DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE BAJO CONSUMO

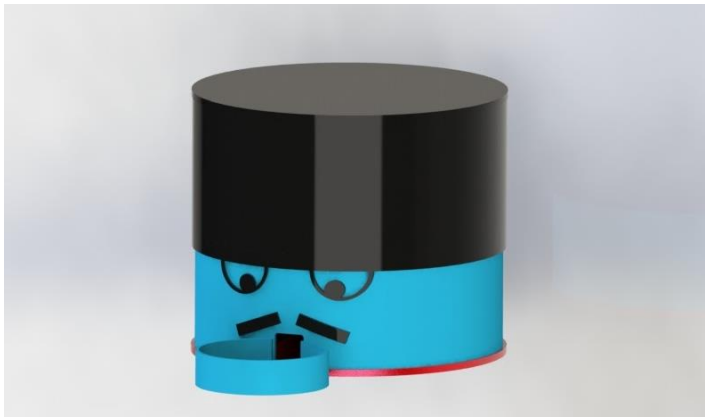
Borja Ortíz Prieto, Álvaro Mora Fraile, Javier Vargas García

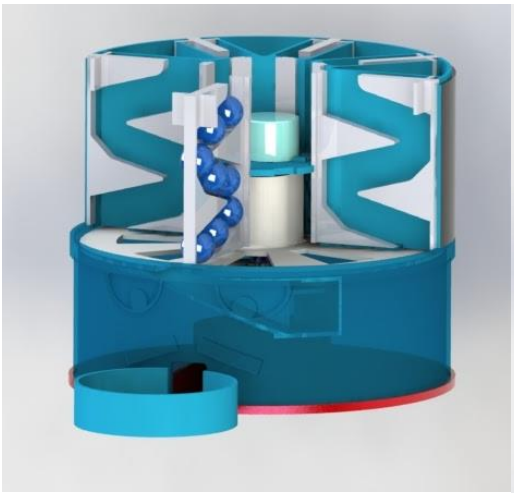
PLANTEAMIENTO DEL SISTEMA

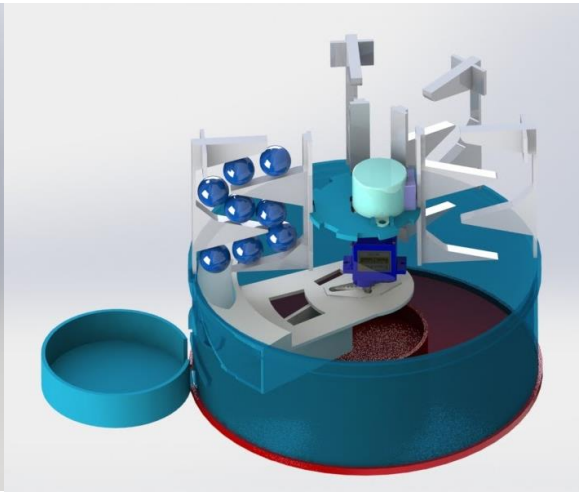
El proyecto consiste en un pastillero automático capaz de suministrar las dosis de medicamentos y avisar al usuario de su toma, con el objetivo de proporcionar un seguimiento del tratamiento facilitando así la labor de cuidadores y médicos.

En este diseño se realiza una implementación de bajo consumo aplicando un algoritmo YDS sobre un ejecutivo cíclico para las tareas de: lectura de sensores, parpadeo de un led indicador, y el control de movimiento de motores. La funcionalidad principal reside en esta última mencionada, la cual se encarga de seguir una secuencia de movimientos accionando un servo y motor paso a paso en base a una monitorización de los sensores para un correcto funcionamiento.

A continuación, se muestra el diseño mecánico realizado, permitiendo así establecer el contexto de ejecución del modelo.







Se observa un total de 8 compartimentos de pastillas independientes que encajan sobre un cilindro rotatorio impulsado por un motor paso a paso. Dicho motor se encarga de colocar el compartimento correspondiente en la posición de reparto mientras que un servo deja caer la pastilla. Durante el proceso se añaden sensores para detectar la presencia del vaso, la apertura de la tapa superior, y la caída de la pastilla.

La interfaz externa debe recoger la información del número de pastillas a repartir por cada compartimento, y realizar su reparto teniendo en cuenta las especificaciones que se plantean a continuación.

ESPECIFICACIONES

Se describen las especificaciones funcionales que permitirán desarrollar sentencias LTL para su futura verificación. Se recogen funcionalidades que contemplan la interacción con el usuario durante el reparto de la pastilla.

ID	*DESCRIPCIÓN*
E1	*Los sensores monitorizan el estado del vaso, la puerta de recarga, y la caída de la pastilla durante el reparto.*
E2	*Un led indicador parpadea durante el reparto de la pastilla.*
E3	*En el caso de que la puerta de recarga esté abierta, no se debe realizar el reparto.*
E4	*En el caso de que el vaso no esté colocado, el servo no debe dejar caer la pastilla.*
E5	*En caso de no ser detectada la pastilla, un máximo de 3 intentos deben ser ejecutados.*
E6	*Si el sensor de puerta no detecta que se cierra o el vaso no es colocado, tras un tiempo de espera, se cancela el reparto y el usuario es alertado con una alarma.*
E7	*El aviso de la toma no se realiza si no se han dispensado pastillas.*
E8	*Tras finalizar el reparto, si alguna pastilla es dispensada, el usuario es alertado de que debe recoger la toma.*
E9	*Si la toma no es recogida, tras un tiempo de espera, se alerta de un error.*

MODELADO DE LAS MÁQUINAS DE ESTADO

Se modelan tres máquinas de estado independientes activadas por tiempo. Utilizan una variable compartida denominada *repartidor_on*, que se establece a uno durante todo el tiempo que el repartidor esté ocupado, sirviendo así como flag de activación para el led y los sensores. Los sensores utilizan variables compartidas para devolver el valor de las lecturas.

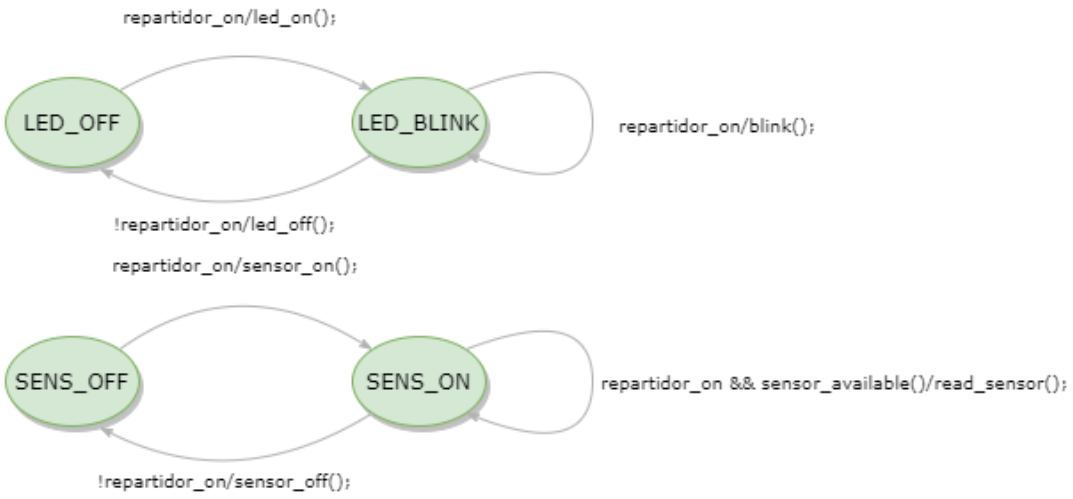
En el esquema, el usuario interactúa con el repartidor, indicando el número de pastillas y las posiciones donde se encuentran.

FSM de leds y sensores:

Son máquinas con tan solo dos estados posibles *activado / desactivado.*

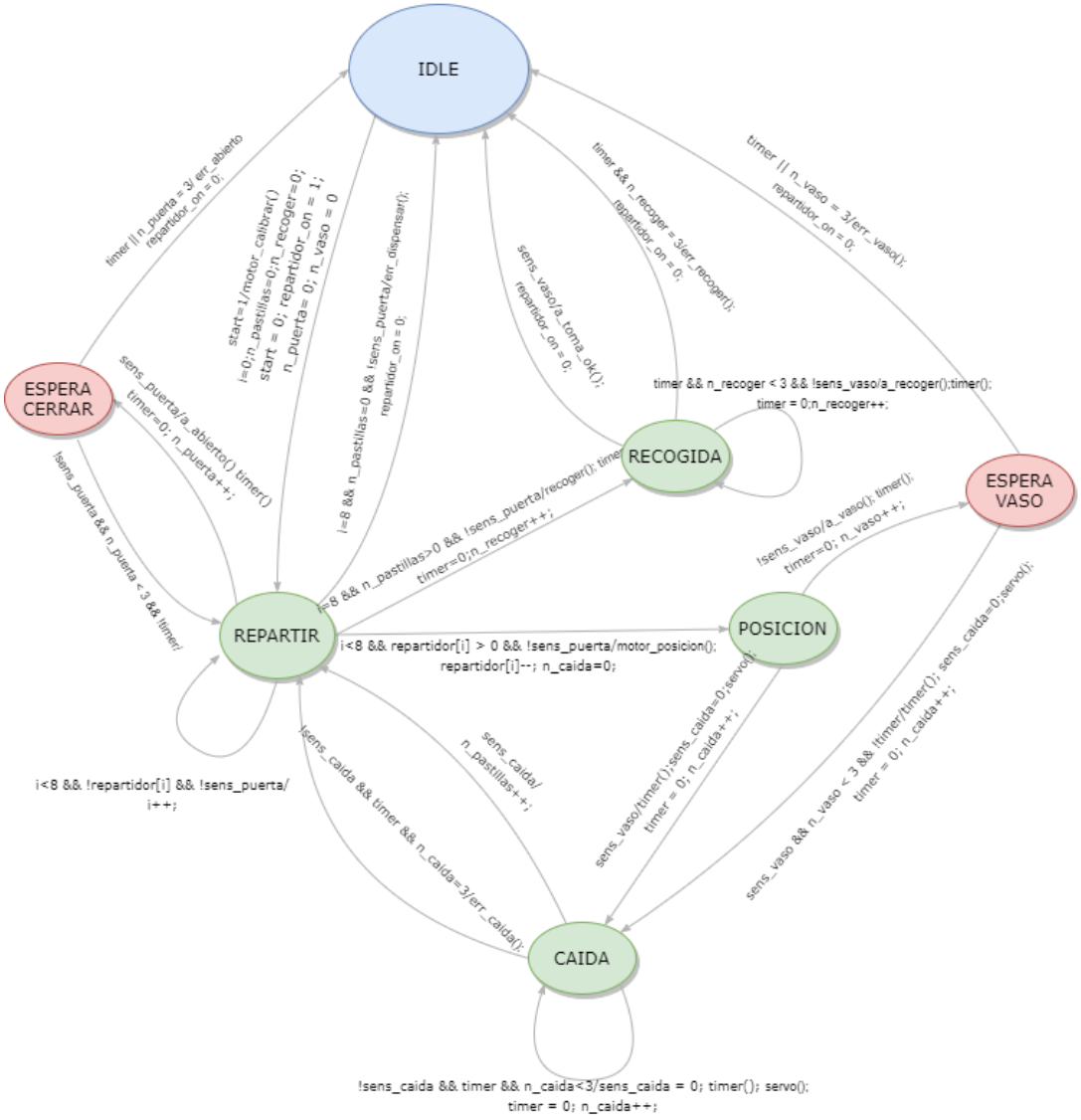
El led utiliza la activación por tiempo para hacer cambiar el estado del led con la función *blink(),* de forma que el periodo de parpadeo coincide con el periodo de ejecución.

Los sensores son solo monitorizados durante el reparto de la pastilla, permitiendo así reducir el consumo. Se observa una función *sensor_available()* que devuelve '1' al detectar algún sensor disponible, y *sensor_read()*, que lee el valor de los sensores, y escriben en la variable compartida *sens_abierto, sens_vaso, y sens_caida.* Utilizada por el repartidor durante su ejecución.



FSM de repartidor:

Realiza la funcionalidad principal del sistema. La interfaz con el exterior es definida con un vector denominado *repartidor[8]*, donde cada posición corresponde a un compartimento determinado, y el número indica la cantidad de pastillas a repartir. Se utiliza la variable *start* para inicializar el sistema.



Las variables internas *i, n_caida, n_pastillas, n_recoger, n_puerta* y *n_vaso,* se utilizan para mantener la funcionalidad interna.

Para el control de motores se utilizan funciones externas a la fsm: *motor_calibrar(),* establece la referencia 0 de posición, *servo()*, encargada de dejar caer una pastilla, y *motor_posicion()*, que mueve el motor paso a paso hasta el compartimento *i*.

Los avisos y errores del usuario también quedan reflejados con llamadas a función, dejando así libertad en la implementación de la respuesta. Para el conjunto de test realizados, las funciones externas utilizan *printf* para dejar las trazas de ejecución.

VERIFICACIÓN FORMAL CON SPIN

El diseño del modelo se ha realizado tendiendo en cuenta el conjunto de especificaciones a cumplir, no obstante, es susceptible de tener fallos que impidan alcanzar los objetivos planteados, por ello se recurre a una verificación formal definiendo el modelo en *promela* y verificando sentencias *LTL* con spin.

Se definen cuatro procesos *active proctype,* tres para las máquinas de estado, y un cuarto que simula el entorno. Creando así un sistema cerrado que permita simular las posibles ejecuciones. En el entorno se tiene en cuenta el estado de encendido de sensores para seleccionar de forma aleatoria valores de los mismos.

El uso de un *timer* se simula junto con el entorno. Creando dos variables *timer_start* y *timer_end* para la habilitación y respuesta respectivamente. Para evaluar su funcionamiento, debemos poner como condición que *timer_end* siempre se activa tras *timer_start*, de no ser así, los estados de espera nunca tendrían retorno, lo cual supone un problema a la hora de verificar sentencias LTL. Dicha condición no supone un problema, tan solo debemos asegurar en la implementación que el *timer* funciona correctamente. A la hora de realizar las especificaciones, se añada *[]timer_end* en algunas sentencias para realizar la comprobación solo cuando el *timer* se mantiene como finalizado.

Traducción de especificaciones:

Previo a la aplicación de las especificaciones funcionales planteadas, verificamos el comportamiento básico del sistema, asegurando que el sistema inicia y finaliza

El sistema se inicia, se alcanza *repartidor_on*

ltl inicio {<>(repartidor_on)}

El sistema finaliza, tras iniciarse

ltl fin {[](repartidor_on) | <>(repartidor_on & <> !repartidor_on))}

E1	*Los sensores monitorizan el estado del vaso, la puerta de recarga, y la caída de la pastilla durante el reparto.*

Si se actictiva *repartidor_on* se alcanza inevitablemente *SENSOR_ON.*

ltl sensores_on {[](!repartidor_on) | <>(repartidor_on & <> estado_sensor == SENSOR_ON))}

E2	*Un led indicador parpadea durante el reparto de la pastilla.*

Si se activa *repartidor_on* se alcanza inevitablemente *LED_BLINK.*

ltl led_on {[](!repartidor_on) | <>(repartidor_on & <> estado_led == LED_BLINK))}

E3	*En el caso de que la puerta de recarga esté abierta, no se debe realizar el reparto.*

En el caso de mantener la puerta abierta y el timer finalizado, nunca se alcanza el estado *POSICION*, encargado de mover los motores para realizar el reparto.

ltl error_abierto {[]([](sensor_puerta) && [](timer_end)) -> [](estado_repartidor != POSICION))}

E5	*En caso de no ser detectada la pastilla, un máximo de 3 intentos deben ser ejecutados.*

Detección correcta de la pastilla si el número de intentos es inferior a 3.

ltl detector_pastilla {[](estado_repartidor == CAIDA && sensor_caida && n_caida < 3) -> <>(estado_repartidor == REPARTIR && !err_caida))}

Si se alcanza el numero de intentos máximos, salta un error.

ltl detector_pastilla_error {[](estado_repartidor == CAIDA && n_caida == 3) -> <>(err_caida))}

E6	*El aviso de la toma no se realiza si no se han dispensado pastillas.*

Si no hay pastillas disponibles, el sistema no alcanza el aviso.

ltl error_dispensar{[]([](n_pastillas == 0) -> [](!aviso_toma))}

E7	*Tras finalizar el reparto, si alguna pastilla es dispensada, el usuario es alertado de que debe recoger la toma.*

Dicha especificación se considera siempre y cuando no haya ningún error, por ello aseguramos que no se puede mantener ningún error y que el timer ha finalizado para evitar bucles.

ltl alerta_recoger{[]((n_pastillas > 0 && [](!err_abierto) && [](!err_vaso) && [](timer_end)) -> <>(aviso_toma))}

El aviso de que la toma se ha realizado correctamente, se envía si se retira el vaso tras entrar en el estado *RECOGIDA.*

ltl recogida_correcta{[](((estado_repartidor == RECOGIDA) && [](sensor_vaso)) -> <>(aviso_recogida_ok))}

E8	*Si la toma no es recogida, tras un tiempo de espera, se alerta de un error.*

Tras entrar en el estado *RECOGIDA*, si no se vuelve a detectar el vaso y el timer finaliza, se activa un error.

ltl error_recogida{[](((estado_repartidor == RECOGIDA) && [](!sensor_vaso) &&[](timer_end)) -> <>(err_recoger))}

Evaluación de resultados

Tras aplicar *spin* se han obtenido algunas observaciones que permiten mejorar el modelo. El sistema incluye varios bucles por construcción, donde estados donde se espera a cerrar la puerta o a colocar el vaso pueden suponer un *loop* si el sensor se activa y desactiva constantemente. Para evitarlo, se han añadido contadores (*n_vasos* y *n_puerta*) para cada entrada a estos estados (de color rojo en el diagrama del modelo), de esta forma solo se puede entrar un número limitado de veces, que tras superarlo, se vuelve a *IDLE,* independientemente del *timer*.

También se han solucionado problemas de determinismo en la evaluación de condiciones, lo cual complicaba la verificación formal. Como solución se asegura que solo una condición de guarda puede cumplirse al mismo tiempo por estado.

IMPLEMENTACIÓN

La implementación se realiza aplicando un ejecutivo cíclico sobre las tres tareas para la activación de las máquinas de estado. Se utiliza la librería *fsm.h* que permite una integración directa del modelo utilizado en *spin*, permitiendo así mantener la validez de la verificación tras la implementación.

Los sensores se simulan utilizando el teclado, dentro de la máquina de estados correspondiente, la *sensor_available()* devuelve un valor distinto de cero si se detecta una entrada del teclado, y *sensor_read()* se encarga de leer la tecla pulsada y traducirla a un valor de las variables de sensores compartidas.

La estructura del código fuente se basa en ficheros independientes agrupados por funcionalidad, si bien las máquinas de estado se definen en *.c* distintos, las funciones de motores y alarma también se definen a parte. Puesto que el programa se ejecuta en un ordenador personal, de nuevo esta interfaz de funciones se describe con *printf*.

El sistema se inicializa con la ejecución del programa pasándole como argumento el vector de reparto deseado, por ejemplo: *./main 1 4 3* ejecuta el reparto de una pastilla del primer compartimento seguido de 4 y 3 pastillas para el segundo y tercero, manteniendo la interacción durante la ejecución con un teclado.

Se puede observar que el modelo simulado en *spin* es totalmente equivalente al modelo implementado, con el único cambio equivalente de realizar una lectura real en la máquina de sensores mientras se mantienen encendidos, lo cual fue contemplado en el entorno simulado de *spin.*

Planificación YDS

Puesto que no se tienen especificaciones concretas de tiempo real, se establecen unos periodos igual a los plazos coherentes para una fluidez en la ejecución. Se trata de una máquina de control que no realiza cálculos complejos, por lo que no debemos tener problemas en la planificación.

Se han obtenido los tiempos máximo de cómputo de forma aproximada, ejecutando las máquinas de estado y midiendo los tiempos de ejecución de *fsm_fire()*, tomando el mayor como caso peor para la planificación.

Tarea	Cómputo A $Fmax$ (ms)	Periodo (ms)	Plazo (ms)
Repartidor	0,2	100	100
Sensor	0,25	100	100
Led	0,05	500	500

Se obtiene el siguiente hiperperiodo:

$$T = mcm(100, 100, 500) = 500ms$$

Se analizan los intervalos de máxima intensidad, tomando grupos de tareas en intervalos que cubren sus plazos y dividiendo tiempos de cómputo entre plazos.

$$SENSOR + REPARTIDOR \rightarrow G[0, 100] = \frac{0.25 + 0.2}{100} = 0.0045$$

$$SENSOR + REPARTIDOR + LED \rightarrow G[0, 1000] = \frac{0.25 + 0.2 + 0.05}{1000} = 0.0005$$

Observamos como las intensidades son muy reducidas, lo cual va a significar mantener la frecuencia de la CPU al mínimo durante la mayor parte del tiempo. Siguiendo con el algoritmo YDS, obtenemos la máxima intensidad agrupando el sensor y el repartidor en el mismo intervalo.

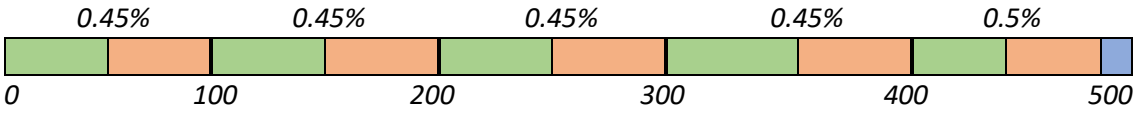
$$F_{sensor} = F_{repartidor} = 0.45\% * Fmax$$

Aplicando de nuevo el proceso con las tareas restantes del resto de periodos, obtenemos los mismos resultados para periodos secundario 1º, 2º y 3º. Quedándonos en el último periodo:

$$SENSOR + REPARTIDOR + LED \rightarrow G[0, 1000] = \frac{0.25 + 0.2 + 0.05}{100} = 0.005$$

$$F_{sensor} = F_{repartidor} = F_{led} = 0.5\% * Fmax$$

Como es lógico, en el último periodo secundario se ejecutan las 3 tareas, lo que supone un pequeño incremento de la frecuencia para alcanzar los plazos establecidos. En la siguiente tabla quedan reflejados los resultados.



Tiempos de respuesta

$$R_{sensor} = max\left[\frac{C_{fmax}}{0.45\%}, \frac{C_{fmax}}{0.5\%}\right] = \frac{0.25}{0.0045} = 55.55ms$$

$$R_{repartidor} = max\left[\frac{C_{fmax}}{0.45\%}, \frac{C_{fmax}}{0.5\%}\right] = \frac{0.20}{0.0045} = 44.45ms$$

$$R_{led} = 500ms$$

Observamos como los tiempos de respuesta son igual o menor a los periodos, por lo que podemos asegurar los plazos de ejecución.

En el código ejecutado sobre un ordenador se utiliza una función *set_frequency()* suponiendo que es capaz de establecer la frecuencia de ejecución de la CPU. En una aplicación implementada sobre un microcontrolador, habría que utilizar la frecuencia mínima disponible justo por encima de los valores obtenidos, no obstante <1% supone una frecuencia demasiado baja, seguramente mucho menor de la disponible, por lo que talvez interese ejecutar el algoritmo a máxima frecuencia para reducir tiempos de cómputo y maximizar el tiempo de dormido de la CPU.

OPTIMIZACIÓN DE MEMORIA

Utilizamos *Valgrind* con la herramienta *cachegrind* para obtener estadísticas de uso de memoria. Ejecutando para una única pastilla y para todos los compartimentos, obtenemos:

valgrind --tool=cachegrind ./main 1

```
==2896== I   refs:      321,786
==2896== I1  misses:      1,402
==2896== LLi misses:      1,263
==2896== I1  miss rate:    0.44%
==2896== LLi miss rate:    0.39%
==2896==
==2896== D   refs:      106,709  (83,121 rd   + 23,588 wr)
==2896== D1  misses:      4,286  ( 3,396 rd   +    890 wr)
==2896== LLd misses:      3,226  ( 2,426 rd   +    800 wr)
==2896== D1  miss rate:     4.0% (   4.1%     +    3.8%  )
==2896== LLd miss rate:     3.0% (   2.9%     +    3.4%  )
==2896==
==2896== LL refs:         5,688  ( 4,798 rd   +    890 wr)
==2896== LL misses:       4,489  ( 3,689 rd   +    800 wr)
==2896== LL miss rate:      1.0% (   0.9%     +    3.4%  )
```

valgrind --tool=cachegrind ./main 2 2 2 2 2 2 2

```
=3164== I   refs:      416,927
=3164== I1  misses:      1,549
=3164== LLi misses:      1,263
=3164== I1  miss rate:    0.37%
=3164== LLi miss rate:    0.30%
=3164==
=3164== D   refs:      143,854  (109,097 rd   + 34,757 wr)
=3164== D1  misses:      4,283  (  3,393 rd   +    890 wr)
=3164== LLd misses:      3,226  (  2,426 rd   +    800 wr)
=3164== D1  miss rate:     3.0% (    3.1%     +    2.6%  )
=3164== LLd miss rate:     2.2% (    2.2%     +    2.3%  )
=3164==
=3164== LL refs:         5,832  (  4,942 rd   +    890 wr)
=3164== LL misses:       4,489  (  3,689 rd   +    800 wr)
=3164== LL miss rate:      0.8% (    0.7%     +    2.3%  )
```

Se observa como al realizar el proceso de repartir más veces (derecha), obtenemos un mayor número de aciertos de caché, tanto de datos como de instrucciones. Para analizar diferentes configuraciones de cache aplicamos la herramienta *cache-sim=yes.* Por ejemplo:

valgrind --tool=callgrind --cache-sim=yes --I1=2048,2,32 --D1=1028,2,64 --LL=8192,1,32

Para bloques de 32 bytes y asociatividad de 2:

Caché I1 (2,32)	Caché D1 (2,32)	Caché LL (2,32)	I1 miss (%)	D1 miss (%)	LL miss (%)
4096	4096	8192	2.94	13.8	4.3
2048	2048	8192	5.13	19.1	5.2
1024	1024	4096	6.5	27.3	8.5

Incrementando el tamaño de bloques a 64 bytes en D1 y LL:

Caché I1 (2,32)	Caché D1 (2,64)	Caché LL (2,64)	I1 miss (%)	D1 miss (%)	LL miss (%)
4096	4096	8192	2.94	12.7	3.4
2048	2048	8192	5.02	17.6	4.1
1024	1024	4096	6.5	23.4	6.7

Observamos como al incrementar el tamaño de bloque, mejora el número de aciertos para una misma capacidad. Tras analizar diferentes configuraciones, se obtiene que un punto aceptable de aciertos frente a capacidad de memoria. Incrementando asociatividad tamaños de bloque, mejoramos resultados.

Caché I1 (4,128)	Caché D1 (4,128)	Caché LL (4,256)	I1 miss (%)	D1 miss (%)	LL miss (%)
1024	4096	2048	3.03	10.8	4.0

Podemos ver exactamente el uso de memoria sobre el código en C utilizando el:

callgrind_annotate --auto=yes callgrind.out.xxxx main.c

Observamos el siguiente uso de memoria en la ejecución de las máquinas de estado:

```
    .   .   .    .    .    .    .    .    .              //Ejecutivo ciclico
  252   .   .    .    .    .    .    .    .              switch (frame) {
    .   .   .    .    .    .    .    .    .                      case 4:
    .   .   .    .    .    .    .    .    .                      set_frecuency(0.005);
   50   0  25    .    .    .    .    .    .                      fsm_fire (estado_sensor_fsm);
3,361 930 446  287   29    7  215   26    7  => /home/javier/Escritorio/src/fsm.c:fsm_fire (25x)
   50   0  25   11    0    0    6    .    .                      fsm_fire (estado_repartidor_fsm);
3,172 1,109  96   24   21    0   20   17    .  => /home/javier/Escritorio/src/fsm.c:fsm_fire (25x)
   50   0  25    .    .    .    .    .    .                      fsm_fire (estado_led_fsm);
5,093 1,356 716  386   69   14  307   66   14  => /home/javier/Escritorio/src/fsm.c:fsm_fire (25x)
    .   .   .    .    .    .    .    .    .                      break;
    .   .   .    .    .    .    .    .    .                      default :
    .   .   .    .    .    .    .    .    .                      set_frecuency(0.0045);
  202   0 101    1    0    0    1    .    .                      fsm_fire (estado_sensor_fsm);
12,427 3,466 1,654 1,067  119   31  801  107   30  => /home/javier/Escritorio/src/fsm.c:fsm_fire (101x)
  202   0 101   42    0    0   42    .    .                      fsm_fire (estado_repartidor_fsm);
45,176 13,320 4,626 1,860 1,888  240 1,445 1,598  206  => /home/javier/Escritorio/src/fsm.c:fsm_fire (101x)
  101   0   0   10    0    0    7    .    .                      break;
    .   .   .    .    .    .    .    .    .              }
1,512   0   0   55    0    0   55    .    .              frame = (frame + 1) % 5;
```

Las tres primeras columnas corresponden a lectura de instrucciones, lectura de datos y escritura de datos respectivamente. Claramente obtenemos un mayor uso con la máquina de estados del repartidor, puesto que tiene más estados y condiciones que comprobar. Aquí el uso del *printf* queda reflejado por un mayor uso de memoria, puesto que no existe procesamiento de datos, no es posible optimizar accesos recurrentes o bucles.

Podemos usar eCACTI para obtener la potencia de acceso para la tecnología de cache donde se implemente, con la potencia de los accesos y los tiempos de acceso, podríamos obtener una medida del consumo de energía. En el actual diseño, tratamos de minimizar los tiempos de acceso como método de optimización del consumo.