El Kernel

Araguaney es un kernel capaz de ofrecer multitarea basada en prioridades de hasta 8 niveles de prioridad, elaborado para el PIC16F877A, el kernel busca que una aplicación sea más intuitiva al programador, al dividirla en pequeñas tareas individuales que puedan depurarse con más facilidad, aprovechar mejor el procesamiento del PIC (por ejemplo, eliminando de la estructura las demoras para temporizar), aprovechando dichas demoras para ejecutar otras tareas, mejorando así los tiempos de respuesta de la aplicación. El núcleo (kernel) además de permitir la multitarea cooperativa, ofrece la flexibilidad de que las prioridades de las tareas sean dinámicas; hace poco uso de la pila, posee varios servicios de sistema (orientados a la temporización casi en su totalidad), temporizadores individuales reciclables, estos permiten que puedan ser compartidos entre tareas si se desea o si no se dispone de mucha memoria RAM.

Fue escrito para un PIC16F877A en lenguaje ensamblador bajo MPASM y usa las características de este ensamblador este sistema, fue diseñado para que su código esté en la posibilidad de ser portado a los microcontroladores PIC de 14bits de menores prestaciones que éste, siempre y cuando posean memoria suficiente para albergar al kernel, los temporizadores y las tareas. Está bajo los términos de software de código abierto, pudiéndose modificar y mejorar y colocarse a la disponibilidad de todos. El software está pensado para que sea usado académicamente en universidades e institutos como material didáctico en el ámbito de la multiprogramación y sus elementos básicos. Al mayor detalle posible lográndose así una estructuración de

programa que aumente la eficacia en la que es usado un microcontrolador comúnmente.

Elementos del Kernel

El kernel en tiempo de diseño:

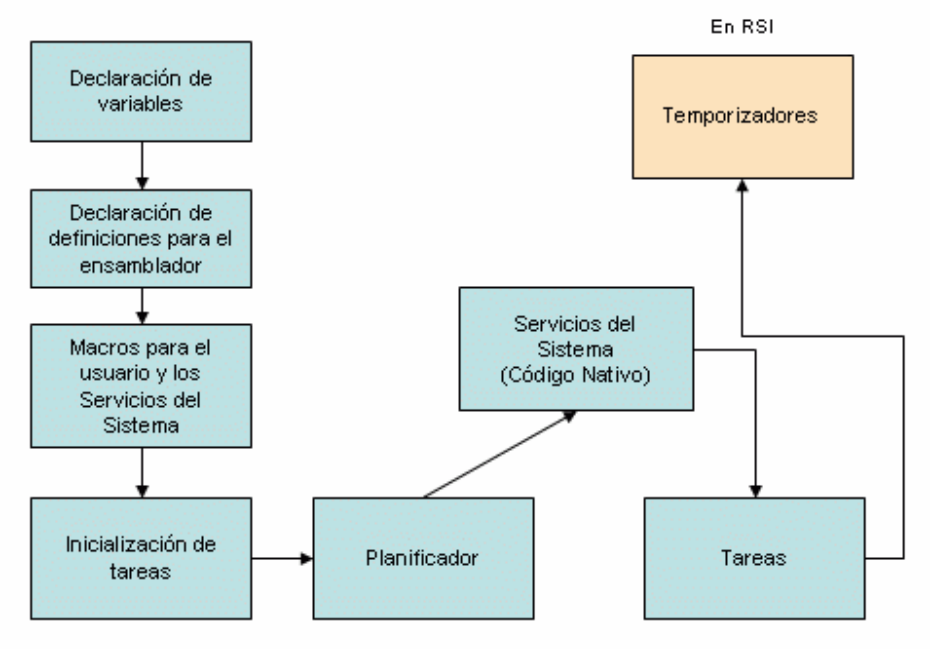


Figura 12. Bloques del kernel en tiempo de diseño

El kernel está elaborado para que sea compilado por el entorno de desarrollo integrado del fabricante del microcontrolador objeto, Araguaney se estructura en tiempo de diseño en varios bloques de manera que MPLAB pueda generar la aplicación correctamente al compilar, el primer bloque es el de *declaración de variables* aquí se declaran las variables en memoria, se debe definir los *bloques de control de tareas* para cada tarea que tendrá la aplicación (variables de superbloque de tareas), incluyendo los dos punteros que sirven como índice en el superbloque de tareas, el superbloque de tareas se encuentra ubicado en el banco 0 del PIC.

Después de definir las variables que usará el kernel para administrar las tareas, se declaran las variables que usará (si se ha de usar) del superbloque de temporización, éste superbloque se encuentra mapeado en el banco 1 del PIC. De esta forma el kernel está preparado para brindar servicios de temporización.

Las variables para la aplicación, semáforos, y elementos de comunicación se definen en ésta sección del kernel, respetando siempre el mapeado de las variables usadas por el sistema.

En la sección *declaración de definiciones para el ensamblador*, es una pequeña sección de código no ejecutable donde se definen pines, igualdades que serán usadas por el ensamblador para sustituir cadenas en el código fuente de la tareas a fin se simplificar su programación.

Seguido de esto están las macros (ya creadas) donde residen los servicios del sistema, están dentro de un archivo para hacer el kernel más modular, estas deben ser incluidas al ensamblador mediante la directiva de compilación del MPLAB "#include", el usuario puede crear macros dentro de éste archivo o aparte, deben colocarse en ésta sección para mantener la coherencia de programación.

Es aquí donde comienza el código ejecutable para el microcontrolador, es en está sección (*Inicialización de tareas*) donde el kernel conoce la ubicación de las tareas, su estado inicial y su prioridad.

El planificador se encuentra después de la sección de inicialización de tareas, la *sección de tareas* es más flexible en su ubicación en ROM, las tareas pueden estar repartidas en cualquier punto de la memoria de programa, a igual que la *sección de temporizadores* (contiene las rutinas que administra a los temporizadores), tiene la misma flexibilidad que la sección de tareas, con la única diferencia que está dentro de

la rutina de servicio de interrupción del microcontrolador y se encuentra dentro de un archivo para que sea incluido a través de la directiva de compilador "#include".

Mapeo de Memoria

El kernel para poder controlar el flujo de ejecución de las tareas depende de una estructura de datos llamada superbloque de tareas, este bloque lo conforman varios BCTs (bloque de control de tareas) consecutivos. El superbloque de tareas se ubica específicamente en el banco 0 del PIC16F877A, como cada BCT ocupa 4 bytes de RAM; Araguaney permite un máximo de 18 tareas, es decir que una aplicación podrá tener un máximo de 18 tareas individuales, se usa al final de los BCTs 6 bytes adicionales, 2 bytes que hacen el papel de punteros del superbloque y 4 bytes destinados al planificador.

Aunque el banco 0 del PIC16F877A comprenda hasta la dirección 7Fh, se usa para el superbloque de tareas hasta la dirección 6Ah, esto sumando los 2 bytes de los punteros del superbloque de tareas. Después del superbloque de tareas se encuentran 4 bytes destinados para las labores de análisis del planificador, las posiciones de memoria desde 70h a 7Fh se usan como variables globales del sistema, aprovechando que estas variables se pueden independientemente del direccionamiento indirecto que tenga el PIC en ese momento. (Figura 13.a).

Dichas variables actúan como variables temporales para almacenar el acumulador, banderas de estado (flags), y el puntero de direccionamiento indirecto del microcontrolador, en muchos de los casos es antes de hacerle servicio a una interrupción. Es posible destinar el espacio sobrante del superbloque (en caso de que la aplicación requiera menos de 18 tareas) para almacenar variables que sean usadas por las tareas de la aplicación, aprovechando así los recursos del microcontrolador en este banco. (Figura 13.b).

Todo este espacio de RAM del PIC está destinado a albergar el Superbloque de tareas	20h ~ 6Bh
Variables del Planificador	6Ch ~ 6Fh
Variables Globales del Kernel	70h ~ 7Fh

Banco 0

Figura 13.a

SUPERBLOQUE DE TAREAS	20h ~ n
Si la aplicación no llena con BCTs este espacio, se puede almacenar variables de la aplicación en ésta área.	n ~ 6Bh
Variables del Planificador	6Ch ~ 70h
Variables Globales del Kernel	70h ~ 7Fh

Banco 0

Figura 13.b

Figura 12.a y 13.b Mapeo de Memoria de Araguaney

El superbloque de tareas está conformado por varios BCTs y cada bloque posee su mapeo de memoria, una estructura de datos con la cual el planificador puede procesar la información inherente a las tareas; cada uno de ellos ocupa 4 bytes de memoria RAM. El mapeo de cada BCT (bloque de control de tareas) es el siguiente:

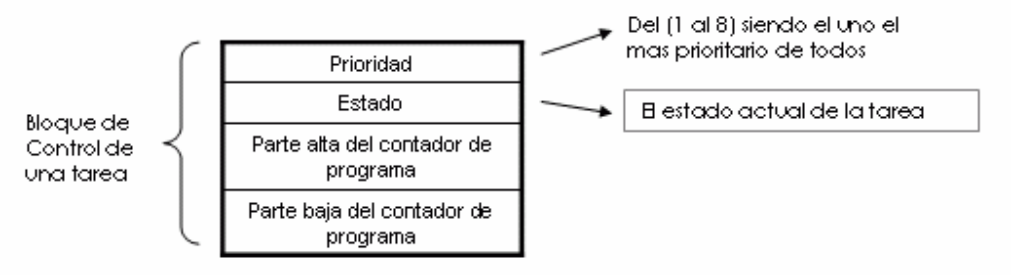


Figura 13: Mapa de memoria de un BCT

El bloque sólo contiene la información necesaria para reanudar la ejecución (Contador de programa) la prioridad de la tarea y su estado, no se guardan pilas de la tarea ni de las funciones, ya que cada tarea posee el procesador completo (multitarea cooperativa) para sí y no será apropiado el procesador por otra tarea sino hasta que la misma tarea devuelva el procesador al planificador, una tarea cederá el procesador en

el punto en que el programador lo desee y éste debe garantizar un estado seguro para el cambio de contexto, esto se puede lograr guardando los datos requeridos en variables pertenecientes a la tarea, por ejemplo, antes de liberar el CPU.

El superbloque contiene dos punteros, que son dos variables al final del él, una variable llamada "puntero de superbloque" siempre apunta al próximo BCT a ser evaluado por el planificador; la otra variable llamada "puntero de tarea anterior" apunta al BCT que contiene los datos de la tarea a la cual se le ha cedido el procesador. (Ver figura 15).

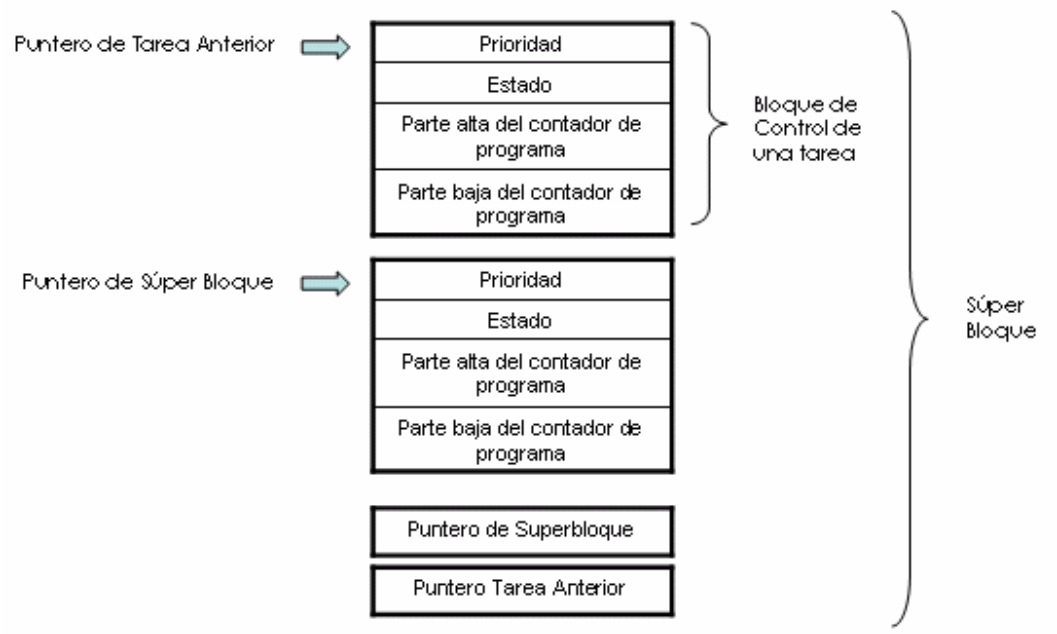


Figura 14: Súper Bloque de Tareas (Mapa de memoria)

En la figura 15, se muestra el superbloque de tareas para una aplicación con dos tareas, se pueden apreciar dos BCT cualquiera, más dos variables al final (los punteros mencionados), como se usa mucho el direccionamiento indirecto del PIC16F877A en la muchas operaciones de acceso a memoria, el planificador gracias

a ellos conoce el punto donde se aprobó la ejecución de una tarea y el punto donde analizará el próximo bloque de control de tareas.

Superbloque de temporización

Otra estructura de datos similar a la del superbloque de tareas es la del superbloque de temporizadores, en este superbloque se guardan los temporizadores individuales de la aplicación, cada temporizador es de 32 bits, permitiendo el conteo de hasta 4.294.967.295 ticks de reloj de sistema, capacitando al kernel de poder temporizar largos períodos de tiempo dependiendo de la duración de tick de reloj de sistema, posee 4 bytes al principio del superbloque de temporización que actúan como variables temporales para rellenar datos de temporización a cada bloque de temporización dentro del superbloque, cada BT (bloque de temporización) ocupa 5 bytes, el primer byte es el estado del temporizador y los 4 bytes restantes es el contador a 32 bits.

Al final del superbloque se encuentra el puntero al superbloque de temporización, la bandera de temporizadores y dos registros temporales usados por las rutinas y servicios del sistema orientados a la temporización.

El máximo de temporizadores individuales es 14, cabe destacar que la asignación de temporizadores se realiza de manera dinámica, es decir que el límite es de 14 temporizaciones simultáneas, el superbloque reside en el banco 1 de los registros de propósito generales del PIC16F877A, ver figura 16.a.

Todo este espacio de RAM del PIC está destinado a albergar el Superbloque de temporización	A0h ~ EFh
Variables Globales del Kernel	F0h ~ FFh

Banco 1

Figura 16.a

SUPERBLOQUE DE TEMPORIZACIÓN	A0h ~ EFh
Si la aplicación no llena con BTs este espacio, se puede almacenar variables de la aplicación en ésta área.	
Variables Globales del Kernel	F0h ~ FFh

Banco 1

Figura 16.b

Figura 15.a y 16.b Mapa de memoria (temporizadores)

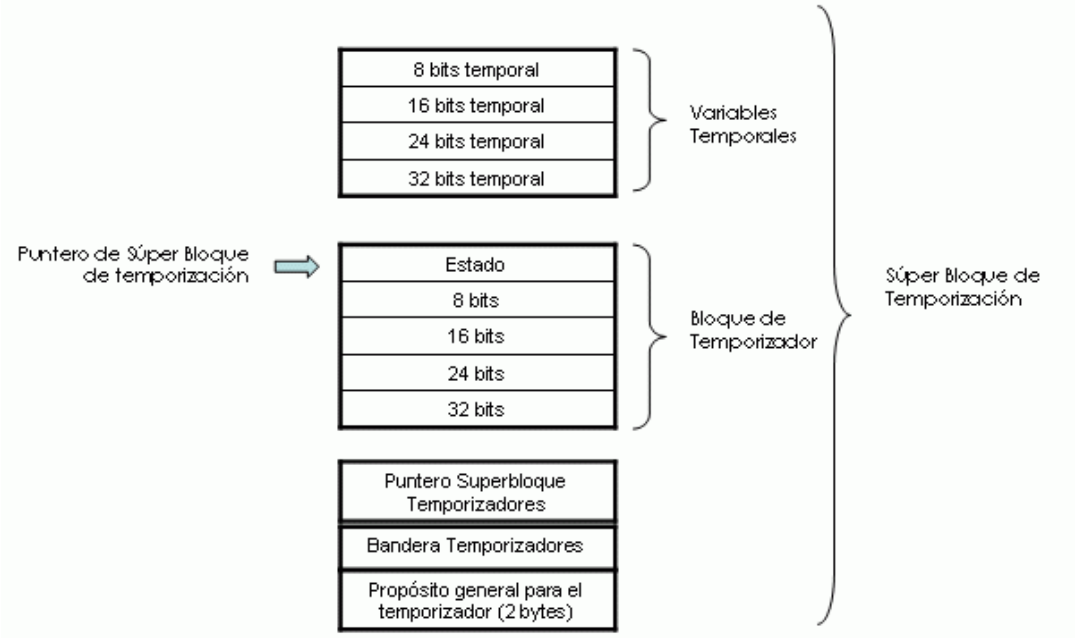


Figura 16: Súper Bloque de Temporización

Se puede almacenar variables en el banco 1 del PIC16F877A si el superbloque de temporización no llena los límites destinados a él (figura 16.b), es decir, si la aplicación no requiere de muchos temporizadores se puede disponer de más espacio para almacenar las variables de las tareas en el banco 1 del PIC.

La figura 17, muestra el detalle de un superbloque de temporización en una aplicación que usa sólo un temporizador.

Variables del usuario

El usuario puede hacer uso de variables en cualquier banco, siempre y cuando éste respete el mapeo de memoria de las variables usadas por el kernel, si el superbloque de tareas no ocupa completamente su región de memoria (figura 10.a) se puede usar el espacio restante para almacenar las variables usadas por las tareas. El mismo caso aplica para el superbloque de temporización, el kernel no hace uso de los bancos 2 y 3 de los registros de propósito general del PIC16F877A, por tanto estos pueden ser usados libremente por el usuario.

Espacio del Kernel (memoria de programa)

El kernel reside en el banco 0 de la memoria de programa del PIC, ya que de por sí es muy ligero, incluyendo los servicios, la rutina de control de los temporizadores y con 2 tareas que dentro de sí tienen un solo cambio de contexto, oscila entre 480 y 520 palabras de instrucción; el PIC16F877A posee 8192 palabras de instrucción, es decir que no ocupa (el kernel bajo las condiciones preescritas) ni la mitad de una pagina de su memoria de programa (1 página del PIC16F877A = 2048 palabras de instrucción). El programador debe tomar las precauciones comunes de la programación en ensamblador del microcontrolador objeto (llamadas y sus páginas, direccionamiento, etc.) a fin de evitar errores de programación.

El Planificador

El corazón del kernel reside en el planificador, éste decide cual tarea correrá y cual no, pero dichas decisiones van acorde a ciertos criterios e involucran varios mecanismos. El planificador está conformado por tres mecanismos independientes que ofrecen a las tareas distintos niveles de prioridad, análisis de sus estados, etc. Permitiéndole tomar acciones en consecuencia.

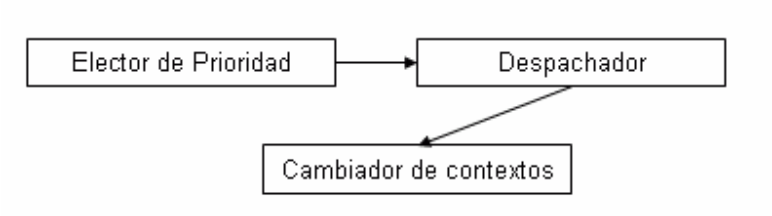


Figura 17: Elementos del planificador

En la figura 18, se muestran los elementos que conforman al planificador, el primer bloque consta del Elector de Prioridades, éste decide que cual es la prioridad que el despachador ha de buscar para su ejecución. El elector de prioridades basa sus criterios en la siguiente expresión:

$$P1>P2>P3>P4>P5>P6>P7>P8$$

Donde P1 es la prioridad 1 (la de mas alta prioridad) y P8 es la prioridad 8 (la prioridad más baja), esto quiere decir que las tareas con prioridad 1 deben obtener el control del procesador más veces que las tareas de una prioridad inferior.

El elector de prioridades usa un mecanismo que satisface dicha expresión de una manera inteligente, ahorrando muchas líneas de código y variables en RAM (3

bytes), el mecanismo (figuras 19.a y 19.b) consta de un contador de carrera libre, dicho contador se incrementará cada vez que se llame al planificador, luego ha de rotar a la derecha el contador de carrera libre a través del acarreo y ha de contar cuantas veces tiene que rotar el contador de carrera libre hasta que el acarreo valga 1. Sólo rotará el contador 9 veces (por las ocho prioridades) si no ha encontrado un 1 al rotar 9 veces, incrementa el contador de carrera libre y se repite la operación. Se puede decir que el elector de prioridades es una función que devuelve en una variable el valor de la prioridad que le toca obtener el control del procesador, con la única diferencia que el elector de prioridades está dentro del planificador y no puede llamársele como se le llama a una función.

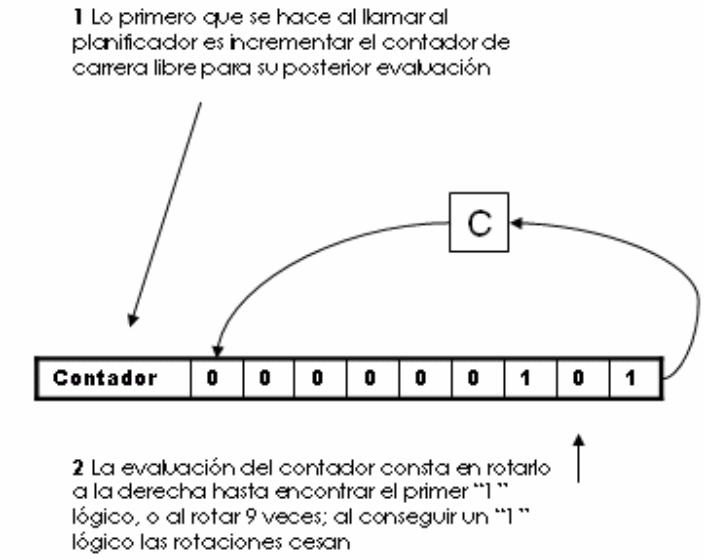


Figura 18.a: Elector de Prioridades (Mecanismo Interno)

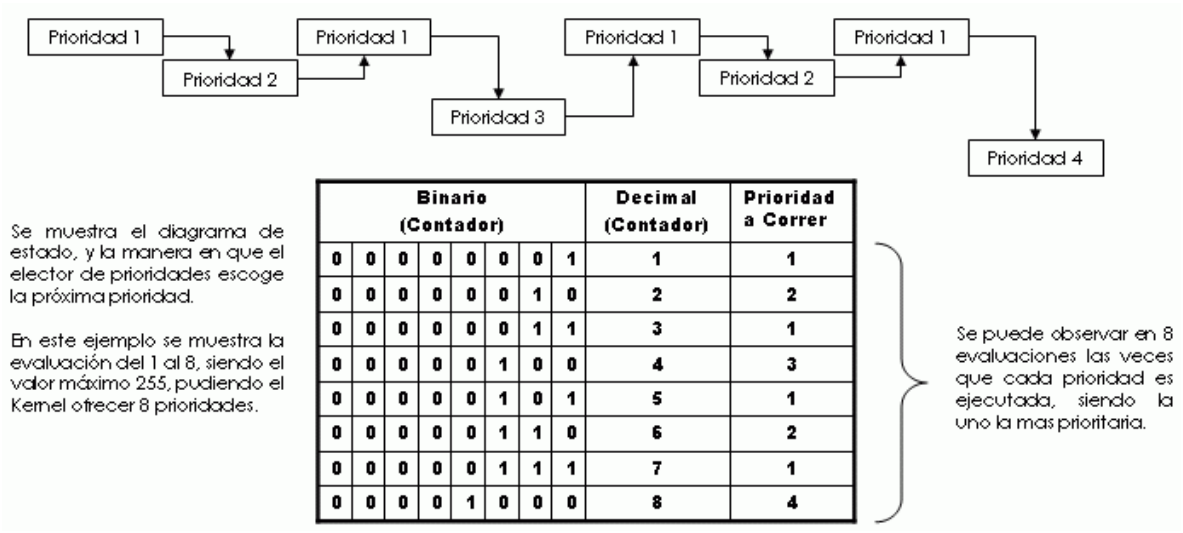
3 Cada vez que se hace una rotación esta variable incrementa en uno su valor, es decir la próxima prioridad a correr es igual a las veces que se necesito rotar para conseguir el primer "1" lógico

Prioridad a Elegir

Binario (Contador)								Decimal (Contador)	Prioridad a Correr
0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	2	2
0	0	0	0	0	0	1	1	3	1
0	0	0	0	0	1	0	0	4	3
0	0	0	0	0	1	0	1	5	1
0	0	0	0	0	1	1	0	6	2
0	0	0	0	0	1	1	1	7	1
0	0	0	0	1	0	0	0	8	4

4 Se puede apreciar en la tabla la evaluación hasta 8 incrementos del contador pudiéndose notar cual prioridad se le será cedida la CPU.

Figura 19.b Elector de Prioridades (Mecanismo Interno)



Binario (Contador)								Decimal (Contador)	Prioridad a Correr
0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	2	2
0	0	0	0	0	0	1	1	3	1
0	0	0	0	0	1	0	0	4	3
0	0	0	0	0	1	0	1	5	1
0	0	0	0	0	1	1	0	6	2
0	0	0	0	0	1	1	1	7	1
0	0	0	0	1	0	0	0	8	4

Figura 20: Diagrama de tiempo (Evaluación Corta)

En la figura 20, se muestra los valores arrojados por el elector de prioridades evaluando solo 8 incrementos del contador de carrera libre para demostrar el funcionamiento del elector de prioridades, las rotaciones cesarán cuando se consiga el primer "1" lógico, mientras otro contador va contando las veces que se rotó el contador, y ese valor es la prioridad que le toca correr, ésta técnica cumple con la expresión que se indicó previamente. (P1>P2…>P8.)

El Despachador

Esta sección del planificador escanea el superbloque de tareas en busca de un BCT que contenga en su variable de prioridad la misma prioridad que el elector de tareas escogió, para luego leer su estado y tomar las acciones pertinentes (ejecutar, hacer elegible, chequear si hay un temporizador libre).

En la figura 21, se muestra la manera en que el despachador accede al superbloque de tareas, el cual es de manera circular, supongamos que el primer escaneo comience en el principio del superbloque [1], el despachador irá escaneando bloque a bloque hasta que encuentre lo que busca, de no encontrarlo, buscará en el siguiente bloque [2], sí, el despachador no consiguió lo que buscaba llegará a escanear todo el superbloque [3]. Supongamos que el escaneo comenzó en un punto intermedio del superbloque [4]. El despachador irá escaneando como lo habíamos explicado, pero al llegar al final del superbloque y todavía el despachador no consiguió lo que buscaba [5], continúa el escaneo retomando desde el principio del superbloque [6], nótese que la idea es que todos los BCT sean escaneados.

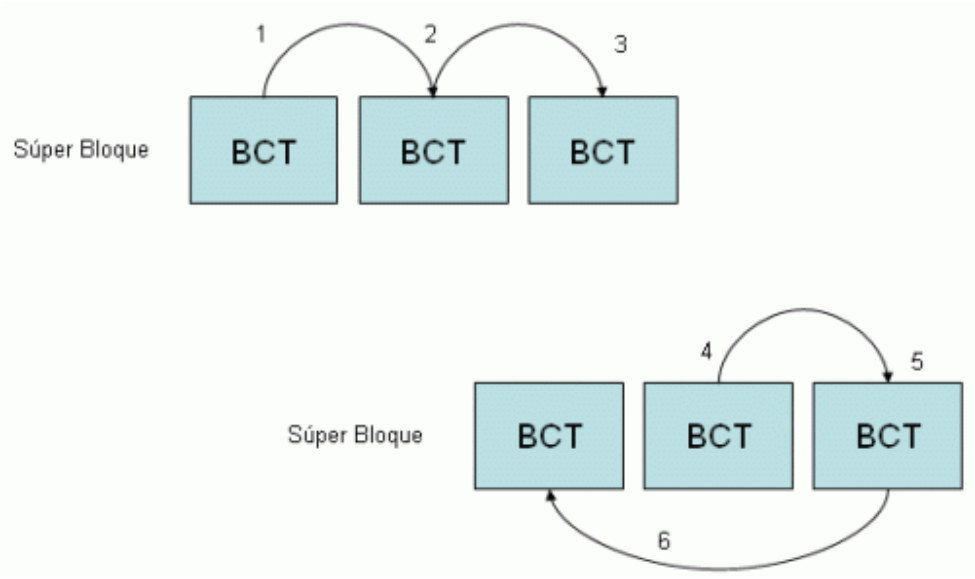


Figura 21: Acceso circular al Superbloque de tareas

Una vez que el despachador consigue un BCT que cumpla con su criterio de búsqueda, detiene la búsqueda, y memoriza el punto en el superbloque donde realizó el último escaneo, pasa a analizar el estado del BCT. El acceso circular brinda la capacidad a las tareas que tienen la misma prioridad de poder ejecutarse de manera cíclica (*Round Robin*), administra la ejecución de las tareas elegibles despachándolas a través del cambiador de contextos.

El Cambiador de contextos

Es la parte más pequeña del planificador, pero no deja de ser importante ya que es la parte encargada de ceder el procesador a una tarea, haciendo uso de una técnica de direccionamiento que consiste que escribir indirectamente al contador de programa del PIC16F877A a través de registros de propósito específicos del PIC destinados a éste fin. Gracias a esta técnica se puede acceder a toda la memoria de programa, es decir que una tarea puede estar ubicada en cualquier punto de la memoria de programa y no sólo en puntos privilegiados.

Estructura de una tarea de Araguaney

Las tareas tienen una estructura modular pero pueden cooperar entre ellas, comunicarse mediante semáforos u otro tipo de señalización, pero como Araguaney es un kernel multitarea cooperativa, cada tarea se ejecuta bajo un bucle infinito y el cual debe tener en algún punto una sentencia que se encargue de ceder el procesador a otras tareas, consideremos los ejemplos:

Tarea 1
    Hacer algo
    Entregar CPU /* Siempre debe existir un cambio de contexto */
Ir a Tarea 1

Tarea 2
    Hacer algo
    Entregar CPU
Ir a Tarea 2

Bajo esta forma se garantiza que las dos tareas tengan oportunidad de correr; pero ahora analicemos un error de programación (omitir un cambio de contexto):

Tarea 1
    Hacer algo
    Entregar CPU /* Siempre debe existir un cambio de contexto */
Ir a Tarea 1

Tarea 2
    Hacer algo
    /* Se omitió el cambio de contexto */
Ir a Tarea 2

Aquí sucede una situación que el programador debe evitar, ha omitido un cambio de contexto y el resultado será, la tarea 1 correrá y realizará sus labores hasta que ceda el procesador al planificador, el planificador le otorga el procesador a la tarea 2, ésta realizará sus labores de manera indefinida y la aplicación completa colapsará debido a que el planificador no puede asignar la ejecución de otras tareas hasta que la tarea que tiene el procesador para sí ***coopere*** a liberarlo, se dice que la aplicación se ha colgado.

Estados de una tarea en Araguaney

En la figura 22 se muestran los estados de las tareas en Araguaney…

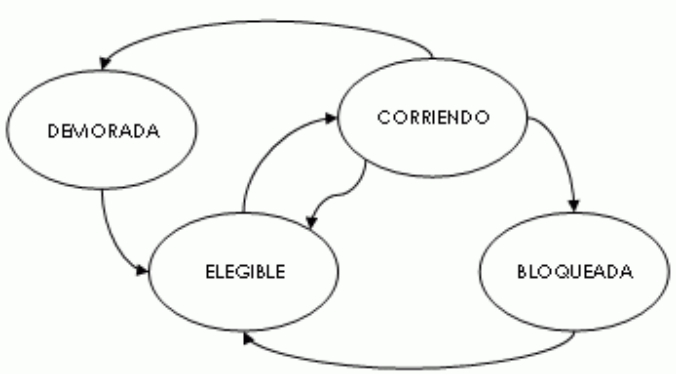


Figura 22: Estados de una tarea (Araguaney)

Elegible: Las tareas que estén listas para que les sean cedidas el procesador se dicen que son *elegibles*, el planificador decidirá el momento de su ejecución dependiendo de su prioridad

Corriendo: Cualquier tarea a la que le sea cedido el procesador estará en este estado y permanecerá así hasta que el planificador se tope de nuevo con ella y al saber que ya fue ejecutada la haga de nuevo elegible.

Demorada: Cuando dentro de una tarea se llama a un *temporizador bloqueante* (ver temporizadores de Araguaney) la tarea que lo llama pasa al estado *demorada*, y el procesador le será cedido a otra tarea. Cuando finalice el tiempo de demora deseado por el programador, la tarea continuará su ejecución normalmente cuando el planificador le otorgue el procesador.

Bloqueada: Un tarea bloqueada no se le permitirá correr y permanecerá en ese estado hasta que una causa externa lo saque de dicho estado, las razones pueden ser variadas. Pueden ser provocadas (porque el programa explícitamente cambie el estado de la tarea) o implícitas, por ejemplo, porque un recurso que la tarea pidió no está disponible, tal cual es el caso con los temporizadores (ver temporizadores de Araguaney), poseen la propiedad de causar el bloqueo de una tarea si ésta al solicitar un temporizador, no hay un temporizador libre, la tarea se bloqueará hasta que se haya liberado un temporizador y éste le sea asignado a la tarea.

Temporizadores de Araguaney

Los temporizadores en Araguaney son estructuras de datos de 5 bytes, éstas estructuras pueden ser reutilizadas por el kernel, a diferencia de algunos modelos de programación donde se le asigna a una tarea uno o más temporizadores de manera fija; Araguaney es muy dinámico con respecto a la etapa de temporización, pues ofrece dos estructuras de programación para temporizadores, la primera son los *temporizadores bloqueantes*, y la segunda los *temporizadores no bloqueantes*; si una tarea quiere hacer uso de un temporizador, ésta debe pedirlo primero a través de un servicio del sistema, si existe un temporizador libre, se le asigna a la tarea, pero si no la tarea es bloqueada hasta que se haya liberado un temporizador.

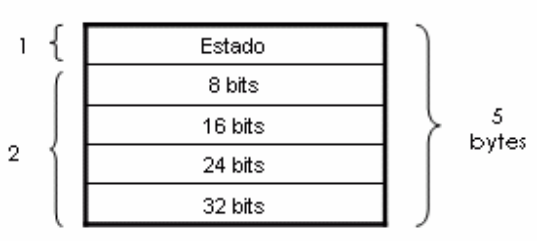


Figura 23: Detalle de un Temporizador (estructura de datos)

La figura 23 muestra el detalle de la estructura de datos usada para temporizar, ésta estructura reside dentro del superbloque de temporizadores, está compuesta por 5 bytes, el primero de ellos [1], el byte de estado, éste almacena la dirección de memoria RAM donde se encuentra la variable de estado de la tarea que pide el temporizador o almacena la *llave* del temporizador, los otros 4 bytes restantes [2], son un contador regresivo de 32bits.

Temporizadores bloqueantes

Como se había mencionado antes, Araguaney permite dos estructuras de programación para manejar las temporizaciones. Una de ellas es el temporizador bloqueante, ésta estructura permite que una tarea pueda esperar un tiempo determinado (en ticks de reloj del sistema) cuando ésta no requiera procesar más nada hasta que ese retardo termine, pues la tarea cambiada al estado *demorada* y no se gastará tiempo de procesador en ella, se puede decir que la tarea está bloqueada, pero con la diferencia de que la tarea pasará al estado elegible cuando su tiempo se termine, esto permite que otras tareas puedan ejecutarse mientras dicha tarea espera a que su tiempo expire.

La estructura de un temporizador bloqueante es así:

```
Tarea Medición
        NOP
        NOP
        ADC  ; Conversión analógica  / digital
        DEMORA 5   ; Esperar 5 ticks sin hacer nada más
        CALL Mostrar_Valor
        GOTO Tarea Medición
```

En éste ejemplo se muestra una tarea sencilla en donde se necesita leer un valor analógico y mostrar los resultados de dicho valor, pero mientras la conversión se realiza sólo se debe esperar un tiempo para después mostrar el valor, pues no se puede mostrar un valor que todavía no se ha medido, el tiempo que puede llevar hacer una medición por ejemplo, puede ser utilizado por otras tareas. La sentencia usada para llamar a un temporizador bloqueante es DEMORA *n*. Donde *n* es el número de ticks de retardo que se quiere; la tarea continuará su ejecución desde el llamado a la rutina que muestra el valor al usuario por ejemplo. Al momento de la demora, el procesador puede ser usado por otras tareas.

Existe un detalle, al momento en que el tiempo que la tarea espera termine, no se le cede el procesador a la tarea inmediatamente, sino que más bien cambia el estado de la tarea a elegible, para que el planificador le ceda el procesador cuando le corresponda, esto evita en cierta forma la inversión de prioridades.

Temporizadores No Bloqueantes

La estructura de los temporizadores no bloqueantes es más compleja que la de los temporizadores bloqueantes, como su nombre lo dice, estos temporizadores son llamados desde una tarea pero no la bloquean, funcionan bajo un método de *llave* y *cerradura*, las tareas al requerir un temporizador no bloqueante deben primero pedir un temporizador, si éste le es concedido, la tarea especificará una llave (un número entero de 0 a 127) dentro del bloque de temporización y es deber de la tarea chequear el temporizador para ver si el tiempo que requiere ha transcurrido (no cede el procesador a otras tareas). Como la asignación de bloque de temporización es dinámica se emplea la llave para que la tarea que pidió el temporizador sea la única en poder acceder y liberar dicho temporizador que es la cerradura. La ventaja es que una tarea puede usar más de un temporizador no bloqueante a la vez, sin que la tarea cese su ejecución.

Los temporizadores no bloqueantes permiten estructuras de programación como la siguiente:

```
Tarea
        Contar 10 Seg bajo la llave X
            {
              (Cualquier cosa)
            }
        Chequear Tiempo y usar llave X
        Encender un Led
        Entregar CPU
Ir a Tarea
```

El ejemplo está diagramado en pseudocódigo de alto nivel, los temporizadores no bloqueantes permiten en este ejemplo, realizar una operación cualquiera durante 10 segundos sin ceder el procesador, una vez finalizado ese tiempo dejar de hacer lo que se estaba haciendo y encender un led, es deber de la tarea chequear si el conteo del temporizador expiró, el kernel emplea el método de llave y cerradura (figura 24) para estos temporizadores, de tal manera que los servicios de temporización sepan a cual temporizador se desea acceder.

Figura 24: Método llave y cerradura

Consideraciones con los temporizadores

En los dos tipos de temporizadores la tarea hace la petición de un temporizador, si hay temporizadores libre dentro del superbloque de temporizadores se les cede, pero de no haber la tarea se bloquea automáticamente y reanudará al ejecución cuando haya un temporizador disponible y entonces éste le sea asignado, de esta manera las tareas se pueden compartir los bloques de temporización y evitar que la aplicación se cuelgue. Claro está que una aplicación que usa mucho los temporizadores y no hayan muchos de estos, afectará el rendimiento de la aplicación, pero no permitirá que se cuelgue.

El byte de estado en un bloque de temporización cumple un doble rol. Puede fungir como puntero al estado de la tarea que pidió el bloque de temporización o como una llave; pues como el superbloque de tareas reside en el banco 0 de RAM del PIC16F877A y su límite superior es la dirección 6Bh; puesto que la variable de estado de un temporizador bloqueante jamás alcanzará un valor superior a este, el kernel reconoce con solo leer la variable de estado que tipo de temporizador se trata. Esto se logra leyendo el octavo bit de la variable de estado en un bloque de temporización, si se lee un "1" lógico el temporizador es no bloqueante y por tanto los otros 7 bits corresponden a la llave del mismo, si por el contrario el octavo bit es "0" entonces el temporizador es bloqueante y el valor completo de la variable estado es el puntero a RAM del estado de la tarea. Ver figura 25.

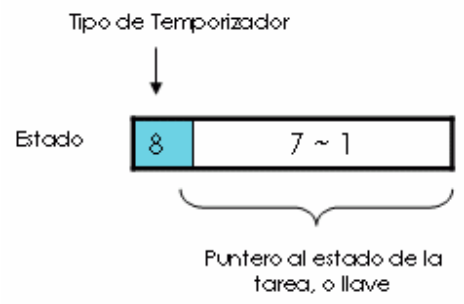


Figura 25: Detalle de la variable de estado de un bloque de temporización

La **precisión** de los temporizadores contiene un margen de error de aproximadamente uno o dos ticks de reloj de tolerancia.

Bandera de temporizador (B_TEMPORIZADOR)

Existe una bandera de sistema que rige a los servicios de temporización del kernel, ésta es la bandera de temporizador (B_TEMPORIZADOR en el kernel); las operaciones relacionadas con temporización reflejan sus resultados en ésta bandera, también la bandera sirve para establecer un tipo de temporizador, la definición de la bandera es:

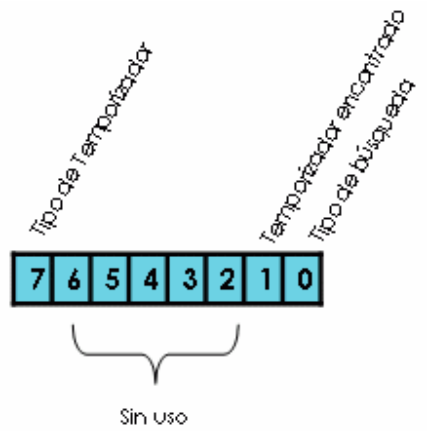


Figura 26: Bandera de Temporizador

Definición de los bits:

<7>: Tipo de temporizador: Este bit define el tipo de temporizador a establecer a una tarea. Si es "1" el temporizador será no bloqueante, si es "0" será bloqueante.

<6-2>: Sin uso

<1> Temporizador encontrado: Vale "1" si el temporizador que se buscaba se encontró, de no ser así vale "0"

<0> Tipo de búsqueda: Si es establecido a "0", antes de buscar un temporizador a través del servicio del sistema, se procede a buscar un temporizador libre, si es establecido a "1" se buscará un temporizador ocupado.

Servicios del sistema

Existen dos tipos de servicio de sistema, uno que yace dentro de la capa del kernel, que consiste en rutinas de manipulación de los temporizadores y se encuentran a un nivel de abstracción mínimo, el otro, consiste en rutinas utilizables desde la capa de aplicación, y constituyen un nivel mayor de abstracción.

Rutinas de bajo nivel

Temporizador

Asigna a la tarea que la llama un temporizador bloqueante o no bloqueante según el caso, esta rutina busca en la bandera de temporizador, dependiendo del valor del bit <7> (ver Bandera de Temporizador) asigna un temporizador bloqueante o no bloqueante a una tarea.

Buscar temporizador

Busca dentro del bloque de temporización un temporizador siguiendo criterios establecidos en la bandera de temporizador (B_TEMPORIZADOR), es una función que coloca el puntero de direccionamiento indirecto (FSR) del PIC en el inicio del bloque de temporización cuando encuentra una coincidencia.

Esta rutina es capaz de buscar temporizadores libres y ocupados dentro del superbloque de temporización, en caso de conseguir ocupados puede discernir entre bloqueantes y no bloqueantes, en el caso de buscar temporizadores no bloqueantes se debe almacenar en la variable temporal (GENERAL_TEMP) el valor de la llave

Chequear temporizador

Esta analiza el bloque de un temporizador no bloqueante para verificar si su contador expiró, ésta requiere como parámetro el valor de la llave del temporizador a chequear, la cual se debe almacenar en la variable (GENERAL_TEMP) antes de llamar a este servicio.

Este servicio hace uso del servicio a bajo nivel "Buscar Temporizador", aprovechándose mejor el código del kernel.

Rutinas de alto nivel

Entrega Cpu

Toda tarea debe tener al menos una instrucción macro-ensamblada como ésta dentro de la estructura de una tarea, pues ésta permite la liberación del procesador por parte de la tarea para que sea ocupado por el planificador.

Demora n

Asigna un temporizador bloqueante a la tarea y detiene la ejecución de la tarea hasta que el tiempo establecido termine, por tanto una tarea solo puede llamar pedir un temporizador bloqueante a la vez; el único parámetro que se le pasa a este servicio es el tiempo de la demora expresado en ticks de reloj del sistema (*n*).

Cuenta tiempo n, m

Asigna un temporizador no bloqueante a una tarea, éste no detiene ni cede la ejecución de la tarea, la tarea al pedir un temporizador no bloqueante debe especificar el tiempo en ticks de sistema (*n*) y una llave (*m*)

Chequear Tiempo n, Etiqueta

Chequea si a un temporizador no bloqueante específico (*n* especifica la llave) le ha expirado su tiempo, de no haber terminado, el flujo del programa salta a la *etiqueta* especificada, si el tiempo expiró entonces la ejecución continúa después de ésta sentencia y libera el temporizador no bloqueante y el bloque asociado queda a disposición.

La aplicación prueba

Para probar el funcionamiento del kernel se construyó una aplicación muy sencilla, la cual consta de 8 leds, los cuales encienden creando un efecto visual giratorio, esto simbolizaría a un proceso cualquiera pero de poca importancia (un calculo, o mostrar una información al usuario) este efecto será continuo, se colocó tres pulsadores (simulando procesos más prioritarios y que requieren una rápida repuesta del sistema), cada pulsador generó un tono, a través del cambio del estado del pin PORTC<5> del PIC16F877A durante un período de tiempo (para el cual se utilizó temporizadores bloqueantes, puesto que no era necesario para la tarea hacer nada más mientras el lapso de tiempo transcurría), y conectando el pin a un amplificador de audio para poder excitar a un altavoz con potencia suficiente para que sea audible; los pulsadores presionados simultáneamente mezclaron los sonidos que individualmente producen en el altavoz sin alterar el "giro de los leds".

La aplicación se estructuró en cuatro tareas:

Leds Giratorios
Prioridad: 6
Temporizadores requeridos: 1

Se encarga de generar el efecto rotatorio de los leds, cambiando el estado del PORTB del PIC16F877A aproximadamente cada 500ms. Como no es algo tan prioritario como recibir una respuesta del usuario se le ha colocado la prioridad 6

Generar Tono 1
Prioridad: 1
Temporizadores requeridos: 1

Genera un tono cercano al Do#

Generar Tono 2
Prioridad: 1
Temporizadores requeridos: 1

Genera un tono cercano al Sol#

Generar Tono 3
Prioridad: 1
Temporizadores requeridos: 1

Genera un tono cercano al Do# pero en una octava más alta.

Las tres últimas tareas reciben la entrada del usuario (los pulsadores) y por tanto requieren la máxima velocidad de respuesta, se les asignó la prioridad más alta que es la 1. El microcontrolador trabajó con un resonador paralelo a cristal de 20MHZ, fue configurado el sistema para que ocurriera un tick de reloj cada 204.8us. Esta base de tiempo permitió establecer temporizadores bloqueantes con resolución suficiente para la generación de los tonos.