TP Final

22.57 - Microprocesadores y Control

Integrantes:

60066 Capparelli, Nicolás

60050 Mujica Buj, Juan Martín

60420 Olivera, Marcos

60184 Torres, Jorge Pedro

60407 Zannier Díaz, Juan José

Índice

Introducción

Se pide desarrollar un sistema basado en dos ECUs, la primera asociada al motor y la segunda asociada al tablero de control. La ECU del Motor controla la velocidad del motor en función de un acelerador, que se encuentra en la ECU del tablero. Para las ECUs se utilizan microcontroladores msp430. Esta información se envía y se recibe entre ECUs mediante el protocolo de comunicación CAN. Esto a su vez requiere de una placa mcp2515 de cada lado de la comunicación CAN, ya que el microcontrolador utilizado no tiene estas capacidades. Para comunicarse con dicha placa se utiliza protocolo SPI.

El eje del motor tiene, solidario al mismo, un encoder óptico formado por una rueda ranurada y un optoacoplador que detecta dichas ranuras. Se utiliza una rueda de 4 ranuras de aproximadamente 10° de tamaño angular. Los pulsos del optoacoplador se detectan mediante el módulo de Input Capture del microcontrolador. Por otro lado, se deberá accionar una bujía por medio del Output Compare entregando un pulso que comienza cuando el eje del motor se encuentra a phi1 grados respecto del PMS, y cuyo ancho es de (phi2-phi1) grados, donde phi1 y phi2 son ángulos predeterminados modificados mediante una PC por conexión UART con la ECU del tablero. Su valor por defecto es de 230° y 260° grados respectivamente.

Además, desde la PC debe poder visualizarse las RPM actuales y la posición del acelerador; y el acelerador debe poder accionarse desde la PC si se activa un modo remoto. El programa en la PC se desarrolla en Matlab. Para el acelerador se utiliza un potenciómetro en la forma de un joystick.

Lógica del programa

El programa consiste en 3 subprogramas, uno de la ECU Tablero, uno de la ECU Motor y uno en la PC, que se ejecutan en un ciclo, leyendo valores de los sensores, las otras placas o la interfaz con el usuario, y generando señales de actuación para los actuadores, todas ejecutando su programa correspondiente cíclicamente. Para evitar problemas de comunicación, los tres subprogramas mandan mensajes por su protocolo correspondiente (CAN entre ECUs y UART entre ECU Tablero y PC) y esperan a recibir una respuesta.

Un diagrama del funcionamiento del sistema junto con sus variables se encuentra a continuación en la **Figura 1.**

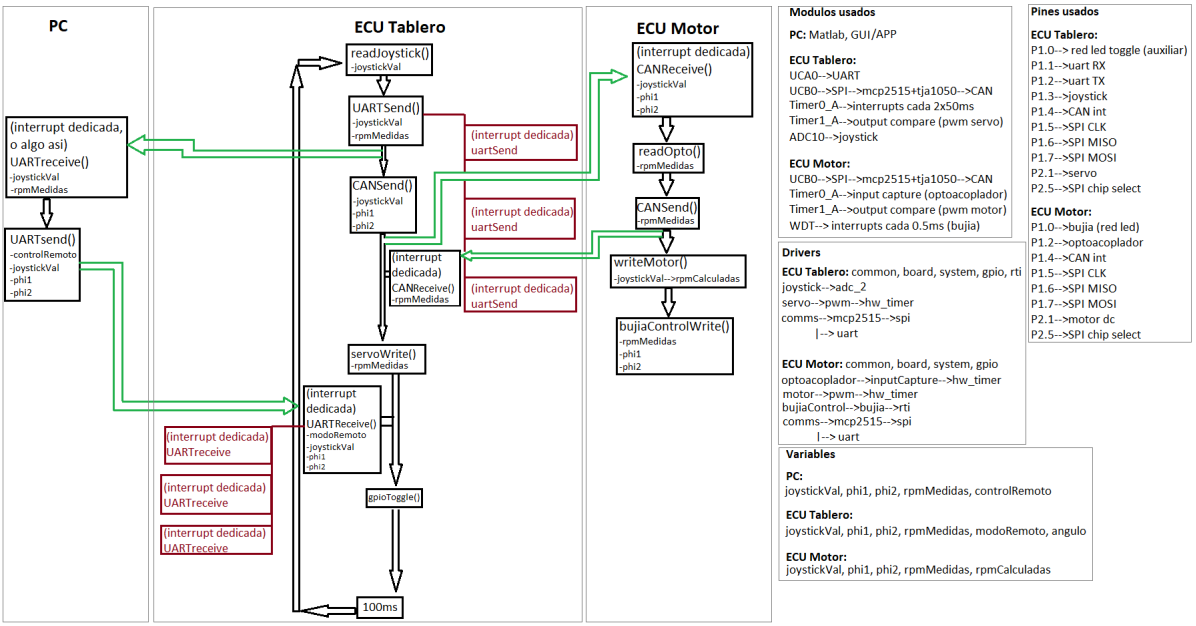

Figura 1. Diagrama de funcionamiento.

ECU Tablero

La ECU Tablero tiene un ciclo cada 100ms. En este ciclo, estando en modo *Manual*, lee el valor del joystick por medio de un ADC y lo escala para su uso entre -10 y 10, manda las variables de interés (valor del joystick, en escala de -10 a 10 y velocidad de giro, en RPM) a la PC por protocolo UART, envía las variables correspondientes (valor del joystick, en escala de -10 a 10, phi1 y phi2, en grados) a la ECU Motor por CAN (por medio de comunicación SPI con la placa traductora) y se queda esperando la respuesta de la ECU Motor por CAN (velocidad de giro medida por el optoacoplador, en RPM). Una vez recibida la velocidad de giro se calcula un promedio de las últimas 4 mediciones para mitigar fluctuaciones en la medición y calcula en qué ángulo debe estar el servomotor para presentar dicho valor en el velocímetro, con una re-escala lineal (0 a 200 RPM pasan a -90° a 90°). Luego, manda la señal correspondiente a dicho ángulo al servomotor y se queda esperando la respuesta de UART (valor de joystick, en escala de -10 a 10, phi1, phi2 y el modo de operación, *'Y'* para *Remoto* o *'N'* para *Manual*). Recibidas estas variables sobrescribe el valor del joystick si el programa se encuentra en modo *Remoto*, donde este valor está dado por el recibido desde la PC. Finalmente, el programa espera el resto de los 100ms para empezar el siguiente ciclo. Para asegurar que sean 100ms se utiliza una interrupción periódica de timer que levanta un flag cada 100ms y se borra al empezar el ciclo.

Nótese el orden en el que el programa manda y recibe datos, primero manda por UART, luego manda y recibe por CAN y luego recibe por UART. Esto se debe a que la comunicación UART es considerablemente más lenta que la comunicación CAN (aun con comunicación SPI intermedia), por lo que se aprovecha el tiempo de espera de UART para mandar y recibir el mensaje por CAN y hacer la lógica correspondiente a este mensaje, y luego esperar hasta que llegue el mensaje UART completo.

ECU Motor

Para el programa de la ECU del Motor, la secuencia es ligeramente diferente. Dado que la ECU Tablero escribe por CAN y luego espera un mensaje, la ECU Motor primero espera el mensaje de la ECU Tablero, obtiene la velocidad de giro medida por el optoacoplador y luego manda su mensaje por CAN, mientras la ECU Tablero está esperando dicho mensaje. Una vez mandadas las RPM, el programa analiza el mensaje recibido y aumenta o disminuye la velocidad del motor (la que se impone por PWM, no la medida) según el valor del acelerador (joystick) y manda la señal correspondiente de PWM por medio del driver del motor. Finalmente, actualiza los valores de velocidad y phi1 y phi2 que utiliza el driver de control de la bujía para saber cuando prender o apagar la bujía. Este ciclo se repite constantemente.

Adicionalmente, cada vez que el optoacoplador detecta un flanco ascendente, es decir, una ranura en la rueda dentada, se impone el ángulo correspondiente (0, 90, 180 o 270, en orden cíclico), para evitar la desviación entre el ángulo de giro calculado y el real.

PC

El programa en la PC es una aplicación ejecutable que espera el mensaje de la ECU Tablero por UART, manda el mensaje de respuesta correspondiente y actualiza sus gráficos y displays. Mediante *sliders* e *input boxes* se permite cambiar los valores a mandar en tiempo real. La **Figura 2** muestra la ventana de la aplicación, y la **Figura 3** explica cada parte de esta.

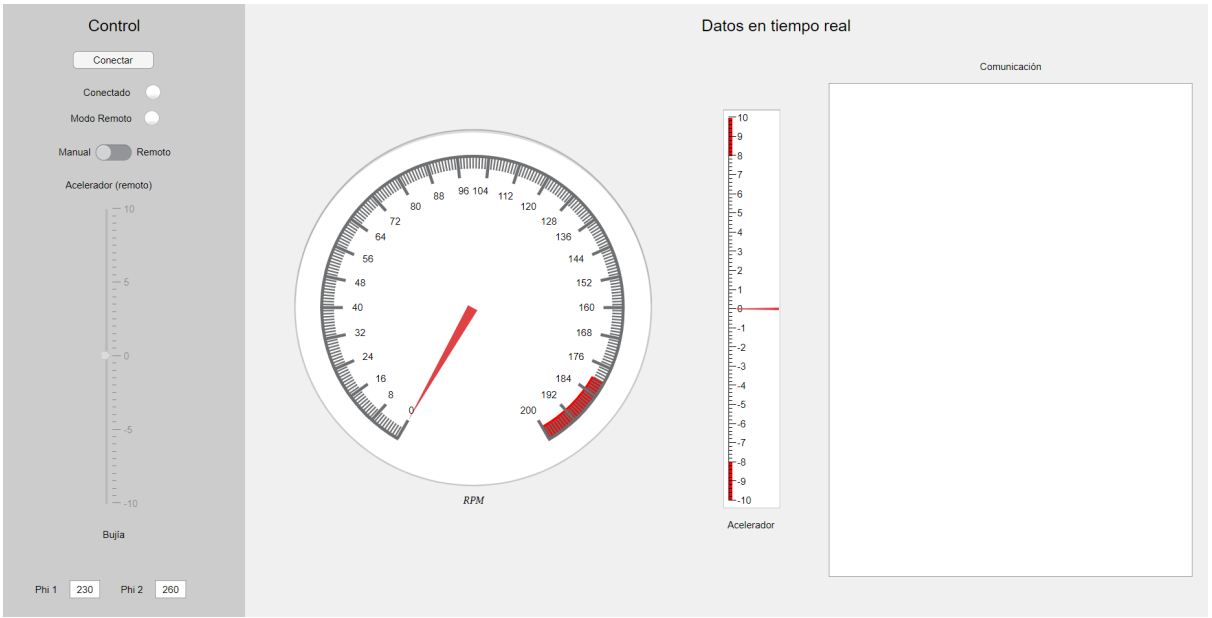

Figura 2. Ventana de control y visualización de datos en la PC.

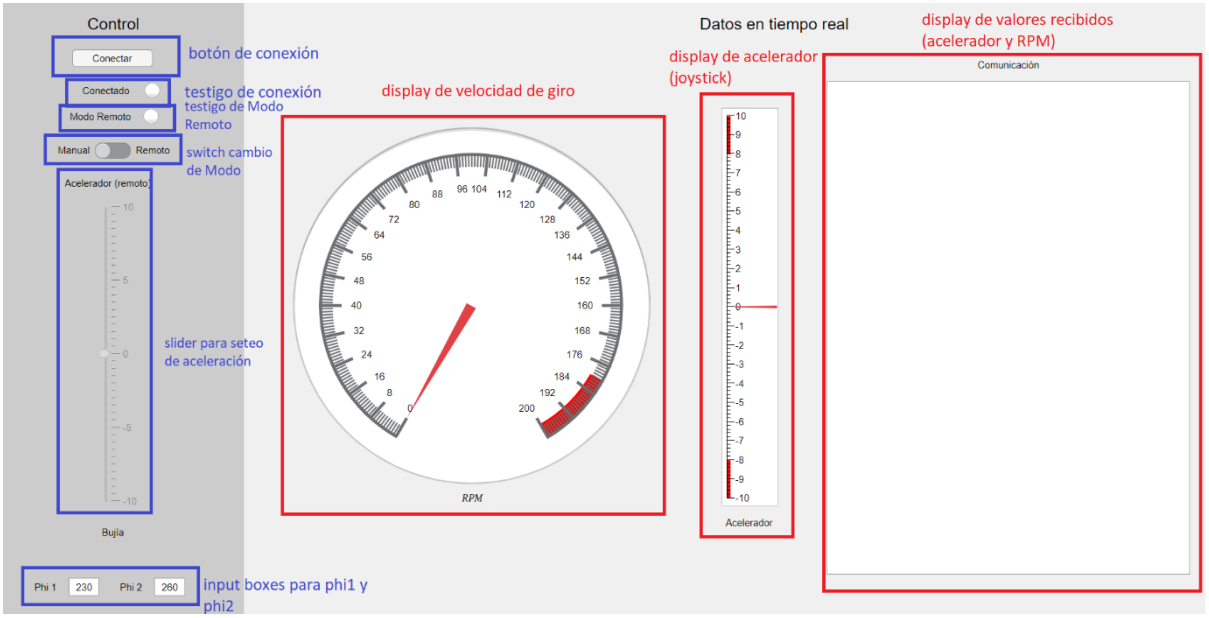

Figura 3. Explicación de ventana de control y visualización de datos en la PC.

Componentes

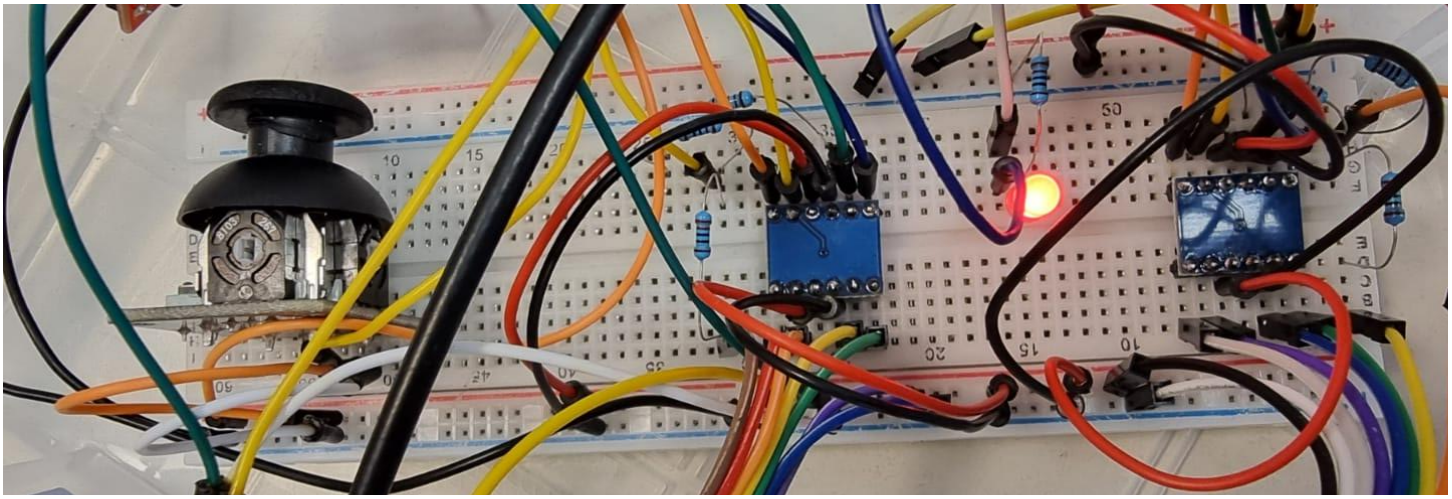

Figura 4. Joystick, conversores de nivel, puente resistivo y led.

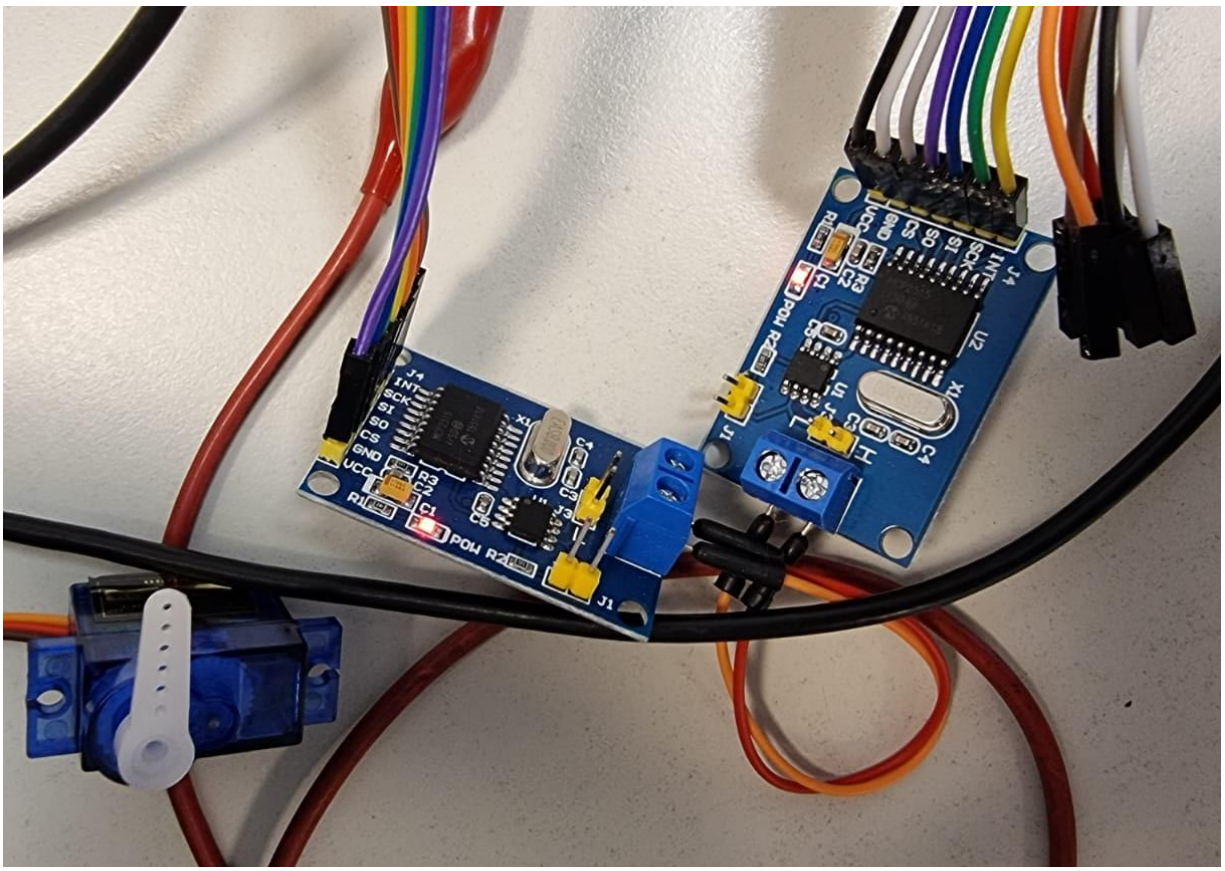

Figura 5. Módulos CAN (mcp2515 y tja1050) y servo.

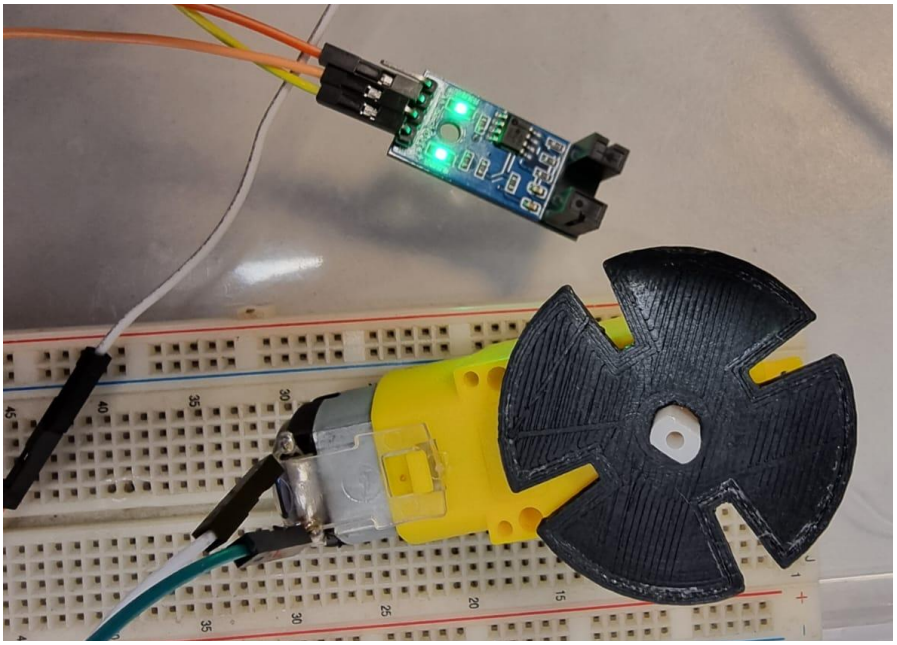

Figura 6. Motor dc con rueda dentada de 4 ranuras y optoacoplador.

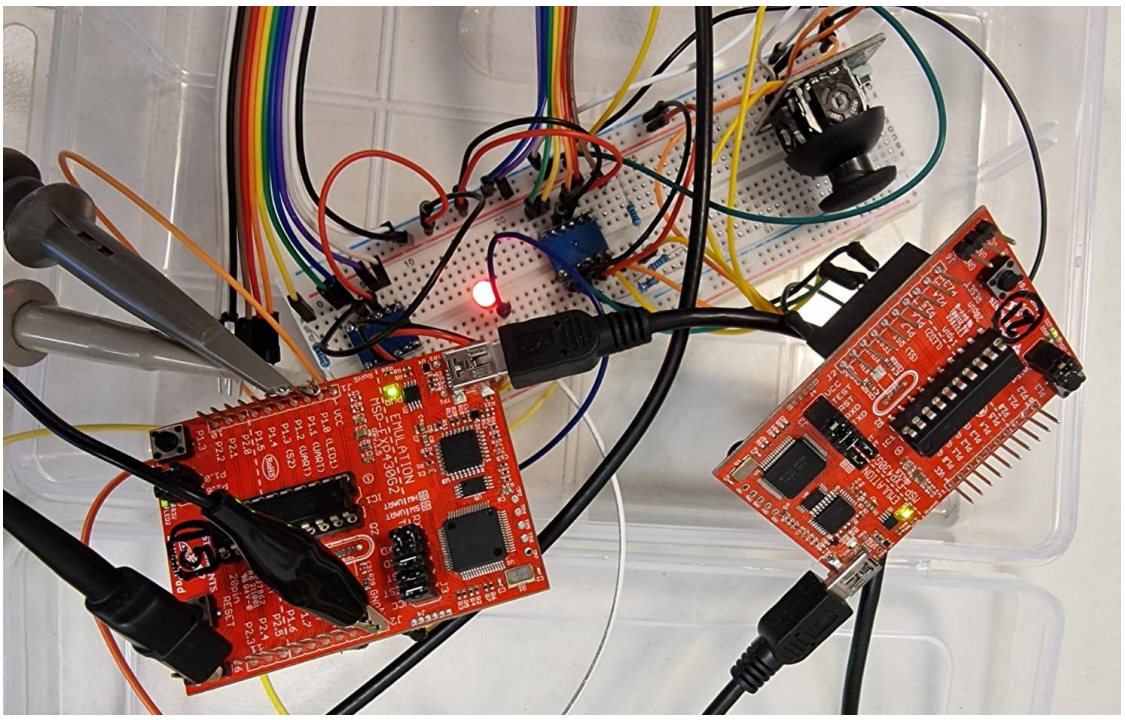

Figura 7. Placas msp430 (ECU Tablero y ECU Motor), con puntas de osciloscopio conectadas para medir señales de bujía y del optoacoplador.

Drivers

Se programan y utilizan una serie de drivers útiles para el sistema en cuestión. Esto incluye drivers de alto nivel para el manejo de los sensores (joystick, optoacoplador) y actuadores utilizados (motor, servo, bujía), así como drivers más particulares como un driver de control de bujía. Además, se incluye una serie de drivers adicionales de más bajo nivel para el manejo los drivers de alto nivel (pwm, input capture, adc) y drivers de aún más bajo nivel para el manejo electrónico de la placa (rti, gpio, hardware timers, mcp2515, system). Por otro lado, también se encuentran drivers de alto nivel para la comunicación (comms) y de bajo nivel para los distintos protocolos (UART, SPI). También se incluyen las librerías common.h y board.h que dan definiciones y macros útiles, así como los pines utilizados en cada placa.

La **Figura 8** muestra un diagrama de los drivers utilizados con su nivel de jerarquía dentro del programa.

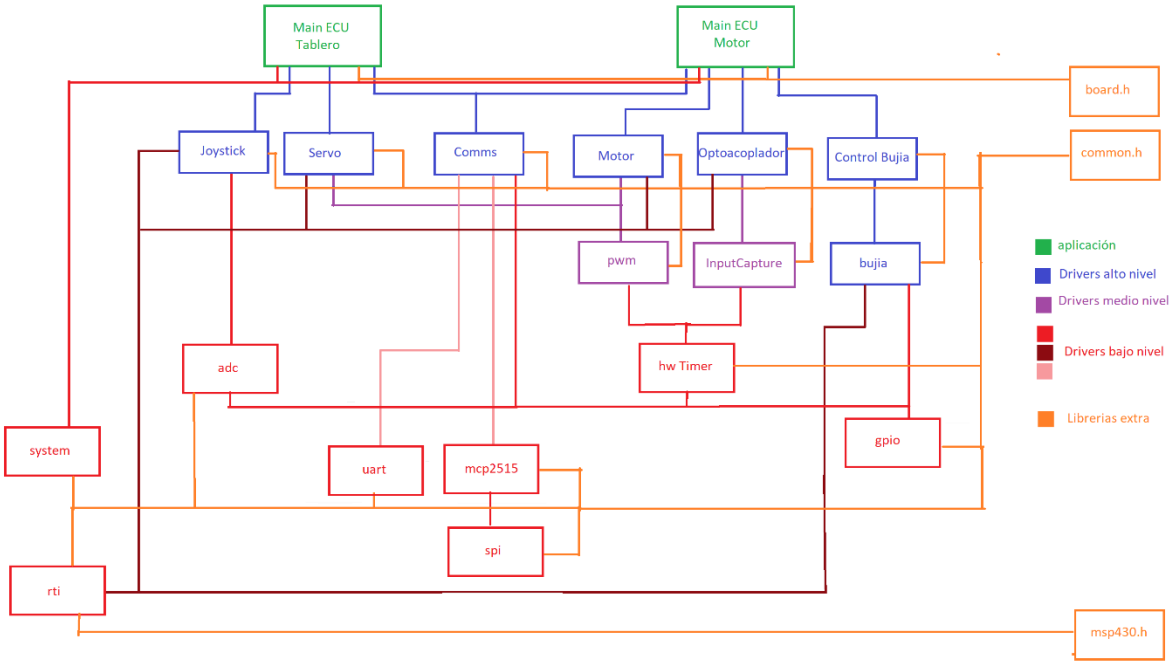

Figura 8. Diagrama de jerarquías de drivers utilizados.

A continuación, se muestra una breve descripción de cada driver con sus funciones.

Bujía

Control de una bujía, señal HIGH para prender la bujía, señal en LOW para apagar la bujía. Incluye una variable *bujiaEncendida* para registrar el estado de la bujía dentro del driver. Tiene la posibilidad de cambiar la señal por medio de interrupciones.

Funciones:

- bujíaInit: inicialización del driver. Requiere el pin de salida y la función de registro a las interrupciones (pasar NULL si no utiliza interrupciones).
- bujiaUpdate: cambia el estado de la señal de salida en el pin dado según *bujiaEncendida.*
- readBujiaVal: devuelve el valor de *bujiaEncendida* registrado.
- writeBujiaVal: sobrescribe el valor de *bujiaEncendida* sin cambiar la señal de salida.
- writeBujia: sobrescribe el valor de *bujiaEncendida* y cambiar la señal de salida.

Joystick

Obtención de valores de un joystick por medio de un ADC. Devuelve un valor de -512 a 511 Incluye una variable *joystickPos* para registrar el valor del joystick dentro del driver. Tiene la posibilidad de leer la señal del joystick por medio de interrupciones.

Funciones:

- joystickInit: inicialización del driver. Requiere el pin de entrada del ADC y la función de registro a las interrupciones (pasar NULL si no utiliza interrupciones). Lee el valor del ADC para usarlo como cero.
- joystickUpdate: lee el valor de la posición del joystick en el pin dado.
- readJoystickVal: devuelve el valor de *joystickPos* registrado.
- writeJoystickVal: sobrescribe el valor de *joystickPos*.
- readJoystick: lee la posición del joystick con el ADC, sobrescribe y devuelve el valor de *joystickPos*

Motor

Control de un motor de corriente continua por medio de una señal de PWM. Se conecta a un driver tipo puente H, no al motor directamente. Incluye una variable *velMotor_x10* para registrar la velocidad actual del motor (multiplicado por 10 para mayor precisión en los cálculos sin utilizar *floats*) dentro del driver. Tiene la posibilidad de cambiar la señal pwm por medio de interrupciones. Utiliza un periodo de pwm de 20ms, con duty cycle variable según *velMotor_x10.*

Funciones:

- motorInit: inicialización del driver. Requiere el pin de salida de pwm, el ID del timer que comanda el pwm y la función de registro a las interrupciones (pasar NULL si no utiliza interrupciones).
- motorUpdate: calcula el tiempo en HIGH dentro del período de PWM, dado *velMotor_x10* (mediante una relación lineal). Limita el duty cycle entre 0.5% y 99.5%.
- readVelMotor: devuelve el valor de *velMotor_x10* registrado, dividio por 10.
- writeVelMotor: sobrescribe el valor de *velMotor_x10* (multiplicando por 10 el valor de entrada) sin cambiar la señal de salida.
- writeMotor: sobrescribe el valor de *velMotor_x10* y cambiar la señal de pwm de salida acorde.
- motorStop: apaga el motor (PWM con duty cycle 0%). Sobrescribe el valor de *velMotor_x10* a 0.

Optoacoplador

Obtención de valor de velocidad de un motor, dado el tiempo entre dos flancos ascendentes de una señal de entrada de un sensor optoacoplador, medida con un módulo *input capture*. Transforma dicha señal en un valor de velocidad, en base a la rueda ranurada utilizada. Incluye una variable *velOpto* para registrar la velocidad medida dentro del driver. Tiene la posibilidad de leer la velocidad por medio de interrupciones.

Funciones:

- optoInit: inicialización del driver. Requiere el pin de entrada del IC, el ID del timer utilizado por el IC, punteros a funciones para ejecutar cuando se detecta un flanco ascendente (que se utiliza para sincronizar el valor de la bujía) y la función de registro a las interrupciones (pasar NULL si no utiliza interrupciones).

- optoUpdate: lee la señal del IC y calcula y sobreescribe *velOpto.* Si el valor del IC es nulo, setea *velOpto* a 0.
- readOptoVal: devuelve el valor de *velOpto* registrado.
- writeOptoVal: sobrescribe el valor de *velOpto*.
- readOpto: lee la señal del IC, calcula, sobrescribe y devuelve el valor de *velOpto*.

Servo

Control de un servomotor por medio de una señal de PWM. Incluye una variable *pos* para registrar la posición angular actual del servo dentro del driver. Tiene la posibilidad de cambiar la señal pwm (y por ende la posición del servo) por medio de interrupciones. Utiliza un periodo de pwm de 20ms, con duty cycle variable según *pos,* de 2,5% (correspondiente a -90°) a 12.5% (correspondiente a +90°).

Funciones:

- servoInit: inicialización del driver. Requiere el pin de salida de pwm, el ID del timer que comanda el pwm y la función de registro a las interrupciones (pasar NULL si no utiliza interrupciones).
- servoUpdate: calcula el tiempo en HIGH dentro del período de PWM, dado *pos* (mediante una relación lineal).
- readServoPos: devuelve el valor de *pos* registrado.
- writeServoPos: sobrescribe el valor de *pos* sin cambiar la señal de salida.
- writeServo: sobrescribe el valor de *pos* y cambiar la señal de pwm de salida acorde.
- servoStop: apaga el servo (PWM con duty cycle 0%). Sobrescribe el valor de *pos* a 0.

InputCapture

Driver para el uso de un timer como input capture. Para medir el tiempo (en base al clock utilizado, 1MHz en este caso) entre flancos ascendentes, flancos descendentes o entre un flanco ascendente y uno descendente. En este caso se utiliza para medir tiempo entre flancos ascendentes. Setea un Timer (especificado en la inicialización del driver) a contar de 0 hasta CCR0 (50.000 en este caso) continuamente y utiliza CCR1 para detectar el flanco ascendente. Genera una interrupción cuando detecta el flanco y guarda el valor en CCR1 y otra cuando detecta el CCR0 y el contador hace overflow. De esta manera se puede calcular la diferencia entre dos valores consecutivos de CCR1 y la cantidad de overflow entre ellos y calcular el tiempo que pasó ente los flancos. Utiliza una variable interna *tBetweenCaptures* que guarda el valor en milisegundos entre flancos.

Funciones:

- inputCaptureInit: inicializa el driver. Requiere el pin a medir, el ID del timer a utilizar, el modo de captura (flancos ascendentes, descendentes o mixto) y funciones extra a llamar cuando se detecta un flanco o overflow.
- inputCaptureRead: devuelve el valor de *tBetweenCaptures*
- rti_Overflow_CCR0: aumenta el numero de overflow ocurridos durante dos detecciones de flancos. Se ejecuta cuando el contador del timer hace overflow. Función privada.

- rti_Capture_CCR1: calcula el valor entre dos flancos detectados, considerando la cantidad de overflows que hubo entre medio. Función privada.

Adc_2

Driver para manejar el módulo ADC del microcontrolador.

Funciones:

- adcInit: Inicializa el driver. Requiere el pin a medir. Realiza todos los seteos correspondientes para una correcta medición del valor de entrada en el pin especificado.
- adcRead: Lee el valor de la señal de entrada, transformado en un valor de 0 a 1023.

Pwm

Driver para la generación de una señal pwm de periodo y tiempo en HIGH (equivalente al duty cycle) especificado. Usa un timer de hardware en modo *Output Compare* (Puede usar Timer0_A o Timer1_A). La generación de la señal se hace a partir de la interrupción del CCR0 (que determina el periodo) y CCR1 (que determina el tiempo en HIGH). Cabe aclarar que sola algunos pines de la placa utilizada pueden utilizarse como salida del *Output Compare*. Este driver soporta hasta 2 señales pwm, uno por timer.

Funciones:

- pwmInit: Inicializa el driver. Requiere el pin de salida, el ID del timer a utilizar, el modo de pwm a generar (HIGH_LOW_PWM), y un periodo y tiempo en HIGH inicial
- pwmWrite: Genera una señal pwm en base al periodo y tiempo en HIGH especificado.
- pwmStop: Apaga la señal pwm.
- update_t_periodo0: Función privada para la formación de la señal de pwm.
- update_t_on0: Función privada para la formación de la señal de pwm.
- update_t_periodo1: Función privada para la formación de la señal de pwm.
- update_t_on1: Función privada para la formación de la señal de pwm.

Comms

Driver de alto nivel para la comunicación CAN y UART. Dados los valores a mandar, genera un mensaje en el formato apropiado, llena los buffers de datos a mandar y da la orden para mandar cuando es necesario y lee los mensajes recibidos y los decodifica. Utiliza los módulos de comunicación UART para comunicarse con la PC y SPI para comunicarse con la placa mcp2515, que realiza la comunicación CAN, ya que el microcontrolador no tiene un módulo de comunicación CAN propio.

Funciones:

- commsInit: inicializa el driver y da la señal de inicializar a los drivers que utiliza. Requiere el largo del mensaje de UART a mandar. Los pines están configurados en la librería board.h

- UARTSend: dado un string de entrada, escribe dicho mensaje en el buffer de transmisión de UART y da la señal de empezar la comunicación.
- CANSend: carga el mensaje a mandar y da la señal de comenzar la comunicación
- UARTRecieve: lee el mensaje recibido en el buffer de recepción de UART.
- CANRecieve: lee el mensaje recibido en el buffer de recepción de CAN de la mcp2515
- UARTStatus: flag de si se recibió un mensaje por UART y está listo para ser leído
- CANStatus: flag en el mcp2515 de si se recibio un mensaje por CAN y esté listo para ser leído.
- CANBuildMsg8: dado un vector de números de 1 byte, genera el mensaje a mandar en el formato correcto.
- CANBuildMsg16: dado un vector de números de 2 byte, genera el mensaje a mandar en el formato correcto.
- CANBuildMsg32: dado un vector de números de 4 byte, genera el mensaje a mandar en el formato correcto.
- CANBuildMsg64: dado un vector de números de 8 byte, genera el mensaje a mandar en el formato correcto.
- CANUnbuildMsg8: dado un mensaje recibido, genera el vector de números de 1 byte correspondiente.
- CANUnbuildMsg16: dado un mensaje recibido, genera el vector de números de 2 byte correspondiente.
- CANUnbuildMsg32: dado un mensaje recibido, genera el vector de números de 4 byte correspondiente.
- CANUnbuildMsg64: dado un mensaje recibido, genera el vector de números de 8 byte correspondiente.

BujiaControl

Driver para el control de una bujía dada una velocidad de giro y dos ángulos, uno de encendido de la bujía y otro del apagado de la misma. Realiza el cálculo del ángulo en el que se encuentra el motor, considerando interrupciones de 0,5 ms (periodo modificable) y la velocidad actual del motor. Los cálculos de ángulos se realizan en centésimas de grado para mayor precisión sin la necesidad de uso de variables de punto flotante. Prende la bujía si el ángulo está entre dos ángulos phi1 y phi2 especificados.

Funciones:

- bujiaControlInit: Inicializa el driver y el driver de bujía que utiliza. Requiere el pin donde se conecta la bujia y los valores iniciales de velocidad y ángulos de encendido y apagado. Chequea que los ángulos provistos sean correctos (phi1<360, phi2<360, phi1<phi2).
- bujiaControlWrite: reescribe los valores de velocidad y ángulos y chequea nuevamente que estos valores sean correctos. Durante esta operación se levanta un flag para evitar que la bujía cambie de estado mientras se están cambiando estas variables. También calcula un delta phi de cuánto se debe incrementar phi en la próxima interrupción, dada la velocidad actual.
- bujiaZero: Seteo a 0 el ángulo del motor.
- bujiaStop: Elimino la función de rti de la lista de funciones registradas para ejecutarse cuando ocurre una interrupción periódica.

- bujiaSetAngle: Sobrescribe un valor específico en el ángulo del motor. Útil para corregir desviaciones entre la realidad y el código.
- rti_bujia: interrupción ejecutada cada 0.5ms, incrementa phi (volviendo a 0 cuando se pasa de 360°) y modifica la bujía cuando corresponda.

Rti

Driver para el manejo de interrupciones de Timer0_A, Timer1_A y WatchDog Timer. Contiene funciones para registrar otras funciones de los drivers para llamar cuando ocurren interrupciones periódicas y funciones para ejecutar dichas funciones. Permite hasta 10 funciones externas en la interrupción. La elección de qué timer usar se realiza por definiciones en common.h (que define si el timer se usa por rti o por el driver hardware timer).

Funciones:

- rtiSubmitCallback: Registra una función para ser llamada en la interrupción periódica. Requiere un puntero a dicha función y la frecuencia cada cual se ejecuta dicha función (se puede determinar que la función se ejecute cada, por ejemplo, 10 interrupciones periódicas. Esto permite hacer varios períodos de ejecución con un solo temporizador).
- rtiClearCallback: Elimina una función del registro anterior.
- recorrerCallbacks: Llama a las funciones registradas una por una, según la frecuencia con la que se ejecuta dicha función. Esta función es la llamada por las interrupciones del timer elegido.

Uart

Driver para utilizar el módulo de comunicación UART nativo a la placa MSP430. Permite mandar y recibir caracteres por comunicación UART. Manda solamente caracteres (valores de 0 a 255). Requiere un acondicionamiento previo del mensaje (realizado en comms). Puede mandar un máximo de 49 caracteres por vez. Se utiliza una comunicación de 9600 baudRate. Versión modificada del driver hecho por Daniel Jacoby.

Funciones

- uartInit: inicializa el driver y el módulo de comunicación UART. Requiere el largo (en caracteres o bytes) del mensaje para inicializar los buffers de transmisión y recepción.
- uartWriteChar: Guarda un caracter en la siguiente posición del buffer de transmisión.
- uartWriteString: Guarda un string de caracteres en el buffer de transmisión.
- uartSend: Da la señal de comenzar la transmisión con el buffer de transmisión actual.
- uartReadChar: Devuele el siguiente carácter (en uint8_t, da el valor ASCII) en el buffer de recepción.
- uartRead: Devuelve todos los caracteres en el buffer de recepción (como un vector de uint8_t del valor ASCII de cada uno).
- uartStatus: Devuelve 1 si hay un mensaje en el buffer de recepción, 0 si no. Borra el flag luego de devolver su estado.
- uart_transmit_rti: rti para la transmisión de datos desde el buffer de transmisión
- uart_receive_rti: rti para la recepción de datos desde el buffer de recepción.

System

Driver para el manejo general del sistema de bajo nivel de la placa msp430, como la inicialización de la misma, la deshabilitación del Watchdog y la configuración de las interrupciones. Versión modificada de driver hecho por Nicolás Magliola.

Gpio

Driver de bajo nivel para manipulación de pines de propósito general (Input/Output). Se utiliza el driver hecho por Nicolás Magliola.

Hw_timer

Driver para el manejo de los timers del MSP430 (Timer0_A y Timer1_A). Permite utilizarse como *Output Compare* e *Input Capture* para generar una señal PWM o interrupciones periódicas (OC) o para contar pulsos o tiempo entre flancos (IC). Versión modificada del driver hecho por Daniel Jacoby.

Mcp2515

Driver para la comunicación cableada con la placa de comunicación CAN (tanto transmisión como recepción). La comunicación se realiza por comunicación SPI. Versión modificada de un driver hecho por K. Evangelos (disponible en https://www.electrodummies.net/en/msp430-2/msp430-can-interface/ fecha de consulta: 5/12/2022).

Spi

Driver para la comunicación con la placa de comunicación CAN. Se encarga de la transmisión y recepción de datos por CAN. Versión modificada de un driver hecho por K. Evangelos (disponible en https://www.electrodummies.net/en/msp430-2/msp430-can-interface/ fecha de consulta: 5/12/2022).

Módulos y pines usados en cada placa

ECU Tablero

Conexiones:

P1.0 --> red led toggle (auxiliar, opcional)

P1.1 --> uart rx

P1.2 --> uart tx
P1.3 --> joystick
P1.4 --> can int
P1.5 --> spi sclk
P1.6 --> spi miso
P1.7 --> spi mosi
P2.1 --> servo
P2.5 --> spi chip select

Módulos:
Timer0_A --> interrupts cada 2x50ms
Timer1_A --> pwm servo
ADC10 --> joystick
USCI_A0 --> uart
USCI_B0 --> spi/can

ECU Motor

Conexiones:
P1.0 --> bujía (red led toggle)
P1.2 --> optoacoplador
P1.4 --> can int
P1.5 --> spi sclk
P1.6 --> spi miso
P1.7 --> spi mosi
P2.1 --> motor dc
P2.5 --> spi chip select

Módulos:
WDT --> interrupts cada 0.5ms para bujía
Timer0_A --> input capture optoacoplador
Timer1_A --> pwm motor dc
USCI_B0 --> spi/can

Notas Adicionales

Comunicación por uart entre ECU Tablero y PC

-ECU Tablero le manda a la PC paquetes de 7 bytes cada 100ms (una vez por ciclo) por UART. Estos contienen: *3 chars de joystickVal* (de -10 a +10: lectura del adc de 0-1023 llevada a +-10), *3 chars de rpmMedidas* (de 000 a 999 rpm, o en verdad hasta 216rpm) y 1 *terminador ('\n')*.

-joystickVal se debe pensar como cuán apretado está el acelerador, el +-10 no tiene en sí unidades, aunque si se quiere, +10 corresponde a +10rpm/100ms = 100rpm/s = 1,67rev/s2 = 10,472rad/s2 (y -10, a -10,472rad/s2).

-Cuando la PC recibe el terminador (el 7mo byte), le responde a ECU Tablero con un paquete de 10 bytes con: con *3 chars de phi1*, *3 chars de phi2* (000 a 360 grados), *1 char de controlRemoto* ('Y' o 'N'), y *3 chars de joystickVal* (de -10 a +10, sólo será tenido en cuenta si controlRemoto='Y').

-Por ejemplo: ECU Tablero manda "+05060\n" (acelJoystick +5→queremos acelerar 5rpm en el próximo ciclo de 100ms; rpmMedidas 60→medimos 60rpm del optoacoplador)

-Y PC responde "240270N+05" (phi1 240°, phi2 270°, controlRemoto: NO/OFF/0, joystickVal (por más que no vaya a ser usada): +5 (5rpm/100ms u otra unidad a definir).

-A un baud rate de 9600 y mandando 10 bits por mensaje de 1 byte (start of frame, 8bit data, end of frame, parity disabled), para un mensaje de 7 bytes tarda 7,29ms y para uno de 10 bytes, 10,42ms. Más un poco más por el tiempo que lleva cargar los transmit buffers/leer los receive buffers.

Comunicación por CAN entre ECU Tablero y ECU Motor

-Así como los mensajes de UART son 'lentos' (baud rate 9600) y una vez que se disparan se van mandando por interrupts dedicadas, los mensajes de CAN son 'rápidos' (usamos 40 kbps, aunque podríamos llegar a 125kbps), una vez que se disparan, el programa solo escribe por SPI al módulo CAN y luego deja que el mcp2515+tja1050 se encargue de codificar el mensaje. La librería mcp2515.h inicializa la comunicación SPI y el módulo CAN, y para enviar o recibir mensajes del otro mcp2515, hace que el msp430 le escriba por SPI en sus registros. Lo que se envía entre mcp2515's son CAN frames, es decir, structs can_t con un id, hasta 8 bytes de data, y otros valores (data length code, extended id). El valor 'status' de la struct can_t guarda data[0] pero no lo usamos.

-ECU Tablero le manda un CAN frame a ECU Motor cada 100ms (una vez por ciclo). Cuando Motor lo recibe por una interrupt dedicada (en la que se setea un status flag), le responde con otro CAN frame. Y ECU Tablero lo recibe por una interrupt dedicada (en la que se setea otro status flag).

-ECU Motor envía 2 bytes (uint16_t) de *rpmMedidas*.

-ECU Tablero envía 5 bytes: 1 de *joystickVal* (int8_t), 2 de *phi1* (uint16_t) y 2 de *phi2* (uint16_t).

-Baud rate del SPI: elegimos 500.000, y del mcp2515: elegimos 40kbps. Para enviar un CAN frame, para un mensaje de 5 bytes, necesitamos escribir 26 bytes por SPI, y luego enviar 87 bits por CAN: tardamos 0.416ms+2.175ms=2,807ms por mensaje (un poco más en realidad porque entre los mcp2515_write() hay 1ms de tiempo de espera, entre los write_many_registers() hay 100us, y cosas así, pero igual 50ms sobran). Así, nos alcanza el tiempo (100ms) para enviar un mensaje desde ECU Tablero, recibirlo en ECU Motor, enviar desde ECU Motor, recibirlo en ECU Tablero, y repetir.

(Con baud rate del SPI de 9600, tardaríamos al menos 22.5ms en dar la instrucción de enviar un mensaje, jugado de tiempo.)

De todas formas, lo que limita el tiempo de ciclo termina siendo la velocidad de la App de Matlab en la PC, que en periodos menores no logra graficar la velocidad ni el acelerador en su GUI.

Sobre rpmMedidas y el servo

La medición del optoacoplador oscila levemente, por ejemplo entre 185 y 193 rpm. Esto hace que el servo y la aguja del plot del velocímetro en la PC con Matlab tambien oscilen de forma poco estética. Como solución, se decide guardar los últimos 4 valores recibidos del optoacoplador, y enviar al servo y a la PC el promedio de esos 4. De esta forma, se obtiene un movimiento más suave de la aguja.

Velocidad del motor dc y bujía

Es un motor de escobillas y reducción 48:1 (tt gearmotor. Brushed dc motor), gira como máximo a 354.16 rpm, en teoría. Cuando hicimos la prueba y lo medimos a 6V con una fuente del laboratorio de mecatrónica, giró a 216rpm aproximadamente. A máxima velocidad, hace una vuelta (360°) cada 277ms aprox.: 0.77ms por grado.

A un ~20% de velocidad por ejemplo (40rpm), hace una vuelta cada 1500ms, 4.16ms por grado.

A menos de 30-20 rpm aproximadamente, se frena, pues el rozamiento vence al torque que le da el pwm.

Para tener, en teoría, buena precisión con los ángulos de la bujía, se debe actualizarla cada menos de 0.77ms, así no se perderán grados.

Sobre las ranuras

Se elige trabajar con una rueda de 4 ranuras en lugar de una sola en el PMS, para poder sincronizar la posición angular del motor con mayor frecuencia (cada 90° en lugar de cada 360°). Además de esto, así se obtiene más cantidad y cercanía de mediciones de velocidad angular.

Para poder visualizar el ángulo del motor (su punto muerto superior en particular) y el encendido y apagado de la bujía, se recurre a dos canales de un osciloscopio: uno mostrará la salida del optoacoplador, y otro la salida de la bujía. Para distinguir la ranura que corresponde al PMS, la misma puede ser más angosta o más ancha. Sólo se exige que cuando el programa se inicie, la primera ranura que vea el optoacoplador sea la del PMS.