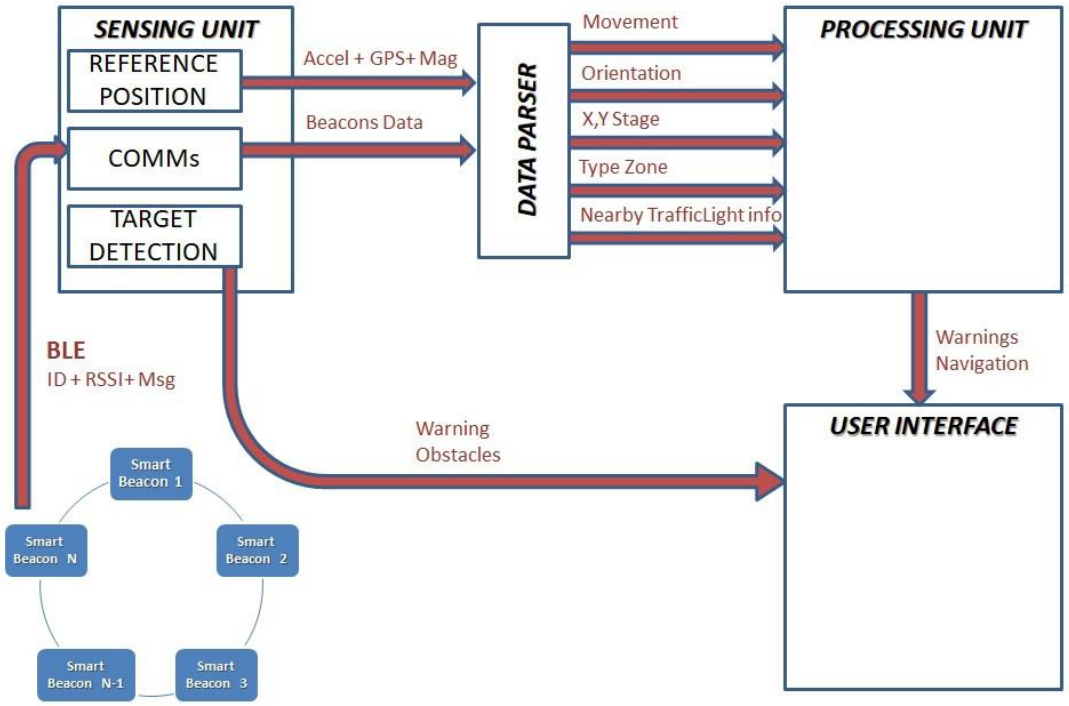
Introducción

El objetivo de este proyecto es desarrollar un sistema para la ayuda al guiado de personas invidentes en entornos urbanos. Para ello, la solución se va a implementar en una Raspberry Pi 3, a la que se conectarán una serie de sensores y módulos de comunicación, a partir de los cuales se tomarán diferentes decisiones. Estos sistemas son: ultrasonidos, GPS, IMU y módulo Bluetooth. Se ha realizado un diseño basado en máquinas de estados para modelar el comportamiento del sistema. En la siguiente figura se muestra la arquitectura, a nivel de diagrama de bloques, del sistema completo:



Justificar la especificación

Se han definido las siguientes especificaciones a cumplir por el sistema:

- **[]((data && validData && (semaphore_state == IDLE_SEM)) -> <>(semId != 0))**

Esta especificación define que cuando hay datos válidos y el estado del semáforo está en IDLE (no hay semáforo seleccionado, es decir, $semId = 0$), eventualmente se seleccionará algún semáforo ($semId\ != 0$).

- **[]((msg_state == CARRETERA_STATE) -> (msg == PELIGRO))**

Cuando la máquina de estados dedicada a la decisión de los mensajes está en el estado CARRETERA_STATE, el mensaje seleccionado para trasmitir es PELIGRO.

- **[](((trigger == 1) && (timer_trigger_end == 1)) -> <> (trigger == 0))**

En el protocolo del sistema de ultrasonidos, hay una señal que, tras finalizar un temporizador, debe bajarse (su valor debe ser 0). Con esta propiedad verificamos que, cuando esa variable previamente se había levantado (con valor 1) y termina el temporizador, la variable toma el valor 0.

- **[](((counter >= (MAX_COUNTER-1)) && (timer_IMU_end)) -> <> (IMU_state == DATA_PROCESSING))**

En la IMU, cuando se han tomado MAX_COUNTER-1 muestras y ha finalizado el timer asociado a este módulo, eventualmente se pasa a procesar la información (IMU_state = DATA_PROCESSING).

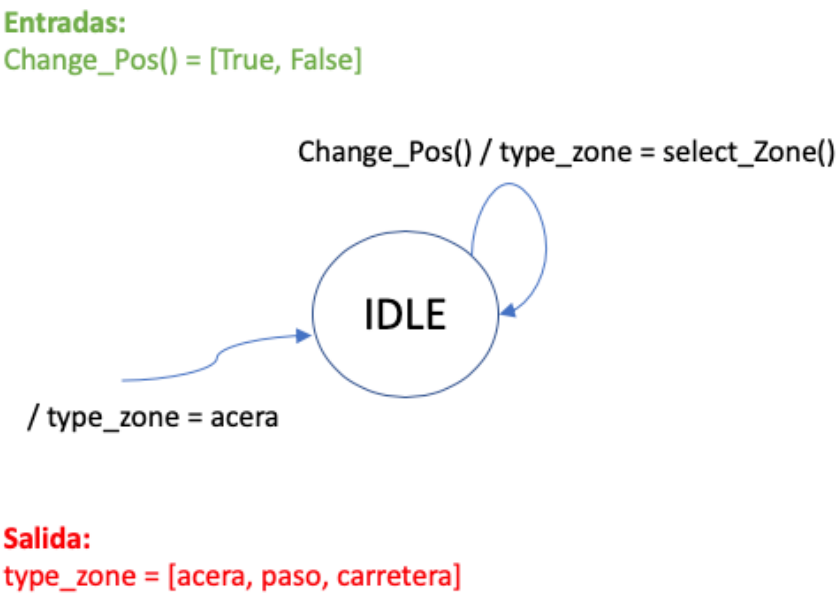
- **[]((timer_IMU_end==1) -> <> (IMU_state == DATA_ACQUISITION))**

Siempre que el temporizador de la IMU finaliza, en algún momento en el futuro se debe retornar al estado DATA_ACQUISITION.

Modelo de máquinas de estados

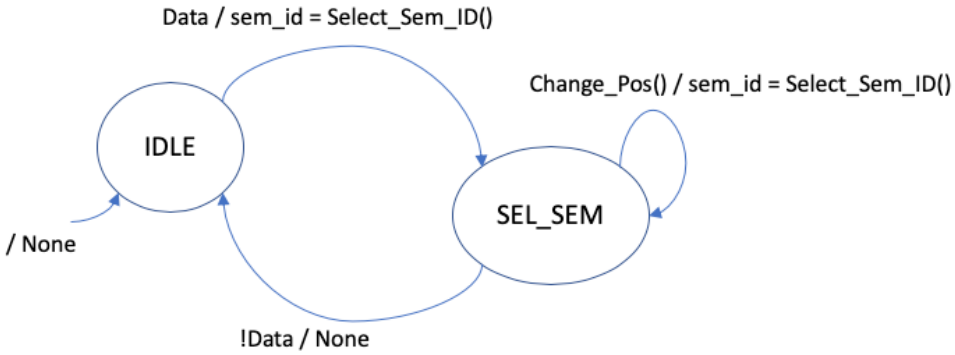
Se ha modelado el sistema utilizando máquinas de estados. Los diagramas asociados a cada una de ellas muestran en las siguientes figuras:

FSM typeZone:



FSM selectSemaphore:

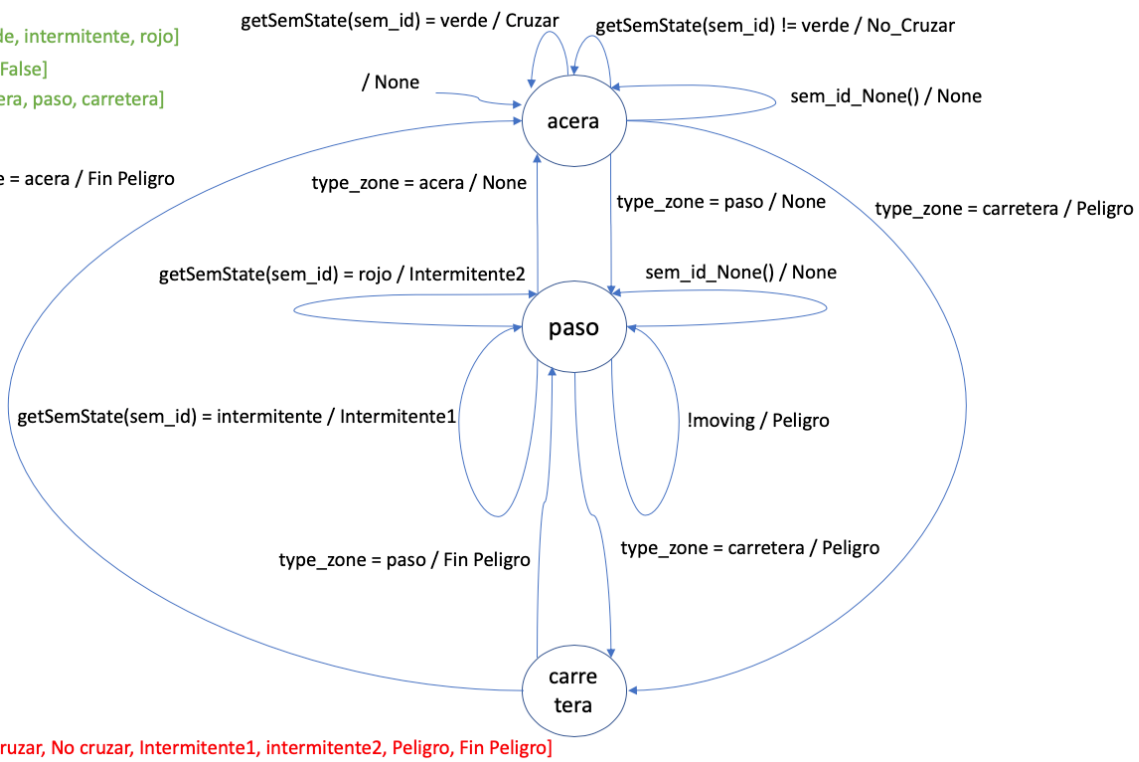
Entradas:
Change_Pos() = [True, False]
Data = [True, False]



Salida:
sem_id

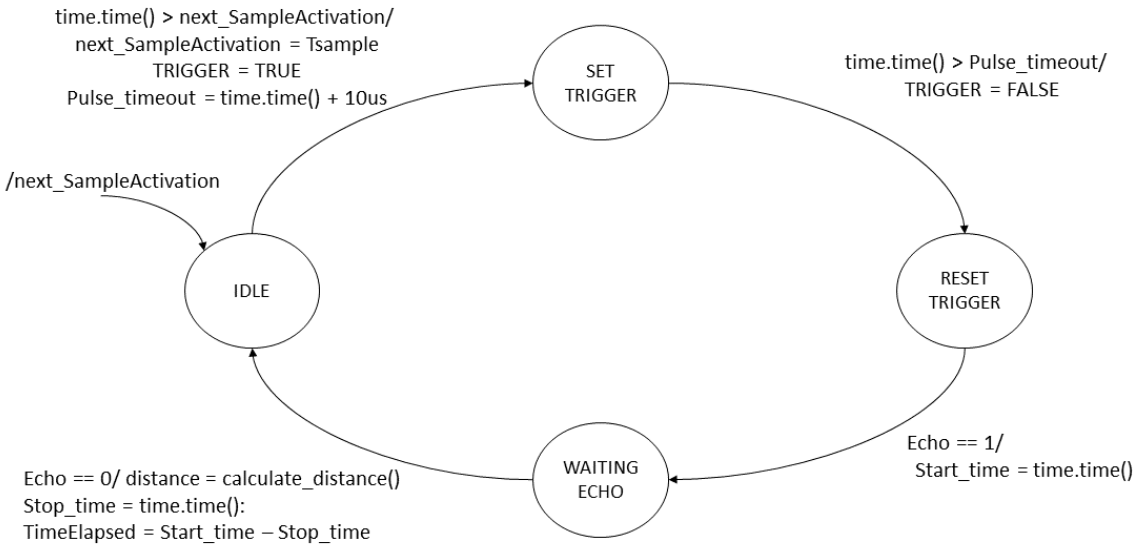
FSM msg:

Entradas:
semester = [verde, intermitente, rojo]
Moving = [True, False]
type_zone = [acera, paso, carretera]
sem_id



Salida:
msg = [None, Cruzar, No cruzar, Intermitente1, intermitente2, Peligro, Fin Peligro]

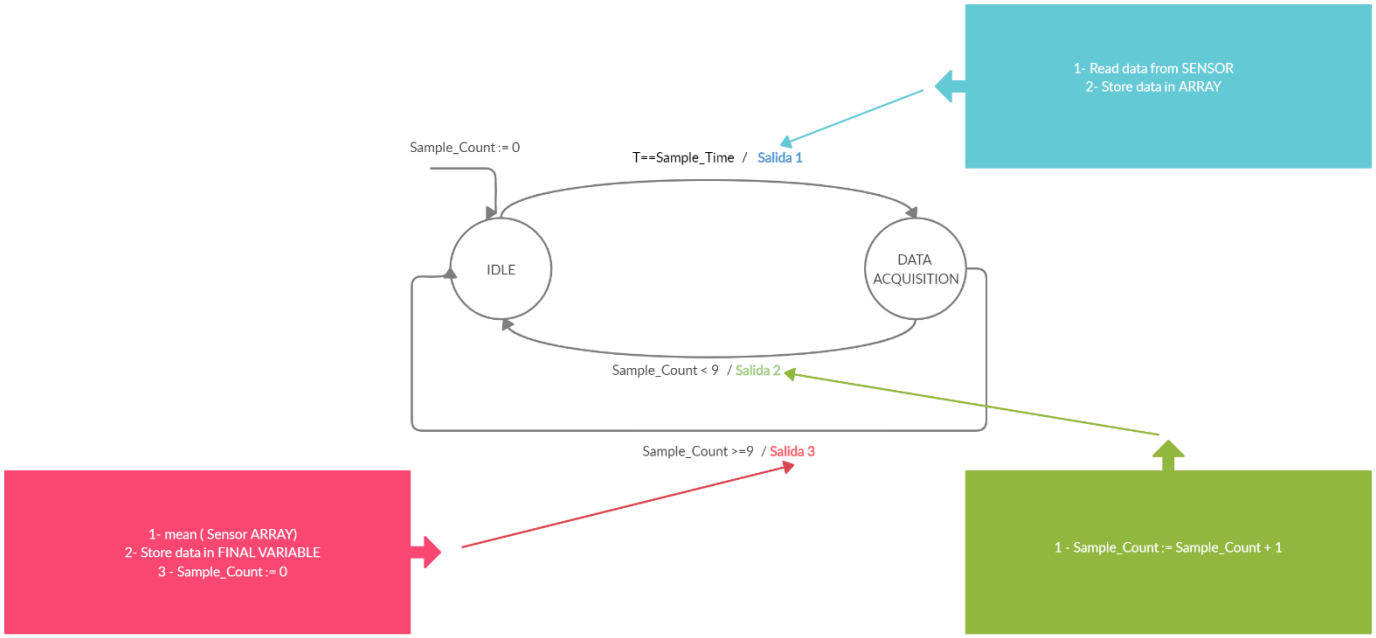
FSM ultrasonidos:



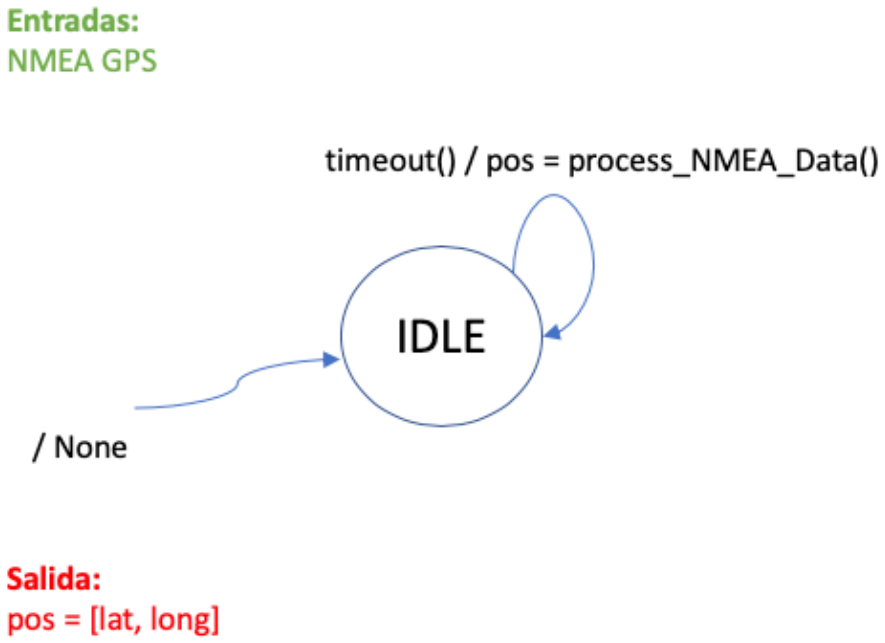
Entradas
Echo = [0, 1]

Salida:
distance ϵ R

FSM IMU:



FSM GPS:



Resultados de la verificación con Spin

En el directorio “spin” de la entrega está disponible el código en Promela de las máquinas de estados anteriormente mencionadas. Por cada una de ellas, se ha generado un *active proctype*. Adicionalmente, se han modelado las entradas con el proceso asociado al entorno. Al generar y ejecutar el verificador, es posible comprobar que todas las especificaciones que debe cumplir el sistema, definidas en el primer apartado de este documento, se comprueba que todas ellas son cumplidas.

Nota: debido a la complejidad del sistema, ha sido necesario modificar la profundidad de exploración de posibilidades de spin (*depth*).

Decisiones de implementación

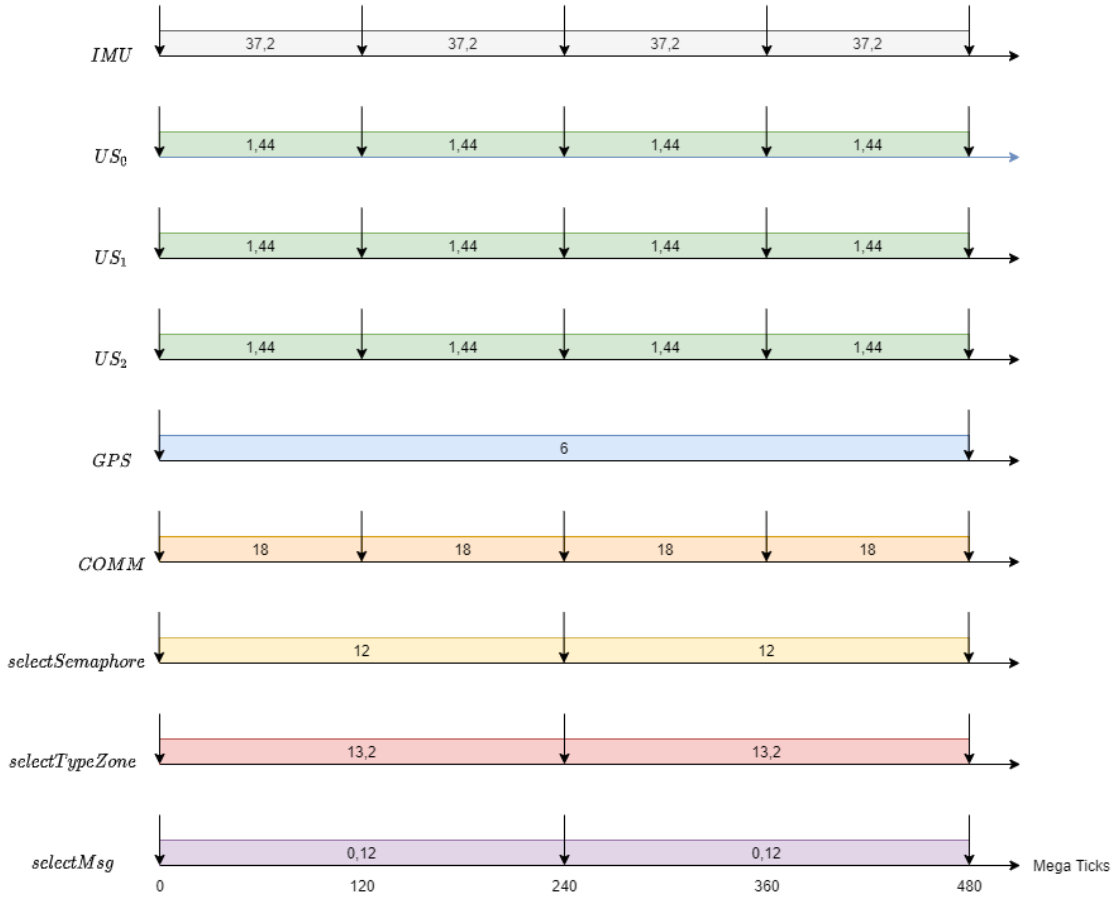
Se ha decidido implementar el sistema en una Raspberry Pi 3, puesto que estábamos trabajando en esta plataforma con anterioridad en la asignatura LSEL. Se ha utilizado con el sistema operativo Raspbian. Se ha realizado todo el desarrollo de planificación suponiendo un sistema de tiempo real, a pesar de que sabemos que este sistema no es de tiempo real. Esto tendrá impacto en los tiempos de ejecución en caso peor medidos en el apartado dedicado a la planificabilidad del sistema, hecho del que somos conscientes. Para la programación del sistema, hemos trabajado con Python, para lo cual se han adaptado las librerías fsm.c y fsm.h implementadas en C, para ser utilizadas en este lenguaje.

Justificación de la validez de la verificación o nueva verificación

Según se ha visto en clase, a la hora de verificar el diseño implementado, basta con demostrar que el comportamiento del sistema está contenido en el lenguaje del modelo realizado en Spin. Para ello, se ha llevado a cabo un proceso riguroso en el modelado de las FSM, para que su implementación en Python sea casi automática.

Justificar el cumplimiento de plazos en el caso peor

Se ha planificado el sistema utilizando el algoritmo de planificación YDS. Para ello, lo primero que se necesita saber es el tiempo de ejecución en caso peor de cada una de las tareas. Se han realizado múltiples medidas del tiempo de ejecución y se he seleccionado el valor máximo de dichos tiempos. Se debe tener en cuenta que la implementación se ha realizado en la Raspberry. Por ese motivo, no se está implementado un sistema de tiempo real puro, sino que hay más tareas que requieren la utilización del procesador. Es por ello que los tiempos son superiores a lo que cabría esperar en un microcontrolador sin otras tareas más que la nuestras propias. Lo siguiente, es conocer el esquema de activación de cada una de las tareas, tal como se muestra en la siguiente figura, en la que los tiempos están especificados en Mega Ticks del sistema:



Por tanto, podríamos resumir todos los tiempos involucrados en la planificación en las siguientes tablas:

MAX FREQ (GHz)	1,200
Intensidad YDS (teorica)	0,614
FREQ YDS teorica (GHz)	0,737
Intensidad real	0,667
FREQ YDS (GHz)	0,800

FSMs	C (s)	C (ticks)	D (s)	T (s)	T (ticks)	C YDS (ticks) teórica	C YDS (ticks) real
imu	0,031	37.200.000	0,100	0,100	120.000.000	60.586.319	55.800.000
us (x3)	0,0012	1.440.000	0,100	0,100	120.000.000	2.345.277	2.160.000
gps	0,005	6.000.000	0,400	0,400	480.000.000	9.771.987	9.000.000
comm	0,015	18.000.000	0,100	0,100	120.000.000	29.315.961	27.000.000
selectSemaphore	0,01	12.000.000	0,200	0,200	240.000.000	19.543.974	18.000.000
selectTypeZone	0,011	13.200.000	0,200	0,200	240.000.000	21.498.371	19.800.000
selectMsg	0,0001	120.000	0,200	0,200	240.000.000	195.440	180.000

Para obtener el intervalo en el que la intensidad es mayor, se han calculado las intensidades en los intervalos en los que la intensidad es diferente, y se ha seleccionado la mayor de ellas. En este caso, la mayor es $G_{[0,480]}$:

$$G_{[0,120]} = \frac{37{,}2 + 1{,}44 + 1{,}44 + 1{,}44 + 18}{120} = \frac{59{,}52}{120} = 0{,}496$$

$$G_{[0,240]} = \frac{2 \cdot (37{,}2 + 1{,}44 + 1{,}44 + 1{,}44 + 18) + 12 + 13{,}2 + 0{,}12}{240} = \frac{144{,}36}{240} = 0{,}6015$$

$$G_{[0,480]} = \frac{4 \cdot (37{,}2 + 1{,}44 + 1{,}44 + 1{,}44 + 18) + 2 \cdot (12 + 13{,}2 + 0{,}12) + 6}{480} = \frac{294{,}72}{480} = 0{,}614$$

Según estas expresiones, obtenemos que la intensidad en ese intervalo es 0,614. Partiendo de la frecuencia original de la Raspberry de 1,2 GHz, con esta intensidad se podría ejecutar el sistema a 737 MHz. No obstante, esta frecuencia no está disponible, por lo que se ha seleccionado la inmediatamente superior, 800 MHz. Escogiendo este valor, es necesario recalcular la intensidad asociada a él: 0,667. A partir de esta intensidad, se han calculado los tiempos de ejecución en caso peor, "C YDS (ticks) real" de la tabla superior, que son los tiempos que requiere cada tarea al planificar con YDS. El último paso es distribuir las tareas a lo largo del intervalo de tiempo disponible para ello, tal como se muestra en la siguiente figura:

Adicionalmente a las tareas expuestas anteriormente, el sistema tiene la labor de emitir a través de una interfaz de audio los mensajes decididos, por medio de un altavoz. Para ello se cuenta con la tarea *userInterface.* Esta tarea invierte una gran cantidad de tiempo en comparación al resto, por lo que se ha decidido que se implementará en otro de los núcleos con los que cuenta la Raspberry Pi, para no interferir en el resto de procesos.