1. Super Simple Tasker

El Super Simple Tasker (SST) scheduler proporciona un planificador conducido por interrupciones que usa una sola pila. Si se modela una tarea como una acción run-to-completion discreta se puede crear un kernel de tiempo real, determinístico, completamente apropiativo, priorizado, llamado Super simple Tasker (SST) con solamente unas cuantas docenas de líneas de código C portable.

2. Implementación de SST

Se presenta una implementación mínima standalone de SST. El codigo presentado es ANSI C portable, con todas las partes de CPU y compilador específicas claramente separadas.

Sin embargo, la implementación omite algunas características que podrían ser necesarias en aplicaciones de la vida real. El principal objetivo en este punto es demostrar claramente los conceptos clave mientras que al mismo tiempo se permite ejecutar el código sobre cualquier PC basada en Windows. El ejemplo se puede compilar usando el IDE Turbo C++ 1.01 en PC con Windows XP, o bien usando el IDE Turbo C++ 4.0 Windows 7 Windows 8 64 bits sobre una PC con Windows 7.

El ejemplo SST demuestra las capacidades de multi-tasking y preemtion de SST (multitareas y apropiatividad de Super Simple Tasker). Este ejemplo consiste de tres tareas y dos ISRs. La ISR del tick de reloj produce un "evento tick" cada 5 ms, mientras que la ISR del teclado produce un "evento key" cada vez que una tecla es liberada y en la razón de autorrepetición del teclado cuando una tecla es presionada y mantenida así por algún tiempo. Las dos "tareas tick": tickTaskA() y tickTaskB() reciben los "eventos tick" y colocan una letra A o B, respectivamente, en una ubicación aleatoria en el panel del lado derecho de la pantalla. La tarea de teclado kbdTask() con la prioridad entre tickTaskA() y tickTaskB(), recibe los códigos de scan del teclado y envía "eventos color" a las tareas tick, las cuales en respuesta cambian el color de las letras que despliegan. También la tarea kbdTask() termina la aplicación cuando el usuario presiona la tecla ESC.

La aplicación de ejemplo SST intencionalmente usa dos interrupciones independientes (el tick de reloj y el teclado) para crear asynchronous preemtions (apropiaciones asícronas). Para incrementar más la probabilidad de preemting a task (apropiación a una tarea) y de interrupt preempting an interrupt (apropiación de interrupción a una interrupción) el código se sazona con llamadas a una función busyDelay(), la cual extiende el tiempo de run-to-completion en un ciclo de cuenta simple. Se puede especificar el número de iteraciones en este ciclo con un parámetro de línea de comando para la aplicación. Se debe ser cuidadoso (careful not to go overboard with this parameter, though) porque valores grandes producirán un conjunto de tareas que no es planificable (schedulable) y el sistema (apropiadamente) empezará a perder eventos.

Las siguientes subsecciones explican el código de SST y la estructura de la

aplicación. El código fuente de SST completo consiste del archivo de cabecera sst.h localizado en el directorio include/ y el archivo de implementación sst.c localizado en el directorio source/. Los archivos del ejemplo están localizados en el directorio example/, el cual también contiene el archivo de proyecto Turbo C++ para construir y depurar la aplicación. (NOTA: dado que la ISR del teclado estándar es remplazada por la versión personalizada de este kernel la depuración de esta aplicación con el IDE Turbo C++ podría ser dificil.)

3. Secciones críticas en SST

SST, como cualquier otro kernel, necesita realizar ciertas operaciones indivisiblemente, la forma más simple y más eficiente de proteger una sección de código de posibles disrupciones es deshabilitar las interrupciones entrando a la sección y habilitar las interrupciones cuando se sale de la sección. Las secciones de código donde se requieren esas deshabilitaciones y habilitaciones reciben el nombre de sección crítica.

Los procesadores generalmente proporcionan instrucciones para deshabilitar / habilitar las interrupciones, y su compilador de C debe tener algún mecanismo para realizar esas operaciones desde C. Algunos compiladores permiten incluir instrucciones de ensamblador inline en el código fuente C. Otros compiladores proporcionan extensiones del lenguaje o al menos funciones de C que se pueden llamar para deshabilitar y habilitar interrupciones desde C.

Para esconder el método de implementación real elegido, SST proporciona dos macros para deshabilitar y habilitar interrupciones. Aquí están las dos macros definidas para el compilador Turbo C++:

```
#define SST_INT_LOCK()    disable()

#define SST_INT_UNLOCK()  enable()
```

En la versión mínima de SST, suponemos la sección crítica más simple posible: una que incondicionalmente deshabilita las interupciones cuando se entra a ella e incondicionalmente habilita las interupciones cuando se sale de ella. Este tipo de secciones críticas simples nunca se deben anidar, porque las interrupciones siempre serán habilitadas cuando se está saliendo de una sección crítica, independientemente de si estaban deshabilitadas o habilitadas antes de entrar a ellas. El scheduler de SST está diseñado para nunca anidar secciones críticas, pero usted debe ser cuidadoso cuando use las macros SST_INT_LOCK() y SST_INT_UNLOCK() para proteger sus propias secciones críticas en las aplicaciones.

Nótese, sin embargo, que la imposibilidad de anidar secciones críticas no necesariamente significa que usted no pueda anidar interrupciones. En los procesadores equipados con un controlador de interrupciones interno o externo, tal como el PIC 8259A en la PC basada en x86, o el AIC en el microcontrolador ARM AT91, usted puede habilitar las interrupciones dentro de las ISRs en el nivel de procesador, evitando el anidamiento de seciones críticas dentro de las

ISRs, y permitiendo que el controlador de interrupción maneje la priorización de interrupciones y el anidamiento incluso antes de que estas alcancen el core de CPU.

4. Procesamiento de interrupciones en SST

Una de las mayores ventajas de SST es el procesamiento de interrupciones simple, el cual realemente no es mucho más complicado en SST de lo que es en un simple "super-loop" (a.k.a. main+ISRs). Debido a que SST no depende en modo alguno del esquema de frame de pila de interrupción, con la mayoría de los compiladores de C para embedidos, las ISRs pueden ser escritas completamente en C. Una notable diferencia entre las ISRs de un simple "super-loop" y las ISRs de SST es que SST requiere que el programador inserte algunas acciones simples en cada entrada y salida de ISR. Estas acciones están implementadas en las macros de SST SST_ISR_ENTRY() y SST_ISR_EXIT(). El fragmento de código en el listado 1 muestra como se usan estas macros en las ISRs de tick de reloj y de teclado de la aplicación de ejemplo definida en el archivo example/bsp.c.

Listado 1: ISRs del ejemplo de aplicación de SST

```
static void interrupt tickISR() {/* every ISR is entered with interrupts locked */
    uint8_t pin;  /* temporary variable to store the initial priority */
    displayPreemptions(SST_currPrio_, TICK_ISR_PRIO);/* for testing, NOTE01 */
    SST_ISR_ENTRY(pin, TICK_ISR_PRIO);

    SST_post(TICK_TASK_A_PRIO, TICK_SIG, 0);     /* post the Tick to Task A */
    SST_post(TICK_TASK_B_PRIO, TICK_SIG, 0);     /* post the Tick to Task B */

    busyDelay();                                 /* for testing, see NOTE02 */
    SST_ISR_EXIT(pin, outportb(0x20, 0x20));
}
/*.....................................................................*/
static void interrupt kbdISR() {/* every ISR is entered with interrupts locked */
    uint8_t pin;
    uint8_t key = inport(0x60);/*get scan code from the 8042 kbd controller */

    displayPreemptions(SST_currPrio_, KBD_ISR_PRIO); /* for testing, NOTE01 */
    SST_ISR_ENTRY(pin, KBD_ISR_PRIO);

    SST_post(KBD_TASK_PRIO, KBD_SIG, key);   /* post the Key to the KbdTask */

    busyDelay();                                 /* for testing, see NOTE02 */
    SST_ISR_EXIT(pin, outportb(0x20, 0x20));
}
```

Nótese que en el listado 1, se incluye la palabra específica de compilador "in-

terrupt", la cual dirije al compilador de Turbo-C para que sintetice el respaldo de contexto apropiado, la restauración del mismo y el prólogo de regreso de interrupción y los epílogos.

Nótense también las macros de entrada y salida de interrupción al principio y al final de cada ISR. (Si la fuente de interrupción requiere ser limpiada, esto debe ser hecho antes de llamar a SST_ISR_ENTRY). Las macros SST_ISR_ENTRY() y SST_ISR_EXIT() están definidas en el archivo de cabecera include/sst.h como se muestra en el listado 2. (El ciclo do .. while(0) alrededor de las macros es solo para que las instrucciones queden sintacticamente agrupadas en forma correcta.)

Listado 2: Definición de las macros de entrada/salida de interrupción de SST

```
#define SST_ISR_ENTRY(pin_, isrPrio_) do { \
    (pin_) = SST_currPrio_; \
    SST_currPrio_ = (isrPrio_); \
    SST_INT_UNLOCK(); \
} while (0)

#define SST_ISR_EXIT(pin_, EOI_command_) do { \
    SST_INT_LOCK(); \
    (EOI_command_); \
    SST_currPrio_ = (pin_); \
    SST_schedule_(); \
} while (0)
```

La macro SST_ISR_ENTRY() es invocada con las interrupciones deshabilitadas y reliza los siguientes tres pasos:

(1) Guarda la prioridad de SST inicial en la variable de pila 'pin_'.

(2) Establece la prioridad SST actual al nivel ISR.

(3) Habilita las interrupciones para permitir anidamiento de interrupciones.

La macro SST_ISR_EXIT() es invocada con las interrupciones habilitadas y realiza los siguientes 4 pasos:

(1) Deshabilita las interrupciones.

(2) Escribe el comando EOI al controlador de interrupción. (por ejemplo, outportb(0x20,0x20) escribe EOI al PIC 8259A master).

(3) Restaura la prioridad SST inicial.

(4) LLama al scheduler de SST, el cual realiza la "apropiación asíncrona," si es necesaria.

Los bloques de control de tarea, como otros kernels de tiempo real, SST mantiene registro de las tareas en un arreglo de estructuras de datos llamadas task control blocks (TCBs). Cada TCB contiene información como el apuntador a

la función de tarea, la máscara de tarea (calculada como (1 << (priority - 1))), y la cola de eventos asociada con la tarea. El TCB ocupa solamente de 8 a 10 bytes, dependiendo del tamaño del apuntador a función. Adicionalmente, usted necesita proporcionar un buffer de cola de eventos del tamaño correcto cuando usted crea una tarea. El TCB usado aquí está optimizado para un número de niveles de prioridad fijo, pequeño y eventos simples —restricciones que tienen sentido en un ambiente embedido clásico. Ninguna de estas limitaciones, sin embargo, es requerida por el algoritmo de planificación de SST.

5. Envío de eventos a las tareas

En esta versión mínima de SST, los eventos son representados como estructuras que contienen dos elementos de tamaño byte: un identificador del tipo de evento (por ejemplo, la ocurrencia de que se presionó una tecla), y el parámetro asociado con la ocurrencia (por ejemplo, el código scan de la tecla). Los eventos son almacenados en colas de eventos organizadas como buffers de anillo estándar. SST mantiene el status de todas las colas de eventos en la variable llamada SST_readySet_. SST_readySet_ es de tamaño byte, lo que significa que esta implementación está limitada a ocho niveles de prioridad. Cada bit en la variable SST_readySet_ representa una prioridad de tarea SST. El bit número 'n' en la variable SST_readySet_ es 1 si la cola de eventos de la tarea de prioridad 'n + 1' no está vacia (los bits están numerados 0..7). Complementariamente, el bit 'm' en SST_readySet_ es 0 si la cola de eventos de la tarea de prioridad 'm + 1' está vacia o el nivel de prioridad 'm + 1' no se utiliza.

El listado 3 muestra la función de envío de eventos SST_post(), la cual usa un algoritmo de buffer de anillo estándar (FIFO). Si el evento es insertado en una cola vacia (1),

```
        if ((++tcb->nUsed__) == (uint8_t)1) {           /* the first event? */
```

el bit correspondiente en SST_readySet_ es puesto a 1 (2),

```
            SST_readySet_ |= tcb->mask__;   /* insert task to the ready set */
```

y el scheduler SST es invocado para checar por "apropiación síncrona" ("synchronous preemtion") (3).

```
            SST_schedule_();            /* check for synchronous preemption */
```

Listado 3: Envío de eventos en SST

```
uint8_t SST_post(uint8_t prio, SSTSignal sig, SSTParam par) {
    TaskCB *tcb = &l_taskCB[prio - 1];
    SST_INT_LOCK();
    if (tcb->nUsed__ < tcb->end__) {
        tcb->queue__[tcb->head__].sig = sig;/* insert the event at the head */
        tcb->queue__[tcb->head__].par = par;
        if ((++tcb->head__) == tcb->end__) {
```

```
            tcb->head__ = (uint8_t)0;                       /* wrap the head */
        }
        if ((++tcb->nUsed__) == (uint8_t)1) {           /* the first event? */
            SST_readySet_ |= tcb->mask__;   /* insert task to the ready set */
            SST_schedule_();            /* check for synchronous preemption */
        }
        SST_INT_UNLOCK();
        return (uint8_t)1;                      /* event successfully posted */
    }
    else {
        SST_INT_UNLOCK();
        return (uint8_t)0;              /* queue full, event posting failed */
    }
}
```

6. El scheduler de SST

El scheduler (planificador) de SST es una función C simple SST_schedule_() cuyo trabajo es eficientemente encontrar la tarea de más alta prioridad que esté lista para correr y ejecutarla, si su prioridad es más alta que la prioridad SST actualmente siendo atendida. Para realizar este trabajo, el scheduler SST usa la variable ya descrita SST_readySet_ y el nivel de prioridad actual SST_currPrio_ mostrado en la figura 1.

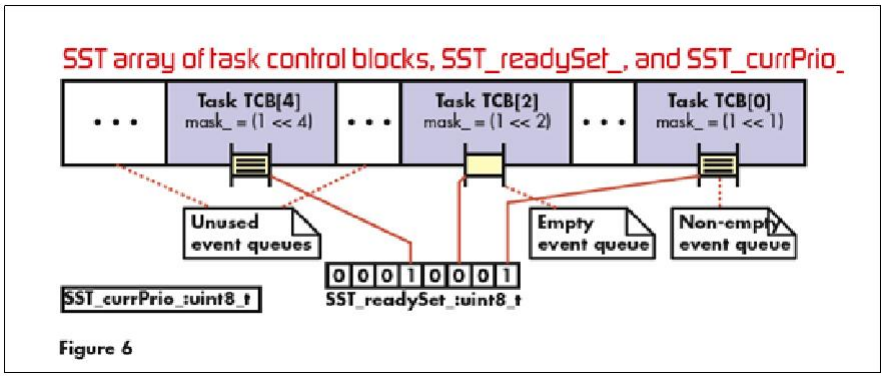

Figura 1: La variable SST_readySet_