7 se observa la secuencia de eventos de la pantalla principal. Los nombres de las variables críticas están normalizados. Además del tipo de variable, de 1, 8, 16 o 32 bits, float etc., las variables poseen otra clasificación relacionada con la auto-organización. Se clasifican como de status de las tareas comenzado con el prefijo **"ST_"**, como globales con **"G_"**, como globales de solo lectura **"GR_"** y como variables locales **L_**. Las variables de conteo se denominan a las variables que se incrementan en la interrupción periódica y comienzan con **TMS_, TS_,** y **"TMIN"** a variables que se incrementan automáticamente cada milisegundo, segundo y minuto respectivamente. Las variables de conteo son particulares a cada tarea y se declaran como locales siempre.

B. Diseño de las tareas

Las tareas se definen por las acciones dentro de cada estado. Recordemos que el compromiso de diseño impide la realización de bucles o lazos. Por lo tanto las acciones en los estados corresponden a asignaciones, operaciones aritméticas o algebraicas e instrucciones de comparación o bifurcación.

La descripción de las acciones dentro de cada estado se realizan dentro de TextBox y si la sentencia termina con "?" se abren dos TextBox uno para verdadero y otro para falso. Recordemos que las tareas son las células básicas de la estructura multitarea, por lo tanto se debe permitir la importación de tareas proveniente de otros proyectos. En el proceso de importación se debe controlar y declarar las variables en uso por dicha tarea, tanto locales como globales.

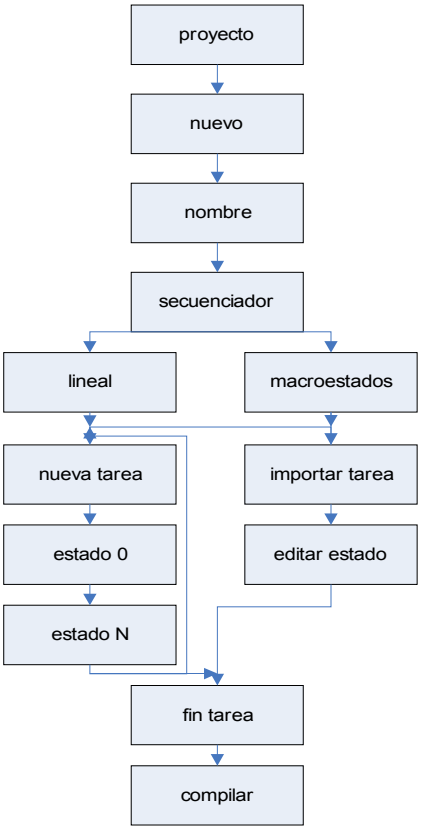

Fig. 7: diagrama de eventos en el diseño asistido.

Una de las virtudes del diseño asistido es que la representación gráfica de las tareas junto con la declaración de las variables y los comentarios de cada estado de las tareas conforma la documentación del proyecto, cumpliendo con el requisito de la auto-documentación.

V. RESULTADOS EXPERIMENTALES

El método propuesto de multitarea estado-cooperativo lo hemos empleado por más de 7 años en diversas aplicaciones. La más compleja fue el control y registro de temperatura en red de 200 tanques de fermentación, con programación local y remota. Se han ensayado programas con hasta 40 tareas con scheduler de macroestados. Las tareas, una vez diseñadas, han sido reutilizadas en otros diseños de manera transparente o modificando la interacción con las nuevas tareas.

Una riqueza adicional del esquema propuesto es la posibilidad de observar y modificar la ejecución de tareas remotamente. Si en el programa se incluye una tarea que mediante, por ejemplo RS232, cumpla la función de leer y escribir en la memoria RAM del microcontrolador, es posible mediante un sencillo programa con una interfase **HMI** adecuada, observar depurar y controlar tareas mediante la lectura de las variables de estado se observa y se controla la ejecución de las tareas. Se obtiene de esta manera un programa de depuración interactuando con el circuito real.

Aunque el método propuesto es independiente del microcontrolador, se ha trabajado principalmente con microcontroladores Microchip 18FX52 y compilador C de la compañía CCS.

Una gran cantidad de alumnos de Técnicas Digitales II y III de la Regional Académica Confluencia, han realizado sus proyectos mediante el esquema propuesto, resultando de gran aceptación por ellos.

VI. CONCLUSIONES

Se ha presentado un esquema organizativo de diseño de software para sistemas embebidos que presenta innumerables ventajas. El sistema propone una plataforma de desarrollo confiable y sistemática, tan sistemática que es relativamente simple diseñar un software que escriba las tareas.

La organización propuesta sobre todo la del secuenciador de macroestados es una opción eficaz para sistemas embebidos complejos. Lo más valioso del diseño es la documentación gráfica que resulta ser independiente de microcontrolador y del lenguaje de desarrollo.

Cada tarea es una porción del sistema operativo, éste se encuentra disperso entre las tareas. Esta dispersión requiere secuenciadores más inteligentes a medida que aumenta la complejidad del diseño.

Se logra que un sistema complejo sea la suma de subsistemas simples y a la vez estos subsistemas se encuentran divididos en estados.

Ante el desafío del diseño de un sistema complejo basado en microcontroladores tenemos dos caminos, emplear un sistema operativo comercial o diseñar nuestro propio sistema operativo multitarea. Si optamos por el segundo camino, la propuesta planteada es una opción válida, confiable y con un control total del desarrollo.

La riqueza didáctica es sorprendente. El diseñador llega a entender completamente a un sistema ya que no hay código oculto y el diseñador o estudiante es el creador del sistema completo.

REFERENCIAS

Belarbi, M. Jacques, R, "Simulation and Verification of embedded Real Time Multitasking Applications", *IEEE Real Time Embedded System Workshop*, 2001.

Curtis, K. "Embedded Multitasking with Small Microcontrollers: Part 2" *Embedded Systems,* diciembre 26, 2006.

Marian, N. y Guo, Y "Model Based Design of Embedded Software" *REM 2006*, Stockholm, Sweden ,2006.

Moore, R "How to Use Real Time Multitasking Kernels in Embedded Systems" Micro Digital Associates, Inc, 2001.

Peatman, J. "Design with Microcontrollers", Mc Graw Hill, New York, 1988.