Solución del Examen final

Problema 1. Acondicionamiento de sensores (3,5 puntos)

1.1. Inclinómetro (1,5 puntos)

Se trata de un sensor integrado con salida en tensión. Por tanto, lo único que hay que hacer es aplicar el método estándar para ajustar la sensibilidad y el cero (Figura 1). Como el rango de salida del sensor es de –20 a 20 mV y el deseado de 0 a 3,3 V, la amplificación necesaria será:

$$A_1 = \frac{3,3}{40 \cdot 10^{-3}} = 82,5 \tag{1}$$

En la configuración de la Figura 1, esto da una salida entre –1,65 V y 1,65 V después del primer amplificador, por lo que habrá que sumar una tensión de *offset* de 1,65 V.

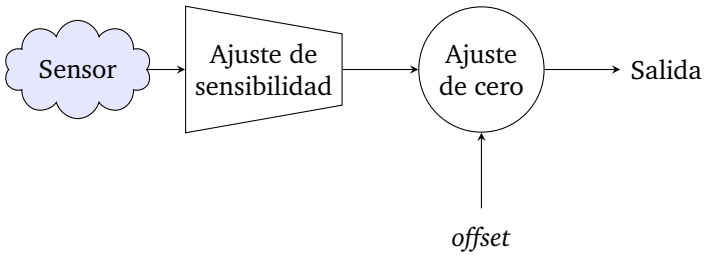

Figura 1. Acondicionamiento genérico de un sensor lineal, aplicable al `IN015`.

Hay que tener en cuenta que el sensor presenta una resistencia de salida no despreciable. En lugar de fiarse de este valor y tenerlo en cuenta en el diseño, conviene conectar el sensor a un amplificador con impedancia de entrada bastante grande, mejor si es infinita. Esto implica usar un amplificador no inversor para la primera etapa. Además, como la amplificación global tiene que ser positiva, se necesita un sumador que no invierta. La Figura 2 muestra una posible implementación (de las muchas posibles) donde se ha llamado v_{in} a la tensión del inclinómetro.

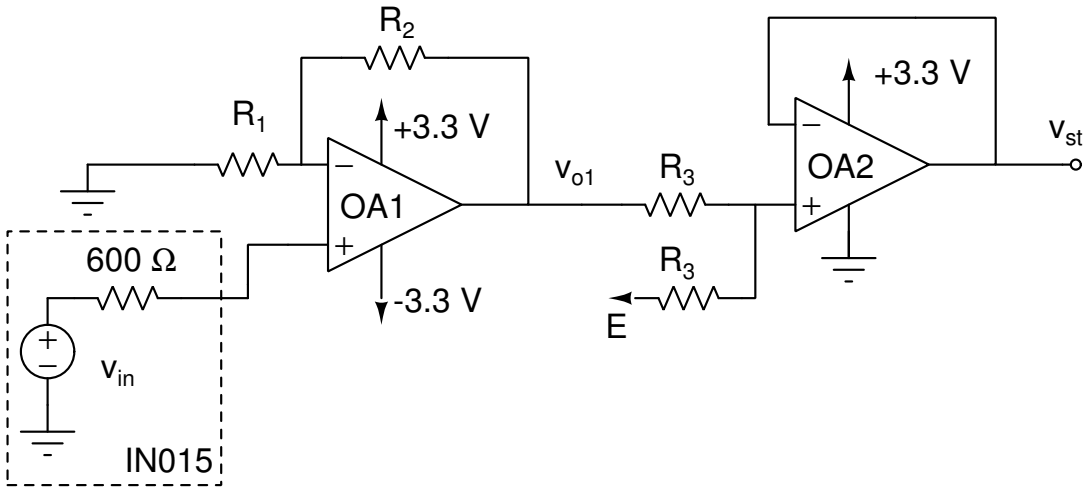

Figura 2. Implementación directa del sistema de la Figura 1.

Las tensiones en los diversos puntos del circuito serán:

$$v_{o1} = v_{in}\left(1 + \frac{R_2}{R_1}\right) \tag{2}$$

$$v_{\mathrm{st}} = v_2^+ = \frac{1}{2}v_{o1} + \frac{1}{2}E = v_{in}\ \underbrace{\frac{1}{2}\left(1 + \frac{R_2}{R_1}\right)}_{\text{ajuste de sensibilidad}} + \underbrace{\frac{1}{2}E}_{\text{ajuste de cero}} \tag{3}$$

Para determinar la expresión de v_{st} se ha aplicado superposición y tenido en cuenta que se han elegido dos valores R_3 iguales para las resistencias del sumador. A partir de aquí se pueden despejar las incógnitas por comparación:

$$\frac{1}{2}\left(1 + \frac{R_2}{R_1}\right) = 82,5 \Rightarrow \boxed{R_1 = 1\,\mathrm{k\Omega}} \quad \boxed{R_2 = 164\,\mathrm{k\Omega}} \tag{4}$$

$$\frac{1}{2}E = 1{,}65 \Rightarrow \boxed{E = 3,3\,\mathrm{V}} \tag{5}$$

Los resultados son valores razonables y además se puede usar la tensión de alimentación directamente para E. Nótese que es bueno que OA2 tenga alimentación entre 0 y 3,3 V; esto evita que en condiciones no previstas (por ejemplo, ángulos de *tilt* mayores de los 20º esperados) la tensión de salida pueda salir del rango admisible para el conversor A/D del microcontrolador.

1.2. Sensor de posición magnético (2 puntos)

a) **(1,5 puntos)** Si se denomina V_p a la tensión del puente, con el positivo en el terminal "+" del amplificador de instrumentación, se puede escribir:

$$V_p = V_{CC}\left(\frac{R_{SZ2}}{R_{SZ1} + R_{SZ2}} - \frac{R_0}{R_0 + R_0}\right) \tag{6}$$

Sustituyendo y simplificando R_0 se llega a:

$$V_p = V_{CC}\left(\frac{1 + \alpha B_{z2}}{1 + \alpha B_{z1} + 1 + \alpha B_{z2}} - \frac{1}{2}\right) \tag{7}$$

Ahora, sabemos que:

$$\left.\begin{aligned} B_{z1} &= \ \ B_0 \sin\theta \\ B_{z2} &= -B_0 \sin\theta \end{aligned}\right\} \Rightarrow V_p = V_{CC}\left(\frac{1 - \alpha B_0 \sin\theta}{1 + \alpha B_0 \sin\theta + 1 - \alpha B_0 \sin\theta} - \frac{1}{2}\right) \tag{8}$$

y finalmente

$$V_p = -\frac{V_{CC}}{2}\alpha B_0 \sin\theta \tag{9}$$

La salida v_{sz} se puede calcular ahora por superposición (ver Figura 3); el amplificador de instrumentación tiene salida con resistencia nula, de modo que es como si fuera un generador ideal de tensión. Con V_{AD} encendido y E apagado (sub-circuito 3(a)), en R_c no pasa corriente —tiene ambos terminales a 0 V, el de la izquierda directamente y el otro por el cortocircuito virtual— así que se trata del clásico inversor.

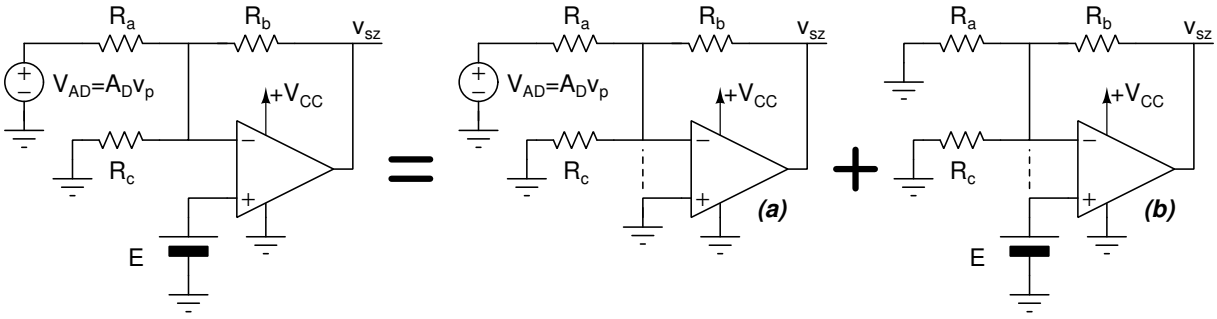

Figura 3. Última etapa del circuito de acondicionamiento. Las resistencias son todas iguales; se han renombrado para poder referenciarlas en el texto más fácilmente.

$$v_{sz}|_a = -\frac{R_b}{R_a} V_{AD} = -V_{AD} \tag{10}$$

En cambio, con E encendido, las resistencias R_a y R_c resultan en paralelo, y la configuración es no inversora:

$$v_{sz}|_b = E\left(1 + \frac{R_b}{R_a||R_b}\right) = E\left(1 + \frac{R}{R/2}\right) = 3E \tag{11}$$

Juntando (9), (10) y (11), se obtiene la fórmula final:

$$v_{sz} = \frac{V_{CC}}{2} A_D \alpha B_0 \sin\theta + 3E \tag{12}$$

La salida tiene que ser 1,65 V para $\theta = 0$, lo cual implica de forma inmediata que:

$$E = \frac{1{,}65}{3} = \boxed{0,55\,\mathrm{V}} \tag{13}$$

y tiene que valer cero para $\sin\theta = -1$ y 3,3 V para $\sin\theta = 1$. Esto implica que:

$$\frac{V_{CC}}{2} A_D \alpha B_0 = 1,65\,\mathrm{V} = \frac{V_{CC}}{2} \Rightarrow A_D = \frac{1}{\alpha B_0} = \frac{1}{1 \cdot 10^{-2} \cdot 0,1} = \boxed{1000} \tag{14}$$

b) **(0,5 puntos)** Aparte de los cambios triviales de simetría arriba-abajo e izquierda-derecha, que cambiarían sólo el signo de la tensión de salida, existen tres configuraciones posibles, como se indica en la Figura 4.

La configuración 4(a) es la que se ha utilizado en el diseño, así que sabemos cómo funciona. Por el contrario, la 4(b) **no** funciona; por ejemplo, un cambio de x positivo haría que la tensión en los dos terminales de medida del puente bajara —dando salida nula por lo menos al primer orden. El circuito 4(c) **sí** puede funcionar; de hecho, es la configuración estándar para los medios puentes.

Tanto la configuración (a) como la (c) son insensibles a los cambios de temperatura. Para la (c) lo sabemos por la teoría, la (a) acabamos de darnos cuenta por el hecho que si se sustituye R_0 por $R_{00}(1 + \beta\Delta T)$ nada cambia en las ecuaciones.

La configuración (c) es algo más flexible, ya que permite diseñar puentes con sensibilidad distinta de la máxima (la configuración (a) proporciona la misma salida sean cuales sean las dos resistencias

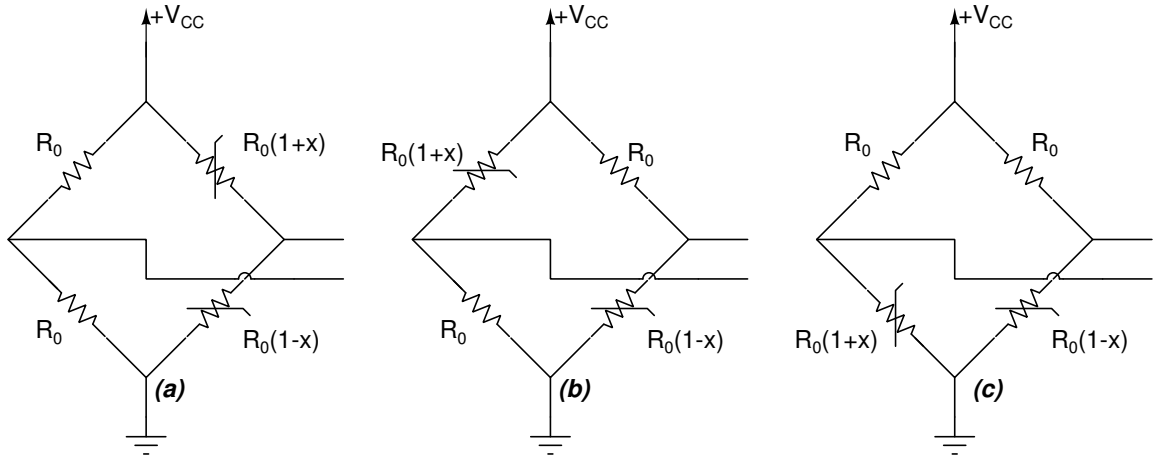

Figura 4. Posibles configuraciones (no triviales) del puente de medida.

del lado izquierdo, siempre que sean iguales), cosa que puede resultar útil si los sensores estuvieran en el límite entre las "pequeñas" y las "grandes" variaciones.

Finalmente, la configuración (a) es más lineal (siempre y cuando las variaciones sean *exactamente* simétricas; el análisis completo es algo más complejo), gracias al hecho de que en la Ecuación 8 las partes variables de los denominadores se anulan entre sí.

Problema 2. Drivers (3 puntos)

a) **(0,5 puntos)** Los únicos circuitos de acondicionamiento que no vienen especificados explícitamente en el enunciado son el del LED infrarrojo y el del motor de corriente continua. Al igual que el pulsador y el interruptor, el LED se ha montado en *pull-up*, aunque la configuración *pull-down* sería igualmente válida siempre y cuando el código sea coherente. En cuanto al motor, sólo hace falta que gire en un sentido por lo que un acondicionamiento con un transistor BJT es suficiente.

El único componente que falta por determinar es la resistencia R_B para que el transistor se comporte como un interruptor digital. Para ello se debe verificar que:

$$I_B \geq \frac{I_{C_{\max}}}{\beta_{\min}} \tag{15}$$

$$I_B = \frac{3,3 - V_{BE}}{R_B} \approx \frac{3,3 - 0,7}{R_B} \tag{16}$$

Suponiendo el caso extremo de que no hay caída de tensión entre el colector y el emisor en saturación ($V_{CE_{\text{sat}}} = 0$):

$$I_{C_{\max}} = \frac{P_{\text{motor}}}{V_{cc} - V_{CE_{\text{sat}}}} = \frac{3}{12 - 0} = 250\,\text{mA} \tag{17}$$

Sustituyendo se llega a que R_B debe verificar:

$$\frac{2,6}{R_B} \geq \frac{250}{50} \Rightarrow R_B \leq 520\,\Omega \tag{18}$$

Por tanto, un valor razonable para estar seguros de que el transistor funciona correctamente en todas las circunstancias sería, por ejemplo, $\boxed{R_B = 270\,\Omega}$.

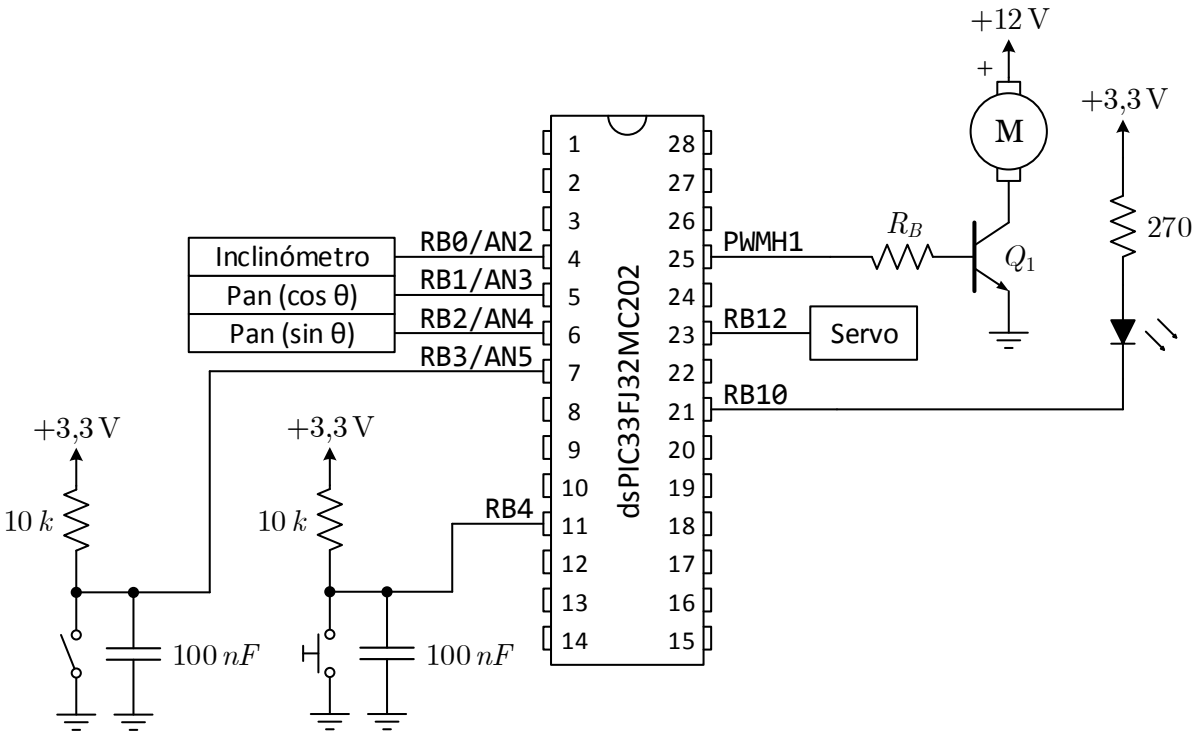

Figura 5. Esquema electrónico del trípode motorizado.

b) **(1,25 puntos)** Ambos sensores son analógicos por lo que hay que emplear el conversor A/D. Dado que la velocidad de conversión no está especificada, se han escogido 500 ksps que es el valor que utilizamos habitualmente. Como se comenta en el enunciado, las operaciones en coma flotante requieren bastantes instrucciones por lo que no es conveniente calcular los ángulos en la interrupción. Una estrategia más inteligente es almacenar simplemente el valor leído por el conversor (en el rango 0 a 1023) y realizar los cálculos complejos sólo cuando se necesite devolver el valor, es decir, en los *getters*:

- El *tilt* se calcula fácilmente ajustando una recta. La única precaución que hay que tener es que si todos los operandos son enteros hay que ordenar las operaciones de tal modo que la división entera no dé siempre cero.
- Del acondicionamiento del sensor magnético de posición se obtienen señales proporcionales a $\sin\theta$ y a $\cos\theta$, que hay que convertir al intervalo [-1, 1] para poder despejar el ángulo de *pan* θ en radianes usando la arcotangente:

$$\tan\theta = \frac{\sin\theta}{\cos\theta} \Rightarrow \theta = \operatorname{atan}\left(\frac{\sin\theta}{\cos\theta}\right)$$

  Finalmente hay que multiplicar por $180/\pi$ para convertir las unidades a grados y cambiar el rango para que vaya de 0º a 359º.

sensores.c

```c
#include <p33FJ32MC202.h>
#include <math.h>
#include "sensores.h"

#define TILT    0  // RB0 (AN2)
#define PAN_COS 1  // RB1 (AN3)
#define PAN_SIN 2  // RB2 (AN4)

#define PI 3.14159265

static unsigned int medida_adc[] = {0, 0, 0};

void inicializarSensores(void) {
   AD1PCFGL &= ~0x001C;
   TRISB |= 0x0007;

   AD1CON3 = 0x0105;     // Muestreo: 1 ciclo, ADCS = 5
   IFS0bits.AD1IF = 0;   // Borrar la bandera
   IEC0bits.AD1IE = 1;   // Habilitar interrupciones
   IPC3bits.AD1IP = 4;   // Prioridad interrupciones
   AD1CON1 = 0x80E0;     // ON, conversión automática
   AD1CHS0 = 2;          // Seleccionar AN2 para empezar
   AD1CON1 |= 1 << 1;    // Empezar a muestrear
}

int getTilt(void) {
   // Otra opción que redondea mejor el resultado sería:
   // return (int)(40 * medida_adc[TILT] / 1023.0 - 20 + 0.5);
   return (40 * medida_adc[TILT]) / 1023 - 20;
}

int getPan(void) {
   float seno   = 2 * medida_adc[PAN_SIN] / 1023.0 - 1;
   float coseno = 2 * medida_adc[PAN_COS] / 1023.0 - 1;
   float angulo = atan2(seno, coseno) * 180 / PI;

   if (angulo < 0.0)
      angulo += 360;

   return (int)(angulo + 0.5); // Se suma 0.5 para redondear los decimales en vez de truncarlos
}

void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void) {
   static unsigned char pin_ad = 2;

   IFS0bits.AD1IF = 0;                  // Borrar la bandera
   medida_adc[pin_ad - 2] = ADC1BUF0;   // Guardar la medida

   pin_ad++;

   if (pin_ad > 4)
      pin_ad = 2;

   AD1CHS0 = pin_ad;    // Seleccionar el nuevo pin A/D
   AD1CON1 |= 1 << 1;   // Empezar a muestrear
}
```

c) **(1,25 puntos)** Como hay que generar trenes de pulsos de frecuencia variable, lo primero es determinar la configuración del timer. En este caso se ha optado por utilizar el Timer 1. Las frecuencias que hay que generar son 1 kHz (T = 1 ms), 2 kHz (T = 0,5 ms) y 4 kHz (T = 0,25 ms). Evidentemente, el máximo común divisor de todos los tiempos anteriores es 0,25 ms, pero no hay que olvidarse de que el factor de servicio de una señal cuadrada es del 50 % por lo que el periodo del timer será de **0,125 ms.**

La única diferencia de este *driver* con respecto a un generador de PWM convencional es que la señal debe generarse durante un tiempo limitado. La forma más sencilla de conseguirlo es encender el timer cuando se quiera enviar una señal y apagarlo al terminar de generar el último pulso. De esta forma se consigue que el timer no esté interrumpiendo inútilmente cuando no es necesario.

disparo.c

```
#include <p33FJ32MC202.h>
#include "disparo.h"

#define PIN_LED 11  // RB11

static unsigned int periodo = 0;

void inicializarDisparo(void) {
   TRISB &= ~(1 << PIN_LED);
   PORTB |= 1 << PIN_LED;  // LED inicialmente apagado (pull-up)

   T1CON = 0x0000;        // Preescalado: 1:1
   PR1   = 5000;          // PR1 = Tiempo * FCY * preescalado = 0,125 ms * 40 MHz * 1:1
   TMR1  = 0;
   IFS0bits.T1IF = 0;
   IEC0bits.T1IE = 1;
   IPC0bits.T1IP = 3;
   T1CON &= ~(1 << 15);  // Mantener el timer apagado
}

void disparoAutomatico(void) {
   periodo = 4;           // T = 4 * 0,125 ms = 0,5 ms (2 kHz)
   T1CON |= 1 << 15;      // Encender el timer
}

void disparoPanorama(void) {
   periodo = 2;           // T = 2 * 0,125 ms = 0,25 ms (4 kHz)
   T1CON |= 1 << 15;      // Encender el timer
}

void construirPanorama(void) {
   periodo = 8;           // T = 8 * 0,125 ms = 1 ms (1 kHz)
   T1CON |= 1 << 15;      // Encender el timer
}
```

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void) {
   static unsigned int ticks  = 0;  // Equivale a 0,125 ms
   static unsigned int pulsos = 0;

   IFS0bits.T1IF = 0;  // Borrar la bandera

   ticks++;

   if (ticks <= periodo / 2) {
      PORTB &= ~(1 << PIN_LED);
   }
   else {
      PORTB |= 1 << PIN_LED;
   }

   if (ticks >= periodo) {
      ticks = 0;
      pulsos++;

      if (pulsos >= 8) {
         pulsos = 0;
         T1CON &= ~(1 << 15);  // Apagar el timer
      }
   }
}
```

Problema 3. Sistema digital (3,5 puntos)

a) **(0,5 puntos)** El sistema tiene dos modos de funcionamiento, automático y panorama. Aunque es posible diseñar una única máquina de estados para ambos modos, queda más claro (y modular) implementarlos por separado. Sin embargo, como los dos modos de funcionamiento no pueden estar activos simultáneamente, hace falta algún mecanismo de coordinación.

Una alternativa es recurrir a lo que se denomina un **gestor de modos**, que no es más que otra máquina de estados (Figura 6). Al arrancar el sistema, el trípode permanece en el estado de reposo hasta que se pulsa el disparador y se inicia la máquina del modo de funcionamiento seleccionado. Cualquier disparo posterior será ignorado hasta que el modo haya terminado. Las máquinas de estado de los dos modos se muestran en la Figura 7.

Como ambas máquinas comparten actuadores, no conviene separar los modos en sendos archivos para no complicar la solución, ya que eso obligaría a desarrollar un *driver* para contar tiempo y controlar el servo. Por ello, el sistema completo se implementará en un único módulo `tripode.h`.

Evidentemente, la señal que determina el periodo del **timer** es la del servo que, según las especificaciones, tiene que funcionar con una resolución de 1º en el rango [–20º, 20º], es decir, que entre 0,5 ms y 2,5 ms debe haber 40 divisiones. Por tanto, el timer se configurará a **50 µs**.

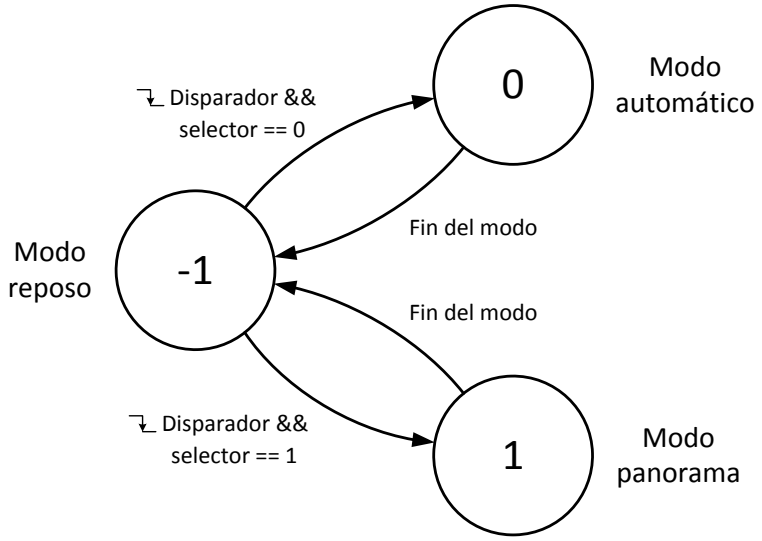

Figura 6. Gestor de modos.

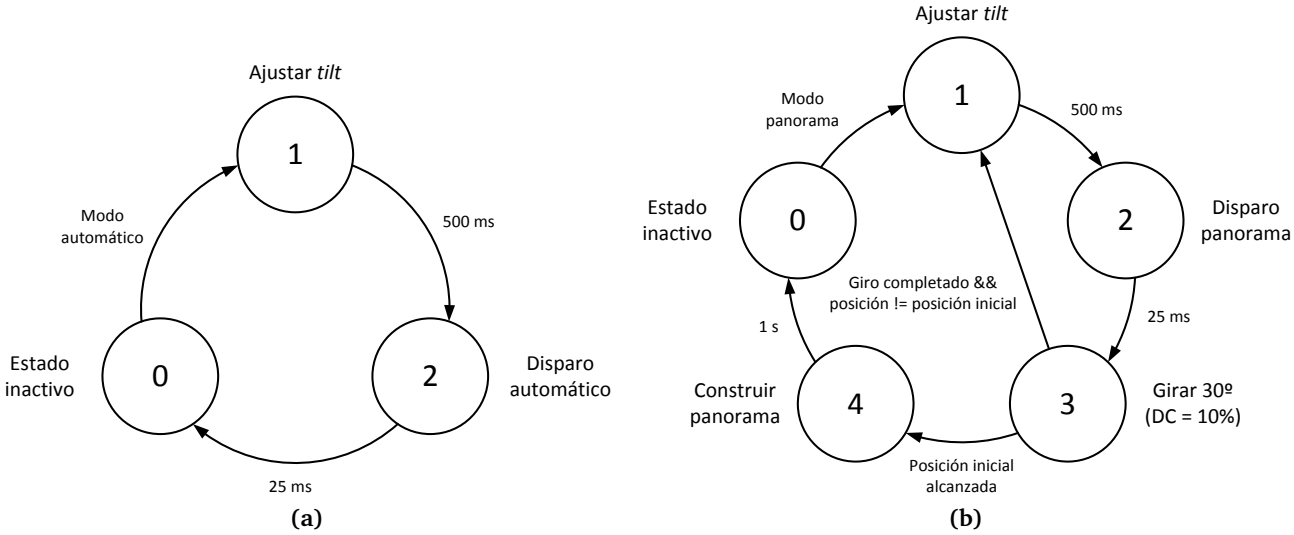

Figura 7. Diagrama de estados de los modos automático (a) y panorama (b).

b) **(3 puntos)**

tripode.h

```
#ifndef _TRIPODE_H
#define _TRIPODE_H

/// Inicializa el trípode motorizado.
/// Utiliza los timers 1 y 2, el periférico generador de PWM
/// y los pines RB0 a RB4, RB10, RB12 y RB14.
void inicializarTripode(void);

/// Máquina de estados del trípode motorizado.
void tareaTripode(void);

#endif
```

tripode.c

```c
#include <p33FJ32MC202.h>
#include "tripode.h"
#include "sensores.h"
#include "disparo.h"

#define PIN_MODO          3  // RB3 (AN5)
#define PIN_DISPARADOR    4  // RB4
#define PIN_SERVO        12  // RB12

#define MODO_REPOSO      -1
#define MODO_AUTOMATICO   0
#define MODO_PANORAMA     1

static int           modo     = -1;
static unsigned int  ticks    =  0;  // Equivalen a 0,05 ms
static unsigned int  ms       =  0;
static unsigned char dc_servo = 30;  // 0º (30 * 0,05 ms = 1,5 ms)

unsigned char modoAutomatico(void);
unsigned char modoPanorama(void);
unsigned int  calcularPosicionObjetivo(unsigned int posicion_actual);
unsigned char comprobarDisparador(void);
void          resetTiempo(void);

void inicializarTripode(void) {
   AD1PCFGL |= 1 << 5;                                  // RB3 - Digital
   TRISB |= (1 << PIN_MODO) | (1 << PIN_DISPARADOR);    // Entradas
   TRISB &= ~(1 << PIN_SERVO);                          // Salida

   T2CON = 0x0000;      // Preescalado: 1:1
   PR2   = 2000;        // PR2 = Tiempo * FCY * preescalado = 0,05 ms * 40 MHz * 1:1
   TMR2  = 0;
   IFS0bits.T2IF = 0;
   IEC0bits.T2IE = 1;
   IPC1bits.T2IP = 3;
   T2CON |= 1 << 15;    // Encender el timer

   configurarPinMotor(PWM_H1);
   inicializarMotor();
   inicializarSensores();
   inicializarDisparo();
}

void tareaTripode(void) {
   unsigned char modo_terminado = 0;

   switch (modo) {
      case MODO_REPOSO:
         if (comprobarDisparador()) {
            modo = (PORTB >> PIN_MODO) & 0x1;
         }
         break;
      case MODO_AUTOMATICO:
         modo_terminado = modoAutomatico();
         break;
```

```
        case MODO_PANORAMA:
            modo_terminado = modoPanorama();
            break;
    }

    if (modo_terminado)
        modo = MODO_REPOSO;
}

unsigned char modoAutomatico(void) {
    static unsigned int estado = 0;

    switch (estado) {
        case 0:
            if (modo == MODO_AUTOMATICO) {
                dc_servo -= getTilt();
                resetTiempo();
                estado = 1;
            }
            break;
        case 1:
            if (ms >= 500) {
                disparoAutomatico();
                resetTiempo();
                estado = 2;
            }
            break;
        case 2:
            if (ms >= 25) {
                estado = 0;
            }
            break;
    }

    return !estado; // TRUE si estado == 0, FALSE en caso contrario
}

unsigned char modoPanorama(void) {
    static unsigned int estado = 0;
    static unsigned int posicion_inicial  = 0;
    static unsigned int posicion_objetivo = 0;

    switch (estado) {
        case 0:
            if (modo == MODO_PANORAMA) {
                dc_servo -= getTilt();
                resetTiempo();
                estado = 1;
            }
            break;
        case 1:
            if (ms >= 500) {
                posicion_inicial = getPan();
                disparoPanorama();
                resetTiempo();
                estado = 2;
            }
            break;
```

```
        case 2:
            if (ms >= 25) {
                posicion_objetivo = calcularPosicionObjetivo(getPan());
                dcMotor(1, 10);
                estado = 3;
            }
            break;
        case 3:
            if (posicion_objetivo == getPan()) {
                dcMotor(1, 0);
                resetTiempo();

                if (posicion_objetivo == posicion_inicial) {
                    construirPanorama();
                    estado = 4;
                }
                else {
                    estado = 1;
                }
            }
            break;
        case 4:
            if (ms >= 1000) {
                estado = 0;
            }
            break;
    }

    return !estado;  // TRUE si estado == 0, FALSE en caso contrario
}

unsigned int calcularPosicionObjetivo(unsigned int posicion_actual) {
    unsigned int posicion_objetivo = posicion_actual + 30;

    if (posicion_objetivo >= 360)
        posicion_objetivo -= 360;

    return posicion_objetivo;
}

unsigned char comprobarDisparador(void) {
    static unsigned char pulsador = 1;
    unsigned char pulsador_ant = pulsador;

    pulsador = (PORTB >> PIN_DISPARADOR) & 0x1;

    return (pulsador < pulsador_ant);
}

void resetTiempo(void) {
    ticks  = 0;
    ms     = 0;
}
```

```
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void) {
   static unsigned int ticks_servo = 0;

   IFS0bits.T2IF = 0;  // Borrar la bandera

   ticks++;
   ticks_servo++;

   if (ticks >= 20) {
      ticks = 0;
      ms++;
   }

   // Generación de la señal de PWM del servo
   if (ticks_servo <= dc_servo)
      PORTB |= 1 << PIN_SERVO;
   else
      PORTB &= ~(1 << PIN_SERVO);

   if (ticks_servo >= 400)
      ticks_servo = 0;
}
```

main.c

```
#include "config.h"
#include "tripode.h"

int main(void) {
    inicializarReloj();
    inicializarTripode();

    while (1) {
        tareaTripode();
    }

    return 0;
}
```