Práctica 2. Clasificador de frutas automático

1. Objetivo

En esta segunda práctica de la parte digital, perseguimos varios objetivos:

- Realizar control pwm con variación de parámetros
- Realizar un pequeño monitor por puerto serie
- Diseñar un sistema con todos los elementos de un S.E.P.A.

2. Material necesario

- Connected Launchpad de Texas Instruments con el microcontrolador TIVA TM4C1294NCPDT
- Manual de la librería de funciones DriverLib y datasheet del microcontrolador
- Placa de conexión TIVA-EVE-L293.
- Servomotor 5V.
- Programa TeraTerm, Putty o similar instalado en el ordenador.

3. Fundamento teórico

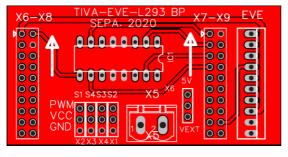
Para la realización de la práctica hará falta conocer el funcionamiento de los diferentes periféricos explicados en clase, en particular los timer, las interrupciones de los mismos, los moduladores pwm, y el puerto serie asíncrono (UART)

4. Realización de la práctica

I. Primer Ejercicio: Manejo de un Servomotor.

Partiendo del ejemplo disponible en Enseñanza virtual, se realizará un programa que maneje un servo conectado al pin PGO. El servo empezará estando en el estado central. Al pulsar el botón 1, hará el recorrido desde el punto central hasta el extremo izquierdo (giro en sentido contrario a las agujas del reloj), esperará 1s y volverá al punto inicial. De manera análoga, al pulsar el otro botón se realizará el giro contrario.

Para ello es necesario conocer el diagrama de la placa de conexión TIVA-EVE-L293, que se adjunta como anexo a la práctica.



Conector	Pin X7
S1	1
S2	2
S 3	3
S4	4

Continuando con el empleo de *buenas prácticas* de programación, se desea que todo el proceso se realice siguiendo un esquema óptimo de consumo de energía, y respondiendo a un modo de programación al estilo de sistemas de tiempo real:

- El sistema se construirá como un sistema síncrono con periodo 50ms. Para ello se programará un timer que marque el final de dicho periodo, y el bucle principal deberá durar siempre mucho menos que dicho periodo.
- En el bucle infinito, poner el sistema en modo de bajo consumo, a la espera que la interrupción del timer *despierte* al sistema.
- Cuando la rutina de interrupción salte (y saque al sistema del modo de bajo consumo), se comprobará si hay algún botón apretado. En caso necesario, ejecutar el movimiento requerido. El movimiento NO SE DEBE realizar en la rutina de interrupción sino en el bucle principal.
- Si se debe esperar un tiempo, se usará un nuevo estado en que se esperará que el tiempo llegue a ese valor (incrementándose a cada vuelta en función del periodo del timer del sistema).

Para poder usar las características de bajo consumo, es necesario usar las siguientes funciones de la librería sysctl.h:

- extern void SysCtlPeripheralSleepEnable(uint32_t ui32Peripheral): especifica que el periférico indicado permanece encendido durante el modo Sleep. Es necesario para que se produzcan las interrupciones que despierten al sistema.
- extern void SysCtlPeripheralSleepDisable(uint32_t ui32Peripheral):
 Deshabilita un periférico específico. Puede ser útil si queremos que un periférico en concreto esté deshabilitado en alguna parte del proceso.
- extern void SysCtlPeripheralClockGating(bool bEnable): Por defecto, todos los periféricos están habilitados, independientemente de lo que digan las dos funciones anteriores, hasta que se ejecute esta instrucción pasando un 'true' (1) como parámetro. Si se le pasa un 'false' (0), se deshabilita el apagado de los periféricos en modo bajo consumo.

Una vez configurado el modo de bajo consumo, para pasar a dicho modo basta invocar la función:

```
extern void SysCtlSleep(void);
```

Una vez que el periférico habilitado (el Timer1) despierte al sistema, cuando retorne de la rutina de interrupción ya volverá en modo normal de funcionamiento. Puede resultar de ayuda consultar el ejemplo2_5.

NOTA: Dado que la función SysCtlSleep() genera problemas cuando se hace Debug, se sugiere hacer uso de una función auxiliar, que espere que se produzca la interrupción:

```
void SysCtlSleepFake(void)
{
    while(!Flag_ints);
    Flag_ints=0;
}
```

De esta forma, en las rutinas de interrupción que quiero que despierten al sistema sólo tendré que dar valor a la variable global Flag_ints. Para facilitar el cambio de una a otra, se sugiere igualmente usar un par de sentencias #define para cambiar fácilmente de una a otra:

```
#define SLEEP SysCtlSleep()
//#define SLEEP SysCtlSleepFake()
```

II. Segundo Ejercicio: Monitorización del proceso.

En este segundo apartado, realizaremos una monitorización del proceso anterior. Suponiendo que cada uno de los botones es un sensor que detecta una pieza tipo A o tipo B, y que en función de eso está moviendo el servo para mandarla hacia un lado o hacia otro, se realizará una modificación en el programa anterior para que, cada vez que se pulse un botón, se mande un mensaje a la consola, informando de que se ha detectado una pieza (de tipo A o tipo B), y la hora (en HH:MM:SS desde que se arranca el proceso). El mensaje podría ser algo como:

```
>> [hh:mm:ss] Pieza tipo A detectada.
>> XX piezas tipo A / YY piezas tipo B
```

Para este apartado, se pueden usar las funciones de consola de uartstdio.c, dado que el mensaje más largo que se va a mandar no superará (en tiempo) los 50ms de periodo del sistema. Si fuera así, para evitar problemas habría que usar la interrupción de transmisión para mandar poco a poco el mensaje, permaneciendo en un estado de espera hasta que se acabase la transmisión.

5. Resultados a entregar.

Al igual que en la práctica anterior, habrá que entregar una memoria suficientemente explícita donde se detalle lo realizado en cada apartado y se comente lo fundamental del código (que se incluirá íntegro en la memoria y como archivo aparte). Asimismo se valorará incluir en la entrega algunas imágenes o videos del funcionamiento del sistema.