INGENIERÍA DE SISTEMAS ELECTRÓNICOS

1. Diseño orientado a modelos (FSM)

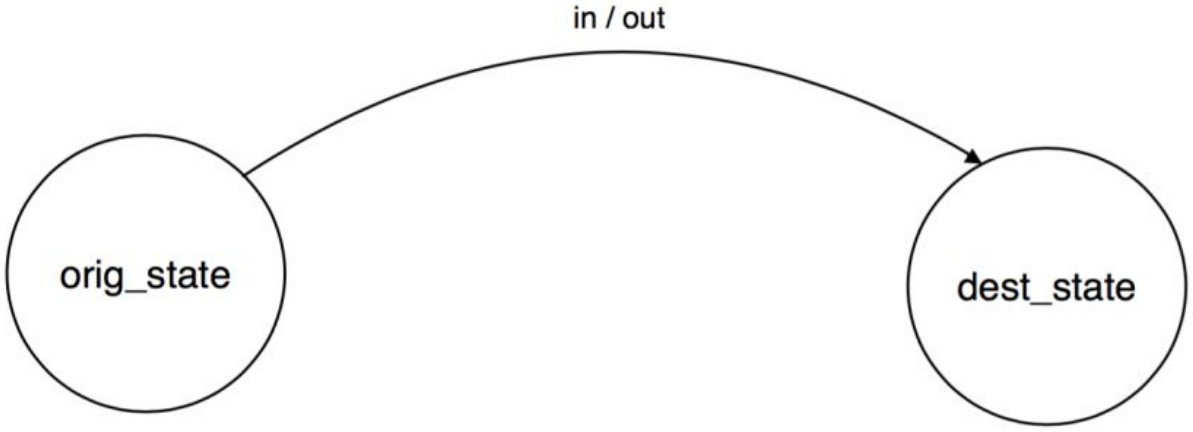
- **Introducción**

  Las máquinas de estados permiten dar soluciones a problemas definiendo una serie de estados en los que se puede encontrar el sistema. Se definen con un conjunto de entradas, salidas, estados y transiciones. En función del estado actual y de las entradas del sistema, se calcula el siguiente estado.
  Un estado se puede definir como un resumen del pasado que tiene efecto en las reacciones del sistema.

- **Ventajas del uso de FSM**
  - Describe el comportamiento del sistema sin ambigüedades
  - Se puede demostrar formalmente la corrección del modelo
  - La implementación es automática desde el modelo
  - El sistema se puede analizar con facilidad para asegurar que cumple los requisitos en el caso peor

- **¿Qué máquinas de estados vamos a utilizar?**

  Las máquinas de estados de **Mealy** se caracterizan porque las salidas del sistema dependen del estado actual y de las entradas al sistema, es decir, de las transiciones.



Utilizamos máquinas de estados **síncronas**, por lo que los cambios de estado sólo se producen cuando hay un flanco de reloj. Las utilizamos porque con mucho más fáciles de analizar y de asegurar que el tiempo de ejecución en el caso peor está dentro de los plazos.

- ## Funciones de guarda o de comprobación

  Las funciones de guarda **comprueban si se han producido las condiciones** para realizar una transición. Es importante que no generen efectos de lado porque no sabemos ni en qué orden ni cuántas veces se van a evaluar.

  Es decir, no pueden asignarse valores a variables ni invocar a funciones que cambien el estado de alguna manera. **Únicamente deben leer variables.**

- ## Máquinas de estados extendidas

  Las máquinas de estados finitas se pueden extender con el uso de **variables** en las funciones de guarda (ej. código en alarma). Cuando añadimos una variable, su valor pasa a formar parte del estado de la FSM. Por tanto el número de estados del sistema se multiplica por el número de valores diferentes que puede tener la variable. Esto **complica la verificación** formal del modelo.

  Sin embargo al añadir variables se **simplifica** enormemente la máquina de estados, por lo que es más fácil de entender, mantener, extender y componer.

- ## Máquinas de estados no deterministas

  Las máquinas de estados no deterministas las **transiciones** se toman de manera **aleatoria** entre aquellas en las que la condición de guarda se verifica.

  Este modelo de computación es equivalente al de las máquinas de estados deterministas cuando siempre hay una única transición posible.

  Pero este modelo funciona y es sencillo de entender porque las condiciones de guarda no alteran el estado de ninguna forma. No podemos saber cuántas veces se evalúan las condiciones ni en qué orden, pero eso no afecta al funcionamiento.

- ## Tipos de FSM

  - **Concurrente**: Síncrona / asíncrona. Variables compartidas.
  - **Jerárquicas**: Transiciones tipo desalojo y reset.
  - **Síncrono-Reactivo**
  - **Dataflow**

2. Sistemas de tiempo real

- **Introducción**

  En este punto nos centramos en el análisis de sistemas para verificar si cumple los **requisitos temporales**. Para ello vamos a planificar el uso de recursos mediante métodos de planificación.
  Estos métodos se caracterizan por tener:
  - Un **método de análisis** que permite calcular el comportamiento temporal del sistema, para estudiar el caso peor
  - Un **algoritmo de planificación** que determina el orden de acceso de las tareas a los recursos del sistema

  El sistema operativo de tiempo real que utilizamos es **FreeRTOS**.

  Un método de planificación puede ser
  - **Estático:** el análisis se puede hacer antes de la ejecución
  - **Dinámico:** el análisis se hace durante la ejecución

  Sintaxis:

N	Número de tareas
T	Período de activación
C	Tiempo de ejecución máximo
D	Plazo de respuesta
R	Tiempo de respuesta máximo
P	Prioridad

  Para un **modelo simple** todas las tareas cumplen:

  $$C_i \leq D_i = T$$

  El **objetivo** es asegurar que se cumpla:

  $$R_i \leq D_i$$

  El **hiperperiodo** se define como el periodo temporal tras el que se vuelve a repetir cíclicamente el comportamiento del sistema:

  $$H = mcm\left(T_i\right)$$

- ## Planificación con ejecutivos cíclicos

  Este tipo de planificación se utiliza si las tareas son periódicas y sus periodos son **armónicos**. Si no fueran armónicos no sería eficiente, pero se podría hacer aún así.

  Se trata de confeccionar un plan de ejecución fijo que se repita, llamado **ciclo principal**, con periodo:

  $$T_M = mcm\left(T_i\right)$$

  El ciclo principal se divide en **ciclos secundarios** con periodo T_S, y se debe cumplir:

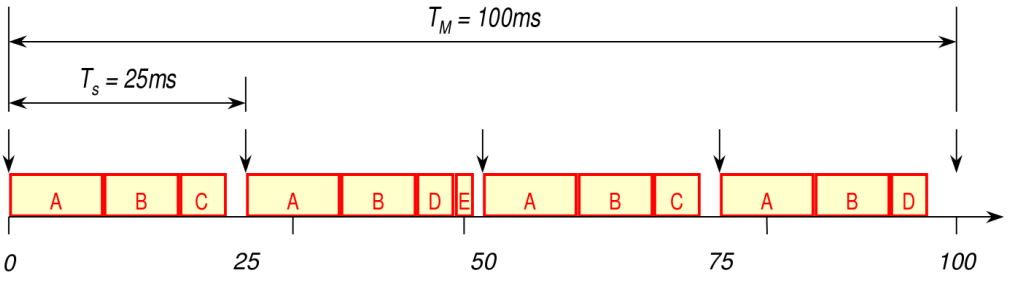
  $$T_M = \left(k \cdot T_S\right)$$

  - ### Ejemplo

```
task1_func (struct event_handler_t* this){
        static const struct timeval period = {0, 25000};
        static int frame = 0;
        switch (frame){
                case 0: fsm_fire(A); fsm_fire(B); fsm_fire(C); break;
                case 1: fsm_fire(A); fsm_fire(B); fsm_fire(D);
                        fsm_fire(E); break;
                case 2: fsm_fire(A); fsm_fire(B); fsm_fire(C); break;
                case 3: fsm_fire(A); fsm_fire(B); fsm_fire(D); break;
        }
        frame = (frame+1)%4
        timeval_add(&this -> next_advalues, &this -> next_activation,
                        &period);
}
```

Tarea	T	C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- El ciclo principal dura 100ms
- Se compone de 4 ciclos secundarios de 25ms cada uno



- **Ventajas**
  - No hay concurrencia en la ejecución, cada ciclo secundario es una secuencia de invocaciones de procedimientos. Por lo tanto, es fácil de depurar
  - No hay exclusión mutua, los procedimientos pueden compartir recursos y datos
  - Es correcto por construcción, no hace falta analizar el comportamiento temporal
  - Optimiza la memoria

- **Inconvenientes**
  - Los periodos deben ser armónicos, si no lo son será una planificación poco eficiente y el plan será difícil de construir
  - Poco flexible y difícil de mantener, las interrupciones son difíciles de tratar y cada vez que cambia una tarea hay que rehacer toda la planificación.
  - No hay concurrencia en la ejecución, se ejecutará en 1 core y el tiempo en caso peor se alarga
  - Si la CPU está demasiado cargada y no nos cabe alguna tarea, tendremos que dividirla "a mano" en subtareas

● Planificación con prioridades fijas y desalojo

Se ejecuta la tarea con **mayor prioridad** de todas que están preparadas para ejecutarse. Las tareas se realizan como procesos concurrentes (paralelos) y una tarea puede estar en varios estados.

La ejecución puede ser:

- Con desalojo, una tarea de mayor prioridad puede interrumpir a otra de menor prioridad que se esté ejecutando
- Sin desalojo, la tarea de mayor prioridad debe esperar a que la CPU quede libre para comenzar a ejecutarse, aunque la esté ocupando una tarea de menor prioridad

Una **asignación de prioridades** típica es la que se conoce como **prioridades monótonas de frecuencia**, que consiste en dar la mayor prioridad a las tareas de menor periodo. Esta asignación es **óptima** para el modelo de tareas simple.

El **Factor de utilización** es una medida de la carga del procesador para un conjunto de tareas. En un sistema monoprocesador debe ser ≤ 1:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}$$

Para el modelo simple los plazos están garantizados si:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N \cdot \left(2^{1/N} - 1\right)$$

La **utilización mínima garantizada** para N tareas:

$$U_0(N) = N \cdot \left(2^{1/N} - 1\right)$$

Para infinitas tareas, tiende a:

$$\lim_{N \to \infty} U_0(N) = \log 2 \approx 0{,}693$$

Se denomina **instante crítico** al instante en que todas las tareas se activan a la vez. Si el instante inicial es crítico, basta con comprobar el primer ciclo de cada tarea.

El **tiempo de respuesta** de una tarea es la suma de su tiempo de cómputo más la interferencia que sufre por la ejecución de tareas más prioritarias.

$$R_i = C_i + I_i$$

La **interferencia máxima** para todas las tareas de prioridad superior es:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

Con la interferencia máxima, podemos calcular el tiempo de respuesta:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

Esta ecuación no es lineal ni continua, y no se puede resolver analíticamente. Para resolverla utilizamos **iteraciones** hasta que $R^{n+1} = R^n$ para que se cumpla, o bien $R^{n+1} > T_i$ y entonces no se cumpliría el plazo.

*Para poder calcular el tiempo de cómputo en un sistema real hay que evitar utilizar estructuras con tiempo de cómputo no acotado.

- **Ejemplo**

Tarea	T	C	P	R
τ1	7	3	3	3
τ2	12	3	2	6
τ3	20	5	1	20

$$\tau_1: \quad w_1^0 = 3;$$
$$\tau_2: \quad w_2^0 = 3 + 3 = 6;$$
$$w_2^1 = 3 + \left\lceil \frac{6}{7} \right\rceil \cdot 3 = 6$$

$$\tau_3: \quad w_3^0 = 5 + 3 + 3 = 11;$$
$$w_3^1 = 5 + \left\lceil \frac{11}{7} \right\rceil \cdot 3 + \left\lceil \frac{11}{12} \right\rceil \cdot 3 = 14;$$
$$w_3^2 = 5 + \left\lceil \frac{14}{7} \right\rceil \cdot 3 + \left\lceil \frac{14}{12} \right\rceil \cdot 3 = 17;$$
$$w_3^3 = 5 + \left\lceil \frac{17}{7} \right\rceil \cdot 3 + \left\lceil \frac{17}{12} \right\rceil \cdot 3 = 20;$$
$$w_3^4 = 5 + \left\lceil \frac{20}{7} \right\rceil \cdot 3 + \left\lceil \frac{20}{12} \right\rceil \cdot 3 = 20$$

- **Ejemplo**

```
xTaskCreate(&fsm1, "startup", 2048, NULL, 1, NULL);  // prioridad = 1
xTaskCreate(&fsm2, "startup", 2048, NULL, 2, NULL);  // prioridad = 2
xTaskCreate(&fsm3, "startup", 2048, NULL, 3, NULL);  // prioridad = 3
        // &fsm1, &fsm2 y &fsm3: Son los "main" de cada tarea, en cada
        // uno de ellos se encuentran las tablas de transiciones de cada fsm
```

- **Ventajas**
  - Muy flexible
  - Usa todos los cores (concurrente)
  - Menor latencia en caso peor
  - Ninguna tarea nos bloquea para siempre
- **Inconvenientes**
  - Necesita una interrupción periódica para comprobar si se han producido errores o bloqueos. Al interrumpir cada poco tiempo aumenta el consumo y se sobrecarga el sistema
  - Necesita un mecanismo de sincronización

- **Tareas aperiódica**

Deben garantizarse los plazos de todas las tareas críticas en el caso peor. Las tareas aperiódicas pueden ser:

- Críticas (hard) o esporádicas: Tienen un plazo de respuesta crítico y se caracterizan por una separación mínima entre dos sucesos de activación consecutivos
- Acríticas (soft): Pueden atenderse más tarde y se caracterizan por un sistema de activación estocástico (ej. Poisson)

- ## Tareas esporádicas

  Para incluir tareas esporádicas hace falta modificar el modelo simple de tareas:

  - El parámetro **T** representa la separación mínima entre dos sucesos de activación consecutivos
  - Suponemos que en el peor caso la activación es pseudo periódica con periodo T
  - El plazo de respuesta puede ser menor que el periodo (D ≤ T)

- ## Prioridades monótonas en plazos

  Cuando los **plazos son menores o iguales que los periodos**, la asignación de mayor prioridad a las tareas de menor plazo de respuesta es **óptima**. Si los plazos son mayores que los periodos no hay método óptimo, hay que probar.

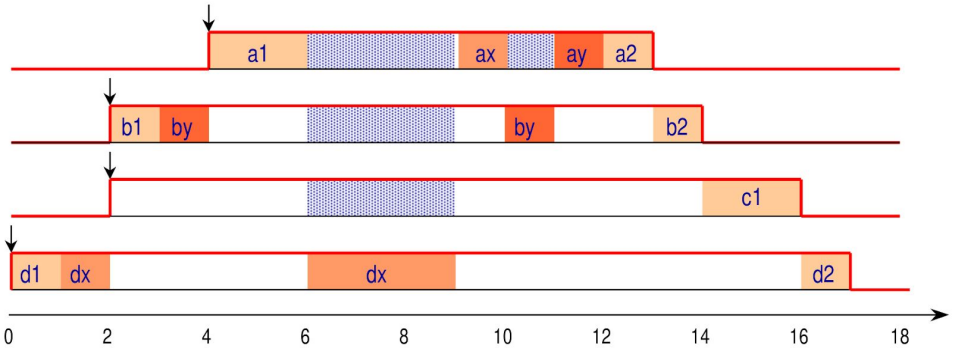
  El tiempo de respuesta se calcula de la misma forma que con la asignación monótona en frecuencia, cuando $R_i^{n+1} = R_i^n$ para que se cumpla, o bien $R_i^{n+1} > D_i$ y entonces no se cumpliría el plazo.

- ## Interacción entre tareas

  En la mayoría de sistemas las tareas intercambian datos o mensajes. Puede ocurrir que una tarea tenga que espera a un suceso de otra tarea menos prioritaria, esto se conoce como **bloqueo** y produce una inversión de prioridad indeseable. No se pueden eliminar completamente pero se puede limitar su duración.

  - ### Herencia de prioridad

    Las prioridades de las tareas varían dinámicamente.
    Cuando una tarea está bloqueando a otra más prioritaria, hereda la prioridad de esta. Es **transitiva**.



La **duración máxima del bloqueo** con herencia de prioridad es:

$$B_i = \sum_{j \in lp(i),k} C_{j,k}$$

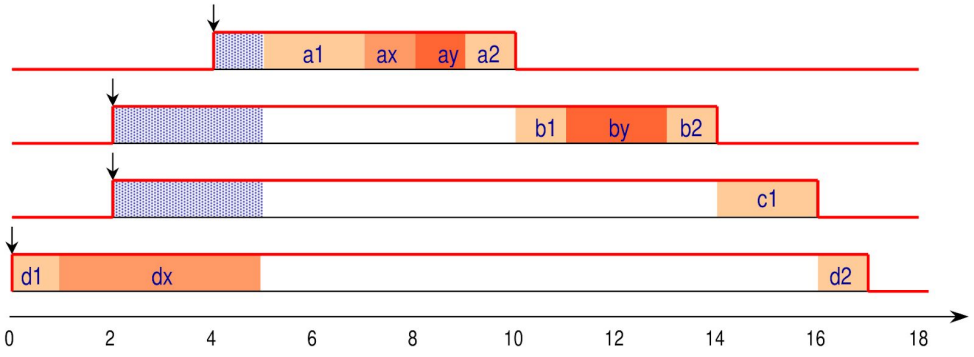
Donde ***k*** es el número de recursos compartidos y $C_{j,k}$ el tiempo durante el cual la tarea *j* accede al recurso *k*.

Cuando hay bloqueos la ecuación del **tiempo de respuesta** se queda así:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

- ## Techo de prioridad inmediato

El techo de prioridad de un recurso es la máxima prioridad de las tareas que lo utilizan. En este protocolo, cuando una tarea accede a un recurso, hereda inmediatamente el techo de prioridad del recurso.



Propiedades:

- Cada tarea puede bloquear una vez como máximo, y solo al principio de cada ciclo
- No puede haber interbloqueos
- No puede haber bloqueos encadenados
- Se consigue una exclusión mutua
- “Cuesta” menos que herencia de prioridad

La **duración máxima del bloqueo** es:

$$B_i = \max_{j \in lp(i),k \in lc(i)} C_{j,k}$$

Donde *lc(i)* es el conjunto de recursos que pueden bloquear *i*

- ## Reactor

  Patrón de diseño para la implementación sencilla de **sistemas de multitarea cooperativa**. Es un patrón de programación **concurrente (paralela)** pero los sistemas estarán formados por un **único hilo** de ejecución.

  Hay un "service handler" que demultiplexa las **peticiones** entrantes (paralelas y asíncronas) y las entrega de forma **síncrona** y "**en serie**" a los "handler" asociados. Se considera una aplicación impulsada por eventos. Una vez la tarea que se está ejecutando termine, se empieza a ejecutar la más prioritaria.

  Se separa completamente el código específico de la aplicación del código de Reactor. Con ello podemos dividir los componentes de la aplicación en partes modulares y reutilizables.

  Debido a la llamada síncrona de los "handlers", Reactor permite concurrencia (paralelización) sin agregar complejidad de múltiples hilos.

  El **tiempo de respuesta** es:

  $$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \cdot C_j$$

  $$R_i = w_i^n + F_i, \text{para } w_i^{n+1} = w_i^n$$

  La variación en los instantes de activación con respecto al esquema idea se denomina ***jitter***. Si lo tenemos en cuenta el tiempo de respuesta es:

  $$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil \cdot C_j$$

  $$R_i = w_i^n, \text{ para } w_i^{n+1} = w_i^n$$

  - ### Ventajas
    - Mejora la planificabilidad
    - Mejora la latencia y tiempos de cómputo
    - Mejora la escalabilidad
    - Mejora la eficiencia
    - Consume casi lo mismo que el ejecutivo cíclico
    - Los periodos pueden no ser armónicos
    - No hace falta mecanismo de sincronización
    - Simplifica el diseño
    - Cada "handler" tiene su propia cola, por lo que no se afectan entre ellos

- **Inconvenientes**
  - El coste de ofrecer un cambio de contexto puede ser alto, por lo que el tiempo de respuesta puede alargarse
  - Si alguna tarea ocupa mucho tiempo, no liberará la CPU hasta que haya terminado por completo (se asemeja a un techo de prioridad inmediato con la CPU como recurso).
    Para solucionar esto se puede introducir cesiones voluntarias de CPU en distintos puntos de la tarea *(F_i)*
  - Una única aplicación puede bloquear el sistema, no hay "watchdog" ni interrupciones periódicas
  - Se pueden perder peticiones si la cola está llena
  - Es más difícil de depurar

- **Ejemplo**

```
task1_func (struct event_handler_t* this){
        static const struct timeval period = {0, 15000};
        fsm_fire(A);
        timeVal_add(&this -> next_activation, &this -> next_activation,
                        &period);
}
int main(){
        EventHandler eh1;
        reactor_init ();
        event_handler_init (&eh1, 1, task1_func);
        reactor_add_handler (&eh1);

        while (1) {
                reactor_handle_events ();
        }
}

// Por cada tarea (fsm) hay un task_func, al igual que en el ejecutivo cíclico
// para el hiperperiodo
// https://github.com/greenlsi/unlessons/tree/master/reactor
```

3. Verificación formal

- **Sintaxis: LTL ⇔ Promela**

G *(globally)*	[ ]
F *(finally)*	<>
X *(next state)*	x
U *(until)*	u

- **Verificación y simulación**

  spin -a model.pml
  gcc -o pan pan.c
  ./pan -a -f -N spec
  spin -t model.pml ⇒ (si ha ocurrido un error y queremos ver)
  spin -p model.pml ⇒ (si se han incluido printf y se desea ver una traza de la simulación)

- **Restricciones para asegurar equivalencia de tipos**
  - Entradas en refinamiento ⊆ entradas del modelo abstracto
  - Salidas del modelo abstracto ⊆ salidas en refinamiento
  - Todos los valores de entrada aceptables para el modelo abstracto también lo son para el refinamiento
  - Todos los valores de salida del refinamiento tienen que ser aceptables en el modelo abstracto
  - Ambos producen las mismas salidas para la misma secuencia de entradas
  - Conjunto de comportamientos del modelo abstracto ⊆ conjunto de comportamientos en refinamiento

4. Errores frecuentes y consejos

- **FSMs en C: Funciones de comprobación**
  - **Ejemplo 1:**

```
int tiempo_maximo (fsm_t* this) {
        if (xTaskGetTickCount() > tiempo_sin_pulsar) {
                if (xTaskGetTickCount() > tiempo_reset) {
                        indice = 0;
                }return 1;
        }return 0;
}
```

→ No se le debe dar valor a la variable 'indice', eso se debe hacer en la función de salida. Además si estás comprobando tiempos distintos es más correcto hacerlo en dos funciones de comprobación distintas.

  - **Ejemplo 2:**

```
int pulsacion (fsm_t* this) {
        if (!GPIO_INPUT_GET(0) ) {
            if (xTaskGetTickCount() > t_antirrebote) {
                t_antirrebote=xTaskGetTickCount() + ANTIRREBOTE;
                tiempo_sin_pulsar=xTaskGetTickCount() + SEGUNDO;
                tiempo_reset=xTaskGetTickCount() + 3*SEGUNDO ;
                c++;
                printf("Valor de c: %d\n", c);
                printf("Valor de indice: %d\n", indice);
                return 1;
             }
        }return 0;
}
```

→ En las funciones de comprobación no se deben hacer printf ni tantas asignaciones de tiempo. Deben ser rápidas de ejecutar.
En este caso además, como estamos mirando la GPIO, algunas asignaciones se pueden hacer en la propia ISR (rutina de atención a la interrupción).

- **FSMs en C: Activación periódica de una tarea**
  - **Ejemplo 1:**

```
portTickType xLastWakeTime;
    while (true) {
        xLastWakeTime = xTaskGetTickCount () ;
         fsm_fire(fsm);
         vTaskDelayUntil(&xLastWakeTime , PERIOD_TICK) ;
}
```

```
portTickType period = 250 /portTICK_RATE_MS;
portTickType last = xTaskGetTickCount () ;
while (1) {
         fsm_fire (light_fsm);
         vTaskDelayUntil (&last , period ) ;
}
```

5. Posibles preguntas de test

- **¿Cuál es la especificación LTL correspondiente a la propiedad p?**
    - **P = Cuando la alarma está encendida, la sirena suena cuando se detecta presencia**

      [] (((state==ON) && presencia) -> <>(sirena==1))
    - **P = En el cruce, en ningún momento pueden estar en verde la calle principal y la secundaria**

      [] (!((principal==verde) &&(secundaria==verde)))
    - **P = La alarma del reloj se apaga cuando se aprieta al botón 1**

      [] ((state==ON) && button1) -> <>(state==OFF))
    - **P = Si la temperatura de la puerta alcanza 100ºC, el horno se apaga**

      [] (((state==ON) && (temp>=100)) -> <>(state==OFF))
    - **P = Si la puerta está abierta, el microondas está apagado**

      [] ((puerta==ABIERTA) && (state==OFF))

- **Un modelo con dos máquinas de estados concurrentes, con M y N número de estados respectivamente, ¿Cuántos estados tendrá de manera general una única máquina de estados equivalente?**

  MxN estados

- **¿Cuándo es necesario añadir un nuevo estado al modelo FSM?**

  Cuando necesitamos que el sistema reaccione de manera diferente a los eventos de entrada

- **En el modelo de tareas simple ¿Qué estrategia de asignación de prioridades es óptima?**

  Asignación de prioridad monótona en frecuencia: a menor periodo (D), mayor prioridad

- **Si los plazos (D) son menores a los periodos (T) ¿Qué estrategia de asignación de prioridades es óptima?**

  Asignación de prioridad monótona en plazo: a menor plazo (D), mayor prioridad

- **¿Y si los plazos (D) son mayores a los periodos (T)?**

  No hay asignación óptima, debemos probar para cada caso

- **Una tarea que no usa recursos compartidos. ¿Puede sufrir bloqueos?**

  Sí, si una tarea menor prioridad tiene un recurso compartido con otra tarea de prioridad mayor, puede heredar la mayor prioridad al acceder al recurso y bloquear la tarea que no accede a recursos compartidos

- **El bloqueo máximo con protocolo de techo de prioridad inmediato, ¿puede se mayor que si usamos herencia de prioridad?**

  No, con techo de prioridad inmediato el bloqueo máximo corresponde con el máximo tiempo de acceso al recurso de las tareas que te pueden bloquear.
  Pero en herencia de prioridad corresponde con el sumatorio de los tiempos de acceso al recurso de las tareas que te pueden bloquear.
  Es decir, con techo de prioridad el bloqueo máximo puede ser MENOR o IGUAL al bloqueo máximo con herencia de prioridad

- **¿Cuántos bloqueos puede sufrir una tarea en cada activación si utilizamos techo de prioridad inmediato?**

  Solo uno y al principio del ciclo

- **¿Y con herencia de prioridad?**

  Tantos bloqueos como recursos compartidos de techo de prioridad mayor que mi tarea

- **Si utilizamos techo de prioridad inmediato y una tarea empieza a ejecutarse, ¿puede sufrir un bloqueo en esa activación?**

  Solo uno y al principio del ciclo

- **¿Cuál es el máximo bloqueo que puede sufrir la tarea de mínima prioridad?**

  Ninguno, solo puede tener interferencias

- **Ordena de menor a mayor tiempo de respuesta**

  Prioridades fijas y desalojo < Reactor < Ejecutivo cíclico

- **Ordena de menor a mayor consumo (periodos armónicos)**

  Ejecutivo cíclico < Reactor < Prioridades fijas y desalojo

- **Ordena de menor a mayor consumo (periodos NO armónicos)**

  Reactor < Ejecutivo cíclico < Prioridades fijas y desalojo

- **Ordena de menor a mayor tamaño de código (periodos armónicos)**

  Ejecutivo cíclico < Reactor < Prioridades fijas y desalojo

- **Ordena de menor a mayor tamaño de código (periodos NO armónicos)**

  Reactor < Ejecutivo cíclico < Prioridades fijas y desalojo

- **Si no hay recursos compartidos en todo el sistema, ¿puede haber bloqueo?**

  No, solo interferencias

- **Si cada máquina de estados es una tarea, ¿Qué determina el tiempo de ejecución en caso peor (C)?**

  fsm_fire();

- **Si cada máquina de estados es una tarea, ¿Qué determina el periodo?**

  Lo definimos nosotros en el código

- **Si cada máquina de estados es una tarea, ¿Qué determina el plazo?**

  Lo definen las especificaciones que deseamos cumplir

- **Para asegurar que el sistema sea planificable, ¿cuáles de estas características debe cumplirse?**

  A. Todas las operaciones deben tener tiempo de ejecución acotado
  B. Los plazos deben ser menores o iguales que los periodos