UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

Máster en Software de Sistemas Distribuidos y Empotrados

Sistemas de Tiempo Real Distribuidos

Programación de STR en STM y FreeRTOS - Sistema de Estabilización de un Dron

Alejandro Casanova Martín - bu0383

Madrid, 14 de abril 2024

Índice

Sistema de Estabilización Básico

El sistema de estabilización básico cuenta con cinco tareas y tres recursos compartidos protegidos por *mutex*. De las cinco tareas, cuatro son periódicas y la quinta es esporádica, siendo ésta activada por una interrupción hardware disparada por uno de los GPIOs del microcontrolador. La activación de la tarea esporádica se realiza mediante un semáforo, que será liberado desde la rutina de servicio de interrupción para desbloquear dicha tarea.

En la Figura 1 se muestra el diagrama de bloques simplificado del sistema, con todos los recursos compartidos, tareas y dispositivos. Las flechas negras indican los accesos a los recursos compartidos, indicándose con el sentido de éstas si se trata de una operación de lectura o escritura, y señalándose en verde el nombre de la variable particular a la que se accede. En el caso de los dispositivos, en verde se indican las operaciones que se realizan sobre estos a través de sus interfaces.

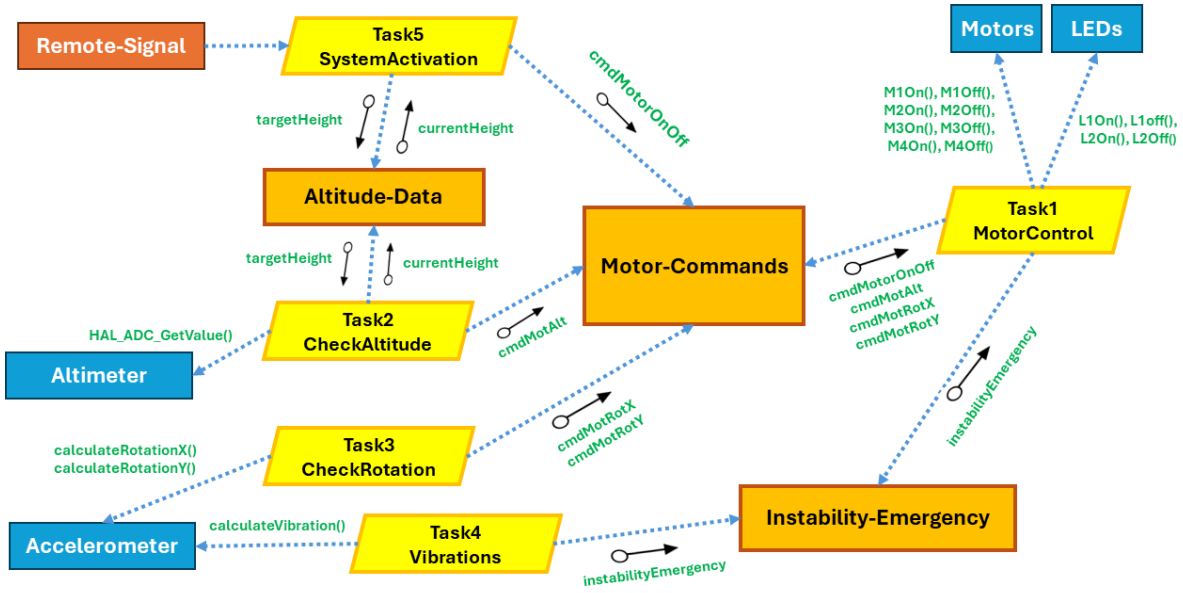

Figura 1. Diagrama de bloques simplificado.

Suposiciones

1. Se ha supuesto que el sistema priorizará corregir la altitud frente a corregir la inclinación. Se trata de una decisión arbitraria, y podría implementarse lo opuesto sin realizar demasiados cambios en el código.
2. Al entrar en el modo de emergencia, se cede el control de los motores a un módulo externo, por lo que desde el sistema de estabilización no podrá abandonarse dicho estado de emergencia de ninguna manera, salvo reiniciando la tarjeta.
3. Al entrar en el estado de emergencia, no sólo se encenderá el LED correspondiente, sino que también se apagará el de estado del sistema, en caso de estar encendido.
4. Se asume que, al activarse el sistema, primero se actualiza la variable de altitud deseada, y a continuación se habilita la actuación sobre los motores, en ese orden.

Tareas

1. **Tarea de control de los motores (MotorControl):**

   La tarea de control de motores maneja únicamente la activación y desactivación de los motores y los LEDs indicadores, en base a los comandos leídos en las variables compartidas escritas por las otras tareas. Al inicio, la tarea verifica si la bandera de *emerFlag* está activa o no. En el caso de que esté activa, se apagan los cuatro motores y el LED de activación. Además, se enciende el LED de emergencia. Caso contrario, se sigue con el programa principal, el cual es leer los valores de las variables compartidas *cmdMotorOnOff, cmdMotRotX, cmdMotRotY* y *cmdMotAlt.* Si es la primera vez que se enciende el dron, se enciende el LED de activación y se activan los cuatro motores. Si el dron ya se encuentra encendido, la tarea solo corrige la altura y enciende los motores individualmente. Finalmente, si el sistema se apaga por el botón externo, se apagan los cuatro motores y el LED de activación.

2. **Tarea de monitorización de altitud (CheckAltitude):**

   La tarea de monitorización de altitud lee siempre los valores del potenciómetro con el uso del ADC. El valor leído en el momento se guarda en la variable compartida *currentHeight.* Luego, verifica si se ha sobrepasado un umbral de pérdida de altitud al comparar el valor leído con el valor de *targetHeight.* En el caso de que la altura actual sea menor que la altura a la que debería mantenerse el dron, se envía el comando para encender los motores. Caso contrario, los motores se mantienen en su posición.

3. **Tarea de monitorización de inclinación: (CheckRotation):**

   La tarea de monitorización de inclinación lee siempre los valores X e Y del acelerómetro. En caso de que X o Y sobrepasen un cierto margen de inclinación, se asigna un valor de 0, 1 o 2 a las variables compartidas *cmdMotRotX* y *cmdMotRotY* que controlan los motores.

4. **Tarea de monitorización de vibraciones (*Vibrations*):**

   La tarea *Vibrations* detecta en tiempo real si el dron ha recibido alguna perturbación o si presenta movimientos bruscos detectados por el acelerómetro. La constante *Vibration_Margin* determina el límite en el cual se presenta o no una vibración por el acelerómetro. Para calcular las vibraciones, la tarea hace lecturas consecutivas del acelerómetro, una tras la otra, para comparar los valores y determinar si el delta es brusco o no. Si la comparación de los valores sobrepasa el límite antes definido, el sistema entiende que se encuentra vibrando bruscamente. Si se presentan más de 3 vibraciones consecutivas, el sistema lo señaliza mediante la variable compartida *Instability-Emergency.*

5. **Tarea de activación/desactivación del sistema (*SystemActivation*):**

   La tarea de activación/desactivación empieza verificando si el sistema se encuentra encendido o apagado. Si inicialmente se encontraba apagado, se asigna la altura de ese momento como la altura objetivo. Luego, verifica que los motores hayan estado apagados previos a su encendido. Como acabamos de actualizar la altitud destino con la altitud actual, el sistema no debería estar corrigiendo altitud en el instante de activarse. Sin embargo, se observó que durante un instante podían parpadear los cuatro motores, debido a un posible comando de corrección de altitud recibido pocos instantes antes a la actualización del valor de la altitud destino. Finalmente, el sistema envía el comando de encendido a la tarea de *MotorControl*. Si el sistema se encuentra encendido inicialmente, solo se envía el comando de apagado a los motores y el sistema se apaga.

Recursos Compartidos

1. **Motor-Commands**: permite al resto de tareas comunicarse con la tarea de control de motores mediante comandos, y consta de las siguientes variables de tipo *uint8_t*:
   - *cmdMotorOnOff*: valdrá 0 o 1 cuando el sistema de estabilización se encuentre activado o desactivado, respectivamente.
   - *cmdMotAlt*: valdrá 0 cuando la altura sea correcta, y 1 cuando deba ser incrementada.
   - *cmdMotRotX* y *cmdMotRotY*: para cada eje de giro, valdrán 0 cuando el ángulo sea correcto, y valdrán 1 o 2 cuando deba corregirse en uno u otro sentido.
2. **Altitude-Data**: almacena la información de altitud y cuenta con las siguientes variables de tipo *int*:
   - *currentHeight*: almacena la altura actual.
   - *targetHeight*: almacena la altura a la que deberá mantenerse el sistema.
3. **Instability-Emergency**: consiste en una única variable de tipo *uint8_t*:
   - *instabilityEmergency*: valdrá 0 normalmente, y 1 cuando se hayan detectado vibraciones y el sistema haya entrado en estado de emergencia.

Dispositivos

1. **Acelerómetro**: realiza medidas de aceleración en los ángulos x, y, z. Su lectura se realiza a través de un bus SPI. Desde el micro se hace un cálculo adicional para derivar de dichas medidas la orientación del dispositivo (de esta manera se simula el giroscopio del que no dispone la tarjeta de desarrollo utilizada). Dado que tanto las tareas de inclinación como de vibraciones hacen uso de él, se trata de un recurso compartido, y su acceso ha sido protegido mediante un *mutex*. De esta manera, también se asegura que los cálculos intermedios realizados sobre sus lecturas se realizan de manera atómica.
2. **Altímetro**: simulado mediante un potenciómetro. Se realiza su lectura mediante el ADC del micro, obteniéndose una lectura entre 0 y 255, dado que el ADC se ha configurado con una resolución de 8 bits.
3. **Motores**: simulados mediante cuatro LEDs de colores de la tarjeta de desarrollo, que son encendidos y apagados mediante los GPIOs del microcontrolador.
4. **LEDs de estado**: uno indica si el sistema de estabilización se encuentra encendido o apagado, y otro indica si el estado de emergencia por vibraciones se encuentra activado.

Montaje (versión 1)

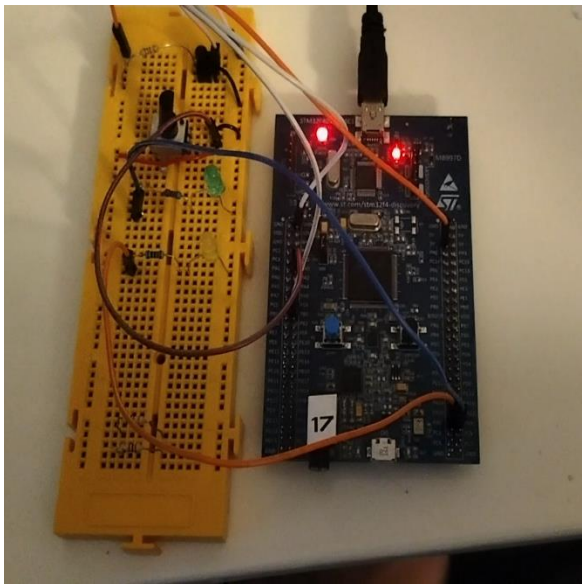

Figura 2. Montaje físico del sistema para las partes 1 y 2 de la práctica.

Estructura del Código (versión 1)

- **Main:** el fichero *main.c* contiene el cuerpo principal del programa. Dentro de la función *main()* se inicializan los periféricos, se crean los semáforos y los *mutex*, y se lanzan las tareas. A continuación, se encuentran las declaraciones de las tareas, cuya implementación quedó descrita en el apartado *Tareas*. También se encuentra aquí la definición de la rutina de manejo de interrupción encargada de liberar el semáforo para despertar a la tarea de activación/desactivación del sistema.
- **Acelerometer:** *accelerometer.c* y *accelerometer.h* contienen las funciones para manejar el acelerómetro. En el fichero de cabecera están definida la interfaz a través de la cual se accederá al dispositivo, para realizar las lecturas tanto de inclinación como de vibraciones. Una función de inicialización incluye la configuración del bus SPI desde el que se accederá al dispositivo, así como la creación del *mutex* que protegerá el acceso al dispositivo desde diferentes tareas. Las funciones de la interfaz del acelerómetro pueden verse representadas en la Figura 1.
- **Led_ctrl:** este fichero de cabecera únicamente encapsula las llamadas a la librería HAL encargadas de encender y apagar, mediante los GPIO del micro, los diferentes LEDs del sistema (tanto los de estado, como los que simulan los motores). Su finalidad es únicamente servir de abstracción para el programador. Dado que las definiciones vienen precedidas de las palabras clave *static inline*, el compilador copiará directamente el cuerpo de las funciones allí donde hayan sido llamadas, de modo que se evitarán saltos de contextos innecesarios en tiempo de ejecución.
- **Shared_resources:** estos ficheros contienen el manejo de los diferentes recursos compartidos. En el fichero de cabecera viene declarada la interfaz de estos, compuesta por sus *getters* y *setters* correspondientes. En la Figura 1 pueden verse las distintas funciones de esta interfaz, y el uso que se hace de ellas desde las tareas. Se cuenta con una función de inicialización, encargada de crear los *mutex* necesarios, que deberá llamarse desde el *main* antes de arrancar el kernel. Para la variable compartida *Altitude-Data*, se ha creado la función *assignTargetHeight()*, encargada de asignar a *targetHeight* el valor de *currentHeight*, de modo que dicha asignación pueda realizarse con una única llamada, y no sea necesario coger el *mutex* dos veces, como ocurriría en caso de tener que llamar primero al *getter* de *currentHeight* para luego asignar dicho valor mediante el *setter* de *targetHeight*.

Implementación con Máquina de Estados

En esta parte se sustituyó la tarea de control de los motores y dos de los recursos compartidos por una máquina de estados o "*Statechart*" (ver la nueva estructura del sistema en la Figura 3). De esta manera se realiza un mejor modelado del comportamiento del sistema, se reduce la carga de trabajo sobre el procesador (dado que se ha eliminado una tarea), se reduce el número de recursos compartidos, y se facilita la depuración del comportamiento del sistema. La propia máquina de estados será un recurso compartido, cuyo valor será su estado actual, y cuyo acceso será protegido mediante un *mutex*.

En la Figura 4 se muestra el diagrama de estados con el que se ha modelado el comportamiento del sistema. En verde se muestran los eventos o entradas del sistema, y en azul/negro las salidas para cada transición. El sistema cuenta con cuatro estados: uno de reposo, uno de posición correcta, y dos de ajuste, para diferenciar si se están realizando correcciones de altitud o de inclinación. En este caso, se ha priorizado la corrección de altitud, por lo que hasta que no se reciba el evento de altitud correcta, no se abandonará el estado de corrección de altitud, y los comandos de corrección de inclinación serán ignorados. En caso de estar en el estado de corrección de inclinación y recibir un comando de corrección de altitud, se abandonará inmediatamente dicho estado para pasar al prioritario, el de corrección de altitud.

El montaje y los dispositivos utilizados son idénticos a los del apartado anterior. El código sufre algunos cambios mínimos, descritos en el apartado *Estructura del Código (versión 2)*.

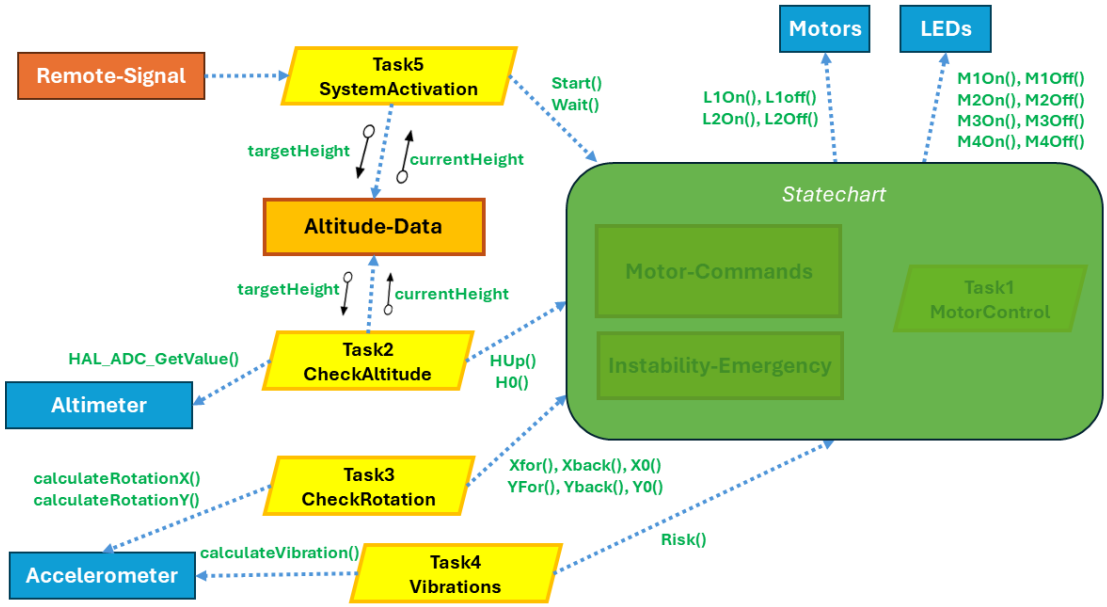

Figura 3. Diagrama de bloques simplificado del sistema (versión con statechart).

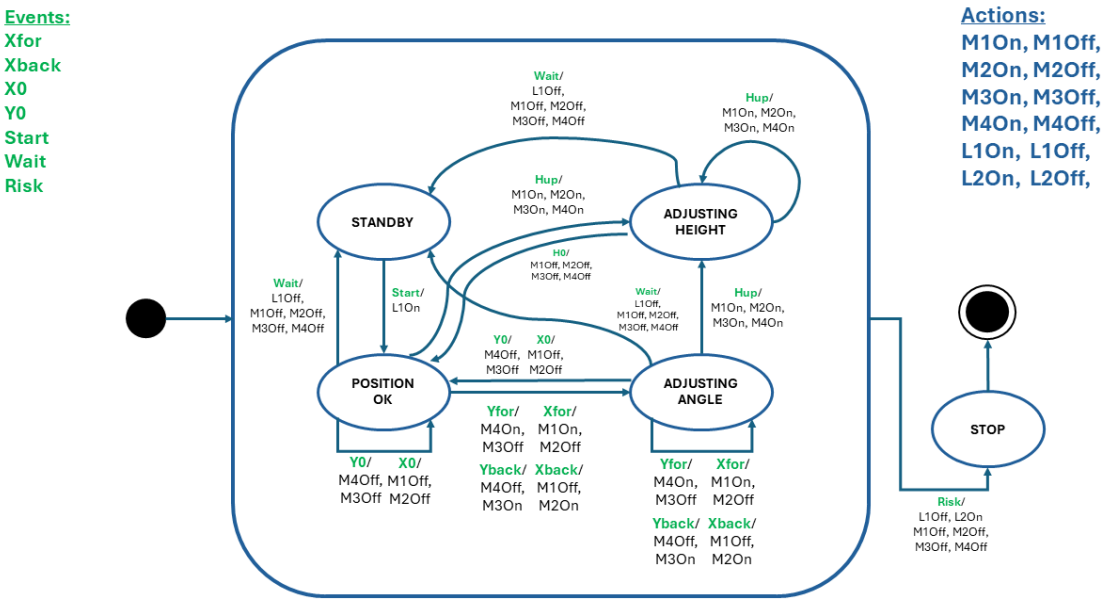

Figura 4. Diagrama de estados del sistema

Estructura del Código (versión 2)

La estructura del código es similar a la del apartado anterior, salvo por algunos cambios, siendo el principal la inclusión de los ficheros *statechart.c* y *statechart.h*, que incorporan la interfaz de la máquina de estados y la definición de sus transiciones (que dependerán del estado actual y del evento recibido). El estado del sistema se almacena en una variable de tipo enumerado, protegida mediante un *mutex*. Una función de inicialización se encarga de la creación del *mutex*, y deberá se llamada desde el *main* antes del arranque del kernel. Las funciones de la interfaz se corresponden con los eventos representados en la Figura 4. En la Figura 3 también pueden verse los diferentes eventos que cada tarea podrá disparar al acceder a la máquina de estados.

El código de las tareas se ha modificado para sustituir las llamadas a los recursos compartidos que han sido eliminados, por las llamadas a los eventos correspondientes del *statechart*. En la Tabla 1 se muestra la equivalencia entre la antigua asignación de valores a las variables compartidas, y los nuevos eventos del statechart.

Variable Compartida	Valor		
	0	1	2
cmdMotOnOff	Wait()	Start()	
cmdMotAlt	H0()	HUp()	
cmdMotRotX	X0()	XFor()	XBack()
cmdMotRotY	Y0()	YFor()	YBack()
instabilityEmergency		Risk()	

Tabla 1. Equivalencia entre los valores de las viejas variables compartidas y los nuevos eventos del statechart.

Implementación Distribuida con Bus CAN

Partiendo del sistema original, se implementó una versión distribuida, separando en un nodo aparte la tarea de control de los motores. El resto de las tareas permanecen en el sistema original, y en lugar de escribirse los comandos de los motores en las variables compartidas, estos son enviados al segundo nodo a través de un bus CAN. Por lo tanto, los recursos compartidos *Motor-Commands* e *Instability-Emergency* han sido sustituidos por el módulo de comunicaciones del bus CAN. Se ha implementado un comando equivalente para cada posible valor de las variables compartidas sustituidas (ver dicha equivalencia en la Tabla 2). En las tareas de ambos nodos, las lecturas/escrituras de las variables compartidas han sido sustituidos por el envío y la recepción de los mensajes por el bus CAN.

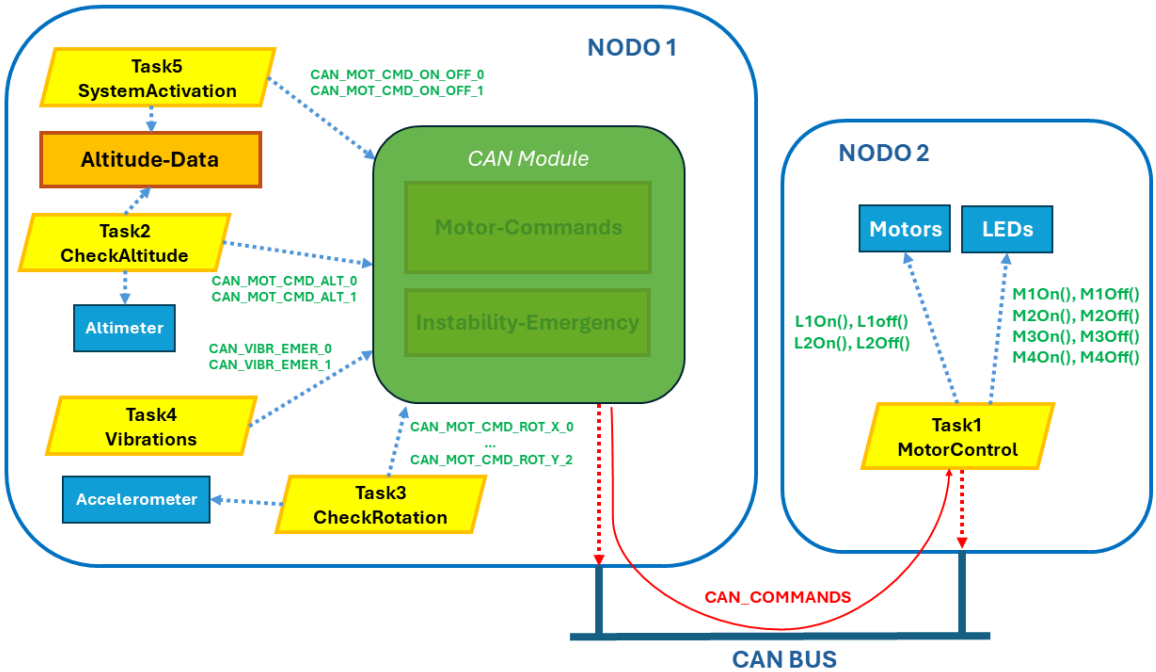

Figura 5. Diagrama de bloques simplificado del sistema (versión distribuida).

Variable Compartida	Valor		
	0	1	2
cmdMotOnOff	CAN_MOT_CMD_ON_OFF_0	CAN_MOT_CMD_ON_OFF_1	
cmdMotAlt	CAN_MOT_CMD_ALT_0	CAN_MOT_CMD_ALT_1	
cmdMotRotX	CAN_MOT_CMD_ROT_X_0	CAN_MOT_CMD_ROT_X_1	CAN_MOT_CMD_ROT_X_2
cmdMotRotY	CAN_MOT_CMD_ROT_Y_0	CAN_MOT_CMD_ROT_Y_1	CAN_MOT_CMD_ROT_Y_2
instability-Emergency	CAN_VIBR_EMER_0	CAN_VIBR_EMER_1	

Tabla 2. Equivalencia entre los valores de las viejas variables compartidas y los nuevos comandos para el bus CAN.

Montaje (Versión 2)

En la Figura 6 se muestra el montaje realizado para la versión distribuida, utilizando una tarjeta STM adicional, y dos transceptores CAN para implementar el bus de comunicación.

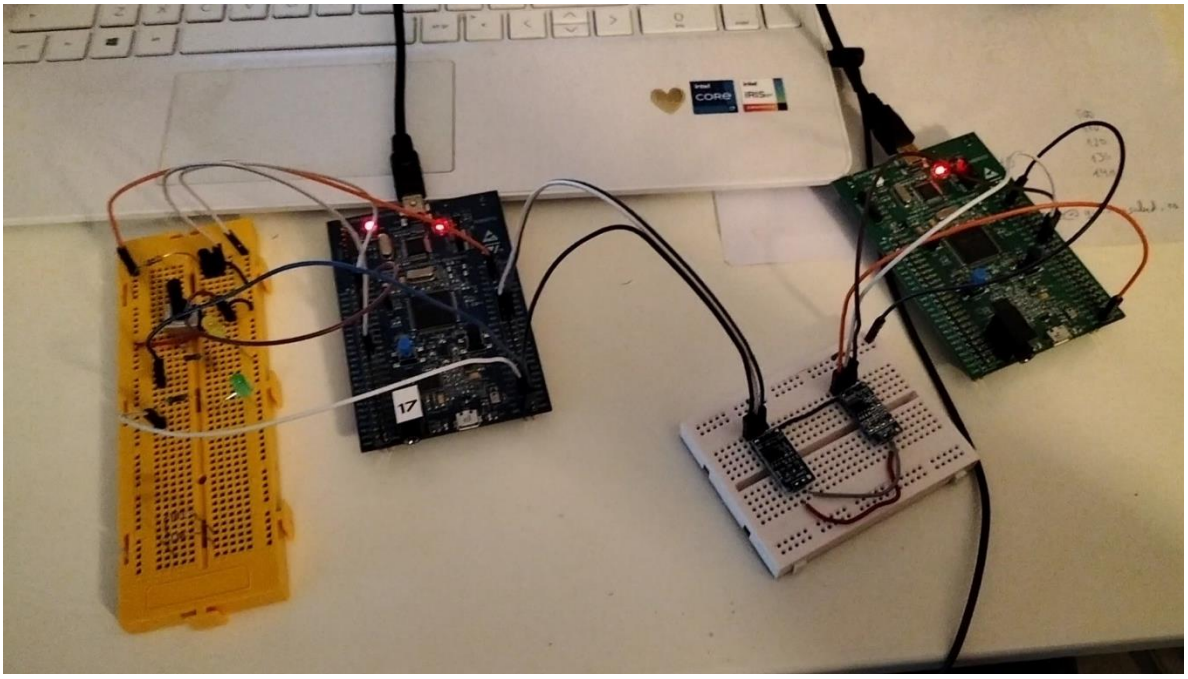

Figura 6. Montaje realizado para la versión distribuida del sistema.

Estructura del Código (Versión 3)

El código es similar al del sistema original, salvo por algunas modificaciones y ficheros adicionales.

- **can_bus:** los ficheros *can_bus.c* y *can_bus.h* contienen la implementación del módulo de comunicaciones para el bus CAN. La interfaz cuenta únicamente con una función de inicialización (que debe ser llamada desde el *main* antes de lanzar el kernel), una función para el envío de mensajes, y un getter para el último valor recibido a través del bus. Finalmente, en el fichero de cabecera también está contenido un enumerado con la definición de los diferentes comandos que podrán ser enviados por el bus.
- **Config:** se ha incluido el fichero config.h para ajustar ciertos valores de configuración del sistema. En este caso, simplemente se ha definido una directiva de preprocesador que indica cuál de los dos nodos se pretende programar. Comentando o descomentando esta línea se conseguirá que se compilen las secciones de código oportunas. En la Figura 7 se puede ver un ejemplo de esto en la inicialización. Dado que la variable CAN_BUS_MODE_TX está definida en el fichero de configuración, se compilará el código encargado de inicializar los periféricos y las variables compartidas, y lanzar todas las tareas menos la de control de motores. En caso de ser comentada la

definición de la variable CAN_BUS_MODE_TX, se compilaría sólo el código de lanzamiento de la tarea de control de motores, que en este caso se encuentra difuminado por la propia IDE. También en el código de inicialización del bus CAN se han incluido directivas de preprocesador para realizar la configuración apropiadamente en función del nodo que se vaya a programar.

- **stm32f4xx_it.**c: en este fichero se encuentra la rutina de manejo de interrupción para la recepción de mensajes por el bus CAN. Dentro de la rutina se ha implementado la señalización del semáforo que se encargará de despertar a la tarea de control de los motores cada vez que se reciba un mensaje.

```
135 #ifdef CAN_BUS_MODE_TX
136   // Initialization for Node 1 (TX)
137   ACC_InitAccelerometer();
138   SHR_Init(); // Init shared resources
139 
140   /* Create tasks */
141   xTaskCreate(ReadRotTask, "readRot", configMINIMAL_STACK_SIZE, 0, PR_TASK3_HOR, 0);
142   xTaskCreate(AltTask, "altitude", configMINIMAL_STACK_SIZE, 0, PR_TASK2_ALT, 0);
143   xTaskCreate(VibrationsTask, "vibrations", configMINIMAL_STACK_SIZE, 0, PR_TASK4_VIB, 0);
144   xTaskCreate(SystemActivationTask, "systemAct", configMINIMAL_STACK_SIZE, 0, PR_TASK5_SYS, 0);
145 #else
146   // Initialization for Node 2 (RX)
147   xTaskCreate(MotCntrlTask, "motCtrl", configMINIMAL_STACK_SIZE, 0, PR_TASK1_MOT, 0);
148 #endif
```

Figura 7. Captura de una sección de código de la función main (versión distribuida). Según la configuración, se compilarán diferentes fragmentos de código, en función de cuál de los nodos se desee programar.