# Capítulo 3

# La Robot Implementation Layer (RIL) y el código asociado

## 3.1 RIL revisited

Decíamos que la realización de la RIL era el objetivo principal del presente proyecto. La *Robot Implementation Layer* es la capa software más cercana al bajo nivel, constituido por el DSP. Está formada en nuestro caso por dos módulos (el Bridge Module y el Status Module) pero no habría problema en añadir más. Estos otros módulos podrían ser procesos independientes, sitos incluso en distintas máquinas, cada uno encargado de una función específica.

La arquitectura impuesta al proyecto nos habla además de una interfaz propia de la relación entre capas, interfaz que será relativamente independiente del robot. Según la jerarquía, la RIL se comunica con el MML (*Module Manager Module*) a través de la IMI (*Inter Modular Interface*). Esta interfaz hace uso de las estructuras definidas en los archivos `GENERAL_IMI_data.h` (comunes a todos los robots) y en `UAV_IMI_data.h` (específicas del UAV). Por otro lado, la relación con el DSP viene definida por la clase `Protocolo_DSP_PC`, en cuyo archivo cabecera vienen además definidas las estructuras utilizadas en el paso de información. Todas estas se ven en la Figura 3.1-1.

Aunque en todo momento se ha buscado la generalidad según la arquitectura general del proyecto, no debemos olvidar que nos encontramos en la zona de dependencia con el tipo de robot. En los siguientes puntos veremos cada uno de los dos módulos de los que se compone la capa RIL.

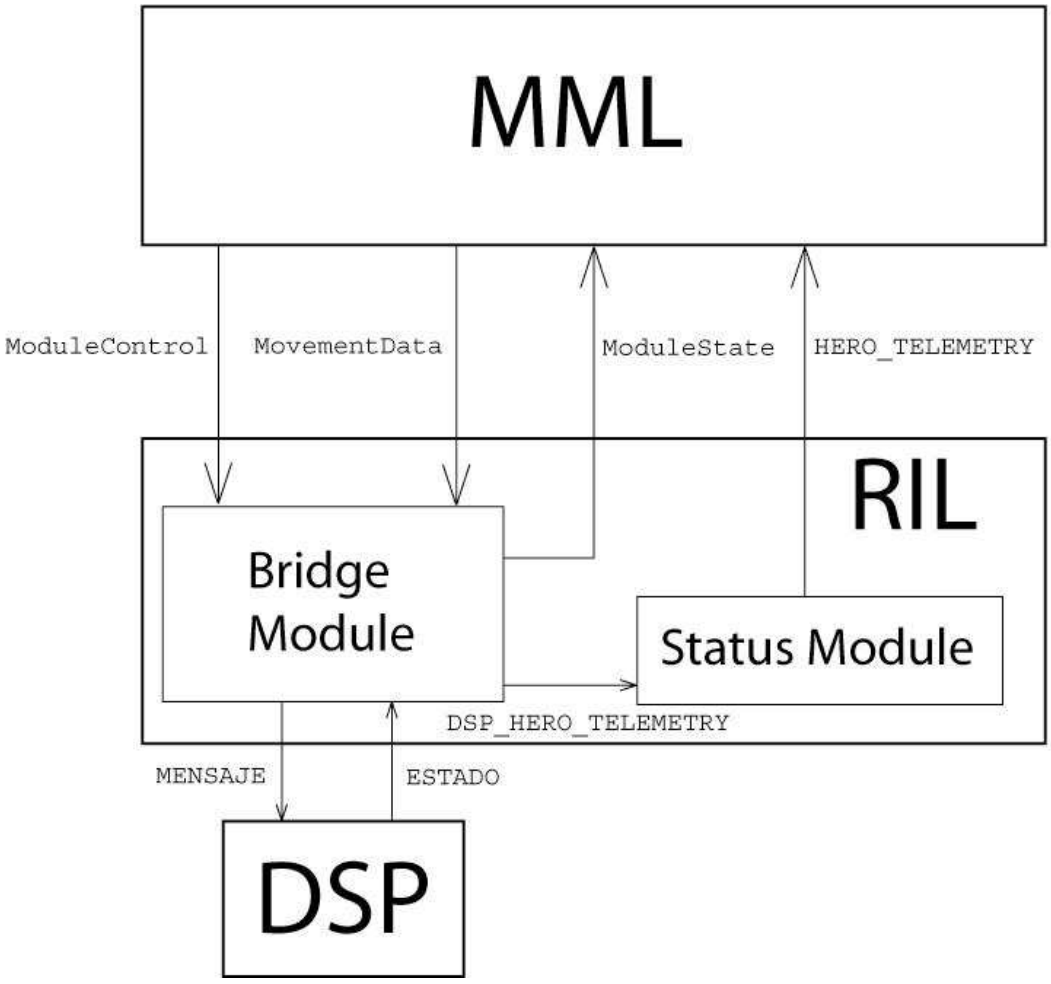


Figura 3.1-1: Tipos de las estructuras que transportan la información

### 3.1.1 Introducción al Software

Para la generación software se contaba con bastante código previo, que se ha intentado reutilizar en la medida de lo posible, adaptándolo a la nueva arquitectura (ver "*Arquitectura general para múltiples robots heterogéneos*" de Antidio Viguria) y en muchos casos al nuevo lenguaje de programación, es decir, portándolo de C a C++. Aunque en éste ultimo caso es sabido que el código no necesita de ninguna modificación en absoluto para funcionar, se ha preferido cambiarlo todo al estilo de C++. Así, se han creado clases con algunas de las funcionalidades ya existentes, después se han creado objetos de dichas clases, a los que dentro del programa nos limitamos a mandar mensajes... en resumen, hemos explotado la esencia de todo lenguaje orientado a objetos.

No hemos podido evitar en el capítulo anterior y ya en este hablar de estructuras de datos, las predecesoras de las clases. Se trata de una potentísima herramienta que ya existía en C, pero que en C++ se vuelve básica. Cuando decimos que el DSP genera un `DSPAck`, un `DSPError` o un `ESTADO_HELICOPTERO`, nos referimos a que genera variables del tipo `DSPAck`, `DSPError` o `ESTADO_HELICOPTERO`. Estos tipos

vienen definidos por nosotros como estructuras, pero a los efectos que nos interesan son como un `int`, un `float` o un `char` (*built-in types*). Para comprender bien el código generado en este proyecto o en cualquier otro, es de extrema importancia conocer qué estructuras se utilizan, dónde se les dan valores coherentes y qué quieren decir los campos que llevan asociados dichos valores.

Durante la reutilización de código han surgido diversos problemas, que en particular iremos comentando si es oportuno a lo largo del texto junto con las soluciones que se han tomado. Pero existe un problemática que es común todo el código: el estilo de programación. El código previo proviene de distintos autores, cada uno con su estilo propio. Lejos de querer comenzar la polémica sobre qué estilo es el mejor, lo que sí es claro es que leer un código que cambia de estilo constantemente es en ocasiones muy confuso. Y dado que en general el código se lee más que se escribe, hemos considerado este punto importante. Por tanto se ha intentado homogeneizar al máximo en este sentido, respetando al mismo tiempo en la medida de lo posible a los autores originales. Un ejemplo de lo que estamos hablando es el idioma en el que se escribe, en lo que se refiere tanto a nombres de variables como a comentarios. Consideramos que lo más conveniente es escribirlo todo en inglés, por una simple cuestión de exportación del código, y así se ha hecho con todo el código nuevo generado en este proyecto. Pero mucho código previo era en español y se ha preferido en esos casos respetarlo y continuarlo así, por lo que en las interactuaciones entre códigos se pueden dar sentencias que parecen sacadas de una canción de *Molotov*, como `dsp_hero_telemetry.heli_volando = NOT_FLYING;`

Para el desarrollo y prueba del software se empezó utilizando dos de los PC disponibles en el laboratorio: *Lostzilla* y *Mercucio*. Concretamente, durante las pruebas, el primero simulaba al DSP mientras que el segundo ejecutaba todas las capas necesarias para el control, comunicándose entre sí vía puerto serie. Por problemas surgidos con estos, tuvimos que pasar a utilizar un único PC, *Motemarte*, que se conectaba a sí mismo ya que contaba con dos puertos serie. Todos tienen instalado *Linux* y las librerías necesarias para el funcionamiento del proyecto (y muy en concreto la librería del *BBCS* que veremos más adelante). La estructura de directorios que contienen al código será siempre la misma y lo más lógica posible; se irá especificando en cada caso. Para ello supondremos que el directorio actual ( `./` ) es el `repository`, y así nuestras rutas no dependerán de dónde se ubique éste en concreto.

## 3.2 Bridge Module

El Bridge Module será el módulo más diferente al resto de módulos de todo el proyecto. Quizá salvando al DSP Simulator, ningún módulo se parecerá a este, ya no en complejidad, sino incluso en arquitectura que como sabemos debería ser prácticamente común al conjunto de módulos. Pero es que el Bridge Module será una excepción en muchos sentidos.

La función del Bridge Module es la de hacer de puente entre el PC y el DSP (de ahí su nombre) pero esto resulta bastante más complejo de lo que pueda parecer inicialmente. Será un puente en el sentido de que pasará los *waypoints* (puntos objetivos) desde capas superiores directamente al DSP, en vez de generar una trayectoria con un criterio determinado. Pero para hacer esto tendrá que codificar y

decodificar según un protocolo establecido, utilizar colas, manejar el puerto serie, gestionar la información que transita y generar información de estado consecuente.

La función de generación de trayectoria se le podría añadir en un futuro conservando la mayoría del código. Toda la gestión de *waypoints* está implementada, y una trayectoria no supone más que una serie de puntos extra que serían generados a partir de los provenientes de capas superiores. Por lo tanto bastaría incluir un hilo generador de la trayectoria y realizar unos pocos cambios para tenerlo en cuenta. Pero para no complicar más por ahora, se ha preferido como hemos dicho, hacer de puente.

Una forma muy sencilla de ver la función del módulo es a través del diagrama siguiente:

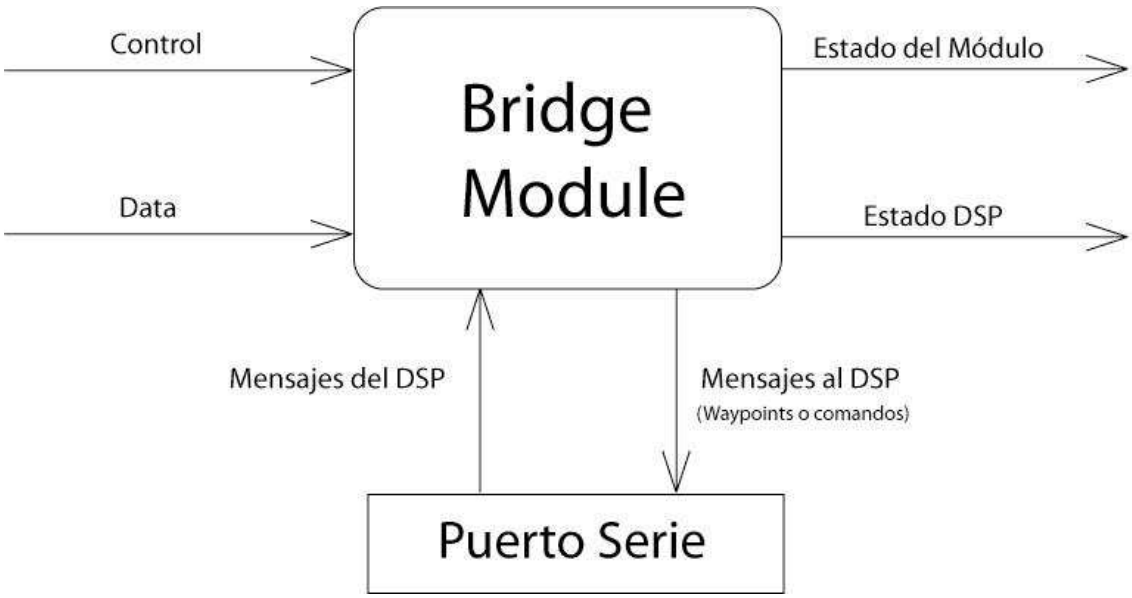


Figura 3.2-1: Bloque del Bridge Module

Es decir, a partir del control y de los datos provenientes de capas superiores (o de otros módulos), el Bridge Module se comunicará vía puerto serie con el DSP (o con un simulador) y devolverá el estado del vehículo según el DSP y el estado del módulo en sí mismo.

## 3.2.1 Implementación software del Bridge Module

La implementación del Bridge Module la lleva a cabo el proceso multi-hilo `BRIDGEMOD`, cuyo único parámetro será el número que identifica al robot al que pertenece. Cuenta con un total de 6 hilos (incluido el `main`) que iremos viendo en esta sección.

**Estructuras**

De la Figura 3.2-1 se observa que, si no contamos las comunicaciones con el puerto serie, nos encontramos con dos entradas (*inputs*) y dos salidas (*outputs*) del módulo. Cada una de ellas tendrá asociada una estructura:

(*input* 1) Control: `ModuleControl` (`./robot_architecture/comms/GENERAL_IMI_data.h`)
Trae información que determina si el módulo debe iniciarse (`START_MODULE`) o pararse (`STOP_MODULE`).

(*input* 2) Data: `MovementData` (`./hero/comms/UAV_IMI_data.h`)
Contiene datos concernientes al movimiento: ordena ir a un *waypoint* (`RS_GOTO_WP`), despegar (`RS_TAKE_OFF`) o aterrizar (`RS_LAND`) y trae los parámetros necesarios para ello.

(*output* 1) Estado Módulo: `ModuleState` (`./robot_architecture/comms/GENERAL_IMI_data.h`)
Como vimos en la descripción de la RIL como sistema del capítulo anterior, la función de un módulo puede: estar ejecutándose (`MODULE_RUNNING`), estar terminada (`MODULE_ENDED`), estar abortada (`MODULE_ABORTED`), o no estar siquiera iniciada (`MODULE_WAITING_DATA_TO_START`).

(*output* 2) Estado DSP: `DSP_HERO_TELEMTRY` (`./hero/comms/UAV_IMI_data.h`)
Contiene información detallada sobre el estado (telemetría) del helicóptero: posición, orientación, velocidades, aceleraciones, señales de actuación, referencias, errores...

**Comunicaciones: generalidades**

Para la gestión de comunicaciones sí se ha llevado una metodología que será común al resto de módulos aunque, como no, nos encontraremos con algunas particularidades para el caso del Bridge Module.

Para la comunicación entre hilos se dispone de una sección crítica, o como en este caso de varias secciones críticas, aunque siempre todas englobadas dentro de un objeto de la clase `CxxxCriticalSection` (para el caso del Bridge Module esta clase se llamará `CBridgeModCriticalSection`). Cada sección crítica implementa un *mutex*. Un mutex es un ente software que permite proteger una región de memoria compartida. Puede hacerse una analogía con un candado de la única puerta a esa región de memoria: cuando hacemos un *Lock* nadie más podrá entrar hasta que no hagamos un *Unlock*. Esto permite que las variables dentro de dicha región sean siempre coherentes a pesar de que varios hilos tengan acceso a ellas.

Las entradas y salidas del módulo (distintas a aquellas que transmiten a través del puerto serie) serán gestionadas por un hilo de comunicaciones que seguirá el estándar propuesto por la "*Arquitectura general para múltiples robots heterogéneos*" y que se vale del protocolo BBCS de comunicaciones.

Las comunicaciones con el DSP son totalmente independientes, según un `Protocolo_DSP_PC` que veremos más adelante y se realizan a través del puerto serie gracias a la clase `PuertoSerie` que también veremos.

*El checksum*

Para cerciorarnos de que la información que circula entre DSP y PC llega a su destino igual que partió de su origen se utilizará el *checksum*. Como la comunicación se realiza por puerto serie carácter a carácter, será muy sencillo y bastante fiable: cada uno de los recibidos (de un mismo mensaje, claro) se van sumando en una variable que llamamos *checksum*. El último carácter del mensaje no se suma, sino que se compara con el *checksum* obtenido: si no ha habido modificaciones indeseadas en el camino o pérdidas de información, ambos valores deben ser idénticos, pues en el último carácter se había metido el mismo *checksum* realizado durante el envío al mensaje. Es decir, tanto el emisor como el receptor del mensaje hacen un *checksum* del mismo (que no es más que una suma de los caracteres que lo componen) y, de comprobarse que ambos valores son iguales, se supone que el mensaje ha llegado tal y como se envió. En realidad el *checksum* no es infalible y en nuestro caso concreto será algo menos fiable debido a que la suma se recorta en todo caso a 7 bits (como todos los caracteres no cabecera de mensaje, como veremos), con lo que si, por ejemplo, el *checksum* de envío y el de recepción debieran diferir sólo a partir del octavo bit, tal circunstancia se perdería y no detectaríamos fallo de transmisión.

**Comunicaciones: BBCS**

El BBCS (*BlackBoard Communication System*) es una API que gestiona las comunicaciones en procesos distintos, independientemente de que estos estén en una misma máquina, en distintas máquinas o bajo sistemas operativos diferentes. Una vez iniciadas y establecidas las comunicaciones, cada nodo ve al resto de nodos iguales, independientemente de la máquina donde se ubique o la forma de comunicarse. Las comunicaciones pueden realizarse en red, según distintos protocolos, o por puerto serie.
Su nombre proviene de la analogía con una pizarra: cada proceso atiende a una serie de variables existentes en la pizarra: cuando un proceso escribe en ella para cambiar el valor de una variable, el resto de procesos que atienden a dicha variable reciben tal información. Aunque esto no es tan sencillo como parece, el caso es que simplifica en gran medida tanto la transmisión de datos como su estructuración.

En la estructura actual, todos los procesos pertenecientes a un robot se comunican con un *Relay Node* propio que se comunica con el resto de *Relay Nodes* según una configuración en estrella. Ni esta tipología ni el código (a excepción de la clase `CXXXCommunication`) dependen en realidad del sistema de comunicaciones, por lo que se podría usar cualquier otro (en vez de BBCS) si se considerara oportuno y no habría que cambiar prácticamente nada. En la carpeta `./BBCS` se encuentra todo el código. Aquí deberíamos comentar que hay que tener cuidado, ya que BBCS se escribe todo mayúsculas. Parece una obviedad, pero *Windows* no siempre opina lo mismo.

Para los dos módulos con los que actualmente cuenta el RIL, el Bridge Module y el Status Module (que veremos en la sección 3.3) las clases que usamos, heredadas de `CBBCSFunctions` (`./BBCS/header/BBCS.h`), son como también veremos `CBridgeModCommunication` y `CStatusModCommunication` respectivamente. Al igual que el simulador más realista del helicóptero, este software ha sido cedido por la *Technische Universität Berlin* (TUB).

En el fichero `./robot_architecture/comms/NetPorts.h` se definen los puertos locales y remotos de todo el sistema, necesarios para la comunicación. Se ha modificado el que existía anteriormente para que contenga los relativos a los módulos del UAV HERO (`HERO_BRIDGE_MODULE_LOCAL_PORT`, `HERO_BRIDGE_MODULE_REMOTE_PORT`, `HERO_STATUS_MODULE_LOCAL_PORT`, `HERO_STATUS_MODULE_REMOTE_PORT`).

En estas comunicaciones existen *slots* seguros y no seguros, según sea importante que el dato llegue a su destino, o que lo importante sea que dicho dato sea lo más reciente posible. Así, del control y los datos nos interesa que no se pierda ninguna información, mientras que de las salidas (estados del módulo y del DSP) lo importante es que la información sea actual. Todos los slots para los módulos del HERO vienen definidos en `./hero/comms/UAV_IMI_slots.h` y siempre dentro del intervalo [3000, 3999]. Para el Bridge Module los slots serán los mostrados en el siguiente cuadro, donde `i/o` indica si son slots de entrada o de salida (*input* / *output*) para el módulo en particular y en la columna de *Secure* se indica con un `1` si es seguro y con `0` de lo contrario:

| | Nombre Slot | Secure | Descripción |
|---|---|---|---|
| i | `UAV_SLOT_MODULE_CONTROL(`*`robot_id`*`)` | 1 | Control |
| | `UAV_BRIDGE_DATA_SLOT(`*`robot_id`*`)` | 1 | Data |
| o | `UAV_SLOT_DSP_SENSORS_STATE(`*`robot_id`*`)` | 0 | Valor de los sensores del DSP (*) |
| | `UAV_BRIDGE_MODULE_STATUS_SLOT(`*`robot_id`*`)` | 0 | Estado del módulo |

(*) Este slot será compartido con el Status Module que veremos más adelante. La diferencia es que para este último será una entrada (como es lógico).

**Hilos**

Las comunicaciones con la capa MML que acabamos de ver las gestiona, como dijimos, el hilo de comunicaciones que siguiendo la nomenclatura de la arquitectura se encontrará en `bridgemodcommunicationthread.cpp`.

El hilo de procesamiento acatará las órdenes de control y movimiento, empaquetando los mensajes que proceda enviar al DSP. Para asegurarnos en lo posible de que dichos mensajes llegan correctamente, el hilo de comunicaciones del DSP (en `dspcommunicationthread.cpp`) los gestionará, recibiendo los *ack* y el control de flujo.

Dos hilos, uno de escritura y otro de lectura se encargarán de estar continuamente mandando y recibiendo información a través del puerto serie.

Todos los hilos serán lanzados por el hilo principal, función `main` que podemos encontrar en el archivo `main.cpp`.

Lo ideal sería leer todo este análisis con el código por delante, pero comprendemos que esto no es siempre posible y en todo caso puede ser lioso. Por ello intentaremos explicarlo todo en un orden y modo tal que sea lo más comprensible posible. A continuación vamos a analizar con detalle todo el código del proyecto concerniente al Bridge Module. Veremos clases, cabeceras e hilos con detenimiento, pero para el caso concreto de las clases se recomienda que se eche un vistazo a los cuadros resumen de su interfaz, que es la parte realmente útil de una clase, en el anexo I. A no ser que se especifique de otra forma, todos los archivos comentados se encuentran dentro de la carpeta `./hero/ril/bridge/source`

**Clase `CBridgeModCommunication`** (en `bridgemodcommunication.h/ bridgemodcommunication.cpp`)

Definición e implementación de la clase, heredada de `CBBCSFunctions` que añade los cuatro slots del módulo (control, data, estado módulo, estado DSP) apropiadamente seguros (`1`) o no (`0`) y define las funciones (dos *Send* - para estado módulo y estado DSP - y dos *Receive* - para control y data -) que sirven de interfaz inmediata con la sección crítica del proceso (o mejor dicho con la parte de ella destinada a la comunicación con BBCS). Para esto último se sirve de las funciones apropiadas de la clase `CBridgeModCriticalSection`. Define además otras dos funciones que realizan la correcta inicialización tanto del objeto en sí como de las comunicaciones, con lo que tenemos 6 en total:

```
bool SendBridgeModuleState(CBridgeModCriticalSection* );
```
Envía el estado del módulo.

```
bool SendBridgeModuleDSPStatus(CBridgeModCriticalSection* );
```
Envía el estado del helicóptero proveniente del DSP.

```
bool ReceiveBridgeModuleControlData(CBridgeModCriticalSection*);
```
Recibe los datos de control para este módulo.

```
bool
ReceiveBridgeModuleMovementData(CBridgeModCriticalSection*);
```
Recibe los datos relacionados con el movimiento.

```
bool Init(void);
```
Inicializa las variables internas para la comunicación.

```
bool InitCommunications(void);
```
Inicializa las comunicaciones.

La clase `CBridgeModCommunication` será únicamente utilizada por el hilo `CommunicationThread` (en `bridgemodcommunicationthread.cpp`). Esto va en total armonía con el resto de procesos de la arquitectura. Al ser heredada puede usar también funciones de `CBBCSFunctions`, como `SetConnections-FromFile`, `Synchronize`, `CheckChannels`, `Finalize`, o `KillChannels`.

**Clase `CBridgeModCriticalSection`** (en `bridgemodcriticalsection.h/ bridgemodcriticalsection .cpp`)

Definición e implementación de la clase. Dicha clase ha ido sufriendo una verdadera evolución a lo largo del proyecto. Teniendo en cuenta que cualquier comunicación entre distintos hilos del proceso `BRIDGEMOD` se realiza a través de la sección crítica (que podemos abreviar en inglés como *CS*), es decir, a través de una estancia de esta clase, cada vez que considerábamos un nuevo tipo de información a transmitir, esta clase crecía.

En un principio se pensó en una clase heredada de `CCriticalSection`, pero pronto nos dimos cuenta de que nos haría falta más de un mutex, así que se optó por la composición de clases. Un objeto de la clase `CBridgeModCriticalSection` contendrá tantos objetos de la clase `CCriticalSection` como mutex sean necesarios. De hecho se tiene un mutex por cada relación entre hilos, además de mutex específicos para variables que no podemos asociar a una interacción concreta entre hilos.

Se declaran los distintos *Set* y *Get* (poner a un valor la variable o tomar el valor de la variable) para las variables, que podrán ser simples variables o conjuntos ordenados de ellas (`CList` o `Cola`). Esto se hace así porque puede o no interesar perder un valor concreto de la variable en un instante. Un ejemplo muy claro son las variables que se conectan con los slots de E/S del módulo: es muy importante distinguir en este caso entre variables `CList` para elementos seguros y simples variables para elementos no seguros, en cada caso del tipo adecuado. Esto tiene su lógica ya que no tendría sentido tener un slot seguro para que después en la sección crítica pudiera ser pisado sin ser leído: es decir, para dejar de ser seguro al llegar a la *CS*.

Tanto los *Set* como los *Get* toman un puntero de la variable a escribir/leer, es decir el paso del parámetro es siempre por referencia.

Tras diversas ampliaciones, finalmente se tienen los mutex:

(*CS1*) Comunicación hilo de procesamiento - hilo de comunicaciones (`BMprocBMcommCS`)

Protege la coherencia de la información de entrada y salida del módulo (distinta de la comunicación serie), es decir: estado del DSP, estado del módulo, control y data.

(*CS2*) Comunicación hilo de procesamiento - hilo de comunicaciones del DSP (`BMprocDSPcommCS`)

Protege la coherencia de la información intercambiada entre los hilos `ProcessingThread` - `DSPCommunicationThread`, es decir, los mensajes que el primero envía conteniendo *waypoints* o conteniendo comandos y los que puede devolver el último en caso de que haya un error de asentimiento (*ack*).

(*CS3*) Comunicación hilo de procesamiento - hilo lectura puerto serie (`BMprocPSreadCS`)

En un principio además de encargarse de los errores de DSP también lo hacía del Estado del DSP, pero se vio que era mejor meter directamente en *CS1* el

DSP_HERO_TELEMETRY ya que el hecho de que fuera actual dicha información era más importante que el ser sistemático.

(*CS4*) Comunicación hilo de comunicaciones del DSP - hilo lectura puerto serie (DSPcommPSreadCS)

Protege la coherencia de la información proveniente del DSP concerniente a asentimiento de comandos y control de flujo.

(*CS5*) Comunicación hilo de comunicaciones del DSP - hilo escritura puerto serie (DSPcommPSwriteCS)

Asegura la coherencia de los mensajes dirigidos al DSP por puerto serie.

(*CS6*) Tiempo (DSPtimeCS)

Protege exclusivamente a la variable dsp_time que por el momento no tiene aplicación y es herencia de una parte del código preexistente, pero que se entiende puede ser utilizada para contener el tiempo local al DSP.

(*CS7*) Estado de la funcionalidad (FncStateCS)

Tiene asignada la seguridad de las variables asociadas al estado de la funcionalidad del módulo, que veremos más adelante qué quiere decir.

Como podemos apreciar, la sección crítica es en este caso bastante más compleja de lo habitual. Contiene un total de 35 funciones (sin contar constructor y destructor) que usan 7 mutex distintos. En las variables de almacenamiento se mezclan CLists, Colas y simples variables (ya sean de nuestros tipos definidos o de *built-in types*) dependiendo de las necesidades. Todo esto se ve mejor en el esquema que mostramos en la Figura 3.2-2.

El constructor se limita a limpiar las CLists (por si acaso) y a hacer los memset necesarios para cerciorarse de que hay espacio suficiente en memoria para todas las variables. El destructor sólo limpia las CLists; en ninguno de los dos casos tenemos que preocuparnos de las Colas, que tienen sus correspondientes constructor y destructor que se encargan de todo.

Toda llamada a un *Get* devuelve un bool que será true en el caso de que hubiera un valor que tomar y false de lo contrario (por ejemplo cola vacía). Los *Set* son distintos: en el caso de la variable sea en realidad una Cola la llamada devuelve un bool que será true en el caso de que se haya podido meter el valor y false de lo contrario (cola llena). Cuando no hablemos de Colas, la llamada a un *set* no devuelve nada (void). A los *Set* y *Get* habituales (cuya mecánica es siempre la misma) se les han añadido funciones que permiten la limpieza de algunas colas. En este caso las colas de *wayoints* (CList) y de errores del DSP (Cola) deben poder ser limpiadas (usando las funciones del template ClearList o limpia_cola). Los casos en los que tal limpieza es necesaria se estudiarán cuando proceda.

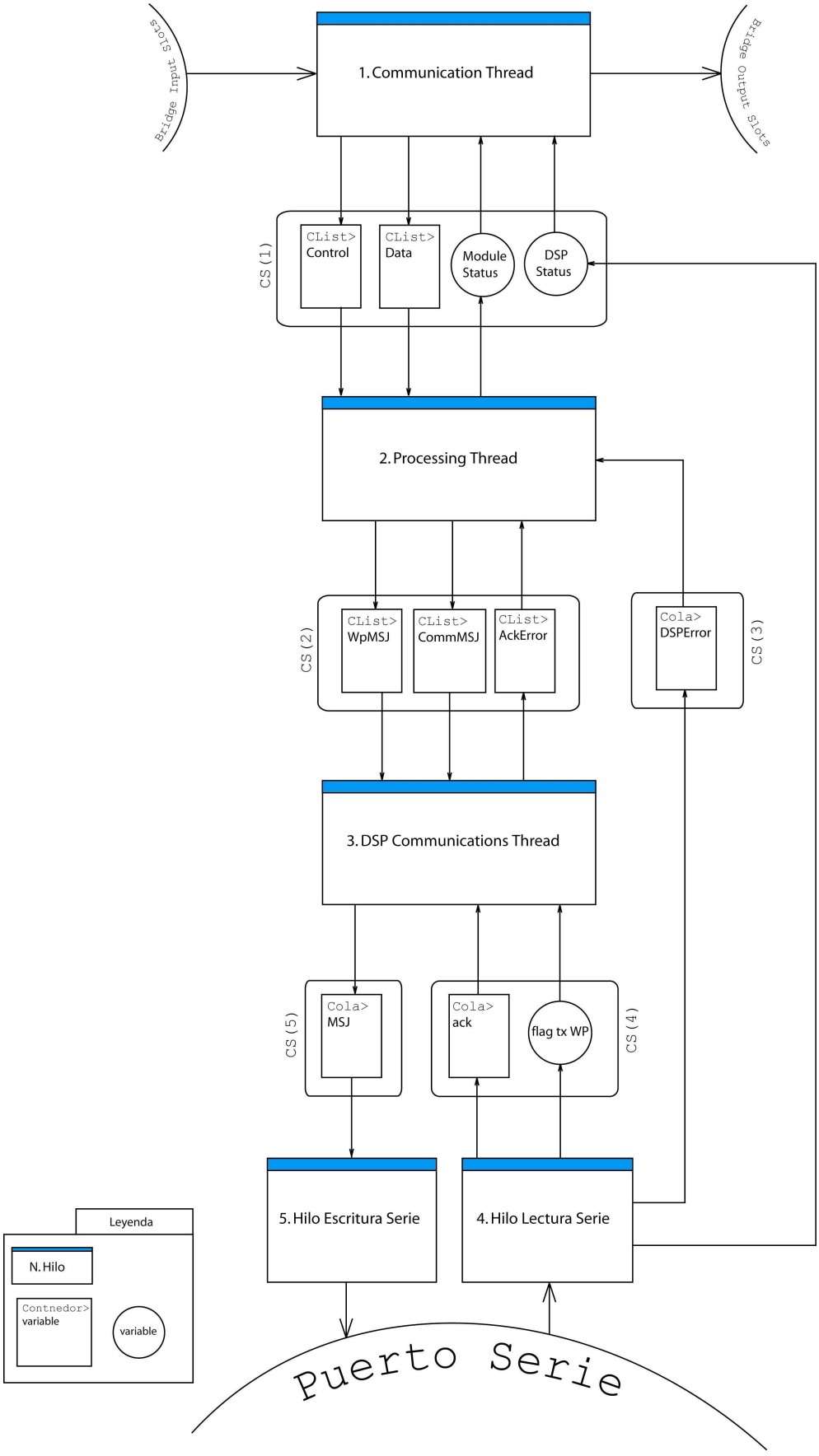


Figura 3.2-2: Esquema de hilos y secciones críticas (CS) del Bridge Module

En la Figura 3.2-2 podemos ver las principales secciones críticas, que responden a relaciones entre hilos. En ella se distinguen dichos hilos, las variables que no se encolan de las sí lo hacen (y en qué tipo de contenedores) y el mundo exterior a la RIL, al que se accede mediante slots de entrada y salida para niveles superiores y por puerto serie en el caso del DSP.

**Clase `Protocolo_DSP_PC`** (en `Protocolo_DSP_PC.h`/ `Protocolo_DSP_PC.cpp`)

Clase que definirá el protocolo de comunicaciones con el DSP. Básicamente permite empaquetar y desempaquetar los datos que transitarán de DSP a PC y viceversa. Para ello se definen dos estructuras: `ESTADO` y `MENSAJE`.

*Estructuras:*

·`ESTADO` contiene en realidad toda la información de los mensajes enviados por el DSP y no sólo el estado del helicóptero, como se podría deducir de su nombre. Cuando se desempaqueta por completo un mensaje proveniente del DSP, esta estructura (variable de este tipo) contendrá información válida del tipo indicado por uno de los flags (de tipo `int`) que contiene con tal finalidad (solo uno puede valer `1`):

- `flag_actualizacion_estado` en el caso de que el mensaje recibido del DSP es un mensaje de tipo estado. Las variables referidas al estado del helicóptero tendrán sentido (bueno, en realidad sólo aquellas definidas en la máscara).

- `flag_ack_comando` en el caso de que el mensaje recibido del DSP sea un mensaje de asentimiento a comando (respuesta *ack*). En este caso, puede que traiga consigo algún dato. De ser así los datos se hallan en las variables miembro "`command_ack_data`" y "`command_ack_data_f`" de la estructura. El identificador del comando al que corresponde el asentimiento se encuentra en "`command_ack_code`" de la estructura. Esta identificación se utilizará en el hilo de comunicaciones del DSP para cerciorarnos de que el *ack* recibido es efectivamente coincidente con el último comando que se envió, para evitar la interferencia de *ack* que se generaron tras el *timeout* (esto lo entenderemos mejor más adelante).

- `flag_control_de_flujo` si el mensaje recibido es de control de flujo para la transmisión de *waypoints*. En este caso la variable entera "`peticion_de_tx`" tiene un valor coherente y con el siguiente sentido:

`0`: No transmitir más *waypoints*. Colas del DSP llenas

`1`: Transmitir un nuevo *waypoint* si se desea. El DSP aún no ha llenado el buffer en el que almacena los *waypoints*.

- `flag_error` valdrá uno en el caso de que el mensaje que se ha recibido sea de tipo error (producido en el DSP). Sólo en ese caso, el código de error se puede leer en la variable entera "`error_code`".

Es decir, el flag que esté a `1` nos indica qué parte de la estructura contiene la información que acabamos de recibir, convirtiendo al resto en escoria que no debemos usar ya que su valor no es válido.

·MENSAJE es una estructura bastante más sencilla que se utiliza para guardar los mensajes que serán enviados al DSP. Contiene un vector de chars de longitud fija (cadena), un entero que indica la longitud de información válida (longitud), un entero que indica el número de bits que deberá recibir en el *ack* correspondiente (bits2receive) y un entero con el código del comando que se ha enviado (command_code):

0 COMANDO_LECTURA_16_BITS
    Lectura de un puerto virtual de 16 bits.
1 COMANDO_ESCRITURA_16_BITS
    Escritura en un puerto virtual de 16 bits.
2 COMANDO_LECTURA_32_BITS
    Lectura de un puerto virtual de 32 bits.
3 COMANDO_ESCRITURA_32_BITS
    Escritura en un puerto virtual de 32 bits.
4 COMANDO_LIMPIEZA_ERROR
    Eliminación de los errores encolados en el buffer de DSPErrors.
5 COMANDO_ACTIVA_TX_ESTADO
    Activación de la transmisión de estado por parte del DSP.
6 COMANDO_INVALIDA_CAMINO
    Detención del helicóptero y eliminación de los puntos encolados.
7 COMANDO_CAMBIO_SEGUIMIENTO
    Activación o desactivación del seguimiento de waypoints.

Nosotros distinguiremos entre MENSAJES de tipo comando y mensajes de tipo *wayoint*, aunque la estructura sigue siendo la misma, por la sencilla razón de que los comandos son asentidos por el DSP mientras que con los *waypoints* en principio no hay forma de saber si han llegado correctamente o no. De hecho existirán colas distintas para cada uno de los dos tipos gestionadas de distinta forma como veremos por parte de los hilos de procesamiento y de comunicaciones del DSP.

Los mensajes que se le pueden enviar al DSP se conforman por medio de unas funciones pertenecientes a la interfaz de la clase. Estas funciones formatean en una cadena de caracteres dichos mensajes de forma que el DSP los entiende. Toman como parámetro una cadena (que ha de tener longitud menor a un máximo establecido) donde construyen el mensaje que ha de ser enviado, carácter a carácter, al DSP a través de un puerto serie. Estas funciones devuelven en la llamada el tamaño real en bytes del mensaje que ha sido formateado en la cadena. Cuando se envía uno de estos mensajes del tipo comando al DSP, éste envía un mensaje de *ack* al PC.

*Conformación de* ESTADO*:*

La decodificación de mensajes provenientes del DSP corre a cuenta de la función procesa_caracter_recibido. Cada vez que llega un carácter por el puerto serie, el hilo de lectura del puerto serie llama a esta función pasándole dicho carácter como primer parámetro. La función devolverá en cada llamada:

0 INCOMPLETE: Si el mensaje aun esta incompleto. Hacen falta mas caracteres para completar el mensaje, es decir, sucesivas llamadas.

-1 CHECKSUMERROR: Ha llegado un mensaje completo, pero se ha producido un error de *checksum* por lo que el mensaje debe ser descartado.

1 ESTADOS: Ha llegado un mensaje completo con información sobre el estado del helicóptero. Entonces, la estructura ESTADO que se le pasa contendrá datos útiles y válidos del estado del helicóptero (posición, ángulos, etc.).

2 CONTROLFLUJO: Se ha completado un mensaje de control de flujo. Hay que mirar la variable "peticion_de_tx", miembro de la estructura ESTADO, que es modificada por esta función, para ver si se puede o no enviar un nuevo *waypoint*.

3 COMMANDACK32: Ha llegado un mensaje completo de tipo respuesta a comando, que incluye un dato de 32 bits. Es decir, un dato codificado en punto flotante (float) acompaña el asentimiento del comando y habrá que mirar en la estructura ESTADO el valor de la variable "command_ack_data_f".

4 COMMANDACK16: Ha llegado un mensaje completo de tipo respuesta a comando, que incluye un dato de 16 bits. Este dato se introduce en la variable miembro de la estructura ESTADO "command_ack_data"

5 COMMANDACK: Se ha completado un mensaje de respuesta a comando, pero no trae consigo dato alguno.

6 DSPERROR: Ha llegado un mensaje de error precedente del DSP. El error en particular se puede ver en la variable miembro de la estructura ESTADO que se le ha pasado: "error_code".

En resumen, únicamente cuando la devolución es mayor que cero, la estructura pasada por referencia como segundo parámetro (p_estado_actual) se rellena con alguna información válida. Un flag a 1 indica qué tipo de mensaje es y asegura que se le han dado valores coherentes a los campos que tengan sentido, dependiendo del tipo de mensaje.

En el caso de que el mensaje contenga el estado del helicóptero, tenemos que tener en cuenta además el parámetro "mascara", que define la composición del mensaje de estado. Es decir, el mensaje de estado puede constar de unas u otras variables dependiendo de esta máscara, de la que tiene que tener constancia tanto el PC como el DSP. Al PC se le pasa como tercer parámetro de la función procesa_

caracter_recibido. Al DSP hay que enviarle un mensaje con el valor de la máscara.

Cada uno de los bits de la máscara (que es un short int) tiene asociado una variable de estado o un conjunto de ellas, de forma que un 1 en dicho bit indica el interés en dicha variable o conjunto, con lo que se realiza la transmisión de ésta o éstas. De lo contrario el bit estará a 0. Las variables asociadas a los bits de la máscara se ven en la figura siguiente:

| | |
|---|---|
| | Bit menos significativo |
| 0 | Tiempo local al DSP. Medido en milisegundos [ms] desde el arranque del DSP. |
| 1 | Posicion actual del helicoptero en coordenadas geograficas. |
| 2 | Posicion actual en helicoptero en coordenadas XY. Aproximacion que hace el DSP tomando como origen la primera lectura valida del GPS y extendiendo el eje Y al norte y el X al este [m] |
| 3 | Velocidad horizontal, vertical y heading dados por el GPS. |
| 4 | QOS del GPS (Quality Of Service). |
| 5 | Angulos de pitch, roll y yaw dados por la IMU [grados]. |
| 6 | Aceleraciones lineales medidas por la IMU [$m/s^2$]. |
| 7 | Velocidades angulares medidas por la IMU [$s^{-1}$]. |
| 8 | Medida de las senales PWM que el piloto envia a los servos. |
| 9 | Medida del sonar. Valor raw, procedente del convertidor. |
| 10 | Nivel de baterias, nivel de combustible y estado del SWITCH Auto/Manual. |
| 11 | Señales de los potenciometros del PAN&TILT. Medida raw procedente del ADC. |
| 12 | Velocidad angular del rotor principal [rpm]. |
| 13 | Medida del altimetro barometrico. Medida raw procedente del ADC. |
| 14 | Valores de actuacion que generan los controladores y que van a los servos. |
| 15 | Referencias de los controladores en cada instante. |
| | Bit mas significativo |

Figura 3.2-3: Significado de los bits de la máscara. Leyenda: n bit | Descripción [unidad].

La máscara se carga desde el archivo de configuración bridge.conf.

*Comandos específicos para enviar al DSP:*

· Comandos de lectura y escritura en puertos virtuales del DSP.

Las variables contenidas en los puertos virtuales de DSP pueden ser leídas y escritas desde el PC mediante el uso de las funciones que se detallan a continuación:

- `int empaqueta_lectura_en_puerto_16(MENSAJE* , unsigned short puerto);`

Esta función formatea un mensaje de tipo comando de lectura de un puerto virtual de tipo entero (16 bits), es decir, de código `COMANDO_LECTURA_16_BITS`. El valor leído será devuelto por el DSP en un mensaje de tipo respuesta a comando con un dato de 16 bits (en `command_ack_data`).

- `int empaqueta_escritura_en_puerto_16(MENSAJE* , unsigned short puerto, short valor);`

Esta función formatea un mensaje de tipo comando de escritura en un puerto virtual de tipo entero (16 bits), es decir, código `COMANDO_ESCRITURA_16_BITS`.

- `int empaqueta_lectura_en_puerto_32(MENSAJE* , unsigned short puerto);`

Esta función formatea un mensaje de tipo comando de lectura de un puerto virtual de tipo *float* (de 32 bits), es decir, de código `COMANDO_LECTURA_32_BITS`. El valor leído será devuelto por el DSP en un mensaje de tipo respuesta a comando con un dato de 32 bits (en `command_ack_data_f`).

- `int empaqueta_escritura_en_puerto_32(MENSAJE* , unsigned short puerto, float valor);`

Esta función formatea un mensaje de tipo comando de escritura en un puerto virtual de tipo float (de 32 bits), es decir, código `COMANDO_ESCRITURA_32_BITS`.

· Otros comandos:

- `int empaqueta_mensaje_mascara(MENSAJE* , short int mascara);`

Esta función crea un mensaje que comunica al DSP la composición deseada del mensaje de estado, mediante el parámetro "`mascara`". Con la definición del parámetro máscara comprenderemos que el estado del helicóptero lo componen muchas variables y que puede que no todas nos interesen en un momento concreto. Con la máscara podremos filtrar sólo aquellas que nos interesan. En realidad, se crea un mensaje de tipo `COMANDO_ESCRITURA_16_BITS`, ya que la máscara en el DSP se encuentra en el puerto virtual `10000` de 16 bits. Cuando aquí decimos tipo nos referimos al código de comando del que hablamos anteriormente.

- `int empaqueta_mensaje_activar_tx_estado(MENSAJE* );`

Esta función conforma un mensaje de tipo `COMANDO_ACTIVA_TX_ESTADO`. Cuando se envía este mensaje al DSP, éste comenzara a enviar periódicamente los mensajes de estado. Las variables válidas dentro de estos mensajes vendrán definidas por la máscara, como acabamos de ver. Por lo tanto, antes de enviar este mensaje, hay que enviar la máscara con el mensaje anterior.

- `int empaqueta_limpiar_codigo_de_error(MENSAJE* );`

Esta función crea un mensaje de tipo `COMANDO_LIMPIEZA_ERROR` que al llegar hará vaciar la cola de errores producidos en el DSP. Esto puede ser especialmente útil tras el proceso de arranque del DSP, durante el cual se producen errores que son de esperar de un estado transitorio y que no deben trascender.

- `int empaqueta_activar_seguimiento(MENSAJE* );`

Esta función construye un mensaje de código `COMANDO_CAMBIO_SEGUIMIENTO` que le dice al DSP que pase a modo seguimiento de *waypoints*. A partir del envío de este mensaje, el DSP empezara a transmitir periódicamente mensajes de control de flujo para el envío de puntos.

- `int empaqueta_desactivar_seguimiento(MENSAJE* );`

Con esta otra por el contrario se empaqueta un mensaje (también de código `COMANDO_CAMBIO_SEGUIMIENTO`) que le dice al DSP que cese de funcionar en el modo seguimiento de *waypoints*. Por tanto, el DSP dejara de transmitir sus mensajes de control de flujo.

- `int empaqueta_invalidar_camino(MENSAJE* );`

Al enviar el mensaje creado por esta función (de código `COMANDO_INVALIDA_CAMINO`) se provoca que el DSP borre su cola de *waypoints* y que el helicóptero se ponga a hacer *hover* entorno al punto en el que se encuentre en el instante actual (instante en que se recibe este mensaje).

Estos comandos ya los habíamos visto por encima cuando estudiamos el DSP como sistema.

*Mensajes de tipo waypoint:*

Los `MENSAJE`s que contengan *waypoints* serán tratados a parte como vimos, ya que no serán asentidos por el DSP. Como la utilización de coordenadas *XY* quedó descartada sólo nos queda una función en la interfaz:

- `int empaqueta_datos_camino_geo(double latitud, double longitud, double altura, char *cadena);`

Esta función conforma un mensaje para mandar un *waypoint* en coordenadas geográficas al DSP. Toma como parámetros los del punto: latitud y longitud en grados, altura en metros.

En resumen, la clase `Protocolo_DSP_PC` implementa el protocolo de comunicaciones entre PC y DSP. Básicamente define las estructuras `ESTADO`, que contiene toda la información proveniente del DSP, y la estructura `MENSAJE` que nos servirá para mandar información a éste último. Contiene las funciones necesarias para procesar lo recibido desde el DSP (`procesa_caracter_recibido`) y además codificar adecuadamente los mensajes dirigidos al DSP (`empaqueta_datos_camino_geo`, `empaqueta_mensaje_mascara`, `empaqueta_mensaje_activar_tx_estado`, `empaqueta_limpiar_codigo_de_error`, `empaqueta_activar_seguimiento`, `empaqueta_desactivar_seguimiento`, `empaqueta_invalidar_camino` y los comandos de lectura y escritura en puertos

virtuales del DSP). Trabaja a nivel de bits, ya que los bytes en realidad sólo llevan 7 bits de verdadera información (el primer bit indica la condición de ser cabecera de un mensaje). Contiene además otra serie de funciones que permiten empaquetar bits de esta manera tan particular, pero estas tendrán la condición de `protected` por no ser necesarias en la misma interfaz. Serán funciones de la interfaz las que harán llamadas a dichas funciones. Esta política de "en la interfaz lo justo y necesario" y sólo funciones, nunca variables, responde a una metodología que explota al máximo el concepto de clase y reduce los riesgos en la reutilización del código.

**Template `Cola`** (en `TCola.h`)

Con el fin de almacenar temporalmente algunos de los datos que circulan en nuestro software teníamos que decidirnos por algún tipo de "contenedor". Se pensó en `CList`, que tan satisfactoriamente se utiliza en otros procesos del proyecto CROMAT. Así nosotros utilizaremos también `CLists` en parte del código. Pero para otras partes se descartó su utilización, ya que como sabemos hace uso de la reserva dinámica de memoria, lo cual hace que el rendimiento sea mínimo en caso de que haya muchas entradas y salidas de memoria. En esos casos se busca algo que se parezca más a un buffer, una región de memoria reservada estáticamente donde introducir o sacar un dato supone tan sólo un desplazamiento de punteros. Entonces decidimos crear nosotros una cola con esas condiciones, a nuestra medida.

Inicialmente se pensó en crear una clase para las dos colas que en principio hacían falta, una de estructuras `ESTADO` y otra de `MENSAJE`. Después se pensó en dos clases de cola, según el tipo de dato a contener, pero definitivamente se vio mucho más interesante y flexible crear un *template*. Así se hizo y creamos la `Cola`. Al ser un template, puede contener cualquier tipo de dato, y de hecho se usará en muchos otros casos además de en aquellos que nos hicieron ver la necesidad de crearlo.

Como variable interna contiene una estructura de tipo `COLA`, que definimos también dentro del template. Dicha estructura consta de:

- una tabla de tipo `T` (donde decimos `T`, el template sustituirá por el tipo de dato que se le especifique en la creación del objeto en concreto) de tamaño definido por el macro `TAMANO_COLA`.

- un par de punteros a `T`, uno apuntará dónde debe escribirse (`primero`) y otro dónde debe leerse (`ultimo`).

- un flag que indica el orden en el que se encuentran los punteros (`pmu`). Esto es importante ya que la cola es circular. `pmu` indica la condición de que el puntero primero es mayor que el puntero ultimo, valiendo `1` en ese caso y `0` de ser menor.

- un entero (`n_elementos`) que lleva la cuenta de los elementos que contiene la cola.

Aunque existen otros algoritmos para la creación de colas circulares, este nos pareció igualmente válido cuando lo encontramos en parte de un código preexistente del DSP. En el esquema de la Figura 3.2-4 podemos ver más claro su funcionamiento.

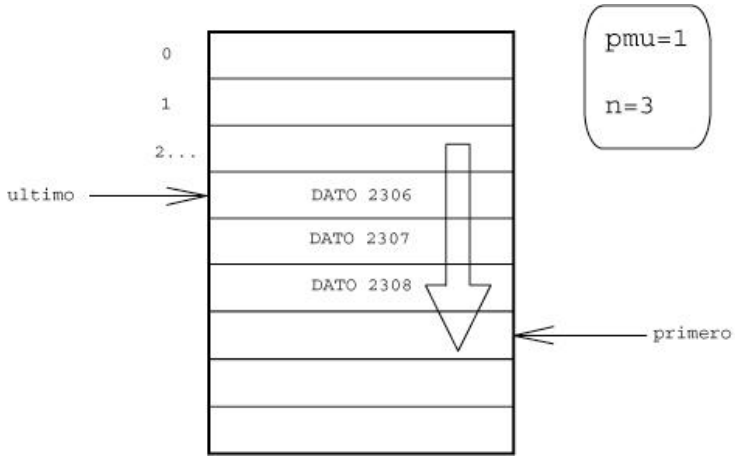


Figura 3.2-4: Esquema `Cola`

El constructor inicializa la cola apuntando primero y ultimo al comienzo de la tabla, poniendo `pmu = 1` (ya que habrá que meter algún dato antes de empezar a sacar, y por lo tanto primero será mayor que ultimo) y `n_elementos = 0`. El destructor por el contrario no debe hacer nada, ya que la reserva estática de memoria no deja huella alguna al salir "*out of scope*".

En la interfaz se definen además las funciones para el usuario:

- `meter_en_cola` y `sacar_de_cola` devuelven un `bool` por llamada que será `false` sólo si no es posible (meter porqué esté llena o sacar porque esté vacía).

- `limpiar_cola` se incluyó para permitir la limpieza de la cola: devuelve en la llamada un `int` con el número de elementos eliminados, lo cual puede ser útil además de que se devolverá cero en caso de que se limpie una cola vacía. Básicamente replica al constructor, es decir pone los punteros al inicio, el indicador `pmu` a `1` y el número de elementos (`n_elementos`) a `0`.

- `leer_de_cola` lee sin eliminar el elemento de la cola, y se creó por una necesidad concreta: se entiende que cuando leemos la velocidad del puerto virtual correspondiente no queremos eliminar el dato. Y como en un principio, por cuestiones que explicaremos en su momento, el puerto virtual de velocidades en el DSP Simulator estaba implementado mediante una cola, usábamos esta función de lectura en lugar de `sacar_de_cola`. En todo momento se entendía que implementar un puerto virtual con una cola era totalmente irreal como veremos, pero en el inicio fue necesario hacerlo así. En todo caso, una vez que se decidió dejar de encolar las velocidades, no se eliminó esta función que podría tener otras aplicaciones y seguir siendo útil.

**Clase `PuertoSerie`** (en `PuertoSerie.h/ PuertoSerie.cpp`)

Clase que permite enviar y recibir datos al puerto serie, para lo cual hace uso de funciones de la librería `<termios.h>`. La implementación es compleja y de bajo nivel, aunque en el código se encuentran bastantes comentarios que ayudan a la comprensión del mismo. Esta clase se realizó partiendo de un código preexistente en C.

En la interfaz nos encontramos con las siguientes funciones:

- Un constructor que, de ser llamado (en realidad auto-llamado en la creación de un objeto de la clase) sin parámetros, toma por puerto y velocidad del mismo los valores por defecto. De lo contrario inicializa el puerto (primer parámetro) a la velocidad que se le indique (segundo parámetro). Se limita a llamar a `init_puerto_serie` para inicializar el puerto serie.

- Un destructor que lo único que hace es llamar a la función `close_puerto_serie` para cerrar el puerto serie.

- `int init_puerto_serie(int baud = DEFAULT_SPEED);`
  Inicializa el puerto de la siguiente forma: lo abre, salva la configuración previa, pone la nueva configuración y activa los cambios. En caso de que haya algún problema en la configuración, esta función no deja al puerto abierto.

- `int close_puerto_serie();`
  Restaura la configuración previa (salvada en `init_puerto_serie`) y cierra el puerto serie.

- `char leer_caracter_puerto_serie();`
  Lee un carácter del puerto serie.

- `void escribir_caracter_puerto_serie(char);`
  Su nombre lo dice todo.

- `void enviar_cadena_puerto_serie(char *cadena);`
  Envía una serie de caracteres al puerto serie.

- `void recibir_cadena_puerto_serie(unsigned char *cadena, int length);`
  Lee un grupo de caracteres del puerto serie. En nuestro proyecto toda la comunicación por puerto serie se hará carácter a carácter, por lo que no utilizaremos ni esta ni la anterior función.

- `void posicionar_principio_puerto_serie();`
  Como su nombre dice, nos posiciona al principio del puerto serie. Tampoco esta función la utilizaremos en nuestro proyecto.

Hemos visto que en la inicialización del objeto se le pueden dar valores al constructor o dejar los valores por defecto en caso de no indicar nada (estos valores por defecto son '`/dev/ttyS0`' para el puerto y `9600` baudios para la velocidad de transmisión). Para especificar el puerto a usar, nosotros usamos una versión modificada y mejorada de la clase `CParser`, que parsea un archivo de extensión '`.conf`' y carga la configuración en una estructura de tipo `CONFIG` (definida en `./hero/ril/bridge/source/config.h`), en la que ahora el puerto a usar será una cadena en vez de un número entero que indicaba la opción a elegir entre varias posibles.

Un objeto `serial_port` de la clase `PuertoSerie` se creará en el `main` del proceso, con el puerto cargado desde '`bridge.conf`' y con una velocidad (actualmente) de `115200` baudios. Este puerto serie se le pasa como parámetro a los hilos de lectura y escritura de puerto serie. Como veremos cuando lleguemos al DSP Simulator, que también se comunica con el puerto serie, el procedimiento para acceder a él es siempre el que acabamos de explicar.

**!** *Permisos sobre el puerto serie:*

Es muy posible que como usuarios sin privilegios no tengamos acceso al puerto serie. En *Linux*, donde el puerto serie se ve como un directorio perteneciente a nuestro sistema de archivos (dentro de `/dev`), basta entones con cambiar los permisos al dispositivo que queramos usar. Es decir, con permisos de superusuario (por ejemplo haciendo un `su`) escribimos:

```
# chmod a+rw /dev/ttyS0
```

o con otro nombre de dispositivo si se da el caso. Con esto damos permisos de lectura y escritura (`+rw`) sobre '`/dev/ttyS0`'a todos los usuarios (`a`). Y ya no deberíamos tener ningún problema con los permisos del puerto.

**Clase `CParser`** (en `CParser.h/ CParser.cpp`)

Para cargar la configuración del archivo `bridge.conf`, que contiene información tan importante como el puerto serie a utilizar en las comunicaciones con el DSP, el valor de la máscara para la transmisión de estados o el valor de los parámetros para los controladores, utilizaremos la clase `CParser`.

Existía un código anterior, del proyecto de David Marcos Rodríguez, que hubo que modificar ligeramente para que al leer el archivo de configuración, tras `puerto_serie` se esperara una cadena de caracteres ('`/dev/ser1...`') en vez de un número entero. Para ello se creó la tabla de caracteres `puerto`, donde se copiaría la cadena que se encontrara en el archivo de configuración tras la especificación `puerto_serie`. Por supuesto, la variable `pto_serie` de la estructura `CONFIG` (en `config.h`) será también una tabla de caracteres. Esto se usará para crear un objeto de la clase `PuertoSerie` con la configuración apropiada en el `main` y multiplica las posibilidades, ya que ahora no estamos forzados a elegir entre las opciones pensadas previamente por el programador, sino que tenemos todas las posibles.

*Funcionamiento del CParser:*

El `CParser` mete todo el archivo de configuración en un buffer para después desmembrarlo poco a poco en pre '=' y post '=', ignorando los comentarios e introduciéndolo desmembrado en una tabla de cadenas de caracteres (es decir, matriz de caracteres) llamada `CommandsList` cuyo esquema vemos en la siguiente Figura:

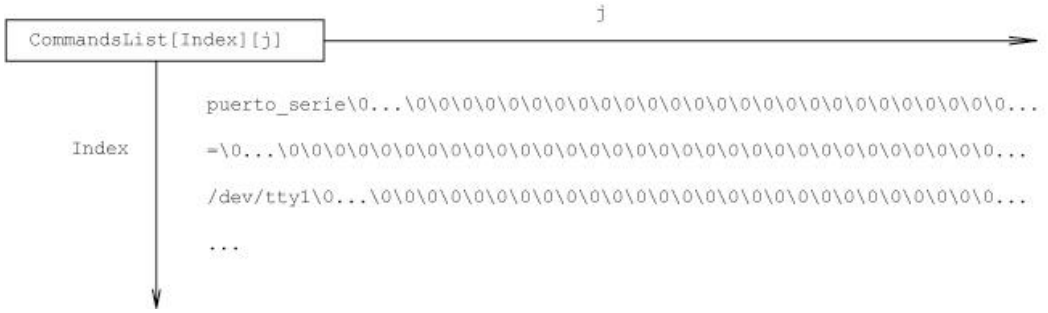


Figura 3.2-5: Funcionamiento de la CommandList

Tras esto es mucho más sencillo meter toda esta información en una estructura de tipo `CONFIG` que guardará definitivamente la configuración. Con `CParser`, dicha configuración no depende de la estructuración del archivo `bridge.conf` (como sucedía antes del `CParser`), sólo hay que respetar la regla "*`campo = valor`*".

En la interfaz nos encontramos las siguientes funciones:

- `CParser();`
  *Default* constructor, se limita a inicializar las variables a cero.

- `CParser(char *FileName);`
  Este otro constructor recibe por parámetro el nombre del archivo de configuración, con lo que ya no es necesario llamar a la función `SetFile`. En ambos casos, se carga el contenido del archivo en un buffer (variable `Buffer`).

- `~CParser();`
  El destructor de la clase (que al no permitir el paso de parámetros es también *default*) elimina la `CommandsList` (después de vaciarla por completo) y el `Buffer`.

- `bool SetFile(char *FileName);`
  Se le pasa el nombre del archivo que meterá en el `Buffer`.

- `int ParseFile(CONFIG* );`
  Crea la `CommandsList` a partir del `Buffer` en el que se metió el contenido del archivo de configuración (pasado al constructor o a `SetFile`), y se van rellenando los campos de la estructura `CONFIG` que se le pasa por referencia con los valores correspondientes. Con ello, ya tenemos una estructura `CONFIG` con toda la información que contenía nuestro archivo de configuración.

Otras funciones no pertenecientes a la interfaz tienen carácter `private` y servirán para ayudar a las anteriores en su trabajo.

*Bug en* `CParser`*:*

Del código anterior se corrigió además un pequeño *bug* que sólo habría dado la cara en circunstancias extremas, pero que supone una incongruencia en el significado de la variable `length` y que no fue difícil de ver una vez encontrado, en la línea 195.

Bug:

```
j=-1;
Length-=j;
```

Lo correcto es:

```
Length-=j;
j=-1;
```

**Clase `CBridgeModManager`** (en `CBridgeModManager.h`/ `CBridgeModManager.cpp`)

La clase `CBridgeModManager` en esencia implementa la máquina de estado del módulo. En función de los datos de control, el estado de la funcionalidad y los posibles errores, es decir, sus entradas, se decide si de bloquea (*block*) o se detiene (*stop*) la funcionalidad y se genera un estado del módulo consecuente.

Cuando hablamos de funcionalidad nos referimos en este caso a todo el proceso de seguimiento de *waypoints*, implementado en el hilo de procesamiento (`Processing-Thread`). Este proceso comprende la recogida e interpretación de los datos referentes al movimiento y la comprobación de llegada a los puntos objetivo. El *manager*, que será una instancia de la clase `CBridgeModManager` tendrá capacidad de bloquear el seguimiento o incluso de pararlo con seguridad en caso de ser necesario. En esta clase se ha añadido además la posibilidad de ignorar la orden de abortar una determinada misión que puede llegarnos desde el Control Center. Esto es gracias a un flag llamado `unable2abort`, que de ser `true` no permite abortar. Por el momento, sólo se utiliza para impedir la detención de un despegue o un aterrizaje, movimientos que sabemos son críticos desde el punto de vista de la seguridad. En la Figura 3.2-6 vemos un esquema de funcionamiento del Module Manager, el hilo de procesamiento será estudiado en profundidad más adelante.

Analicemos sus entradas y salidas:

(*input* 1) El Control puede mandarnos arrancar o parar el módulo: `START_MODULE` o `STOP_MODULE`.

(*input* 2) El estado de la funcionalidad puede ser: `FUNCTIONALITY_RUNNING` o `FUNCTIONALITY_ENDED`. Además se nos incluye información sobre los errores que puedan haberse generado en la funcionalidad, o que provengan del DSP (`DSPError`).

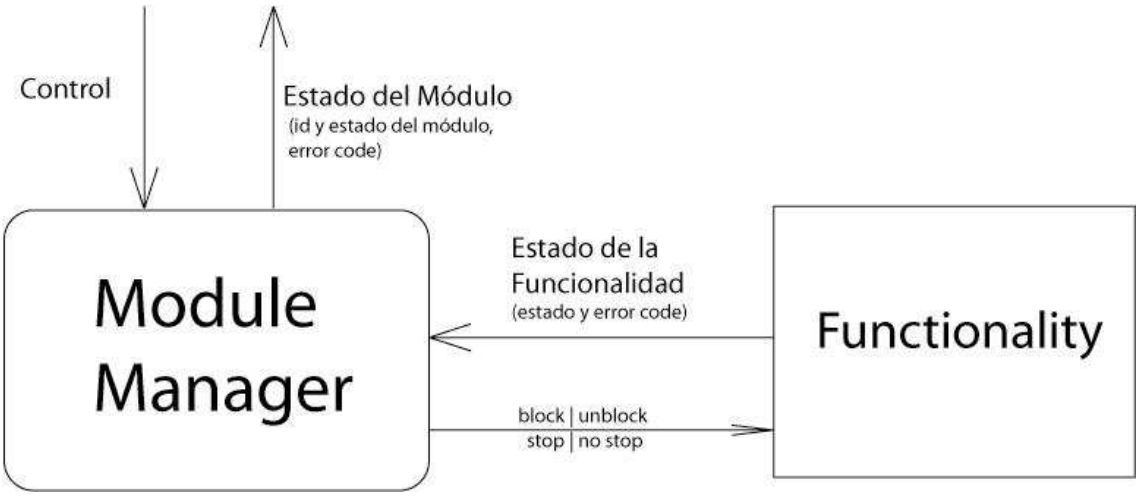


Figura 3.2-6: Esquema de funcionamiento del Module Manager

(*output* 1) El estado del módulo (identificado en este caso como `HERO_BRIDGE_MODULE_ID`) puede ser: `MODULE_RUNNING`, `MODULE_ENDED`, `MODULE_ABORTED`, `MODULE_WAITING_DATA_TO_START`. También aquí se transmiten errores, que pueden ser del tipo: `NO_ERROR_IMI`, `MODULE_RUNNING_TO_RESTART_STOP_IT_BEFORE`, `ERROR_IMPOSSIBLE_TO_STOP_MODULE`, o bien el traducido de un error de la funcionalidad o el DSP.

(*output* 2) Por último la decisión, es decir, la acción a realizar sobre la funcionalidad: `BLOCK_MODULE_FUNCTIONALITY`, `UNBLOCK_MODULE_FUNCTIONALITY` o `NOTHING_TO_DO`. También se puede ordenar la detención de la funcionalidad, con lo que se anula tanto el conjunto actual de *waypoints* como todos los ya introducidos en la lista para el DSP (`WPMessagesToDSPCommList`) y se manda un mensaje de tipo `invalidar_camino`. Esto se hace cambiando el flag oportuno de `NO_STOP` a `STOP`.

Todo esto se ve más claro en el esquema siguiente (Figura 3.2-7), en el que se especifica el software subyacente al esquema anterior.

La función que nos interesa de esta clase es `SetModuleActionAnd-UpdateModuleState`, que es la que lee el estado del módulo y el de la funcionalidad, decide qué hacer con la funcionalidad y finalmente actualiza el estado del módulo.

La orden de hacer `STOP` se produce en caso de que llegue algún error o una orden de abortar (estando permitido abortar) desde el Control Center.

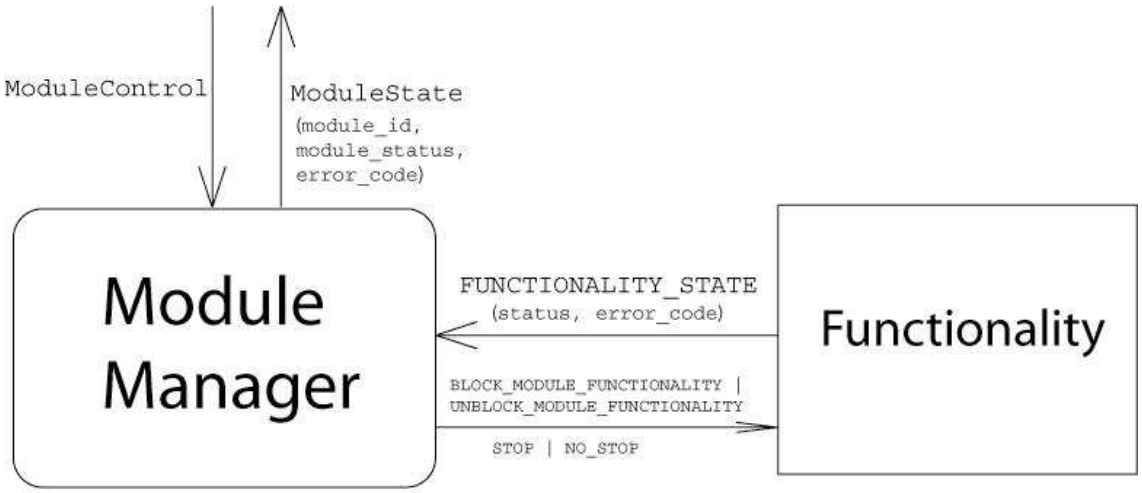


Figura 3.2-7: Esquema software del funcionamiento del Module Manager

Los errores provenientes del DSP, que una vez traducidos por la función `Estado2DspError` están en el intervalo [201, 500], pasan a través de la funcionalidad al Module Manager, donde deben ser mapeados por la función `TranslateErrorCode` de esta clase. Por mapear entendemos tomar la decisión de qué errores deben trascender y que errores no son importantes como para hacerlo. El criterio con el que se toma esta decisión es muy sencillo: son lo suficientemente importantes los errores que se pueden interpretar como un mal funcionamiento fatal o permanente del DSP. Todos los errores que el DSP puede enviarnos se verán a continuación en un cuadro resumen, pero adelantamos que el mapeado actualmente implementado hace que trasciendan los errores:

`8` (Error de escritura de las salidas PWM) y `18` (Error en *scheduler*), ambos bajo el código más general de `HERO_DSP_FATAL_ERROR`.

`102` (Error de *checksum*) como `COMMUNICATION_FATAL_ERROR`.

Los errores *ack* que también llegan desde la funcionalidad, sin embargo, trascienden inmediatamente ya que suponen siempre un grave fallo en las comunicaciones: `COMMUNICATION_FATAL_ERROR`.

**Cabecera `datatranslation.h`**

Como una de las misiones del Bridge Module es la de traducir tipos de datos, se ha optado por crear un archivo cabecera que contenga a todas las funciones que lo que hacen es exclusivamente traducir. El prototipo general de estas funciones es:

```
bool TipoOrigen2TipoDestino(TipoOrigen* , TipoDestino* );
```

La llamada devuelve `true` si todo ha ido bien y `false` en caso contrario. La traducción es especialmente útil para desmembrar la gran estructura `ESTADO` en otras más pequeñas que contengan sólo la información válida de ésta (señalada por uno de los cuatro flags). Esta cabecera se ha situado finalmente en la carpeta `./hero/ril`

`/comms/` porque será utilizada también en el Hero Status Module (función `DSPHeroTelemetry2HeroTelemetry`).

De especial importancia es la función `Estado2DspError`, que además de extraer la información concerniente al error de la estructura `ESTADO`, la traduce para que sea compatible con los demás errores de la arquitectura. La información sobre los errores del DSP, que tuvimos que deducir del código del DSP del que disponíamos, se resume en el siguiente cuadro:

Cuadro 3.2-1: Resumen errores del DSP

1: Cola cola_rx_sci_a llena (interrupt).
2: Cola cola_rx_sci_b llena (interrupt).
3: Error al leer puerto sci_a con espera (cola vacía).
4: Error al leer puerto sci_b con espera (cola vacía).
5: Error al escribir en el puerto sci_a sin/con espera (cola llena).
6: Error al escribir en el puerto sci_b sin/con espera (cola llena).
7: Puerto no válido al enviar dato por el sci.
8: Error de escritura de las salidas PWM por ser nwords=!1.
9: uart no válido en escritura/lectura, inexistente (turno).

10: Cola cola_rx_uart_1 llena.
11: Cola cola_rx_uart_2 llena.
12: Error (rec_word&0x400) en uart1 (interrupt).
13: Error (rec_word&0x400) en uart2 (interrupt).
14: Cola uart vacía.
15: Cola uart llena.
-
18: Error en scheduler, el proceso foreground ha durado más de 1 segundo y por tanto ha sido interrumpido para atender una rutina más prioritaria.

20: Rutina por defecto para llamada del driver de lectura.
21: Rutina por defecto para llamada del driver de escritura.
22: Rutina de servicio de interrupción por defecto.

30: Comando no válido.

40: Error de checksum al procesar carácter del IMU.
-
45: Error en el CRC al procesar carácter del GPS.
46: Estado inesperado al procesar carácter del GPS.

100: Cabecera desconocida al procesar carácter del PC.
-
102: Error de checksum al procesar carácter del PC.

300: Error de checksum en de-codificador de mensajes del radio-receptor DDS-10.

La incompatibilidad llegaba por la numeración de los errores, que coincide con la de los errores generales de la IMI (`GENERAL_IMI_error_code.h`, de 0 a 99) y los particulares del UAV (`UAV_IMI_error_code.h`, de 100 a 200). Esto se soluciona sumando una constante en el momento de esta primera traducción, en este caso 200, con lo que los errores del DSP pasan al intervalo numérico de 201 a 500, que aún no está utilizado por ningún tipo de error interno a la RIL.

**Hilo principal** (en `./hero/ril/bridge/main.cpp`)

El hilo principal del proceso `BRIDGEMOD` será el encargado de lanzar al resto de los hilos, además de declarar dos importantes variables globales que compartirán todos los hilos:

- La sección crítica (`BridgeModCS`) de la clase `CBridgeModCriticalSection`.

- El entero `end`, que indica la condición de terminación del programa. Todos los bucles internos de los hilos se finalizan cuando esta variable toma un valor distinto de `0`. En un principio el usuario producía esta condición introduciendo '`q`' por el teclado, pero ahora es un manejador de la señal generada por `Ctrl+C` el que pone `end = 1;` es decir, el usuario deberá ahora pulsar `Ctrl+C` para terminar el programa.

El proceso resultante (`BRIDGEMOD`) tiene como parámetro indispensable la identidad del robot (`robot_id`), un entero mayor de cero que distingue a un robot de otro. Es decir, debe ejecutarse: `./BRIDGEMOD [`*`robot id`*`]`

En `main` se carga la configuración del archivo `bridge.conf` con el `CParser`. Ésta se almacena en la variable `configuracion` de tipo `CONFIG`. Además se usa para la creación del objeto `serial_port` de la clase `PuertoSerie` que nos permitirá las comunicaciones con el DSP.

También se altera la máscara del proceso, que no debemos confundir con la máscara que define las variables de estado. La máscara del proceso se refiere a las señales que pueden o no interrumpir el funcionamiento de un hilo. Así aquí se desbloquea la señal `SIGINT` (generada al pulsar `Ctrl+C`) para que sea tratada por el manejador y se bloquea `SIGRTMIN` por razones que veremos más adelante. La máscara de proceso es heredada por los hilos que sean lanzados: en este caso hemos dicho que el main lanza todos los hilos del `BRIDGEMOD`, por lo que todos estos hilos heredan esta máscara de proceso.

El lanzamiento de los hilos se realiza con los siguientes parámetros:

*1.* `CommunicationThread` con un puntero a `robot_id`. Lo primero que se hace es abrir las comunicaciones. Este hilo gestiona las relacionadas con el resto de módulos y capas de la arquitectura software, por lo que necesita la identidad del robot para distinguirse del resto de módulos homónimos de distintos robots.

*2.* `ProcessingThread` con un puntero a `configuracion`. El hilo de procesamiento necesita concretamente la máscara de estado (ojo otra vez, no confundir

con la de proceso), contenida dentro de la configuración, para pasársela al DSP en un mensaje antes de empezar a pedir la transmisión del estado.

*3*. `DSPCommunicationThread` con un puntero `NULL`. Como no necesita nada, se le pasa un `NULL`, es decir, nada.

*4*. `HiloLecturaPuertoSerie` con un puntero a `serial_port`. Este hilo gestiona la lectura del puerto serie, por lo que necesita acceso a este objeto ya inicializado por `main`, que implementa nuestro puerto serie.

*5*. `HiloEscrituraPuertoSerie` con un puntero a `serial_port`, por razones idénticas a las anteriores.

Una vez lanzados el resto de los hilos, el `main` no hace más que esperar la terminación del proceso: dentro del `while(!end)` no hay más que un `usleep` de 300ms. A la terminación del programa (`end = 1`), se recogen los hilos. El orden en el que se recogen los hilos es inverso al orden en el que se lanzan.

(*1*) **Hilo `CommunicationThread`** (en `bridgemodcommunicationthread.cpp`)

Hilo de comunicaciones del proceso con otros módulos y capas del software. Su modo de operar es análogo al del resto de hilos de comunicaciones de la arquitectura definida para el proyecto: se añaden los slots del módulo (para el caso del Bridge Module son 4 como vimos), se crea el archivo de configuración de las conexiones (`connectionsBM_` *`robot_id`*`.conf`) en el que se incluye la conexión al Relay Node del robot (ver artículo "*Arquitectura general para múltiples robots heterogéneos*") y finalmente se realizan las conexiones según el archivo de conexiones recién creado.

Una vez iniciadas las comunicaciones y el `BridgeModManager`, entramos en el bucle `while(!end)`:

- Se realiza una sincronización y se duerme el proceso durante 50ms con la función `Synchronize`.

- Se comprueban los canales de conexión con `CheckChannels`.

- Se reciben datos de los slots de entrada con los *Receive* de `CBridgeMod-Communication`. Esta información pasa a la `CS(1)`.

- Se envía a los slots de salida la información que corresponde, procedente de la `CS(1)`. Esto se hace con los *Send* de `CBridgeModCommunication`.

- El `BridgeModManager` actualiza el estado del módulo y decide si la funcionalidad debe bloquearse o no.

Tanto cuando se recibe la condición de terminación del programa como en caso de error, se finalizan correctamente las comunicaciones gracias a las funciones heredadas `Finalize` y `KillChannels`.

Resumiendo, este hilo realiza E/S BBCS (4) y decide el estado de la funcionalidad cada 50ms. Usa en este caso concreto la clase `CBridgeModCommunication`.

(*2*) **Hilo `ProcessingThread`** (en `bridgemodprocessingthread.cpp`)

El hilo de procesamiento es quizá el más importante del módulo, puesto que como hemos visto, es el que entre otras cosas desarrolla la funcionalidad del mismo: el seguimiento de *waypoints* (WP).

Antes de entrar en el bucle de control, hacemos la inicialización del DSP (enviar máscara, activar la transmisión de estado, limpiar el código de error, invalidar camino y activar seguimiento, como se nos dice en el documento *"Acciones al arrancar el proceso UAV y con independencia de los módulos funcionales"*) y damos valores iniciales a una serie de flags que nos serán de gran ayuda. Sus nombres y los valores que pueden tomar hablan por sí solos, por tanto no decimos que no haya otras formas más eficientes de desarrollar la funcionalidad, pero sí que nuestro esfuerzo no se ha enfocado a ello, sino a una fácil comprensión en la lectura del código:

- `vector_flag` se compone de dos campos enteros (`received` y `next`) que pueden tomar los tres valores: `NOT_FIRST`, `FIRST` y `LAST`.

- `arriving` y `arrived` son dos variables booleanas que nos hablan del acercamiento o la llegada al objetivo.

- `block` nos indica como vimos la decisión del Module Manager de bloquear o no la funcionalidad y otra booleana, `unable2abort`, la imposibilidad de abortar el movimiento actual. También habíamos visto al estudiar el Module Manager los valores `STOP` y `NO_STOP` que puede tomar el entero `stop_flag`.

Ya en el bucle de control ( `while(!end)` ) lo que hacemos es preparar mensajes para el DSP y leer los provenientes de éste. Nuestra política al respecto estará siempre basada en la cautela:

- En el caso de nos llegue un paquete de *waypoints*, es decir, una serie de puntos que tienen sentido sólo como conjunto, los vamos almacenando en una tabla pero no los mandamos al DSP hasta que no llegue el último. Así ignoramos paquetes que no estén encabezados o terminados e intentamos evitar la propagación de posibles incoherencias. Veremos que la condición de primer y último punto de un paquete viene expresada por dos variables booleanas que contiene la estructura `GoToWP_Params`, contenida en `MovementData`.

- En caso de que enviemos un mensaje al DSP, nos aseguraremos de que éste ha llegado correctamente antes de enviar otro. De esto se encargará en realidad el hilo `DSPCommunicationThread` que estudiaremos en el siguiente punto.

En principio tanto comandos como *waypoints*, una vez codificados para el DSP, van empaquetados dentro de estructuras `MENSAJE`, con lo que a nuestros ojos son iguales. Pero pronto nos dimos cuenta de la necesidad de poder distinguir entre ambos; el problema surge por dos diferencias fundamentales entre comandos y WP:

- Los comandos son asentidos por el DSP, es decir, podemos tener constancia de que han llegado correctamente. Los WP no.

- Los WP se guardan en un buffer que es fácil que se llene en algún momento, ya que lo normal es que se manden muchos WP para por ejemplo seguir una trayectoria determinada. Por ello se implementa un control de flujo a través de un flag que indica si se pueden trasmitir o no más puntos. Precisamente este flag se estuvo estudiando como posible asentimiento de WP, pero finalmente de descartó por diversos motivos: entre otras cosas, no podemos estar seguros de qué WP originó el asentimiento ya que el flag no contiene ninguna otra información. Para los comandos, por el contrario, no tiene sentido el control de flujo.

Por estas dos razones es importante distinguir entre comandos y WP, ya que no hacerlo podría llevarnos a situaciones graves a la vez que absurdas: por ejemplo, quizá ya he recibido el asentimiento del último comando que mandé, pero al estar el buffer de WP del DSP lleno, ya no puedo mandar una orden que es precisamente la de limpieza de dicho buffer. Es resumen, hay que evitar que comandos y WP se bloqueen entre sí. Esto se consiguió metiéndolos en colas distintas (`CommandMessagesToDSPComm-List` y `WPMessagesToDSPCommList`) antes de que el `DSPCommunication-Thread` los mandara ya al `HiloEscrituraPuertoSerie` para ser enviados al DSP.

Ya hemos adelantado el concepto de paquete de *waypoints*: no es más que un conjunto de puntos. En realidad, lo que nos llegará siempre son paquetes, ya que un único *waypoint* puede ser considerado como un conjunto de un solo punto. Decíamos también que este concepto va ligado a los flags enteros `first_waypoint_data` y `last_waypoint_data` de la estructura `GoToWP_Params` que pueden tomar los valores `0` ó `1`. Así podemos distinguir los *waypoints* según el valor del par (`first_waypoint_data,last_waypoint_data`):

(`1,1`): es un paquete de un solo punto. En el mismo momento se manda al DSP.

(`1,0`): es el primer punto de un paquete; inmediatamente tras él deben llegar más puntos de tipo (`0,`*x*) pertenecientes al mismo paquete.

(`0,0`): es un punto intermedio de un paquete; en algún momento anterior debió llegar un punto de par (`1,0`) e inmediatamente tras él debe venir un (`0,`*x*) también del conjunto.

(`0,1`): es el último punto del conjunto. El paquete ya puede ser enviado al DSP.

Con lo que respecta al despegue y al aterrizaje del helicóptero, en un principio no eran más que WP del tipo (1,1). Ahora se hace ya de una forma más realista, a través de la escritura en puertos virtuales. En todo caso, ambas acciones tienen la propiedad de no poder ser abortadas (unable2abort = true).

Ahora estamos en condiciones de ver resumidamente pero con más detalle qué entendemos por funcionalidad: en caso de no encontrarnos bloqueados y sí en reposo, tomamos una nueva orden de movimiento y miramos el tipo de movimiento (en el campo type_of_request) de ser:

- RS_TAKE_OFF,
en las fases iniciales del proyecto, se tomaba la altura que indicada por Movement-Data.MovementParams.take_off_ params.altitude, la posición actual en el plano horizontal y se conformaba un WP que me se enviaba este hilo a sí mismo, para luego ser procesado en la próxima iteración por RS_GOTO_WP. La posición actual la leíamos del estado del DSP disponible en la Sección Crítica en ese momento. Este método era ingenioso, pero no realista.

Una vez que se implementaron los puertos virtuales en el DSP Simulator, ya se pudo proceder de una manera más parecida a la que se usará en el caso real. Concretamente se realiza una escritura con el valor 1 en el puerto virtual 10121. Esto lo interpreta el DSP Simulator como una orden de despegue, y es él mismo el que conforma el WP, de una manera muy parecida a la que acabamos de describir.

- RS_LAND,
se opera con total analogía al caso anterior, sólo que ahora el puerto virtual a utilizar es el número 10122. Además, la altura de un aterrizaje es siempre conocida, y sencillamente igual a 0.

- RS_GOTO_WP,
se actúa en dos fases:

*1)* Si el WP que leo es realmente el primero (1,*x*) comienzo a rellenar una tabla dinámica (malloc, realloc) que contendrá el paquete que acaba de comenzar a transmitirse. En el esquema siguiente (Figura 3.2-8) vemos la estructura típica de una de estas tablas.

Cuando me llega el último de los WP (*x*,1) lo guardo en una variable a parte, ya que éste será el punto objetivo final. Entonces la tabla resulta estar completa, como mínimo con un WP del tipo (1,1), y se codifica (protocolo.empaqueta_datos_camino_geo) y se mete en la WPMessagesToDSPCommList para el DSPCommunicationThread: supuestamente el helicóptero comienza a moverse (arriving = true) y ya podemos pasar la siguiente fase.

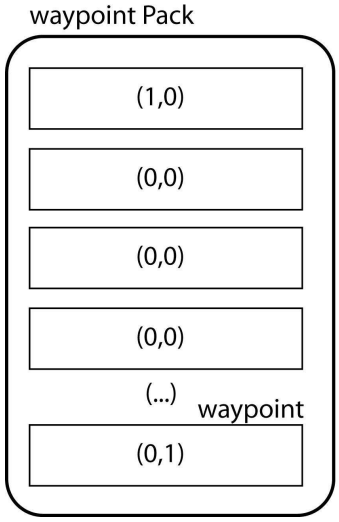


Figura 3.2-8: Tabla de WP

Antes hay que comentar que cada vez que llega un nuevo (`1,`*x*) se comienza una tabla, pisándose cualquier otro paquete que no haya sido concluido y por tanto inmediatamente enviado. Pero tal circunstancia no se pierde, es decir, el código sabe que estamos anulando el paquete anterior sólo que no hace nada al respecto, salvo anunciar que así sucede (`"WayPoint overwriting!"`). Esto quiere decir que si se decide realizar otra acción en este caso (por ejemplo generar un error) no hay más que añadir código en esta parte.

*2)* Voy leyendo el estado actual (`BridgeModCS.GetBridgeModDSP-Status`) para compararlo con la variable que me guardé como objetivo (último WP válido). ¿Me encuentro suficientemente cerca de mi objetivo? De ser la respuesta positiva, puedo considerar mi tarea terminada (`state.status= FUNCTIONALITY _ENDED`). Si por el contrario me llegara una orden de `STOP`: vacío la tabla actual, la lista de puntos del PC, y el buffer del DSP (`protocolo.empaqueta_ invalidar_camino`). Elimino además los errores del DSP y doy igualmente por concluida mi tarea.

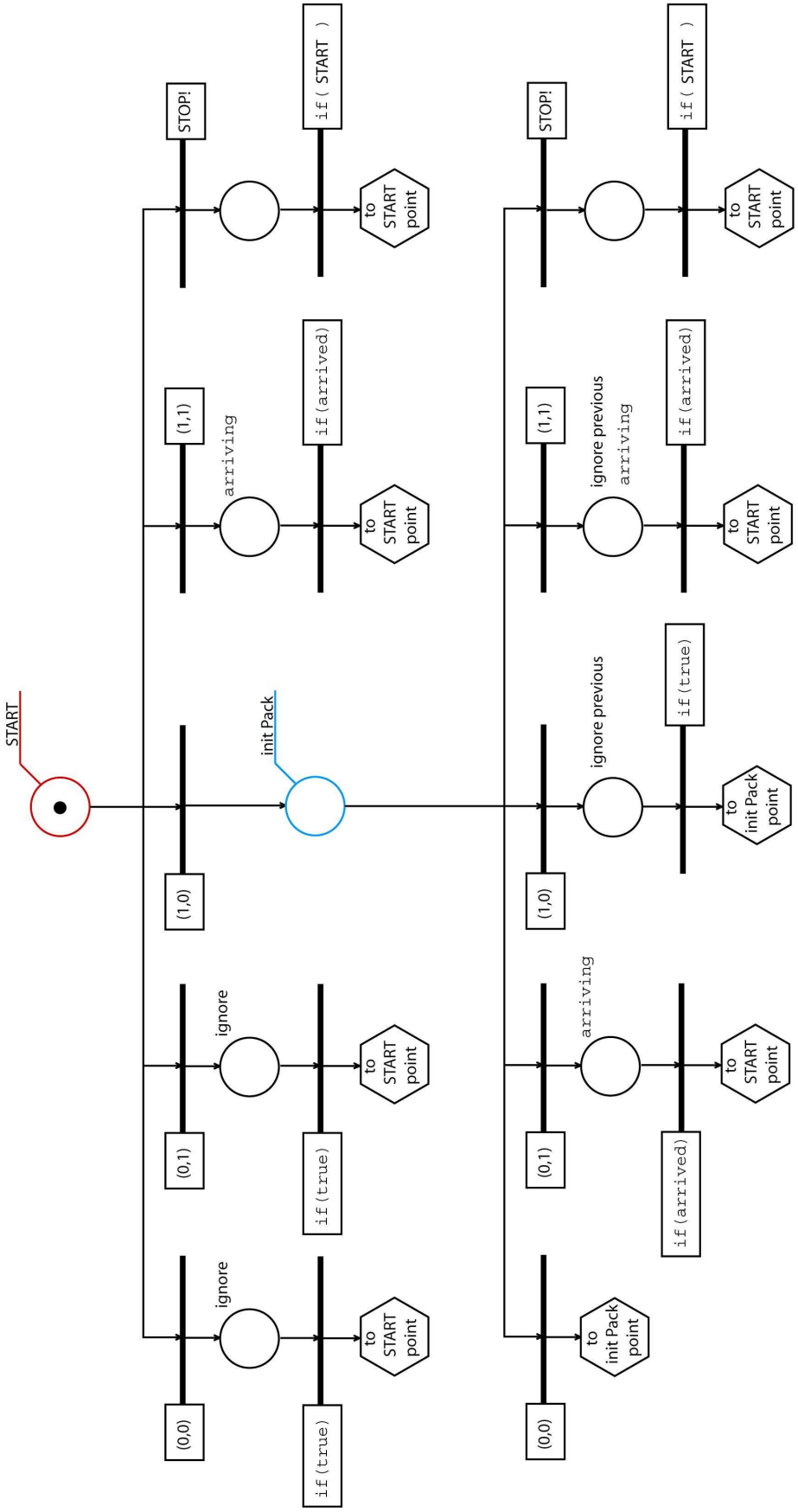


Figura 3.2-9: Árbol Diagrama de Petri que describe la posible casuística

La comprobación de proximidad a nuestro objetivo se hace de una forma muy sencilla. Se define una tolerancia angular para la longitud y la latitud (`DELTA_DEG`) y otra lineal para la altura (`DELTA_M`) que definirán el tamaño del "cubo" dentro del cual consideraremos alcanzado nuestro objetivo. No es exactamente un cubo, no sólo porqué sus lados no tienen porqué ser iguales, sino porque dos de sus dimensiones no son ni siquiera lineales. De todas formas, para las magnitudes que manejamos en este proyecto que comparadas con el radio terrestre son despreciables, la figura resultante es muy parecida.

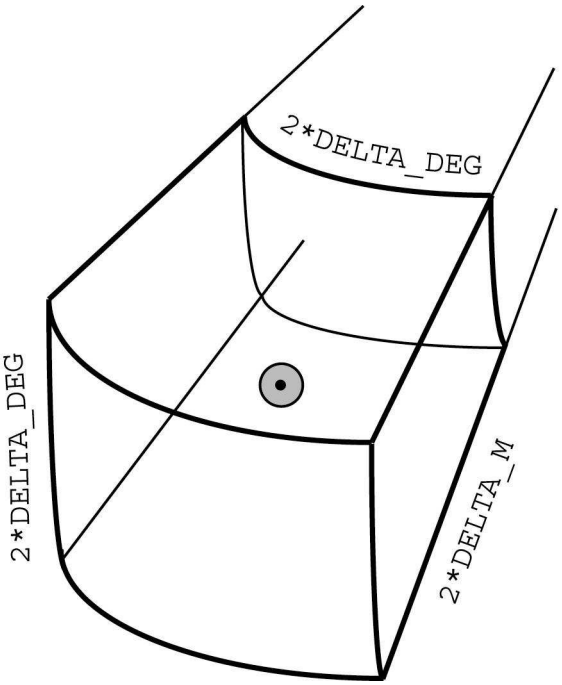


Figura 3.2-10: Cubo de tolerancias

Como disponemos de una macrofunción para el cuadrado de un número (que podríamos cambiar por la de valor absoluto que se definió más adelante) la ecuación que usaremos para esta región del espacio es muy sencilla:

$$|x_{tar} - x| < \delta \Rightarrow (x_{tar} - x)^2 < \delta^2$$

donde $x$ representa cualquiera de las tres magnitudes medidas, $x_{tar}$ corresponde al objetivo (*target*) y $\delta$ a la tolerancia asignada para el caso.

Por último, e independiente de que estemos bloqueados o no, en reposo o en movimiento, el `ProcessingThread` lee los errores (ya sean de tipo `AckError` o provenientes del DSP) y los manda al Module Manager. El Module Manager los traducirá convenientemente, subirá al estado del módulo aquellos que proceda por su gravedad y los tendrá en cuenta a la hora de tomar decisiones.

(*3*) **Hilo `DSPCommunicationThread`** (en `dspcommunicationthread.cpp`)

Este hilo será el encargado de gestionar los mensajes del hilo de procesamiento al DSP, tanto si son comandos como si son WP.

Para los WP lo único que tiene que tener en cuenta es el control de flujo, que vigila que el buffer del DSP no se desborde. Si el flag `peticion_de_tx=1`, entonces mando el mensaje para que el hilo de escritura en puerto serie se lo pase por fin a su verdadero destinatario, el DSP.

Para los comandos el procedimiento es más delicado. Tomamos uno y ponemos en marcha un temporizador que hemos programado previamente; entonces seguimos con nuestras tareas, que comprenden comprobar que el comando ha llegado correctamente, además de mandar mensajes con *waypoints* como hemos visto. Esta comprobación se lleva a cabo leyendo cada *ack* que llega, y comprobando que responde al tipo de mensaje que enviamos. Si todo va bien, recibiremos el *ack* correspondiente y podremos apagar el temporizador, tomar otro comando si lo hay y volver a empezar. Pero si salta el temporizador, hemos de suponer que el comando no ha llegado correctamente al DSP; lejos de desesperarnos, lo intentaremos otra vez, volviendo a mandar el mismo mensaje. Y así lo haremos hasta tres veces: si después de tres intentos consecutivos seguimos sin obtener respuesta alguna del DSP, mandaremos un `AckError` al hilo de procesamiento que terminará su funcionalidad con un `COMMUNICATION_FATAL_ERROR`.

Un flag llamado `timeout` nos indicará si el temporizador ha saltado, y otro llamado `received` si hemos recibido asentimiento del último comando enviado. Para llevar la cuenta de los intentos de envío sin éxito de un mismo comando, se dispone además del entero `counter`.

Hemos visto previamente que en `main` se bloquea la señal `SIGRTMIN`. Al ser main la que lanza todos los demás hilos, estos heredarán su máscara de señales. Después únicamente en `DSPCommunicationThread` se desbloquea `SIGRTMIN`, con la que se programa el temporizador, que es de disparo único. La acción programada a realizar cuando se recibe esta señal es pasar a un manejador que pone `timeout=1`.

(*4*) **`HiloLecturaPuertoSerie`** (en `hilolecturapuertoserie.cpp`)

Su propio nombre deja muy clara su misión: este hilo será encargado de recibir por puerto serie los mensajes del DSP. El objeto de la clase `PuertoSerie` que le servirá en tal tarea se le pasa como parámetro desde el hilo principal, pero no obstante, le será necesario cargar la configuración desde `bridge.conf` (otra vez con `CParser`) para conocer además el valor de la máscara de estado. Después, en cada iteración hace llamadas a `lee_caracter_puerto_serie` y a `procesa_caracter_recibido`, función de la clase `Protocolo_DSP_PC`. Esta última devuelve un entero que indica qué tipo de mensaje nos ha llegado. En todos los casos se traduce la estructura `ESTADO` que ha sido rellenada (funciones `Estado2`*`Xxx`* en `datatranslation.h`) y se hace en la Sección Crítica el *Set* apropiado, tal y como se vio en la sección *Conformación de* `ESTADO` cuando se estudió la clase `Protocolo_DSP_PC`:

CHECKSUMERROR: Ha habido un error de *checksum* y el mensaje recibido debe ser descartado.

INCOMPLETE: El mensaje está incompleto, debemos seguir iterando porque la estructura ESTADO aún no está convenientemente rellena.

ESTADOS: Se traduce ESTADO a DSP_HERO_TELEMETRY y se hace un SetBridgeModDSPStatus. En este caso se realiza además una comprobación de que los estados tienen un tiempo coherente, es decir, que los que nos van llegando son realmente más actuales que los anteriores. El método lo hemos obtenido de un código previo: tendremos un vector de tres estructuras ESTADO en el que tratamos de tener en todo momento un control sobre la actualización, de modo que el dato que se ha comprobado es más reciente se envía a través de est[1].

| **\step** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8...** |
|---|---|---|---|---|---|---|---|---|---|
| Est[0].t | 0 | t0 | t1 | t2 | t3 | t2**!** | t4 | t5 | t6 |
| Est[1].t | 1 | **t0** | **t1** | **t2** | t3 | t2 | **t4** | **t5** | t6**?** |
| Est[2].t | 0 | 0 | t0 | t1 | t2 | t2 | t2 | t4 | t5 |

Te la tabla ejemplo anterior, sólo se envían los estados cuyos tiempos se encuentra en negrita. Para el último caso, el que su estado se mande o no dependerá del valor de Est[0].t en el siguiente paso.

CONTROLFLUJO: Control de flujo, traduce ESTADO a DSPFlagPeticionTxWP y hace un SetDSPFlagPeticionTxWP.

COMMANDACK32: Respuesta 32 bits, traduce ESTADO a DSPAck con nbits = 32 y hace un SetDSPAck. Esto quiere decir que se ha recibido una respuesta a comando que contiene un dato de 32 bits, que encontraremos en el campo command_ack_ data_f.

COMMANDACK16: Respuesta 16 bits, análogo al caso anterior, pero el dato es ahora de 16 bits y se encuentra en command_ack_data.

COMMANDACK: Respuesta sin datos asociados, análogo a los casos anteriores, pero sin ningún dato (nbits = 0).

DSPERROR: Error de DSP, traduce ESTADO a DSPError y hace un SetDSPError. Hemos visto que dicha traducción hace además que los códigos de error del DSP sean compatibles con otros códigos de error del PC.

(*5*) **`HiloEscrituraPuertoSerie`** (en `hiloescriturapuertoserie.cpp`)

Este hilo, encargado de escribir los mensajes para el DSP en el puerto serie, sigue un algoritmo de lo más sencillo:

·1) En el caso de que haya algún `MENSAJE` en la cola `MessagesToSerial-WritingCola`, lo escribe en el puerto serie (y lo escribe carácter a carácter con `escribir_caracter_puerto_serie` pero completo en una sola iteración del bucle de control).

·2) En caso contrario, "duerme" un rato (50 ms) para no seguir consumiendo tiempo de CPU.

·3) Vuelve al punto 1.

Como veremos cuando estudiemos el DSP Simulator, el procedimiento de escritura en puerto serie será totalmente análogo en ambos casos.

**Cabecera `UAV_BM_data.h`**

Define una serie de estructuras en las que el hilo de lectura del puerto serie descompondrá la información recibida en forma de `ESTADO`. Estas estructuras de datos sólo se utilizarán dentro del Bridge Module.

**Cabecera `config.h`**

Define la estructura `CONFIG` en la que se recogerán los datos cargados desde el fichero de configuración.

**Archivo de Configuración `bridge.conf`** (en `./hero/ril/bridge/`)

Contiene la configuración del Bridge Module. Esto incluye, entre otros datos, la ruta del puerto serie a utilizar en las comunicaciones con el DSP y la máscara de estados que define qué variables de estado nos interesa transmitir.

**Archivo `Makefile`** (en `./hero/ril/bridge/`)

Contiene una lista de todos los archivos objeto del proyecto, así como una relación de dependencias de éstos con los archivos fuente, de manera que podemos ahorrar tiempo de compilación gracias a la utilización de `make`. Este será el método para la compilación de todos los procesos que creemos.

## 3.3 Hero Status Module

El Status Module es el encargado de pasar el estado del helicóptero a la capa software superior (MML). A partir de toda la información disponible, su finalidad es la de generar un estado lo más preciso posible en un formato que será aún característico del robot, pero al menos más general que los puros valores de los sensores. Por el momento la única fuente de información es el Bridge Module, que se limita como sabemos a hacer de puente con el DSP, por lo que el Status Module simplemente generalizará el estado proveniente del DSP. Es decir, traducirá el estado de formato `DSP_HERO_TELEMETRY` a `HERO_TELEMETRY`. Esto no quiere decir que en un futuro no se puedan incluir más entradas para que comparándolas se pueda llegar a una mayor precisión en la estimación del estado.

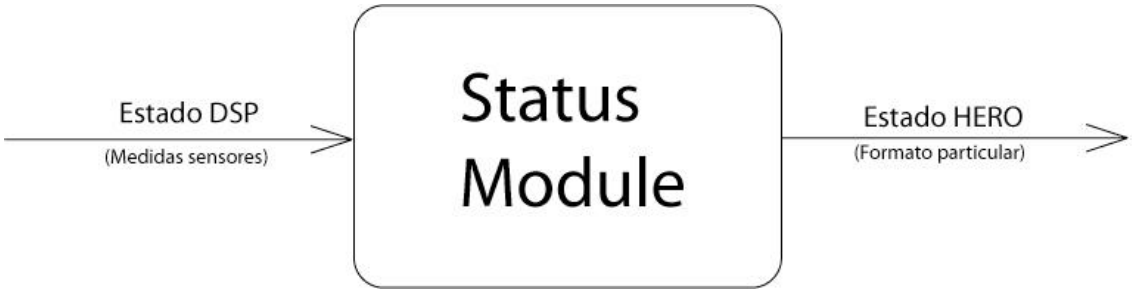


Figura 3.3-1: Bloque del Status Module

Si se desea, este mismo módulo es capaz de generar un log con los campos que se deseen de su entrada (es decir, de la estructura `DSP_HERO_TELEMETRY`). Para ello existe un archivo `log.conf` en el que basta poner a uno los campos de los que se desea hacer log. Este archivo nos recuerda a la sección perteneciente a la máscara de transmisión de estado del archivo `bridge.conf`. Esto es porque se ha usado idéntico criterio a la hora de agrupar las variables, y de hecho se ha usado una versión modificada del `CParser` (llamada `CParser_HSMLog`) para cargar dicho archivo. Esto nos lleva además a hacer una llamada a la precaución, ya que si se activa el log de un campo del que no se ha activado la transmisión, la información tendrá un comportamiento no definido y seguramente en parte falso.

Resumiendo, el Status Module pasa el estado a la capa superior en formato correcto y, en caso de estar activado algún campo en `log.conf`, genera un log con fecha y hora en su nombre (por ejemplo: `hero(1)_28_08_06-20_39_26`) con las entradas recibidas de dichos campos. Los logs se guardarán en la carpeta `./hero/ril/status/log/`.

### 3.3.1 Implementación software del Status Module

La implementación del Status Module la lleva a cabo el proceso multi-hilo `STATUSMOD`, cuyo único parámetro será el número que identifica al robot al que pertenece. Cuenta con tan solo 3 hilos (incluido el `main`) que veremos a continuación.

**Estructuras**

De la Figura 3.3-2 se observan las estructuras que tienen asociadas las señales de entrada y salida de este módulo:

(*input*) Estado DSP: `DSP_HERO_TELEMTRY` (`./hero/comms/UAV_IMI_data.h`)
Contiene información detallada sobre el estado (telemetría) del helicóptero: posición, orientación, velocidades, aceleraciones, señales de actuación, referencias, errores... todo en el formato propio del DSP.

(*output*) Estado HERO: `HERO_TELEMTRY` (`./hero/comms/UAV_IMI_data.h`)
La información es en esencia la misma de la entrada, sólo que ahora se expresa en el formato particular de HERO, que ya es apto para subir a la siguiente capa software.

Por lo demás, tanto las clases como los procedimientos de este módulo son análogos a los empleados en el Bridge Module (sólo hay una clase totalmente nueva), por lo que no vemos interés en repetirlos. A continuación sólo veremos los hilos y destacaremos las diferencias. Si no se dice lo contrario, todos los archivos se encuentran en la carpeta `./hero/ril/status/source/`.

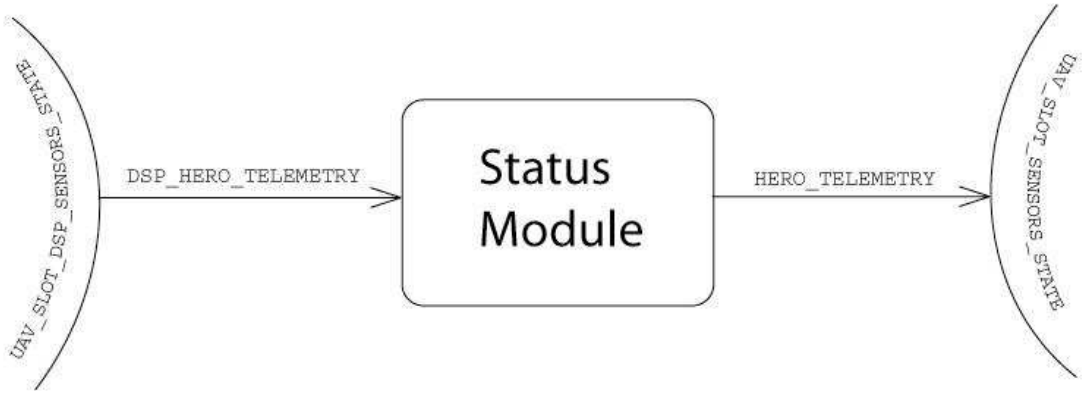


Figura 3.3-2: Bloque del Status Module (punto de vista software)

**Hilo principal** (en `./hero/ril/status/main.cpp`)

Como de costumbre, se limita a lanzar el resto de los hilos (en este caso el `CommunicationThread` y el `ProcessingThread`) y a esperar `'Ctrl+C'` para terminar correctamente la ejecución. La sección crítica que usará este proceso será heredada de `CCriticalSection`, ya que en este caso sólo necesitamos un mutex, se ha preferido la herencia frente a la composición, aunque sea por el simple hecho de la comodidad.

**Hilo `CommunicationThread`** (en `statusmodcommunicationthread.cpp`)

El hilo de comunicaciones también presenta una estructura idéntica a la del homónimo del Bridge Module. La configuración de la conexión se guardará en `connectionsHSM_`*`robot_id`*`.conf`. En este caso sólo se necesitan dos slots, ambos no seguros.

| | Nombre Slot | Secure | Descripción |
|---|---|---|---|
| i | `UAV_SLOT_DSP_SENSORS_STATE(`*`robot_id`*`)` | 0 | Valor de los sensores del DSP (*) |
| o | `UAV_SLOT_SENSORS_STATE(`*`robot_id`*`)` | 0 | Estado en formato particular |

(*) Como vimos, compartido con el Bridge Module

Con total analogía al caso del Bridge Module, la funcionalidad de este hilo se implementa casi por completo en la clase `CStatusModCommunication`.

**Hilo `ProcessingThread`** (en `statusmodprocessingthread.cpp`)

Lo primero que hace el hilo es cargar la configuración desde `log.conf`. Para ello hace uso de la clase `CParser_HSMLog`, modificación de la conocida `CParser` (ya sin el comentado bug). Esta clase carga la configuración en una estructura `LOGCONFIG`, que se define en `HSMLog.h`, cabecera de una clase que se ha creado para trabajar con el log. En la creación de un objeto de esta clase (`HSMLog`) se le pasa una configuración (tipo `LOGCONFIG`) con lo cual ya se decide si se hace log o no (en el caso de todos los campos a cero). En caso afirmativo, se crea un archivo y se coloca ya una primera línea con los nombres de los campos de los que se hará log. Y a través de este objeto será como escribiremos en dicho log cada vez que se pase un nuevo dato de estado. El destructor de la clase se encargará de cerrar el archivo de log (en el caso de que se haya abierto, claro) en cuanto salga '*out of scope*'. Se ha elegido hacer una clase porque es mucho más cómodo, y cabría la posibilidad de hacer varios logs a la vez, con distintos campos cada uno, como objetos independientes que serían.

| heli_volando | tiempo_local | latitud | longitud | altura | heading_GPS | velocidad_horizontal | velocidad_vertical | qos_ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0 |
| 0 | 0 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0 |
| 0 | 0 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0 |
| 0 | 126 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 127 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 128 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 129 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 130 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 131 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 132 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 133 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 134 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 135 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 137 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 138 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 139 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 140 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |
| 0 | 141 | 10.000000 | 10.000000 | 0.000000 | 1.570796 | 0.000000 | 0.000000 | 0 |

Figura 3.3-3: Ejemplo de un log

La función encargada de la traducción es `DSPHeroTelemetry2Hero-Telemetry` y se encuentra definida en `./hero/ril/comms/datatrans-lation.h`, cabecera que vimos con anterioridad en el Bridge Module.