**Lógica del programa**: ver “logica del programa.png”. Super resumido, la **ECU Tablero** tiene un ciclo cada **50ms** en el que se comunica por **UART** con la **PC** y por **CAN** con la **ECU Motor**. La **ECU Tablero** hace de **master**, las otras dos de **slave** y le responden a sus mensajes.

-Tenemos la placa **ECU Motor**, la placa **ECU Tablero** y la **PC**.

-**ECU Tablero** hace un ciclo cada **50ms**. Tiene las variables internas **joystickVal**, **anguloServo**, **modoRemoto**, **rpmOpto**, **phi1** y **phi2**. ¿Qué hace en el ciclo?

-Si la variable **modoRemoto** vale **‘N’**, lee el **joystick** (con joystick\_2.h, que usa a adc\_2.h) y guarda su valor en la variable **joystickVal**.

-Arma un paquete de chars y envía **joystickVal** y **rpmOpto** por **UART** a la **PC** (con comms.h, que usa a uart.h).

-Arma un paquete de bytes y envía **joystickVal**, **phi1**, **phi2**, por **CAN** a **ECU Motor** (con comms.h, que usa a mcp2515.h, que usa a spi.h).

-Espera una respuesta por **CAN** de **ECU Motor** hasta que la recibe. Dicha respuesta contiene un valor de **rpmOpto**. Lo guarda.

-Calcula **anguloServo** en base a **rpmOpto** y actúa sobre el **servo** (con servo.h, que usa a pwm.h, que usa a hw\_timer.h).

-Espera una respuesta por **UART** de la **PC** hasta que la recibe. Dicha respuesta contiene un paquete con valores de **phi1**, **phi2**, **modoRemoto** y **joystickVal**. Separa el paquete y guarda **phi1**, **phi2** y **modoRemoto**.

-Si la variable **modoRemoto** vale **‘Y’**, guarda el **joystickVal** que recibió por UART**.**

-Espera hasta que comience el nuevo ciclo.

-**ECU Motor** tiene un pseudociclo que comienza con los mensajes que recibe por **CAN** de **ECU Tablero**. Tiene las variables internas **joystickVal**, **phi1**, **phi2**, **rpmMotor** y **rpmOpto**. El pseudociclo es el siguiente:

-Espera un mensaje por **CAN** de **ECU Tablero**. El mensaje contiene un paquete con valores de **joystickVal**, **phi1** y **phi2**. Los guarda.

-Lee el último valor guardado del **optoacoplador** (con opto.h, que usa a hw\_timer.h) y lo guarda en **rpmOpto**.

-Responde con un mensaje de **CAN** a **ECU Tablero**. El mensaje contiene un valor de **rpmOpto**.

-Calcula **rpmMotor** en base a **joystickVal** y actúa sobre el **motor dc** (con dc\_motor.h, que usa a pwm.h, que usa a hw\_timer.h).

-Guarda valores de **rpmOpto**, **phi1**, **phi2** dentro del driver de la **bujia** (bujia.h, que usa a rti.h).

-Mientras se corre este ciclo, **ECU Motor** tiene **interrupciones periódicas cada 0.5ms** (generadas por el **Wathcdog Timer**, según se pide en system.h). En dichas interrupts, usando el valor de **rpmOpto** y el tiempo entre interrupciones, el driver bujia.h calcula el ángulo **phi** del motor. Si dicho ángulo está entre **phi1** y **phi2**, se enciende la bujía (led, pin de salida). (Con 0.5ms, estamos jugados de tiempo pero alcanza. Así, para aprox 200rpm, podemos tener precisión de 1°).

-El driver opto.hestá constantemente esperando una ranura para medir las rpm’s del motor dc (el driver contiene guardada la última medición, que es la que se pide una vez por ciclo para enviar por **CAN**). Cuando detecta una ranura, se dispara una interrupt dedicada en la que se fija el valor de **phi** del driver bujia.h: se usa la detección de la ranura para sincronizar el trackeo del ángulo phi. Esto se realiza cada 90°, si se tiene un disco con 4 ranuras (espaciadas en 90° entre sí).

-La **PC** tiene otro pseudociclo, si así se lo puede llamar:

-Espera un mensaje por **UART** de **ECU Tablero**. El mensaje contiene un paquete con valores de **joystickVal** y **rpmOpto**. Los guarda y grafica/imprime.

-Responde con un mensaje por **UART** a **ECU Tablero**, que contiene un paquete con valores de **phi1**, **phi2**, **modoRemoto** y **joystickVal**.

-Mientras tanto, posiblemente con una **GUI**, la PC actualiza en tiempo real los valores de **phi1**, **phi2**, **modoRemoto** y **joystickVal** (éste último sólo si **modoRemoto**=**’Y’**) que le está enviando constantemente por **UART** a la **ECU Tablero**.

Otras cuestiones:

Comunicación por uart entre ECU Tablero y PC:

-**ECU Tablero le manda a la PC** paquetes de **7 bytes** cada **50ms** (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 rpmOpto* (de 000 a 999 rpm… o más realista de 30-50 a 215 aprox) 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/50ms = 200rpm/s = 3.33rev/s2 = 20.94rad/s2 (y -10, a -20.94rad/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 serán tenidos en cuenta si controlRemoto=‘Y’).

Alternativa: PC responde después de recibir el primer byte (no sé cómo sería eso). Depende de lo que pueda hacer el Matlab.

-Por ejemplo: ECU Tablero manda “**+05060\n**” (acelJoystick +5→queremos acelerar 5rpm en el próximo ciclo de 50ms; velOpto 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/50ms 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 poquito 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, aka 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 **50ms** (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 *rpmOpto*.

-**ECU Tablero** envía **5 bytes**: **1** de *acelJoystick* (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 asi, pero igual 50ms sobran). Así, nos alcanza el tiempo (50ms) 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.)

Sobre los OC-PWM de la ECU Motor: andamos medio justos de timers pero alcanzan!

Usamos WDT para interrupts periódicas (cada 0.5ms) para la bujía, usamos Timer0\_A para el dc\_motor, y usamos Timer1\_A para el optoacoplador (o al revés Timer0 y 1, no recuerdo pero está en el código).

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

Pero a un ~20% de velocidad por ej (40rpm), hace una vuelta cada 1500ms, 4.16ms por grado

**Módulos usados y pines ocupados 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

Modulos:

Timer0\_A --> interrupts cada 50ms

Timer1\_A --> pwm servo

ADC10 --> joystick

USCI\_A0 --> uart

USCI\_B0 --> spi/can

**ECU Motor**

Conexiones:

P1.0 --> bujia (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

Modulos:

WDT --> interrupts cada 0.5ms para bujia

Timer0\_A --> input capture optoacoplador

Timer1\_A --> pwm motor dc

USCI\_B0 --> spi/can

**Lista de drivers**

Usados para el TP:

bajo nivel: board, common, gpio, rti, system

medio nivel: adc\_2, hw\_timer (FALTA PROBAR), mcp2515, spi, uart

alto nivel: bujia, comms, dc\_motor, joystick\_2, opto (FALTA HACER), pwm, servo

Otros:

medio nivel: adc, i2c, spi\_old\_09\_2022

alto nivel: array7seg, button, counter, display7seg, encoder, joystick, status\_leds, sw\_timer

**Resumen por driver**

*(template)*

*NOMBRE.H*

*llama a: <driver1.h>, <driver2.h>*

*func publicas: func1(), func2()*

*defines: DEFINE\_1, DEFINE\_2, MACRO\_1(), MACRO\_2()*

*enums: lista1, lista2*

*typedefs: typedef1\_t, typedef2\_t*

*otros (como en common)*

*NOMBRE.C: llama a nombre.h, driver3.h. DEFINE\_3, DEFINE\_4. func\_privada1(), func\_privada2(). Otros*

*comentarios.*

*//Más comentarios sobre el driver en general, cómo funciona, cómo llamar a sus funciones, posibles //mejoras/ampliaciones y limitaciones, etc*

Usados para el TP:

BOARD.H

llama a: gpio.h

defines: RED\_LED\_MSK,BUTTON\_MSK,ENCODER\_A\_MSK,SIETE\_SEG\_MSK,DISPLAY\_PORT,etc

COMMON.H

llama a: <stdint.h>, <stdbool.h>, <stdio.h>, <msp430.h>

defines: TIMER0\_USED\_BY\_RTI, TIMER0\_USED\_BY\_HWT, TIMER1\_USED\_BY\_RTI,

TIMER1\_USED\_BY\_HWT, DELAY\_1s, 500ms, 100ms, 10ms, 1ms, 100us, 10us, 1us.

macros utiles, uint8\_t, printf, etc

GPIO.H

llama a: common.h

func publicas: gpioMode(), gpioWrite(), gpioToggle(), gpioRead(), gpioWritePort(), gpioWrite7bit()

defines: LOW,HIGH,INPUT,OUTPUT,INPUT\_PULLUP,PORTNUM2PIN(),etc

typedefs: gpio\_t

GPIO.C: llama a gpio.h

//se puede agregar interrupciones en algun pin

RTI.H

llama a: common.h

func publicas: rtiSubmitCallback(), rtiClearCallback()

typedefs: rti\_callback\_t

RTI.C: llama a rti.h, <msp430.h>. MAX\_CALLBACKS. Usa interrupts dedicadas (para

generar interrupts periodicas) con Timer0\_A, Timer1\_A o Watchdog Timer.

//rtiClearCallback() no esta super chequeado.

SYSTEM.H

llama a: common.h

func publicas: systemInit()

defines: enable\_interrupts(), disable\_interrupts()

enums: clock\_frec, rti\_period, rti\_timer

SYSTEM.C: llama a system.h. stopWDT(), setCLK(), setRTI()

//genera interrupts periodicas con Timer0\_A, Timer\_1 o WDT, segun el timer

//elegido (el periodo debe ser el adecuado para ese timer).

-----------------------------------------------------------------------------------------------------------------------------------------------

ADC\_2.H

llama a: common.h

func publicas: adcInit(), adcRead()

ADC\_2.C: llama a adc\_2.h, gpio.h

//Funciona sin interrupciones. Cuando se llama a adcRead(), se toma unos us para

//realizar la medicion en el momento.

//Por ahora, lee una sola entrada analogica. Se puede programar para que lea

//y guarde mas.

HW\_TIMER.H

llama a: common.h

func publicas: hwTimerInit(), hwSetTimer(), hwTimerGetTicks(), hwTimerIncCCR0(),

hwTimerIncCCR1(), hwTimerSetCCR0(), hwTimerSetCCR1(), hwStopTimer(),

enums: hw\_timer\_mode, hw\_timer\_count\_mode, hw\_timer\_output\_mode,

hw\_timer\_capture\_mode, hw\_timer\_out\_pin

HW\_TIMER.C: llama a hw\_timer.h, gpio.h. Puede usar interrupts dedicadas y funciones

de callback en los interrupt de CCR0 y CCR1.

//Puede usarse en modo INPUT CAPTURE (para contar pulsos o tiempo entre flancos) o

//OUTPUT COMPARE (para pwm o interrupts periodicas).

//Puede usar el Timer0\_A y/o el Timer1\_A.

//Si usa ambos timers, puede usarlos en el mismo modo o en distintos modos (IC/OC).

//Tiene interrupts con CCR0 y CCR1/CCR2. El uso de CCR2 para OC no esta programado,

//no esta contemplado en el driver, idem para los pines de salida que usan Out2 con

//OUTMOD.

//TAR (free running counter) es uint16\_t, de 0 a 65mil. En 1MHz, hace overflow a los

//65ms. Tener en cuenta para ambos modos de uso (IC, OC).

//El modo input capture idealmente usa 1) count up mode, ccr0 callback para logica

//de overflows, ccr1 callback para logica de conteo de tiempo o pulsos; o 2)

//continuous mode, ccr1 callback para logica de overflows, ccr0 callback para logica

//de conteo de tiempo o pulsos.

//Mucha atencion a los pines habilitados para cada modo y cada CCRx a usar.

MCP2515.H

llama a: common.h

func publicas: MCP2515\_init(), MCP2515\_CanVariable\_init(), MCP2515\_can\_tx0(),

MCP2515\_can\_rx0(). En verdad solo esas se usan. Otras:

MCP2515\_spi\_test(), MCP2515\_reset(), MCP2515\_bit\_modify(),

MCP2515\_write(), MCP2515\_write\_many\_registers(), MCP2515\_write\_id(),

MCP2515\_read(), MCP2515\_read\_many\_registers(), MCP2515\_read\_id(),

MCP2515\_can\_tx1(), MCP2515\_can\_tx2(), MCP2515\_can\_rx1(),

MCP2515\_clear\_rx0(), MCP2515\_clear\_rx1(), MCP2515\_int\_clear()

defines: DELAY\_1s,etc, BOOL,TRUE,FALSE, MCP2515\_CS\_LOW, MCP2515\_CS\_HIGH,

mil defines propios del mcp2515 que podrian estar en el .h

typedefs: can\_t

MCP2515.C: llama a: mcp2515.h, spi.h

//Realmente solo se usan las primeras 4 funciones (init, canVariableInit, tx0, rx0).

//Su correcto funcionamiento esta muy unido al driver de spi. Son un paquete que va

//junto.

//Tiene hardcodeado el uso del pin P2.5 como chip select (tambien hardcodeado en

//driver de spi). Debe definirse un pin con interrupciones para la salida INT del

//mcp2515 (por ejemplo, P1.4, ver driver comms).

//Driver para la placa mcp2515+tja1050, cuyo nivel logico alto es 5v (y no 3.3v).

//Para el oscilador nativo de 8MHz, usa 40kbps para CAN (debe estar entre 20 y 125,

//si no recuerdo mal) y 500kbps para SPI.

SPI.H

llama a: common.h

func publicas: SPI\_init(), SPI\_transmit()

SPI.C: llama a spi.h

//Forma un paquete junto con mcp2515. Funciona a 500kbps.

//Para la placa msp430, usa el modulo USCI\_B0 y los pines P1.5, P1.6, P1.7 (SCLK,

//MISO, MOSI).

//Tiene hardcodeado el pin P2.5 para chip select.

UART.H

llama a: common.h

func publicas: uartInit(), uartWriteChar(), uartWriteString(), uartSend(),

uartReadChar(), uartRead(), uartStatus(), uart\_transmit\_rti(),

uart\_receive\_rti()

UART.C: llama a uart.h. uartPutChar(), uartGetChar()

//Usa interrupciones dedicadas para recibir/enviar un char tras otro. Usa USCI\_A0.

//Init() habilita interrupts para recibir: los chars recibidos se acumulan en un

//receive\_buffer. uartRead() lo lee hasta hallar un '\0' (para usar en algun main).

//uartReadChar() solo lee el ultimo char recibido. uartWriteChar(),

//uartWriteString() escriben en un transmit\_buffer. uartSend() habilita interrupts

//para enviar, y va mandando uno tras otro todos los chars acumulados en

//transmit\_buffer (hasta hallar un '\0'). El '\0' no se lee del buffer ni se envia

//por uart, solo lo usan las funciones para saber cuando parar de leer/escribir.

//Si enviara periodicamente chars, puedo usar un uartRxStatus(), uartTxStatus(), con

//los que miro (cada 1ms x ej) si ya se libero el TXBUF. Misma logica para RX.

//necesitaria los status y deshabilitar interrupts dedicadas. Así podría ser otra

//version de este driver.

//Atencion que USCI\_A0 y USCI\_B0 (para uart y spi) tienen el mismo interrupt vector

//y siempre se debe ver quién lo activo. El spi de este paquete de drivers no usa

//interrupts pero en caso de hacerlo, debe haber otro driver que maneje el vector

//de interrupciones de ambos drivers.

-----------------------------------------------------------------------------------------------------------------------------------------------

BUJIA.H

llama a: common.h

func publicas: bujiaInit(), bujiaWrite(), bujiaZero(), bujiaSetAngle(), bujiaStop()

BUJIA.C: llama a bujia.h, rti.h, gpio.h. RTI\_INTERVAL, PERIODO.

//Usa interrupts periodicas para actualizar la posicion angular en funcion de la

//velocidad de giro. Y conociendo la posicion angular, prende una salida (la bujia)

//si se esta entre dos angulos dados.

//Para 200rpm aprox, se recomiendan interrupts cada 0.5ms (para tener una definicion

//de 1°). Para un clock de 1MHz de la msp430, no soporta periodos menores.

//Dentro del driver, para evitar error grande de redondeo, se trabaja con phi\_x100:

//el angulo multiplicado por 100. Asi, trabajando con ints, se tiene precision hasta

//la centesima. Error de redondeo en cada actualizacion menor a 0.005°.

//bujiaStop() utiliza rtiClearCallback(), no esta muy probado.

//El driver soporta una sola bujia. Se podria ampliar.

COMMS.C

llama a: common.h

func publicas: commsInit(), UARTSend(), CANSend(), UARTReceive(), CANReceive(),

UARTStatus(), CANStatus()

COMMS.H: mcp2515.h, uart.h, gpio.h. CAN\_INT.

//Maneja las comunicaciones por UART y CAN. Tiene hardcodeado el uso del pin P1.4

//Para las interrupciones dedicadas de CAN dadas por la señal INT de la placa

//mcp2515+tja1050.

//con commsInit(0) se inicializa solo la comunicacion CAN y no la UART

DC\_MOTOR

llama a: common.h

func publicas: dcMotorInit(), dcMotorWrite(), dcMotorStop()

defines: DC\_VEL\_MAX

enums: dc\_motor\_out\_pin

DC\_MOTOR.C: dc\_motor.h, pwm.h. PERIODO.

//Controla un motor dc por pwm en un pin de salida. Ese pwm usa un timer (0 o 1).

//Atencion con los pines de salida: por las caracteristicas de pwm y hw\_timer, solo

//algunos estan habilitados.

//El driver soporta un solo motor dc. Se podria ampliar.

//dcMotorStop() llama a pwmStop() que llama a hwStopTimer(). No esta chequeado el

//comportamiento de la salida del OUTMOD (si queda apagada o si encendida/apagada

//segun en que valor se freno el free running counter).

JOYSTICK\_2.H

llama a: common.h

func publicas: joystickInit(), joystickRead()

JOYSTICK\_2.C: llama a joystick\_2.h, adc\_2.h. adc2joystick.

//Junto con adc\_2, trabaja sin interrupts periodicas. Solo toma una medicion cuando

//se le pide (y la convierte de 0-1023 a una escala de +-10, con calibracion del 0

//en joystickInit).

//Este driver solo lee la posicion en X del joystick (no nos interesa Y ni SW).

**OPTO.H (FALTA HACER)**

**llama a: common.h**

**func publicas: optoInit(), optoRead(), optoStatus(), optoStop().**

**OPTO.C: llama a opto.h, hw\_timer.h**

**//(FALTA HACER)**

**//agregar a hw\_timer: hwTimerGetCCR0(), hwTimerGetCCR1(), hwTimerGetCCR2().**

**//optoInit elige el timer (0 o 1), el pin (de input), los flancos a detectar (rising, falling o both) y acepta un callback. En base al pin, elige entre hwTimer en modo count up (ccr1 callback cuenta pulsos o tiempo y ccr0 callback cuenta overflows) o continuous count (ccr1 callback es llamada por los overflows y cuenta overflows, ccr0 callback cuenta pulsos o tiempo). Y al callback que cuente los pulsos o el tiempo (el que detecta los flancos), le puede agregar una funcion de callback extra.**

**Si lo queremos hacer bien ad hoc, en un flanco comienza a contar, en el siguiente termina y guarda el valor. En el tercer flanco de nuevo comienza a contar, en el cuarto termina y guarda el valor. Eso sirve para las ranuras de 10°. Esta corriendo y contando constantemente, desde que le doy a optoInit() (y hasta que le doy a optoStop, que llama a hwTimerStop y a chequear qué hace). Con optoRead, levanto el ultimo valor guardado (puedo ver con optoStatus si hay un valor nuevo).**

**Con la funcion de callback extra, puedo llamar a bujiaSetAngle() para sincronizar el angulo de la bujia!**

PWM.H

llama a: common.h

func publicas: pwmInit(), pwmWrite(), pwmStop()

enums: pwm\_mode, pwm\_out\_pin

PWM.C: llama a pwm.h, hw\_timer.h. update\_t\_periodo0(), update\_t\_on0(),

update\_t\_periodo1(), update\_t\_on1().

//Usa hw\_timer en modo OUTPUT COMPARE. Puede usar Timer0\_A o Timer1\_A.

//Actualmente solo esta habilitado el modo HIGH\_LOW (los otros son LOW\_HIGH y

//CENTERED).

//Genera un pwm del periodo y tiempo\_on deseados. Si se desea actualizar estos

//valores, esto se hace en la interrupcion del CCR0 (fin del periodo) y del CCR1

//fin del tiempo\_on) para evitar comportamientos extraños en el ciclo presente y

//actualizar para el siguiente ciclo.

//Atencion con los pines habilitados para pwm, por como esta armado hw\_timer, solo

//se pueden usar algunos.

//El driver soporta hasta dos pwm's, uno por timer. Se podria ampliar.

//pwmStop() llama a hwStopTimer(). No esta chequeado el comportamiento de la salida

//del OUTMOD (si queda apagada o si encendida/apagada segun en que valor se freno

//el free running counter).

SERVO.H

llama a: common.h

func publicas: servoInit(), servoWrite(), servoStop().

enums: servo\_out\_pin.

SERVO.C: llama a servo.h, pwm.h. PERIODO.

//Controla un servo por pwm en un pin de salida. Ese pwm usa un timer (0 o 1).

//Atencion con los pines de salida: por las caracteristicas de pwm y hw\_timer, solo

//algunos estan habilitados.

//El driver soporta un solo servo. Se podria ampliar.

//servoStop() llama a pwmStop() que llama a hwStopTimer(). No esta chequeado el

//comportamiento de la salida del OUTMOD (si queda apagada o si encendida/apagada

//segun en que valor se freno el free running counter).

-----------------------------------------------------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------------------------------------------------

Otros:

ADC.H

llama a: common.h

func publicas: adcInit(), adcRead(), adcStatus()

ADC.C: llama a adc.h, gpio.h, rti.h. RTI\_INTERVAL.

//usa interrupciones periodicas para leer datos cada cierto periodo.

//por ahora, lee una sola entrada analogica. Se puede programar para que lea

//y guarde mas.

I2C.H (ESTA EL TEMPLATE PERO VACIO)

llama a: common.h

func publicas: i2cInit(), despues parecido a uart? (o a una version vieja de uart)

I2C.C: llama a i2c.h

SPI\_OLD\_09\_2022.H

llama a: common.h

func publicas: spiInit(), spiWriteChar(), spiWriteString(), spiSendReceive(),

spiReadChar(), spiRead(), spi\_send\_receive\_rti().

enums: spi\_mode.

SPI\_OLD\_09\_2022.C: llama a spi\_old\_09\_2022.h. spiPutChar(), spiGetChar(), MAX\_CHARS.

//Usa USCI\_B0. Modo master o slave: el master manda el clock, el slave lo recibe.

//Send y receive se dan al mismo tiempo. Slave siempre tiene habilitadas interrupts

//dedicadas de RX; master sólo cuando se quiere comunicar. Funciones: misma logica

//que uart (o a una version vieja de uart).

-----------------------------------------------------------------------------------------------------------------------------------------------

ARRAY7SEG.H

llama a: common.h

func publicas: arrayInit(), arrayWrite(), arrayClear(), arrayBlink()

enums: array7seg\_symbols

ARRAY7SEG.C: llama a array7seg.h, gpio.h, rti.h. NSEG, NSEL, NDISPLAYS,

RTI\_INTERVAL, array7seg\_blink\_rti(), array7seg\_multiplex\_rti().

//usa interrupts periodicas para multiplexar entre los 4 displays, y para

//arrayBlink(). Para esta ultima, se puede desuscribir.

BUTTON.H

llama a: common.h

func publicas: buttonInit(), setButton(), getDataButton(), getStatusButton()

BUTTON.C: llama a button.h, gpio.h, rti.h. MAX\_BUTTONS, RTI\_INTERVAL,

button\_read\_rti()

//usa interrupts periodicas, pero no guarda su callback\_id, pues no tiene

//funcion para desuscribirse

COUNTER.H

llama a: common.h

func publicas: getCounter(), incCounter(), decCounter(), sumToCounter(),

setCounter(), resetCounter(), readCounter().

COUNTER.C: llama a counter.h. MAX\_COUNTERS

DISPLAY7SEG.H

llama a: common.h

func publicas: displayInit(), displayWrite(), displayClear(), displayBlink()

enums: display7seg\_symbols

DISPLAY7SEG.C: llama a display7seg.h, gpio.h, rti.h. NSEG, display7seg\_rti().

//usa interrupts periodicas para displayBlink(). Se puede desuscribir.

ENCODER.H

llama a: common.h

func publicas: encoderInit(), getStatusEncoder(), getDataEncoder()

ENCODER.C: llama a encoder.h, gpio.h, rti.h. RTI\_INTERVAL, encoder\_read\_rti().

//usa interrupciones periodicas para leer datos cada cierto periodo.

JOYSTICK.H

llama a: common.h

func publicas: joystickInit(), joystickRead()

JOYSTICK.C: llama a joystick.h, adc.h

//por ahora es igual a adc.h, pensar que puede tener distinto/extra.

//para este tp, solo lee posicion en X (no nos interesa Y ni SW)

STATUS\_LEDS.H

llama a: common.h

func publicas: statusLedsInit(), statusLedsWrite()

enums: status\_leds\_estados

STATUS\_LEDS.C: llama a status\_leds.h, gpio.h. N\_SEL\_LEDS

SW\_TIMER.H

llama a: common.h

func publicas: swTimerInit(), swSetNewTimer(), swTimerStart(), swTimerStop(),

swTimerContinue(), swGetTime(), swTimerExpired(), swTimerDelay()

defines: SW\_TIMER\_SINGLESHOT, SW\_TIMER\_PERIODIC.

SW\_TIMER.C: llama a sw\_timer.h, rti.h. RTI\_INTERVAL, MAX\_TIMERS, sw\_timer\_rti(),

timer\_callback\_t

//usa interrupciones periodicas para actualizar sus timers.

//es util para arrancar un timer y ver cuando termina. O para iniciarlo y pedir el

//tiempo transcurrido (algunos segundos) en algun momento para ver cuanto paso.