Implementación de Slack Stealing y una Tarea Planificadora en nxtOSEK sobre Lego Mindstorms NXT 2.0

Sebastián Lucas[1], José M. Urriza[1], Francisco E. Páez[1]

Universidad Nacional de la Patagonia San Juan Bosco
Facultad de Ingeniería, Departamento de Informática
Sede Puerto Madryn
josemurriza@unp.edu.ar, fpaez@unpata.edu.ar

Resumen Este trabajo presenta una implementación, utilizando el Sistema Operativo de Tiempo Real nxtOSEK y el kit de robótica educativa Lego Mindstorms NXT 2.0, en configuración *segway*, de un método de Slack Stealing para la administración del tiempo ocioso en un Sistema de Tiempo Real, junto con una Tarea Planificadora que aprovecha el mismo para la ejecución concurrente de tareas periódicas y esporádicas, sin que las primeras pierdan sus vencimientos. Mediante este desarrollo, se pueden ejecutar tareas no críticas de manera concurrente, sin afectar las tareas críticas que mantienen el *segway* en equilibrio.

1. Introducción

Este trabajo presenta la implementación de una técnica de administración de tiempo ocioso, denominado Slack Stealing (SS), junto con una Tarea Planificadora (TP), que implementa una política de planificación que utiliza dicha técnica para usar el tiempo ocioso para la ejecución concurrente de tareas, en un Sistema Operativo de Tiempo Real (SOTR) basado en el estándar OSEK/VDX, utilizado en la industria automotriz[1]. Como plataforma de prueba se utiliza un Lego Mindstorms NXT 2.0, en configuración *segway*, donde el sistema de control, realimentado con sensores que provee la plataforma, es implementado mediante un Sistema de Tiempo Real (STR), con un sistema de control remoto vía Bluetooth y detección de obstáculos.

El trabajo se organiza de la siguiente manera: en la sección 2, se realiza una introducción a los STR y a los métodos de SS. En la sección 3 se comentan trabajos previos. Luego, en la secciones 4 y 5 se describe el estándar OSEK/VDX y la plataforma Mindstorms. Seguidamente, en las secciones 6 y 7 se presenta el diseño general e implementación del sistema. Finalmente, la sección 8 presenta las conclusiones y los trabajos futuros.

[1] Actualmente se está desarrollando a nivel nacional el FreeOSEK bajo dicho estándar, utilizado en el proyecto CIAA (Computadora Industrial Abierta Argentina).

2. Introducción a los Sistemas de Tiempo Real

En un STR los resultados, además de ser correctos aritmética y lógicamente, deben producirse antes de un instante denominado *vencimiento* [18].

Según que tan crítico es el cumplimiento de los vencimientos, los STR se clasifican en tres tipos. El primer tipo, denominado *duro* o *crítico*, no tolera la pérdida de vencimientos. El segundo tipo permite la pérdida de algunos vencimientos, y se denomina *blando*. Finalmente, el tercer tipo, *firme*, tipifica las pérdidas según un criterio estadístico.

Los STR del tipo *duro* son utilizados en sistemas donde la pérdida de un vencimiento tiene consecuencias graves (aviónica, control industrial, soporte a la vida, etc.). Por lo tanto, en etapa de diseño se garantiza el cumplimiento de los vencimientos con un *test de planificabilidad* [10, 1, 7, 22]. De ser exitoso, se dice que el STR es *planificable*. En [10] se probó que el *peor instante de carga*, en sistemas mono-recurso, es cuando todas las tareas solicitan ejecución simultánea (*instante crítico*). Si el STR es planificable en dicho instante, lo será en cualquier otro.

Un STR utiliza un *algoritmo de planificación* para determinar qué tarea ejecutará en un instante dado, que puede ser *estático* o *dinámico* [2]. Los algoritmos *dinámicos* asignan una prioridad a cada una de las tareas, que puede variar en tiempo de ejecución (*prioridades dinámicas*) o no (*prioridades fijas*). Los algoritmos de planificación dinámicos por prioridades fijas más utilizados son Rate Monotonic (RM) [10] y Deadline Monotonic (DM) [8].

Una tarea de tiempo real i (τ_i) es generalmente caracterizada mediante su periodo (T_i), su vencimiento relativo (D_i) y su peor caso de tiempo de ejecución (C_i). Luego un STR con n tareas es definido como el conjunto de ternas:

$$S(n) = (C_i, T_i, D_i), ..., (C_n, T_n, D_n) \tag{1}$$

En el caso de las tareas esporádicas, el valor del periodo (T) representa el mínimo tiempo entre dos activaciones consecutivas de la tarea.

Un STR ejecuta, por lo general, un conjunto predefinido de tareas periódicas con requerimientos de tiempo real, a las que se denominan Tareas de Tiempo Real (TTR). El STR puede poseer también tareas sin requerimientos de tiempo real, denominadas Tareas de No Tiempo Real (TNTR). A este tipo de sistemas se lo denomina STR Heterogéneo (STRH). La ejecución de las TNTR no debe causar pérdida de vencimientos de las TTR, pero a su vez pueden tener algún requerimiento especial, como atención prioritaria. Para lograr una planificación concurrente eficiente, es necesario un método que aproveche el tiempo ocioso dejado por las TTR para la ejecución de las TNTR.

2.1. Introducción a los Métodos de Slack Stealing

Los métodos de SS permiten, en STR que cuentan con tiempo ocioso en algún momento de su ejecución, identificar y aprovechar parte del mismo [19, 20]. Este tiempo ocioso es denominado Slack Disponible (SD). Este es el tiempo que, en

un determinado intervalo, las TTR no utilizarán, suponiendo que cumplan con su peor caso de tiempo de ejecución. Este tiempo ocioso se puede aprovechar para ejecutar otros requerimientos, retrasando las TTR sin comprometer su planificabilidad. Por ejemplo, puede utilizarse para mejorar la calidad de servicio de tareas no criticas, que de otra manera tendrían que esperar a que los intervalos ociosos ocurran naturalmente (atención en *background*).

Varios trabajos han presentado implementaciones de SS [4, 17, 6, 21, 9], que difieren entre sí según si realizan el cálculo en tiempo de ejecución o de inicialización, o bien de manera exacta o aproximada. Los primeros métodos de cálculo exacto eran costosos, y su implementación en tiempo de ejecución, inviable. Sin embargo, trabajos más recientes presentan nuevas técnicas que reducen considerablemente el costo de cálculo [21].

3. Estado del Arte

Existen trabajos previos de implementación de métodos de SS en un SOTR. En [14] se realiza una implementación de una variante del algoritmo aproximado presentado en [3], en el SOTR MaRTE OS[2]. En [5] se desarrolla, también sobre MaRTE OS, una implementación del algoritmo exacto y de menor costo, propuesto en [21]. Un *framework* para la implementación de técnicas de SS sobre el SOTR FreeRTOS[3] es presentado en [15]. En los trabajos [12, 11, 13] se desarrolla una implementacion de un método aproximado sobre RTSJ (*Real-Time Specification for Java*).

El diseño e implementación de una TP fue presentado en [16], donde se describe además un desarrollo de referencia sobre FreeRTOS, que implementa un planificador basado en SS como ejemplo.

4. Estándar OSEK/VDX

El estándar OSEK/VDX[4] es un conjunto de normas para Sistemas Embebidos (SE) en automóviles, para el diseño del Sistema Operativo (SO), comunicación (COM), administración de redes (NM) y lenguaje de implementación (OIL). Nace de la fusión, en 1994, del estándar OSEK, impulsado por automotrices alemanas (BMW, Opel, VW, DaimlerChrysler, entre otras), y el proyecto VDX, desarrollado por Renault y PSA. Actualmente, la industria se encuentra migrando hacia un nuevo estándar, basado en OSEK/VDX, denominado AUTOSAR[5].

La especificación del estándar para el diseño del SO se denomina OSEK-OS. Especifica que el SOTR debe ser de tipo *estático*, lo que implica que las tareas, sus prioridades, alarmas, eventos, la cantidad de memoria a emplear, etc., son

[2] https://marte.unican.es
[3] http://www.freertos.org
[4] http://www.osek-vdx.org/
[5] http://www.autosar.org/

definidos previo a la compilación, durante un proceso denominado *generación*. La configuración del sistema se realiza mediante el lenguaje *OSEK Implementation Language* (OIL), y las funcionalidades particulares con el lenguaje C. Luego, mediante un *Generador* se transforma los archivos OIL en equivalentes C, y junto con los archivos fuente que implementan la funcionalidad, se compila un archivo binario, que contiene el sistema final.

4.1. Planificación de Tareas en OSEK/VDX

Una tarea provee un marco para la ejecución de un conjunto de acciones. El SOTR permite la ejecución concurrente de múltiples tareas, según una política de planificación. El estándar OSEK/VDX especifica dos políticas:

1. *Apropiativa*: la tarea actualmente en ejecución puede ser desalojada de la CPU por otra de mayor prioridad.
2. *No Apropiativa*: la tarea en ejecución no puede ser desalojada hasta que finalicen su ejecución o se bloquee.

Cada tarea tiene una *prioridad fija*, definida por el desarrollador, indicada mediante un valor entero. Cuanto más grande el valor, mayor prioridad tiene la tarea, siendo cero la prioridad más baja. Si dos o más tareas tienen idéntica prioridad, se las asigna a la CPU en el orden en que fueron activadas.

Una tarea en OSEK/VDX puede estar en alguno de los siguientes estados:

1. *Running*: la tarea está ejecutándose en la CPU.
2. *Ready*: la tarea está lista para ejecutar y a la espera de la CPU.
3. *Waiting*: la tarea está a la espera de un evento que reanuda su ejecución.
4. *Suspended*: la tarea finalizó, y está a la espera de ser activada nuevamente.

El estándar define dos tipos de tareas, *básicas* y *extendidas*. Las tareas *extendidas* pueden estar en cualquiera de los estados anteriores. Las tareas *básicas*, en cambio, no cuentan con el estado *Waiting*.

4.2. Eventos y Alarmas

Los *eventos* son un mecanismo de sincronización para las tareas *extendidas*. Una tarea puede esperar, en el estado *Waiting*, por un evento particular generado por otra tarea o por una alarma. Los eventos pueden ser despachados desde cualquier tipo de tarea, pero sólo una tarea *extendida* puede tener asignados eventos y esperar por uno.

Las *alarmas* son un mecanismo que permite activar tareas o lanzar eventos. Las alarmas pueden ser únicas (*one-shoot*) o bien ser periódicas. La programación de estas alarmas se basa en contadores (*Counters*), cuyo valor es expresado en *ticks* del sistema.

5. Plataforma de Desarrollo y Prueba

5.1. Lego Mindstorms NXT 2.0

El Lego Mindstorms es un kit de robótica educativa, con múltiples configuraciones, que cuenta con una computadora, denominada *brick*, programable en varios lenguajes, como NXT-G, C/C++, Java, Lua y Ada, entre otros. El *brick* cuenta con cuatro puertos para sensores, y tres para motores, una pantalla LCD, cuatro botones y conectividad Bluetooth y USB. Posee un microcontrolador Atmel AT91SAM7S256 de 32 bits, con 256 KB de memoria flash y 64 KB de RAM, y un coprocesador Atmel AVR ATmega48, con 4 KB de memoria flash y 512 bytes de RAM. Para este trabajo, se emplearon los siguientes sensores:

1. Un sensor ultrasónico, que permite medir la distancia hacia un objeto, hasta un máximo de 255 cm.
2. Un sensor táctil, de tipo pulsador, que envía una interrupción al *brick* al ser presionado.
3. Un giroscopio de un eje, con un resonador de cuarzo, que permite leer la velocidad de rotación unas 300 veces por segundo.

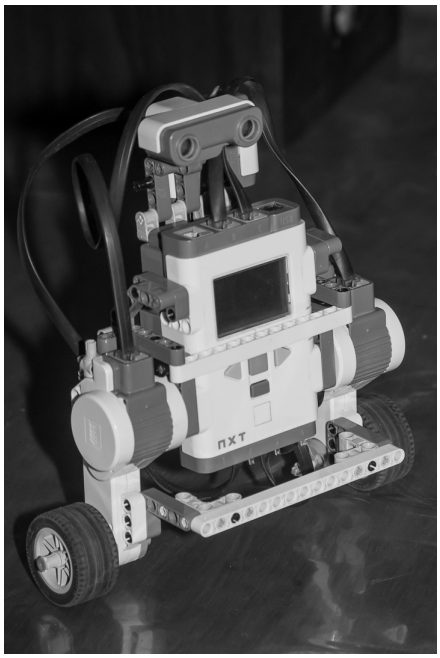

Figura 1. Lego Mindstorms NXT 2.0 en configuración *segway*.

5.2. Plataforma nxtOSEK

Para la implementación sobre el Lego Mindstorms se escogió, como implementacion específica del estándar OSEK/VDX, a nxtOSEK[6], que es una plataforma de código abierto especialmente diseñada para este *kit*. Provee una implementación de un SOTR bajo el estándar OSEK/VDX (TOPPERS/ATK), junto con un nuevo *firmware* provisto por el proyecto LeJOS, y la API ECRobot C/C++. Además, nxtOSEK permite utilizar TOPPER/JSP, que es una implementación del estándar μITRON.

Cabe notar que nxtOSEK no ofrece la posibilidad de administrar una interrupción por *hardware*. Por lo tanto, se implementó un *polling*, desde una de las tareas periódicas del sistema, para obtener el estado del sensor táctil.

5.3. Configuración y Modelo Físico del Segway

Se empleó la configuración *segway* del Lego Mindstorms, que es un péndulo invertido, que se mantiene erguido mediante el accionar de un controlador Proporcional-Integral-Derivativo (PID). Este controlador emplea un modelo definido por un conjunto de parámetros, dependientes de la forma y tamaño del *segway*.

Como diseño del controlador PID, se utilizó el modelo utilizado en el proyecto NXTway-GS[7]. Dado que se utilizaron ruedas de distinto radio, se realizaron modificaciones a los valores originales de los parámetros del modelo, y se lo recalculó mediante Matlab. Se consiguió así un balance más estable del péndulo con las nuevas ruedas. Dado que los motores que controlan las ruedas no trabajan de manera pareja, se realiza un cálculo adicional para corregir el error de sincronización entre ambos motores.

6. Diseño

La TP es la encargada de planificar las tareas del sistema, y se activa periódicamente cada 1 ms. El resto de las tareas, al finalizar la ejecución de una instancia, se suspenden indefinidamente. La TP las activa posteriormente, cuando corresponda, para que continúen su ejecución. Para esto hace uso del API[8] de nxtOSEK. Para el caso de las tareas periódicas, la TP las despertará periódicamente. En cambio, las tareas esporádicas serán activadas ante ciertos eventos y serán ejecutadas únicamente si existe suficiente SD.

Es importante notar que la TP solo realiza la activación de las instancias. Si la misma activa dos o más tareas, la asignación de la CPU a una instancia particular la realiza el planificador del SOTR, según las prioridades asignadas a las tareas.

[6] http://lejos-osek.sourceforge.net/

[7] https://la.mathworks.com/matlabcentral/fileexchange/19147-nxtway-gs–self-balancing-two-wheeled-robot–controller-design

[8] Application Programming Interface

6.1. Tareas del Sistema

La principal tarea es la TP, denominada *TaskPlanificadora*. Es de tipo *extendida*, y tiene la mayor prioridad del sistema (prioridad 6), por lo que siempre obtiene la CPU. La tarea se encuentra asociada al evento *EventPlanificadora*, que la ejecuta de manera periódica cada 1 ms mediante una alarma denominada *TaskPlanificadora_Alarm*. El detalle de su implementación se describe en la sección 7.1.

El resto de las tareas periódicas, todas de tipo *básica*, descriptas con detalle en la sección 7.2, son:

1. *TaskBalancer* (TTR τ_1), encargada de mantener el *segway* erguido. Tiene prioridad 4 y es crítica, ya que su fallo causa la pérdida de equilibrio del *segway*.
2. *TaskEco* (TTR τ_2), mide la distancia entre el *segway* y un obstáculo, y actuá si la distancia está por debajo de un umbral preestablecido. Ejecuta con prioridad 3.
3. *TaskBluetooth* (TTR τ_3), recibe comandos de movimiento desde una PC vía Bluetooth. Ejecuta con prioridad 2.
4. *TaskLCD* (TTR τ_4), presenta información acerca del funcionamiento del robot en la pantalla del *brick*. Ejecuta con prioridad 1.

Las tareas aperiodicas son activadas al ser presionado el sensor táctil, pero la TP las ejecuta sólo si existe SD. Así las tareas críticas puedan ser retrasadas sin que el *segway* pierda estabilidad. Estas tareas, descritas con más detalle en la sección 7.3, ejecutan con prioridad 5 y son:

1. *TaskAperiodicGiro* (TNTR τ_5), de tipo *extendida*, hace que el robot gire sobre su eje.
2. *TaskAperiodicTouch* (TNTR τ_6), de tipo *básica*, emite un sonido.

Cuadro 1. Tareas del Sistema.

Tarea	Tipo	Tipo (OSEK)	Prioridad	Periodo (ms)
TaskPlanificadora	TTR	Extendida	6	1
TaskAperiodicGiro	TNTR	Extendida	5	-
TaskAperiodicTouch	TNTR	Básica	5	-
TaskBalancer	TTR	Básica	4	4
TaskEco	TTR	Básica	3	20
TaskBluetooth	TTR	Básica	2	40
TaskLCD	TTR	Básica	1	200

6.2. Diseño de las Tareas

Las tareas del sistema cuentan con dos *modos* de ejecución:

1. *Configuración* (SETUP). Las tareas arrancan en este modo, desde el cual inicializan sus variables.
2. *Ejecución* (EXECUTE). Este es el modo normal de ejecución de la tarea, en el cual realizan sus actividades.

Además, una tarea puede estar en una *etapa* particular, que representa el estado de ejecución de la misma para la TP (que puede diferir de su estado en el SOTR). Las *etapas* son:

1. *Espera* (W): la instancia actual de la tarea terminó de ejecutarse, y está a la espera de su siguiente periodo o instancia.
2. *Ejecución* (E): la tarea se está ejecutando, y se mantendrá en este estado hasta que termine su ejecución, aún cuando haya sido desalojada.
3. *Lista* (L): la tarea queda en esta etapa al finalizar el modo SETUP.

Las tareas, cuando terminan de ejecutar el modo SETUP, quedan en la etapa *Lista*. La TP cambia a modo EXECUTE sólo si todas la tareas están en la etapa *Lista* (ver sección 7.1). En modo EXECUTE, una tarea cambia entre las etapas *Ejecución* y *Espera*.

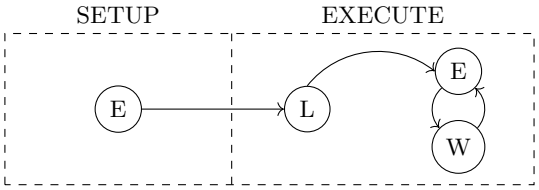

Figura 2. Modos y Etapas.

6.3. Control de Balance del Segway

Para el control del balance del *segway* se utilizó la implementación de NXTway-GS provista por nxtOSEK. Las funciones empleadas son:

1. *balance_init*. Inicializa las variables internas del API y define el punto de referencia inicial del giroscopio. Durante este proceso el *segway* debe estar quieto y en una posición erguida.
2. *balancer_control*. Implementa el sistema de control, encargado de establecer la posición actual del sistema,y realizar los cálculos para la corrección de las variables que controlan los motores. También se encarga de recibir los datos para realizar los movimientos de avance, retroceso y giro. Debe ejecutarse con un periodicidad de al menos 4 ms, para mantener en equilibrio el *segway*.

6.4. Cálculo de Slack Disponible

Se implementó el método de SS exacto presentado en [21], como un módulo que provee las siguientes funciones a la TP:

1. *SetupSlack*. Calcula los peores casos de tiempos de respuesta de cada tarea, requeridos para el cálculo del SD.
2. *CalculoSlack*. Calcula el SD de una tarea, usando el método presentado en [21], con el tiempo actual y el tiempo de ejecución de la misma como parámetros.
3. *SlackSistema*. Calcula el SD del sistema, para un instante de tiempo determinado, como el mínimo SD entre las tareas.
4. *CalculoSlackInicial*. Realiza el cálculo del SD de cada tarea en el instante inicial.
5. *ActualizacionContadoresSlack*. Actualiza los contadores de SD de cada tarea.

7. Implementación de las Tareas

7.1. Tarea Planificadora

En esta sección se presenta cómo funciona la TP en los distintos modos de ejecución.

En modo SETUP, la TP primero invoca la función *setupSlack*. Luego, activa el resto de las tareas para que ejecuten su modo SETUP, donde inicializan sus variables y estructuras de datos, y pasa a un estado *Waiting* durante 1010 ms, cediendo la CPU a las tareas. Este lapso es resultado de esperar 1000 ms para dar tiempo a la tarea *TaskBalancer*, más 10 ms para las restantes tareas. Si al finalizar este lapso de tiempo alguna tarea no se encuentra en la etapa *Lista*, la TP vuelve a pasar al estado *Waiting*, repitiendo el proceso hasta que todas estén en dicha etapa.

Cuando todas las tareas ejecutaron su modo SETUP, se realiza el cálculo del SD de cada una en el instante $t = 0$, y posteriormente se toma el tiempo de ejecución del *kernel*, el cual será la base para el tiempo de inicio del sistema. En este momento la TP pasa al modo EXECUTE, ejecutándose periódicamente cada 1 ms.

En cada instancia, en primer lugar obtiene el tiempo actual del sistema, menos el tiempo empleado en el modo SETUP. Luego, controla que tarea ejecutó, su estado actual, y el tiempo ejecutado por la misma hasta dicho instante. Dado que la granularidad del sistema es de 1 ms, esta será la resolución máxima de la medición del tiempo de ejecución. Luego calcula el SD de la tarea y del sistema.

Dado que hay situaciones en las que la TP instancia más de una tarea en el mismo *tick*, si una tarea periódica finaliza su ejecución en menos de 1 ms, es posible que otra tarea que se encuentre en estado *Ready* pase al estado *Running*, y finalice, también, antes del próximo *tick*. Consecuentemente, la próxima instancia de la TP debe procesar la finalización de una o más tareas. Para poder identificar cuáles tareas se ejecutaron, se mantiene un arreglo de etapas de

ejecución, donde cada tarea periódica, al finalizar, cambia su valor de LISTA a EJECUTADA. Cuando una o más tareas finalizaron, se realiza el cálculo del SD, de cada una, en ese *tick*. Los contadores de las tareas que no hayan ejecutado, son reducidos en una unidad, y se vuelve a asignar LISTA a todas las variables del arreglo de etapas de ejecución.

Luego, se controlan las banderas de interrupción de las TNTR. En caso de haber alguna bandera levantada, y de existir suficiente SD, se activa la TNTR correspondiente a dicha interrupción. Solo se permite la ejecución de una sola instancia de cada TNTR.

En este punto, la TP define cuál tarea se debe ejecutar. Para esto utiliza una *cola de eventos futuros* que registra los *ticks* restantes para la próxima instancia de cada tarea. La lista se inicializa con los periodos de las tareas, y los mismos se reducen en una unidad con cada ejecución de la TP. Cuando uno o más de estos contadores llegan a cero, se activan las respectivas tareas, y se reinician con el valor de periodo correspondiente. Si el contador de una tarea llega a cero, y la misma aún se encuentra en la etapa *Ejecución*, significa que la misma no cumplió con su vencimiento.

7.2. Tareas Periódicas

TaskBalancer. Esta tarea ejecuta el PID provisto por NXTway-GS para mantener el *segway* en equilibrio, por lo que se ejecuta cada 4 ms. En el modo SETUP, invoca la función *balance_init*, e inicializa los motores para tomar el punto de referencia inicial. Luego, toma los datos del giroscopio durante un segundo, y los promedia. El valor resultante es utilizado como la posición inicial del *segway*. En modo EXECUTE, invoca la función *balance_control* que chequea la posición del robot y realiza los cálculos del lazo de control y actúa sobre los motores para mantener el equilibrio.

TaskEco. Esta tarea mide la distancia entre el *segway* y un obstáculo, y tiene un periodo de 20 ms. En el modo SETUP no realiza ninguna acción. En el modo EXECUTE, mide la distancia a un obstáculo, y cuando la misma es menor a un umbral (determinado por un parámetro en tiempo de compilación), invoca la función *balance_control* para que el *segway* retroceda hasta que el obstáculo este fuera del rango.

TaskBluetooth. Esta tarea se encarga de la comunicación con una PC vía Bluetooth, desde la cual, mediante el software NXT GamePad, se puede cambiar la dirección de movimiento del robot. La tarea tiene un periodo de 40 ms. En el modo SETUP crea un *buffer* para almacenar temporalmente los comandos de movimiento entrantes. En el modo EXECUTE, recibe los comandos de movimiento y los guarda en dos variables que indican a *balance_control* si el robot debe debe avanzar o retroceder, y si debe girar a la derecha o la izquierda.

TaskLCD. Esta tarea se encarga de presentar información del sistema en la pantalla del *brick*, y cuenta con un periodo de ejecución de 200 ms. Dado que nxtOSEK no permite administrar interrupciones por *hardware*, desde esta tarea se realiza un *polling* del estado del sensor táctil. Si se detecta que fue pulsado, levanta una bandera que indica a la TP que debe activar la tarea aperiódica correspondiente.

7.3. Tareas Aperiódicas

TaskAperiodicGiro. Esta tarea se encarga de que el *segway* realice un giro sobre su eje vertical. Como su tiempo de ejecución es extenso, cede periódicamente la CPU. Durante su ejecución, establece el valor del giro del *segway*, utilizado por la función *balance_control*, para girar el *segway* una pequeña cantidad de grados y pasa a la etapa *Waiting*. Luego, la TP, en su próxima ejecución, si existe suficiente SD en el sistema, reanuda la ejecución de la tarea. De esta manera, la tarea requiere de múltiples activaciones para completar el giro deseado. Si se apropiara completamente del uso del CPU, el *segway* perdería el equilibrio. La TP, administrando el tiempo ocioso, permite que la misma cumpla con su ejecución, retrasando tareas críticas dentro de límites aceptables.

TaskAperiodicTouch. Esta tarea se encarga de emitir un sonido cuando es presionado el sensor táctil. Solo es activada si existe suficiente SD en el sistema. Tiene la misma prioridad que la tarea TaskAperiodicGiro, y es planificada por la TP de la misma manera.

7.4. Ejemplo de Ejecución

A continuación se presenta un ejemplo de ejecución del sistema, representado en el gráfico 3. Al inicio, la TP instancia las tareas periódicas para que ejecuten su modo SETUP, y espera por 1010 ms. Al cabo de este tiempo, todas las tareas quedan en etapa lista (el eje t no está a escala). Al despertar, la TP terminará de ejecutar su modo SETUP, y queda a la espera de ser activada nuevamente por la alarma periódica. Esto se da en el instante $t = 2$ (2 ms), donde la TP activa todas las tareas, que ya están en modo EXECUTE, constituyendo el *instance crítico* del sistema. Más adelante, en el instante $t = 200$, la tarea TaskLCD (TTR τ_4) detecta que se presionó el sensor táctil, y levanta una bandera. En la siguiente ejecución de la TP, como existe suficiente SD, se ejecuta la TNTR. Notar que su ejecución retrasa la siguiente instancia de la tarea TaskBalancer (TTR τ_1), sin que la misma pierda su vencimiento.

8. Conclusiones y Trabajos Futuros

Este trabajo comprueba la factibilidad de implementación de un método de SS dinámico y exacto, para administrar el tiempo ocioso, junto con un esquema de planificación mediante una TP, en un SOTR basado en el estándar

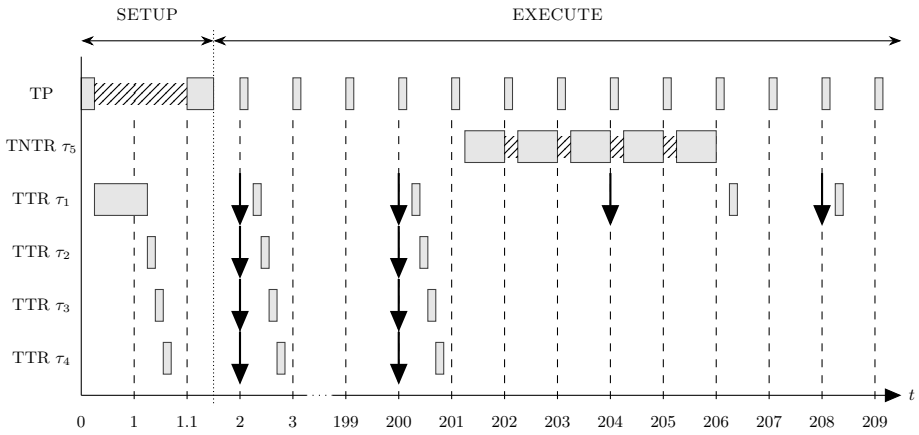

Figura 3. Ejemplo de una traza de ejecución del sistema.

OSEK/VDX, que permitió implementar una nueva política de planificación sin modificar el SOTR.

Las implementaciones disponibles del estándar no ofrecen métodos para la administración del tiempo ocioso, más allá del uso de tareas en segundo plano. La técnica de SS permite planificar tareas esporádicas y aperiódicas, pero también implementar técnicas como ahorro de energía, tolerancia a fallos, etc. La TP permite el desarrollo de sistemas heterogéneos, donde se puedan ejecutar tareas sin requerimientos estrictos de tiempo real de manera concurrente a las tareas críticas.

Como trabajos futuros, se planea replicar la implementación en otros SOTR basados en el estándar OSEK/VDX, como por ejemplo FreeOSEK[9], empleado en el proyecto CIAA, o en otros estándares, más modernos, como AUTOSAR. También se trabajará en la integración del método de administración de tiempo ocioso directamente en el núcleo del SOTR, para un mejor desempeño y ahorro de recursos del sistema.

Agradecimientos

Los autores quieren agradecer a los revisores por sus oportunas correcciones y recomendaciones.

Referencias

[1] Bini, E., Buttazzo, G.C., Buttazzo, G.M.: A Hyperbolic Bound for the Rate Monotonic Algorithm. In: 13th Euromicro Conference on Real-Time Systems (ECRTS 2001), 13-15 June 2001, Delft, The Netherlands, Proceedings. pp. 59–66. IEEE Computer Society (2001), https://doi.org/10.1109/EMRTS.2001.934000

[9] https://github.com/ciaa/Firmware/

[2] Burns, A.: Scheduling hard real-time systems: a review. Software Engineering Journal 6(3), 116–128 (1991), https://doi.org/10.1049/sej.1991.0015
[3] Davis, R.I.: Approximate slack stealing algorithms for fixed priority pre-emptive systems. University of York, Department of Computer Science (1993)
[4] Davis, R.I., Tindell, K., Burns, A.: Scheduling slack time in fixed priority pre-emptive systems. In: Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, December 1993. pp. 222–231 (1993), http://dx.doi.org/10.1109/REAL.1993.393496
[5] Díaz, L.A., Páez, F.E., Urriza, J.M., Orozco, J.D., Cayssials, R.: Implementación de un Método de Slack Stealing en el Kernel de MaRTE OS. In: Proceeding XLIII Jornadas Argentinas de Informática e Investigación Operativa (43 JAIIO) - III Argentine Symposium on Industrial Informatics (SII). pp. 13–24 (2014)
[6] José Manuel Urriza, Javier D. Orozco, R.C.: Fast Slack Stealing methods for Embedded Real Time Systems (2005)
[7] Joseph, M., Pandya, P.K.: Finding Response Times in a Real-Time System. Comput. J. 29(5), 390–395 (1986), http://dx.doi.org/10.1093/comjnl/29.5.390
[8] Leung, J.Y., Whitehead, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. Perform. Eval. 2(4), 237–250 (1982), https://doi.org/10.1016/0166-5316(82)90024-4
[9] Lin, C., Brandt, S.A.: Improving Soft Real-Time Performance through Better Slack Reclaiming. In: Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005), 6-8 December 2005, Miami, FL, USA. pp. 410–421. IEEE Computer Society (2005), http://dx.doi.org/10.1109/RTSS.2005.26
[10] Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM 20(1), 46–61 (1973), http://doi.acm.org/10.1145/321738.321743
[11] Masson, D., Midonnet, S.: Slack time evaluation with RTSJ. In: Wainwright, R.L., Haddad, H. (eds.) Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008. pp. 322–323. ACM (2008), http://doi.acm.org/10.1145/1363686.1363769
[12] Masson, D., Midonnet, S.: Userland Approximate Slack Stealer with Low Time Complexity. In: RTNS 2008. pp. 29–38. Rennes, France, France (Oct 2008), https://hal-upec-upem.archives-ouvertes.fr/hal-00620349
[13] Midonnet, S., Masson, D., Lassalle, R.: Slack-time computation for temporal robustness in embedded systems. Embedded Systems Letters 2(4), 119–122 (2010), http://doi.ieeecomputersociety.org/10.1109/LES.2010.2074184
[14] Minguet, A.R.E.: Extensiones al Lenguaje Ada y a los Servicios POSIX para Planificación en Sistemas de Tiempo Real Estricto. Ph.D. thesis, Universidad Politécnica de Valencia (2003)
[15] Paez, F., Urriza, J.M., Cayssials, R., Orozco, J.D.: Métodos de Slack Stealing en FreeRTOS. In: JAIIO (ed.) 44 Jornadas Argentinas de Informática (JAIIO). SADIO (2015)
[16] Páez, F.E., Urriza, J.M., Cayssials, R., Orozco, J.D.: FreeRTOS user mode scheduler for mixed critical systems. In: Embedded Systems (CASE), 2015 Sixth Argentine Conference on. pp. 37–42 (Aug 2015)
[17] Santos, R.M., Urriza, J.M., Santos, J., Orozco, J.: New methods for redistributing slack time in real-time systems: applications and comparative evaluations. Journal of Systems and Software 69(1-2), 115–128 (2004), http://dx.doi.org/10.1016/S0164-1212(03)00079-7

[18] Stankovic, J.A.: Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. Computer 21(10), 10–19 (Oct 1988), http://dx.doi.org/10.1109/2.7053

[19] Thuel, S.R., Lehoczky, J.P.: Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems Using Slack Stealing. In: Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994. pp. 22–33. IEEE Computer Society (1994), http://dx.doi.org/10.1109/REAL.1994.342733

[20] Tia, T.S., Liu, J.W.S., Shankar, M.: Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-priority Preemptive Systems. Real-Time Syst. 10(1), 23–43 (Jan 1996), http://dx.doi.org/10.1007/BF00357882

[21] Urriza, J., Paez, F., Cayssials, R., Orozco, J., Schorb, L.: Low cost slack stealing method for RM/DM. International Review on Computers and Software 5(6), 660–667 (2010)

[22] Urriza, J., Paez, F., Orozco, J., Cayssials, R.: Computational Cost Reduction for Real-Time Schedulability Tests Algorithms. IEEE Latin America Transactions 13(12), 3714–3723 (Dec 2015)