CAPÍTULO 5. MEJORA DEL PROGRAMA DE PARTIDA Y ADAPTACIÓN DEL MISMO A LA NUEVA ESTRUCTURA SOFTWARE PARA *CROMAT*.

5.1 Nueva estructura software para el *UAV*.

5.1.1. Justificación del cambio.

El software que se venía utilizando para el *UAV*, y que se ha descrito en el capítulo anterior, tenía varios defectos. El código estaba escrito de una manera muy poco clara, no resultaba demasiado intuitivo para una persona que necesite retocarlo o ampliarlo. Además tenía algunas deficiencias. Existía un *bug* que hacía que el proceso colapsara cada cierto tiempo. El hilo encargado de la comunicación con el *BBCS* había sido comentado casi en su totalidad y retocado para que pudiera funcionar. El archivo de configuración se leía mal, y cualquier pequeño cambio que se le hiciera (como insertar una línea vacía) resultaba fatal para la comprensión del mismo por parte del programa.

Desde el punto de vista del proyecto *CROMAT* se vio que era conveniente tener una aplicación que fuese más genérica, ya que el programa en cuestión no tenía sentido alguno para otro robot que no sea el helicóptero; y menos centralizada, puesto que el proceso *UAV*, con sus diferentes hilos, pretendía hacer todas las funciones que el *UAV* necesita a alto nivel. Se prefiere una estructura software que involucre a varios procesos que se comuniquen entre sí, que podrán estar en diferentes máquinas. De esta manera se pueden hacer ampliaciones con mucha más facilidad, y el conjunto de programas resulta mucho más modular y sencillo de entender.

5.1.2. Descripción de la nueva estructura.

5.1.2.1. Descripción general.

La nueva estructura tiene varios niveles de abstracción, que van desde el centro de control (usuario final) hasta la comunicación con el *DSP*. En cada nivel de abstracción habrá una serie de módulos, que desarrollarán las tareas necesarias.

5.1.2.2. Niveles de abstracción.

Para el proyecto *CROMAT* se han diseñado tres niveles de abstracción dentro de cada robot: *RAL*, *Module Manager* y el conjunto de módulos.

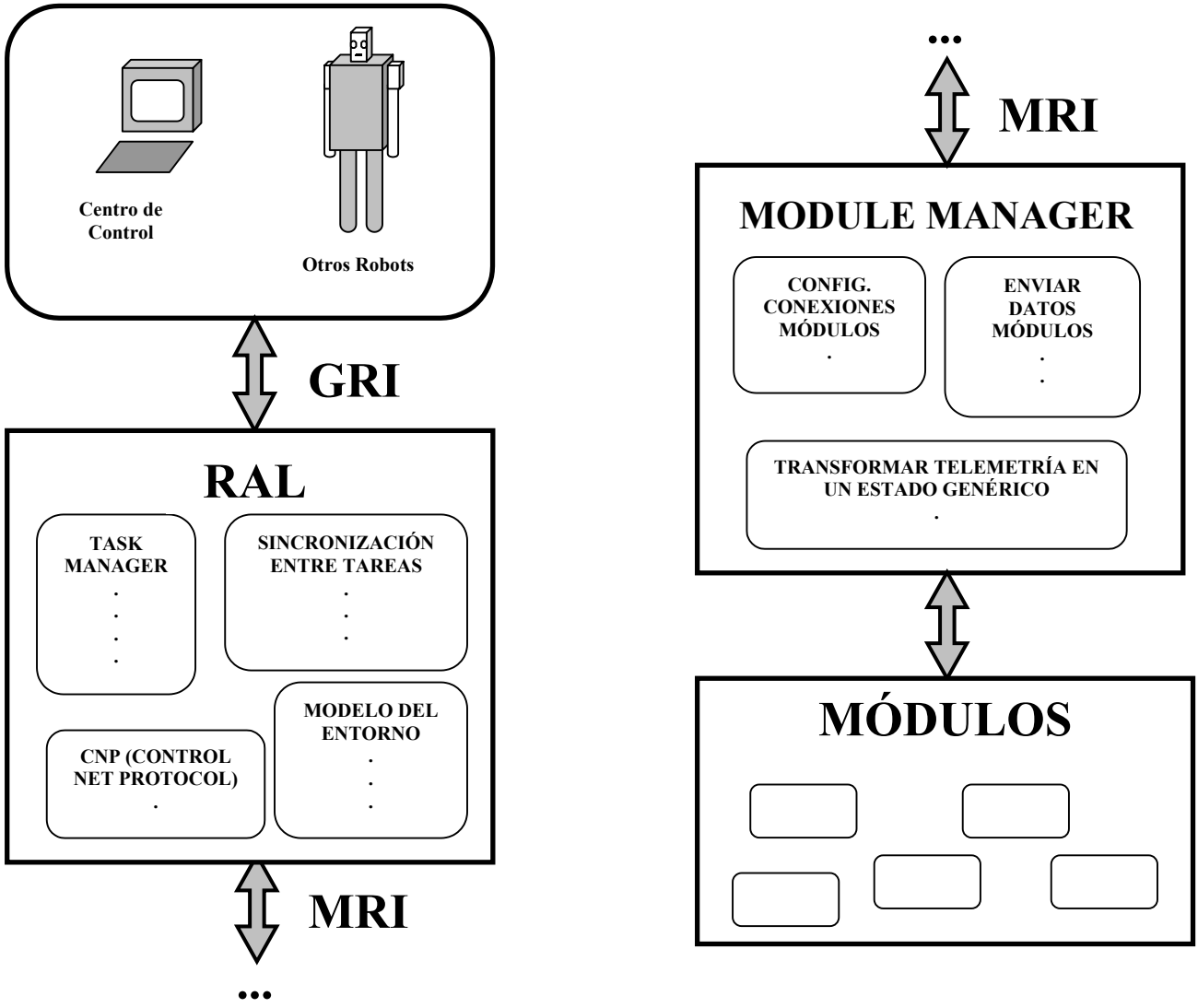

Figura 5-1. Niveles de abstracción.

5.1.2.2.1. *RAL*.

RAL son las siglas de *Robot Abstraction Layer* (Capa de Abstracción del Robot). Este nivel será el mismo en cada robot. En ella se incluyen funciones como un manejador de tareas, la asignación distribuida de tareas, la sincronización entre los diferentes robots o los modelos del entorno. Este nivel se comunica con en centro de control y el resto de robots mediante el interfaz *GRI*, *General Robot Interface* (Interfaz General de Robot). Permite ver desde fuera a todos los robots de la misma forma. El *RAL* se comunica también con su nivel inferior (*Module Manager*) a través del *Management Robot Interface* (Interfaz para el Manejo del Robot) o *MRI*, que al igual que el *RAL* es independiente del robot.

5.1.2.2.2. Module Manager.

El *Module Manager* (Manejador de Módulos) se encarga de manejar las tareas y coordinar el trabajo de los diferentes módulos del nivel inferior, y que posteriormente se describirán. Es el primer nivel que sí es particular para cada robot. Esta capa configura las conexiones de los módulos, envía los datos que necesiten, o transforma la telemetría que recibe de éstos en un estado genérico del robot. Es importante destacar que el *Module Manager* sólo tratará una tarea de cada tipo de las que se realicen en el conjunto de módulos. Para el caso del *UAV* los tratamientos se harán para tres tipos de tareas: percepción, movimiento y *pan & tilt*. Este número de tareas puede ser ampliado en un futuro, según las necesidades que vayan surgiendo.

5.1.2.2.3. Módulos.

Se trata de unidades independientes encargadas cada una de una tarea particular. Estas unidades pueden ser hilos, conjuntos de hilos o más comúnmente procesos. La comunicación entre módulos, salvo excepciones, se hará mediante *BBCS*, facilitando de esta manera la portabilidad y permitiendo que diferentes módulos puedan hallarse en máquinas distintas.

En el caso del *UAV* existirán una serie de módulos con funciones de visión, uno para el control del *pan & tilt*, otro para el control del movimiento, uno más para la exportación de la telemetría, y finalmente un generador de trayectorias. No obstante, la disposición por módulos, así como el interfaz de comunicaciones entre los mismos permiten la ampliación del número de módulos, bien para nuevas tareas, bien como apoyo a los ya existentes, de una forma sencilla y estructurada.

Cada módulo podrá recibir información de otro módulo o del *Module Manager*, y en algún caso también del *DSP*, a través del puerto serie. Si un módulo puede recibir el mismo tipo de datos de varios *slots* diferentes, será el *Module Manager* el que le indique cuál debe tener en cuenta. Por ejemplo, el generador de caminos puede estar escribiendo una trayectoria en su *slot* de salida a la vez que algún módulo de visión haga lo propio en el que le corresponda. El *Module Manager* indicará al módulo receptor el *slot* del que ha de leer.

5.2. Mejora y ampliación del software de control existente para el *UAV*.

5.2.1 Depuración del programa.

Como se dijo al principio, el software para el control del *UAV* tenía algunos *bugs*, que hacían caer al proceso después de un tiempo de ejecución. Tales errores fueron detectados y corregidos. Hasta la fecha, y después de horas de prueba, el programa ha funcionado a la perfección, por lo que se entiende que ya está libre de errores.

5.2.1.1. Variables de condición.

La aplicación que se está viendo utiliza *mutex* y *variables de condición* para el acceso a zonas comunes de memoria. Revisando el código a portar se descubrió un error en el uso de las variables de condición. Había algunos casos en los que se hacía la espera de la condición (*pthread_cond_wait*) sin haber cerrado el *mutex* previamente. La propia ayuda de *QNX* explica que es necesario cerrar el *mutex* antes de la espera. Se recuerda que *pthread_cond_wait* abre el *mutex*, permitiendo de esta manera que otro hilo entre en la sección crítica. Si un hilo estaba ya en esta zona, con el *mutex* cerrado, puede darse el caso de que dos hilos estén trabajando con la memoria compartida, lo cual lleva a un fallo del proceso. Esto se muestra en el siguiente ejemplo.

Se tienen tres hilos. Dos de ellos acceden a una zona compartida, y un tercero necesita esperar una variable de condición. Se supone que en el hilo principal está declarado el *mutex* (mut) y una variable global, que puede ser de cualquier tipo, por ejemplo un vector de enteros llamado *cola*, y de tamaño 1024.

En el gráfico anexo se aprecia cómo hay un momento en el que los hilos 1 y 3 están a la vez en la zona crítica. Este tipo de situaciones son precisamente las que se tratan de evitar con el *mutex*. La solución, como se comentó, pasa por añadir *pthread_mutex_lock* y *pthread_mutex_unlock* antes y después de la espera de la condición respectivamente en el segundo hilo. En tal caso el hilo 2 no abriría el *mutex* ya que se quedaría esperando junto al hilo 3 a que el primer hilo saliera de la zona crítica.

Hilo	Hilo 1	Hilo 2	Hilo 3

Rutina	Void * hilo1(void* arg)
	{
	<Código inicial del hilo>
	...
	/*Clausura del *mutex*/
	pthread_*mutex*_lock(&mut);
	...
	<código de la **sección crítica**. Aquí utiliza "cola">
	...
	/*Reapertura del *mutex*/
	pthread_*mutex*_unlock(&mut);
	...
	<Código final del hilo>
	}
		void * hilo2(void* arg)
		{
		<Código inicial del hilo>
		...
		/*Espera de la condición*/
		pthread_cond_wait(&cond,&mut);
		...
		<Código final del hilo>
		}
			void * hilo3(void* arg)
			{
			<Código inicial del hilo>
			...
			/*Clausura del *mutex*/
			pthread_*mutex*_lock(&mut);
			...
			<código de la **sección crítica**. Aquí utiliza "cola">
			...
			/*Reapertura del *mutex*/
			pthread_*mutex*_unlock(&mut);
			...
			<Código final del hilo>
			}

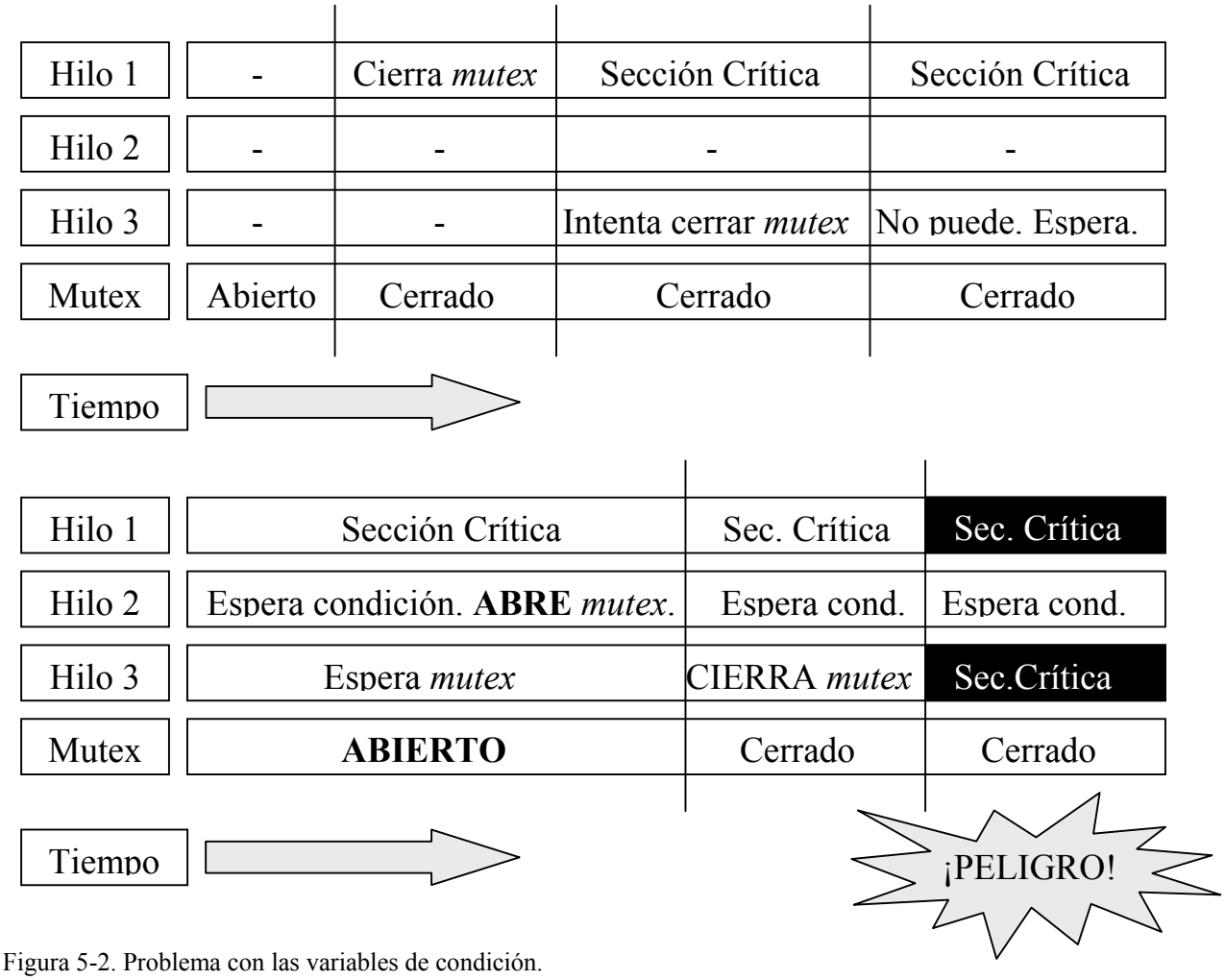

Figura 5-2. Problema con las variables de condición.

5.2.1.2. Índice de cadena fuera de rango.

Aparte del problema con las variables de condición, el programa presentaba un *bug* en el hilo de procesado de los caracteres recibidos. El índice de un buffer en ocasiones superaba el tamaño del mismo, provocando un error de segmentación. En efecto, el mencionado índice incrementaba su valor cada vez que recibía un nuevo carácter, fuese éste el que fuera, poniéndose a cero cada vez que llegaba una nueva cabecera, y con ella un nuevo mensaje. Mientras el puerto serie recibiera bien los caracteres no había problema. Sin embargo en ocasiones se pierden caracteres, o se reciben erróneos. En tal caso puede suceder que se pierda una cabecera, o que, también por fallos del puerto serie, reciba una cabecera que no reconozca. En ambos casos el índice seguiría subiendo hasta superar el tamaño del buffer.

El problema no era sencillo de detectar, ya que se daba de una manera aleatoria, pues, como es evidente, el puerto serie no falla de forma periódica. La solución al problema ha sido poner a cero el índice en caso de desbordamiento. Con ello se pierde la información almacenada en el buffer. Sin embargo esto no supone un problema, ya que

tal información es inútil, pues el desbordamiento viene provocado por una recepción errónea de los datos.

5.2.2. Mejora del programa.

5.2.2.1 Creación de una clase de lectura del archivo de configuración.

Lo primero que hacía el programa al iniciarse era abrir un archivo de configuración, donde leía datos como los parámetros que debe enviar el *DSP* (telemetría), o las direcciones de las máquinas con las que se iba a comunicar.

5.2.2.1.1. Función de lectura del archivo de configuración.

Para leer el archivo de configuración (*uav.conf*) el programa utilizaba una función que leía línea por línea el archivo, y guardaba los datos según el número de línea que hubiera leído. Las líneas que tuvieran como primer carácter una almohadilla no contaban. La lectura de los caracteres y su conversión a enteros se hacía uno a uno mediante bucles, desde el final de la línea hasta el principio.

```
for(i=longitud-1; i>=0; i--)
{
    dato_int+=((buffer[i] - '0')*j);
    j*=10;
}
```

Este método, aparte de inefectivo, resulta muy inconveniente. En principio cualquier carácter extraño que hubiera al final o al principio de cada línea daba problemas. Por esta razón fue necesaria la detección del retorno de carro que se describió en el capítulo anterior. Aun así cualquier espacio colocado al principio o al final de una línea, así como una línea vacía, daba como resultado una mala lectura del archivo de configuración, y consecuentemente un funcionamiento incorrecto del programa. Para ilustrar los problemas citados vale como ejemplo la lectura de las direcciones *IP* para las comunicaciones. Todas llevaban un carácter adicional, pues se leían mal: '*172.16.1.65x*'. Ante la escasa fiabilidad el código de partida obviaba las direcciones *IP* recogidas del archivo. Era necesario escribirlas directamente en el código, siendo necesario compilar cada vez que hubiera que cambiarlas.

Otro problema que conllevaba el uso de esta función de recogida de datos era que no existía relación directa entre la descripción del dato y el dato en sí. En efecto, el programa diferencia si se trata de un dato u otro según el número de línea en el que esté ubicado (sin contar las líneas con almohadillas al principio). Para que el usuario sepa de qué dato se trata ha de leer el comentario que hay escrito antes del dato, lo cual puede ser confuso, implica que no se puede cambiar el orden, y lleva al desastre cuando por cualquier causa se cambie mal o se pierda alguno de los comentarios. Tampoco ayuda a la hora de hacer ampliaciones al fichero, ya que habría que escribir los nuevos datos al final del fichero, o cambiar muchas líneas de código de la función recolectora. De esta forma los datos nuevos quedarían desordenados, y separados de otros datos similares a ellos, que podrían formar un conjunto.

5.2.2.1.2. Creación de una clase para leer uav.conf, y reestructuración del archivo.

Para solucionar todos los problemas derivados de la lectura del fichero de configuración se ha decidido hacer una nueva clase que lea e interprete el mismo.

El nuevo archivo *'uav.conf'* tendrá una serie de indicadores predefinidos que se colocarán delante del dato, y entre ellos un carácter *':'*, o un *'='*. A diferencia que en el fichero antiguo, aquí se pueden insertar comentarios al final de cualquier línea sin que ello produzca problemas. Se pueden crear también grupos de datos, con identificadores y llaves.

Archivo antiguo	Archivo nuevo

`#dato 1:`
`25`
`#grupo 1, dato 1:`
`42`
`#grupo 1, dato 2:`
`35`
`#dato 2:`
`2`
`...`
	`Dato1: 25  #comentario (opcional)`
	`Grupo: grupo1  # ...`
	`{`
	`  dato1: 42 # ...`
	`  dato2: 35 # ...`
	`}`
	`datos 2: 2 # ...`
	`...`

Para la recogida de datos se ha utilizado el código de la clase '*CBBCSParser'*, que recoge conjuntos de caracteres, y los va guardando en un tabla de cadenas. La clase entiende como conjuntos de caracteres aquéllos que estén separados por espacios, caracteres de fin de línea, o cualquiera de los siguientes: `':', '{', '}', '[', ']', '(', ')' y '='`. También detecta las almohadillas, y desecha los caracteres que queden entre una de ellas y el final de esa línea.

A partir de aquí se irán leyendo los conjuntos, actualizando las variables de configuración que correspondan a cada uno de los identificadores que se vayan leyendo.

De esta manera el archivo de configuración queda con una estructura mucho más intuitiva y cómoda cara al usuario, más aún cuando sigue el modelo de archivos de configuración que suele utilizarse en Linux, y *UNIX* en general. Además este sistema le confiere una gran seguridad a la lectura del archivo, puesto que se terminan los problemas con los caracteres extraños o sobrantes y con el orden en el que se pongan los datos.

5.3. Adaptación del programa a la nueva estructura software *CROMAT*.

Como se explicó en el primer apartado de este capítulo, se ha definido una nueva estructuración del software para *CROMAT* en general, y en particular para el *UAV*. Como es lógico el programa del que parte este proyecto no cumplía con las especificaciones ahora creadas, por lo que se ha procedido a una adaptación a las mismas. También se ha aprovechado para incluir algunas funciones que anteriormente no tenía.

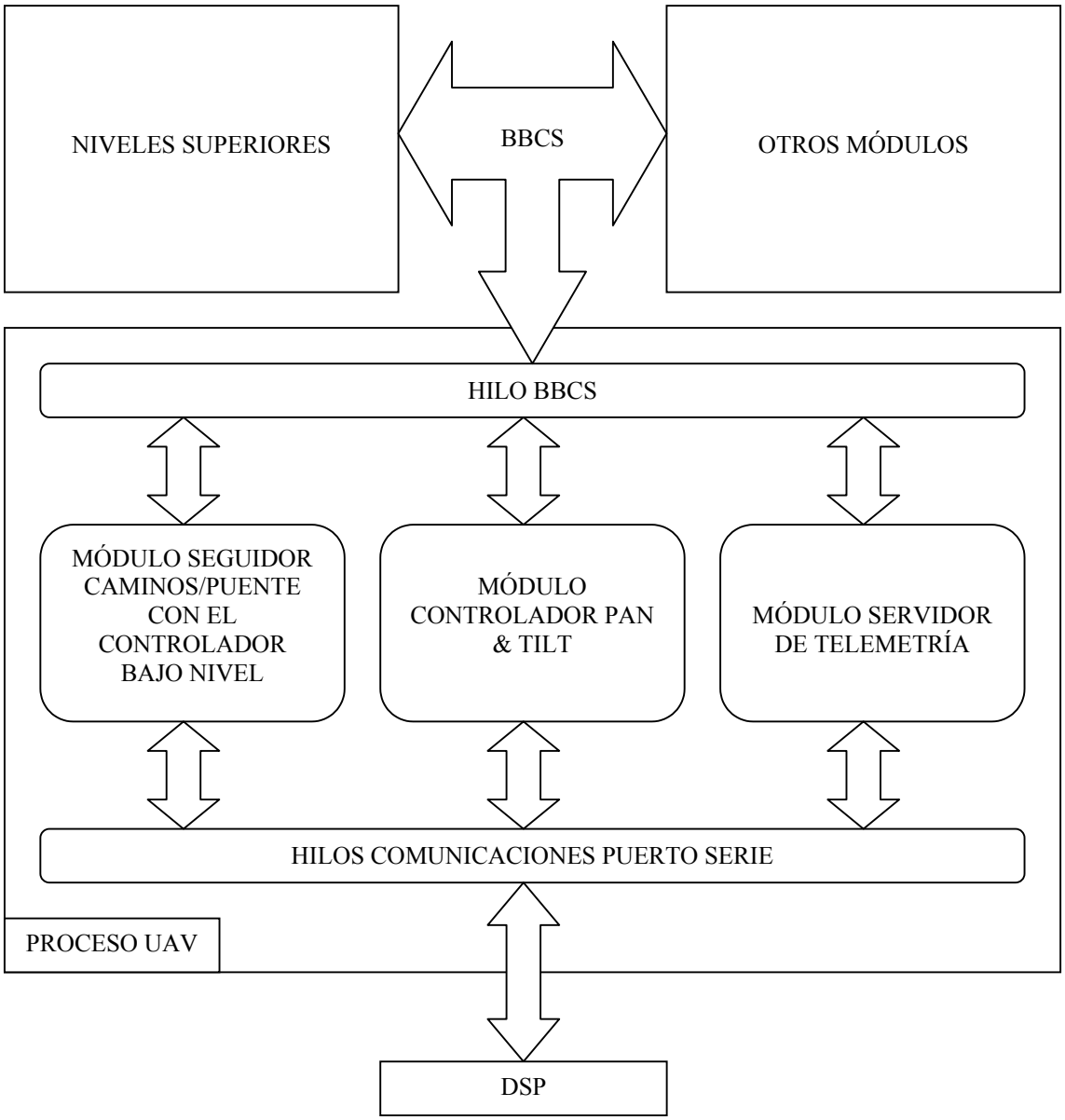

Figura 5-3. Proceso UAV para la nueva estructura software

El software de control del *UAV* se encuentra en el nivel de abstracción más bajo, formando parte de los módulos. Aunque se trate de un único proceso, éste va a ser diferente al resto de los de su nivel, ya que es el que hace las comunicaciones con el *DSP*. Por esta razón este proceso lleva asociados tres módulos diferentes, cuando lo habitual será que cada proceso defina un único módulo.

Los módulos asociados a este software son los del envío de telemetría, control de *pan & tilt* y servidor de caminos o puente con el controlador de bajo nivel. Dichos módulos compartirán los hilos de comunicaciones, tanto con el *DSP* (escritura y lectura del puerto serie) como con el resto de módulos y el nivel superior, a través del *BBCS*. Además, claro está, todos los módulos tendrán en común el hilo principal.

5.3.1. Cambios en los hilos comunes.

5.3.1.1. Hilo principal.

Aparte de las tareas que realizara el hilo principal, ahora deberá hacer algunas más.

En principio, y como medida de seguridad, se pondrá a cero la velocidad de navegación del helicóptero. Esto se hará justo después de haber lanzado el resto de hilos.

Una vez anulada la velocidad se procede a enviar la máscara al *DSP*, activar la transmisión de estado, limpiar el código de error, desactivar el seguimiento e invalidar el camino que tuviera programado en ese momento el *DSP*.

Posteriormente el hilo principal deberá indicar al *DSP* el tipo de helicóptero que va a controlar, según la configuración de servos del rotor principal. Este parámetro se habrá recogido del fichero de configuración, para lo que se ha tenido que crear un nuevo campo en el mismo.

Se crearán también en el fichero de configuración una serie de grupos de variables, correspondientes a los parámetros de los controladores del *DSP*. Tales parámetros se enviarán por el puerto serie antes de que se inicie el bucle de control. Se muestra a continuación el grupo de variables correspondiente al control del ángulo *pitch*, como ilustración de lo que es un grupo en el fichero de configuración, así como de un grupo de parámetros de controlador en particular:

```
PID: PITCH
{
      Kp=1 #Cte. Proporcional.
      Ki=2 #Cte. Integral.
      Kd=3 # Cte. Derivativa.
      valor_int=4 # Valor de la intengral del término integral (suma). Excluyendo la Ki.
      max_awu=5 # Valor máximo de la integral (anti wind-up)
      min_awu=4 # Valor mínimo de la integral (anti wind-up).
      referencia=3 #Referencia.
      offset=2 # Offset que se suma a la salida del PID
      sat_encima=9 # Saturación por encima.
      sat_deb=8 # Saturación por debajo.
}
```

5.3.1.2. Hilo de comunicaciones *BBCS*.

Ya se comentó anteriormente que el hilo de comunicaciones por *BBCS* presentaba unos defectos notables, y para que funcionara de alguna manera el código tuvo que ser comentado casi en su totalidad y se le tuvieron que añadir algunas líneas. Además, en el software antiguo este hilo únicamente se encargaba de transmitir la telemetría. Con el nuevo sistema de módulos el hilo de comunicaciones por red adquiere una importancia mucho mayor, ya que será éste el medio que se utilice para el intercambio de información entre los módulos, y entre éstos y el nivel superior.

Para la programación del sistema de comunicaciones se ha seguido el modelo que se viene utilizando en *CROMAT*. Se trata de utilizar un único hilo de comunicaciones por cada proceso, que intercambiará la información de la transmisión con los hilos que la necesiten a través de una sección crítica protegida con un *mutex*. Habrá tantas secciones críticas como tipos de datos a enviar o recibir.

El procesamiento de los datos puede ser más lento que su recepción. En tal caso se podría perder información, ya que mientras se procesa un dato si llega más de uno del mismo tipo se sobreescribirían en el buffer. En caso de recibirse la información por un *slot* no seguro el problema carecería de importancia, ya que se entiende que esa información no es del todo importante. Sin embargo en los *slots* seguros podría ser muy perjudicial perder información. Para evitarlo se crean una serie de listas en la sección crítica (una por cada tipo de dato con *slots* seguros), y de esta forma es poco probable que se pierda información.

Para el manejo de las funciones *BBCS* se hace uso de la clase *CBBCSFunction*. Esta clase permite, entre otras cosas, que se realicen las conexiones a partir de un archivo de configuración. De esta manera los datos de la conexión que había en el archivo de configuración de la aplicación antigua se pasan al fichero para el *BBCS*.

5.3.2. Programación de los módulos.

Tras poner a punto los hilos comunes del proceso se pueden programar los tres módulos en que se divide el proceso de partida, y que se enumeraron en un apartado anterior.

5.3.2.1. Módulo servidor de telemetría.

Estará siempre activo. Se dedicará simplemente a exportar la telemetría al nivel superior y al resto de módulos, si es que alguno la necesitara. Hará uso de un *slot* no seguro.

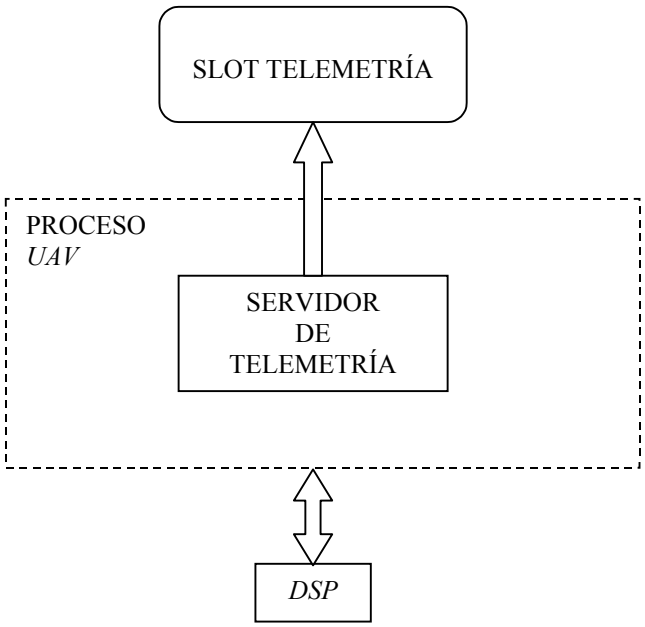

Figura 5-4. Módulo servidor de telemetría.

5.3.2.2. Módulo controlador de *pan & tilt*.

Estará siempre activo. Pondrá accesible a cualquier módulo el control del *pan & tilt* en sus diferentes modos de funcionamiento, que más adelante se describirán. Se enumeran a continuación los diferentes *slots* que interactuarán con el módulo, y la funcionalidad de cada uno.

5.3.2.2.1. *Slot CONTROL ENTRADA*.

Se trata de un *slot* de entrada, de tipo *state* seguro. Este *slot* llevará asociada una estructura del tipo *CONTROL_ENTRADA_PT*:

```
typedef struct
{
	int slot_entrada_control_modo;
	int slot_entrada_datos;
} CONTROL_ENTRADA_PT;
```

Los campos de la estructura hacen referencia a los *slots* de donde el módulo habrá de recoger los datos. Puede que existan varios módulos que aporten el mismo tipo de información. Esta información se incorporará en el *slot* de salida de cada módulo que la facilite. Sin embargo el módulo controlador de *pan & tilt* sólo podrá hacer caso a un módulo por cada tipo de datos recibidos, por lo que sólo recogerá información de un *slot* por tipo. Ése es el *slot* que se le indicará en cada campo de *CONTROL_ENTRADA_PT*.

Podría suceder que no se requiriera información de alguno de los *slots*. En tal caso se recibiría un -1, que el programa entenderá como un *slot* al que no habría que atender.

5.3.2.2.2. *Slot CONTROL MODO.*

Se trata de un *slot* de entrada, de tipo *state* seguro. Este *slot* llevará asociada una estructura del tipo *MODO_FUNCIONAMIENTO_PT*:

```
typedef struct
{
	int modo_de_funcionamiento;
} MODO_FUNCIONAMIENTO_PT;
```

El único campo de la estructura define el modo de funcionamiento del *pan & tilt*, que puede ser:

0: Se dan las referencias de *pan & tilt* por la interfaz de datos en cada instante (referencia de ángulos pan y tilt).

1: Se apunta a un objetivo o destino. El punto se tomará de la interfaz de datos. El dato se pasará con sus coordenadas *GPS* (latitud, longitud, altura).

2: Modo mantener la orientación del *pan & tilt* respecto a unos ejes de referencia locales. Se mantendrá la orientación que haya en ese momento.

Cuando se reciba un dato por este *slot*, simplemente se le pasará tal cual al *DSP*.

5.3.2.2.3. *Slot DATOS.*

Se trata de un *slot* de entrada, de tipo *state* no seguro. Este *slot* llevará asociada una estructura del tipo *DATOS_PT*:

```
typedef struct
{
	float ref_pan;
	float ref_tilt;
	double latitud;
	double longitud;
	double altura;
} DATOS_PT;
```

Los elementos de la estructura aportarán las consignas para el controlador del *pan & tilt*. Los dos primeros campos son las referencias que se le pasan al controlador cuando se activa el modo 0, y los tres últimos definen el punto hacia el que han de mirar las cámaras en el modo 1 (ver *slot CONTROL_MODO*).

Al recibir un dato del *slot*, el módulo enviará las referencias al *DSP* si el modo es 0, enviará las coordenadas del punto en el caso de que el modo sea 1 o no hará nada para modo 2.

Rangos de las variables:

float ref_pan:	[-180, 180]
float ref_tilt:	[-90, 90]
double latitud:	[-90, 90]
double longitud:	[-180, 180]
double altura:	[-∞,+∞]

5.3.2.2.4. *Slot ERROR.*

Se trata de un *slot* de salida, de tipo *state* seguro. Este *slot* llevará asociada una estructura del tipo *ERROR_PT*:

```
typedef struct
{
     int error_pan_tilt;
} ERROR_PT;
```

El único campo de la estructura es un entero, que llevará el código del error que se haya producido. Este error se encolará, y el hilo de comunicaciones enviará los errores encolados, siempre que haya alguno.

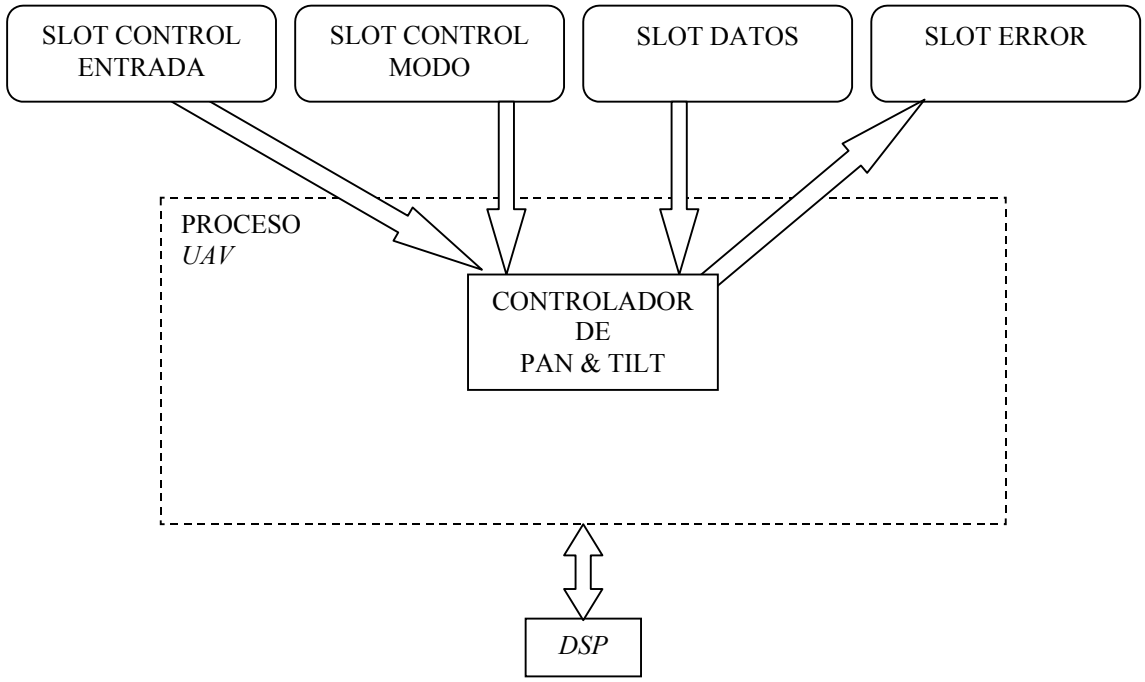

Figura 5-5. Módulo Pan & tilt.

El único tipo de error que podría darse para el caso del *pan & tilt* sería la recepción de algún dato fuera de rango. Los primeros datos que habrán de ser revisados son los que lleva la estructura *CONTROL_ENTRADA_PT*, ya que si los *slots* que se van a mirar son incorrectos no se puede continuar con el resto de tareas. Esta operación se hará en el propio hilo de comunicaciones, puesto que es éste el que ha de utilizar los mencionados *slots*. Las estructuras del resto de *slots* se comprobarán en el hilo del *pan & tilt*. Es importante destacar que ningún dato será computado, enviado o utilizado en caso de existir cualquier tipo de error que le afectara directa o indirectamente.

5.3.2.3. Módulo seguidor de caminos o puente con el control de bajo nivel.

Estará siempre activo. Ofrecerá los servicios de movimiento al *DSP*. Éstos podrán estar en forma de "seguidor de caminos", o bien como "puente con el control de bajo nivel". Al igual que se hiciera con el módulo anterior, se enumeran a continuación los diferentes *slots* que interactuarán con el módulo, y la funcionalidad de cada uno.

5.3.2.3.1. *Slot CONTROL_ENTRADA.*

Se trata de un *slot* de entrada, de tipo *state* seguro. Este *slot* llevará asociada una estructura del tipo *CONTROL_ENTRADA_SEG_CAM*:

```
typedef struct{
     int slot_entrada_control_modo;
     int slot_entrada_heading;
     int slot_entrada_speed;
     int slot_entrada_datos;
} CONTROL_ENTRADA_SEG_CAM;
```

Slot análogo al de entrada de datos del *pan & tilt*. Los campos de la estructura dictan los *slots* a tener en cuenta cuando hay varios con un mismo tipo de datos. Un -1 indicará que se trata de un *slot* que no será necesario atender.

5.3.2.3.2. S*lot CONTROL_MODO.*

Se trata de un *slot* de entrada, de tipo *state* seguro. Este *slot* llevará asociada una estructura del tipo *CONTROL_MODO_SEG_CAM*:

```
typedef struct{
     int modo_o_comando;
     int modo_de_funcionamiento;
     int numero_de_comando;
} CONTROL_MODO_SEG_CAM;
```

Mediante este interfaz se le indicará al controlador de bajo nivel si se le está enviando un nuevo modo de funcionamiento o un comando, y el valor de ese modo o ese comando. Los valores de los componentes de la estructura podrán ser:

modo_o_comando:

0: El mensaje es relativo al cambio del modo de funcionamiento.

1: El mensaje es un comando.

modo_de_funcionamiento:

0: Modo seguimiento de caminos. El *DSP* calculará el *heading* de forma automática. El helicóptero siempre se orientará hacia el siguiente punto del camino.

1: Modo seguimiento de caminos . El *heading* del helicóptero será el que se le aporte por el *slot HEADING*, que se verá más tarde.

2: Modo semiautomático: El *DSP* será el que estabilice el aparato, si bien el piloto podrá dar las referencias a los controladores a través del radiomando.

3: Reservado para un uso futuro.

4: Modo despegue.

5: Modo aterrizaje.

6: Modo en el que el módulo se comporta como un puente con el controlador de bajo nivel.

numero_de_comando:

0: Abortar.

El módulo actuará de la siguiente manera cuando reciba un mensaje por este *slot*:

Modo o comando	**Modo de funcionamiento**	**Numero de comando**	**Acción**
1	**-**	**0**	Comando aborto. Se vaciarán las colas de puntos tanto del *DSP* como del *PC-104*.

Modo o comando	Modo de funcionamiento	Numero de comando	Acción
0	**0**	**-**	Se le pasará al *DSP* el modo de funcionamiento, y se activará el seguimiento.
	1	**-**	Se calculará el *heading* del helicóptero en ese momento, y junto al modo de funcionamiento se le pasará al *DSP*. Activación del seguimiento.
	2	**-**	Se le pasará el modo al *DSP*, se desactivará el seguimiento, y se invalidará el camino.
	3	**-**	Por ahora nada. En un futuro se utilizará para señalar el uso de un seguidor de caminos alternativo al del *DSP*, y que iría programado en este módulo.
	4	**-**	Se desactivará el seguimiento, se invalidará el camino, y se pondrá el *DSP* en modo despegue.
	5	**-**	Idem modo aterrizaje.
	6	**-**	Se pasará al *DSP* el modo puente.

5.3.2.3.3. *Slot HEADING.*

Se trata de un *slot* de entrada, de tipo *state* no seguro. Este *slot* llevará asociada una estructura del tipo *HEADING_SEG_CAM*:

```
typedef struct{
     float heading;
} HEADING_SEG_CAM;
```

El único elemento de la estructura indica un valor del *heading* en radianes.

Rango: $[-\pi, \pi)$

Si está activado el modo 1, cuando se reciba algo por este *slot* el programa pasará de radianes a grados el valor del *heading*, y lo mandará al puerto serie. En caso de estar activado cualquier otro modo, el módulo no tendrá en cuenta la recepción de esta estructura.

5.3.2.3.4. S*lot VELOCIDAD*.

Similar al anterior, el *slot VELOCIDAD* es no seguro, de entrada y de tipo *state*. Llevará asociada una estructura del tipo *SPEED_SEG_CAM*:

```
typedef struct{
     float speed;
} SPEED_SEG_CAM;
```

El único campo de la estructura indica la velocidad que deberá llevar el helicóptero.

Rango: [0, *MAX_SPEED*], donde *MAX_SPEED* es una variable que se leerá del fichero de configuración.

Cuando se reciba algún dato por este *slot* éste se le comunicará siempre al *DSP*, sin importar el modo en curso.

5.3.2.3.5. *Slot DATOS*.

Sin duda el *slot* más extraño de todos. Es de tipo *stream*, de entrada y seguro. El tamaño del mensaje es desconocido a priori, por lo que el buffer tendrá que irse ampliando en caso de que el tamaño de éste sea menor que el de los datos.

La información que se recibe tendrá el siguiente formato:

unsigned int nº puntos.
DATO_PUNTO_SEG_CAM punto 1
DATO_PUNTO_SEG_CAM punto 2
...
DATO_PUNTO_SEG_CAM punto n

Donde *DATO_PUNTO_SEG_CAM* es el siguiente tipo:

```
typedef struct{
     double latitud;
     double longitud;
     double altura;
}DATO_PUNTO_SEG_CAM;
```

La razón de recibir los datos en este formato y no en otro más lógico es que de esta manera el módulo tendrá información de la trayectoria entera, lo cual es importante para una hipotética implementación de un seguidor de caminos en el *PC-104*.

Los campos de *DATO_PUNTO_SEG_CAM* indicarán las coordenadas *GPS* de un *waypoint*.

Rangos:

unsigned int nº puntos:	$[1, 2^{32}-1]$
double latitud:	[-90, 90]
double longitud:	[-180, 180]
double altura:	$[-\infty, +\infty]$

En caso de estar activados los modos 0 ó 1, cuando se reciba un dato de este tipo se deben desempaquetar los datos, e irlos enviando al *DSP*. En caso contrario no se actuará.

Para cumplir estas especificaciones ha habido que cambiar las funciones que empaquetaban los datos para mandarlos por el puerto serie y la función que los encola, ya que en la versión anterior del programa el *heading* entrada dentro de la estructura del punto. Se ha creído conveniente sacarla para que dos módulos diferentes puedan pasarle al controlador uno las consignas del *heading* y el otro las de los *waypoints*.

Antes de enviar los datos al *DSP*, para el caso de la latitud y la longitud hay que multiplicarlos por 10.000 y pasarlos a enteros. Esto se debe a que el *DSP* no utiliza flotantes de doble precisión, y podría haber problemas con la resolución al usar precisión simple.

5.3.2.3.6. *Slot REFS.*

Se trata de un *slot* de entrada, de tipo *state* no seguro. Este *slot* llevará asociada una estructura del tipo *REFS_SEG_CAM*:

```
typedef struct{
     float ref_altura;
     float error_longitudinal;
     float error_lateral;
     float ref_heading;
     float ref_rpm_motor;
} REFS_SEG_CAM;
```

Los campos de la estructura definen las referencias que se le pasan al *DSP* cuando está en modo puente.

Rangos:

float ref_altura:	[-∞, +∞]
float error_longitudinal:	[-∞, +∞]
float error_lateral:	[-∞, +∞]
float ref_heading:	[-π, π)
float ref_rpm_motor:	[0, 1.800]

Los datos recibidos por este *slot* sólo cobrarán sentido en el modo 6. En tal caso el programa se limitaría a enviar los valores de la estructura al *DSP*.

5.3.2.3.7. *Slot ERROR.*

Se trata de un *slot* de entrada, de tipo *state* seguro. Este *slot* llevará asociada una estructura del tipo *ERROR_SEG_CAM*:

```
typedef struct{
     int error_seg_cam;
} ERROR_SEG_CAM;
```

Al igual que pasaba en el módulo controlador del *pan & tilt*, el único elemento de la estructura *ERROR_SEG_CAM* es el código del error que se haya producido en el módulo. El funcionamiento es análogo al caso del *pan & tilt*, con la única diferencia de que también se enviarán algunos errores que devuelva el *DSP*. En particular se mandarán al *slot* aquellos errores que afecten de manera directa a las tareas ordenadas por los niveles superiores.

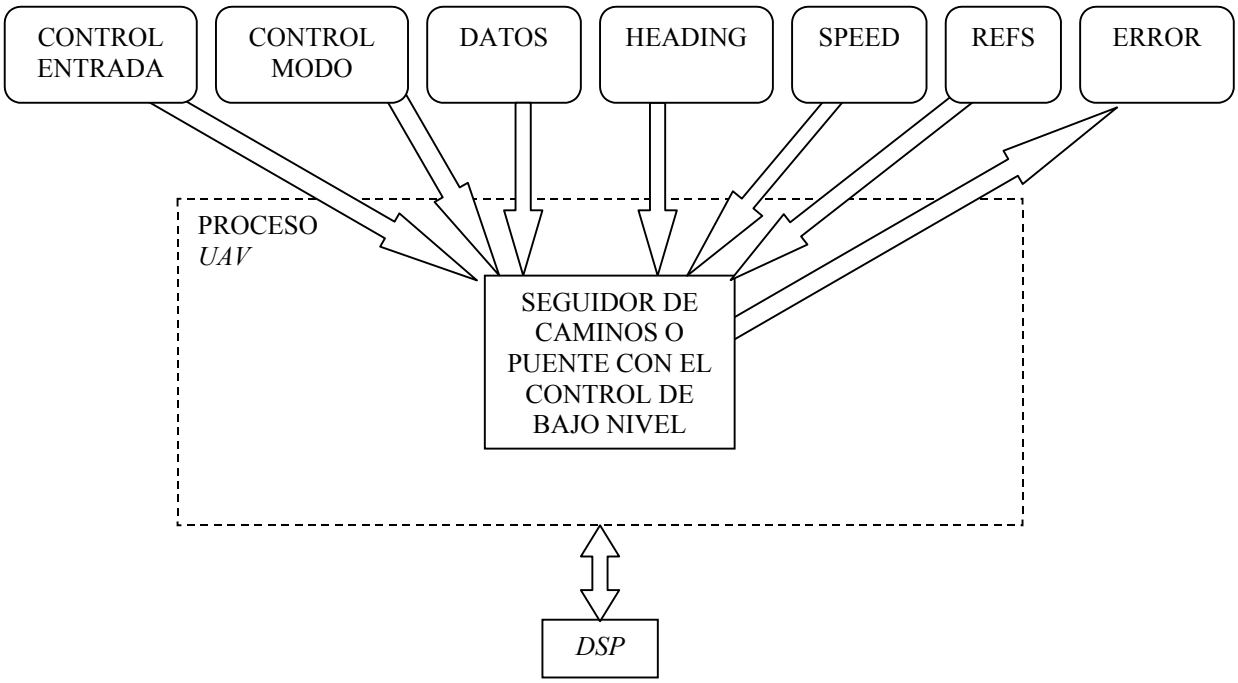

Figura 5-6. Módulo seguidor/puente.

5.4. Conclusiones.

En lo que respecta a la mejora del software antiguo, se puede decir que tras el cambio el programa es mucho más robusto, útil y fiable. Hasta la fecha el funcionamiento ha sido exitoso en todas las pruebas realizadas, por lo que se entiende que los fallos de programación han sido todos reparados.

En cuanto a la implementación de la nueva estructura software, ésta se ha tratado de escribir de la manera más ordenada posible y siguiendo la filosofía de programación de *CROMAT*. Sacrificando eficacia en favor de estructuración. Esto unido a la abundancia de comentarios en el código hacen mucho más sencilla las labores de modificación y/o ampliación que en un futuro pudieran hacerse.