Capítulo 3

La Robot Implementation Layer (RIL)

3. LA ROBOT IMPLEMENTATION LAYER (RIL)

La capa RIL, *Robot Implementation Layer*, es la que se encarga de trabajar con el hardware del ROMEO-4R, por lo que se encuentra en el nivel más bajo (menos abstracto) de esta arquitectura.

En la arquitectura general de los robots, presentada en el capítulo anterior, se puede observar como esta capa es la única que depende por completo del robot en cuestión sobre el que está trabajando. De esta forma, cada robot tendrá una capa RIL distinta, que dependerá del hardware exclusivo que posea cada uno.

En dicha capa se encuentran todos los drivers, funciones y módulos que controlan todos los dispositivos, tanto sensores como actuadores, del vehículo.

Para estudiar esta capa con un orden que haga más comprensible al lector su entendimiento, se seguirá un esquema común en todos los módulos. Esto no sólo es un método pedagógico, sino que se basa en una forma de programación sistemática que se ha adoptado a lo largo de toda la programación de este Proyecto Fin de Carrera, así como en toda la arquitectura del ROMEO-4R y demás robots que la usarán. Así, será más fácil y comprensible la adición de nuevos módulos al código, donde un programador de alto nivel tendrá unas pautas de programación a seguir que faciliten dicha tarea.

La RIL estará formada por los módulos que se muestran en la Figura 3-1, donde en rojo se indica el módulo correspondiente al simulador del HAM, el cual sustituirá al módulo del HAM cuando se realicen pruebas en las que sea necesaria la simulación del hardware del robot (este módulo se estudiará en el siguiente capítulo). El código asociado a esta capa se encuentra dentro del repositorio, siendo este el punto de partida del sistema, en la carpeta *./romeo/ril*.

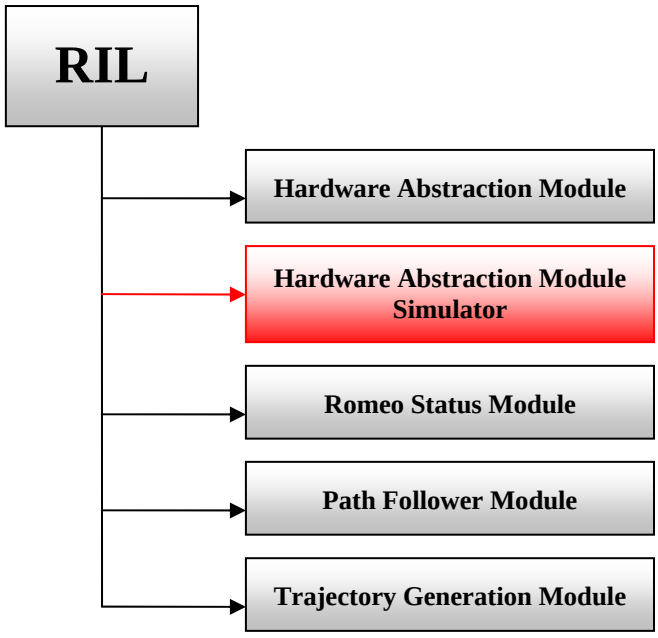


Figura 3-1: Esquema con los módulos de la capa RIL

Tal y como se comentó anteriormente, existe una estructuración del código que seguirá, básicamente, el esquema presentado en la Figura 3-2, donde se encuentran los archivos que tendrán los diferentes módulos de la RIL. Únicamente la parte indicada en rojo, correspondiente a la clase que permite manejar el estado del módulo (*Manager*) y a algunas funciones de utilidad para el programa principal, será la que no aparecerá en determinados módulos.

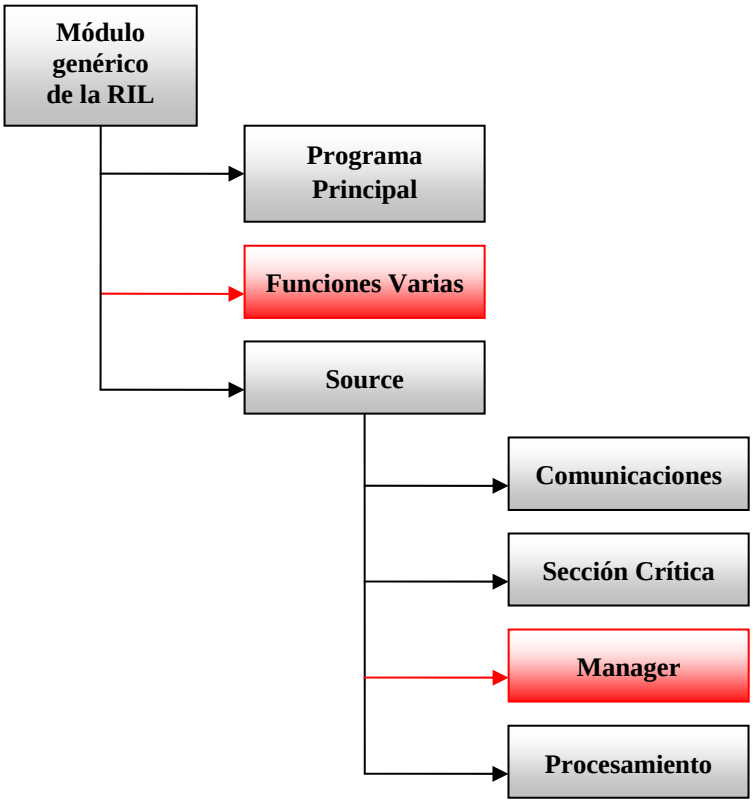


Figura 3-2: Esquema de un módulo genérico de la capa RIL

A continuación se realiza una descripción detallada de los distintos módulos de la RIL, los cuales se introdujeron en el capítulo anterior, con las correspondientes definiciones de las clases y funciones que conforman los mismos.

3.1. Hardware Abstraction Module (HAM)

Se trata del módulo de menor nivel de abstracción de toda la arquitectura. Estando íntimamente ligado al hardware del robot, y dependiendo por tanto del robot en cuestión sobre el que trabaja, se encarga de la gestión de los dispositivos del ROMEO-4R, tanto sensores como actuadores, recibiendo las lecturas de datos de los primeros, de forma que estén disponibles para otros módulos, y enviando a los segundos las referencias de control que le lleguen.

Al igual que ocurrirá en todos los módulos, contará con su sistema de comunicaciones BBCS y con una sección crítica, términos que será comentados en profundidad posteriormente.

Por lo tanto a este módulo se le tienen que pasar las referencias de los controladores y tienen que ser capaz de enviar por el sistema de comunicaciones los datos sin procesar de los distintos sensores.

Para la compilación de este módulo, existe el fichero *Makefile* (*./romeo/ril/HAM/Makefile*), que contiene una lista de todos los archivos objeto del HAM, así como una relación de dependencias de éstos con los archivos fuente, de manera que se puede ahorrar tiempo de compilación gracias a la utilización del comando *make*.

Esto proporcionará un programa ejecutable, al que se llamará mediante el siguiente comando:

>> ./ROMEO_HAM [robot id]

donde *[robot id]* es el identificador del robot con el que se está trabajando.

Por último, comentar que este módulo interactúa, por una parte, con el *Romeo Status Module*, teniendo como salida la información procedente de los dispositivos, y por otra parte, con el *Path Follower Module*, teniendo como entrada las referencias de control para los actuadores. A su vez, tiene las entradas y salidas correspondientes a la interactuación con el hardware del robot.

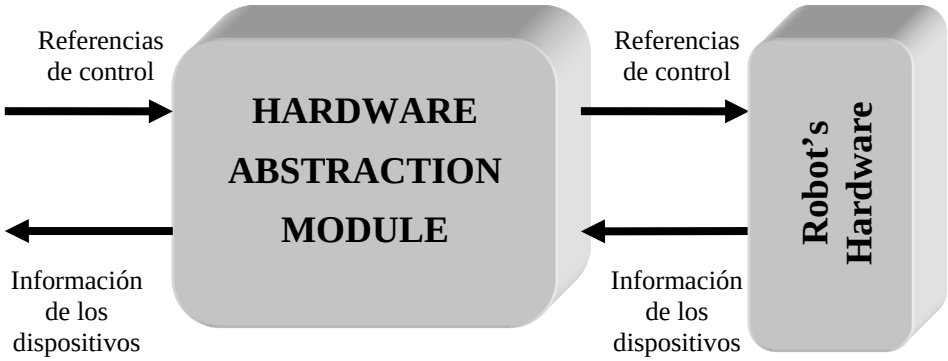
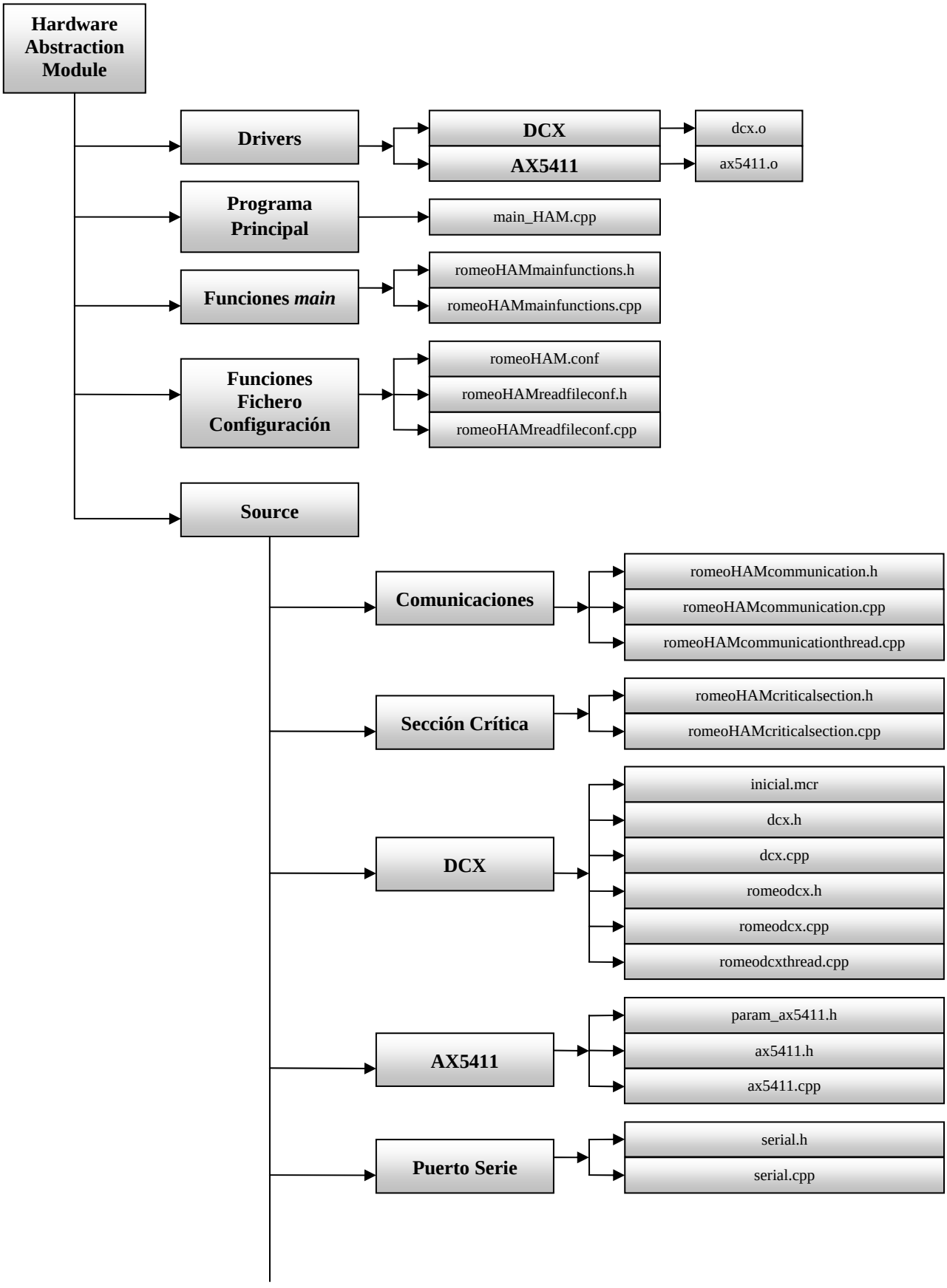
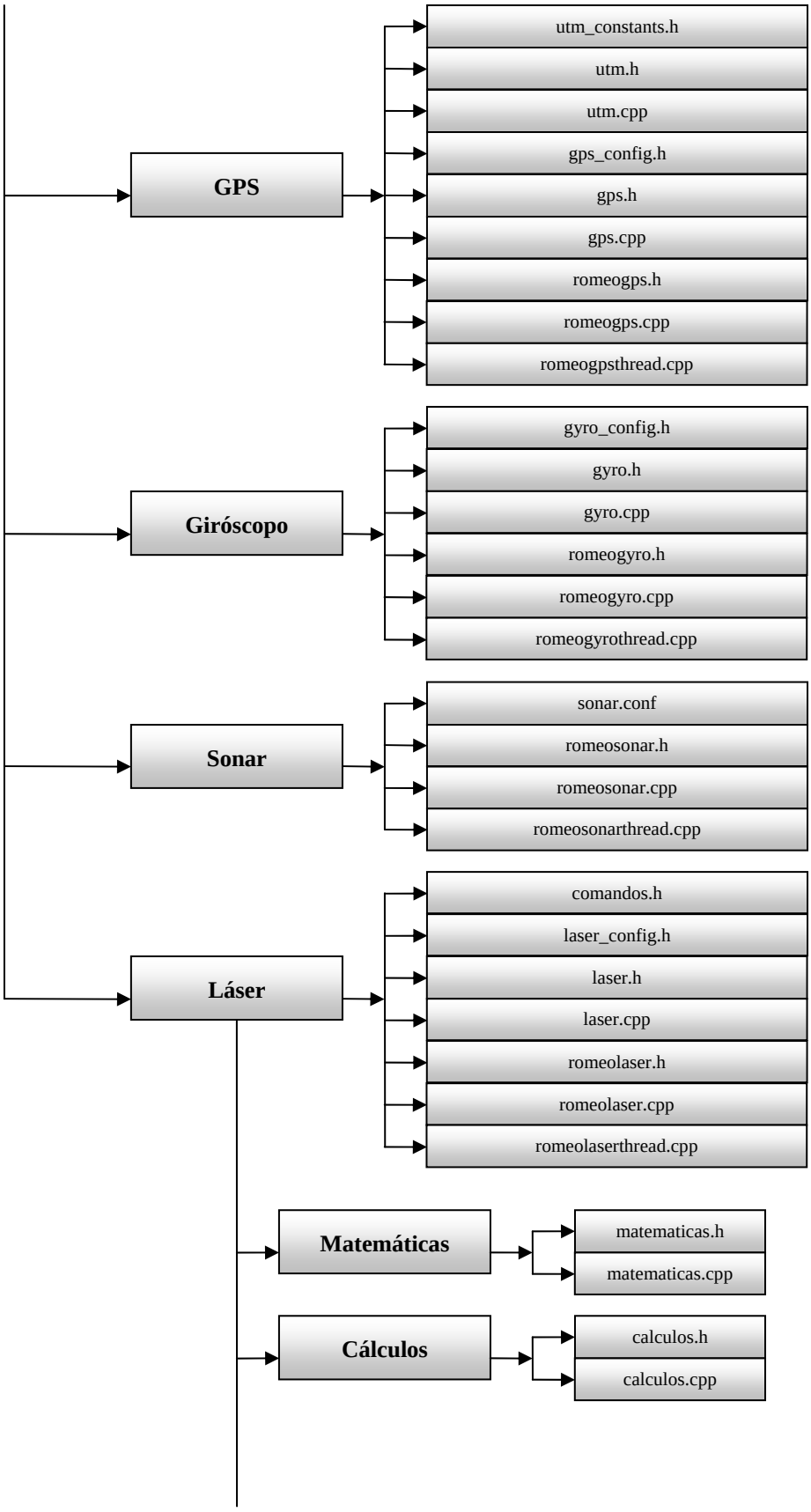


Figura 3.1-1: Entradas y salidas del Hardware Abstraction Module

La estructura del *Hardware Abstraction Module* viene definida por el siguiente esquema:




GPS
utm_constants.h
utm.h
utm.cpp
gps_config.h
gps.h
gps.cpp
romeogps.h
romeogps.cpp
romeogpsthread.cpp
Giróscopo
gyro_config.h
gyro.h
gyro.cpp
romeogyro.h
romeogyro.cpp
romeogyrothread.cpp
Sonar
sonar.conf
romeosonar.h
romeosonar.cpp
romeosonarthread.cpp
Láser
comandos.h
laser_config.h
laser.h
laser.cpp
romeolaser.h
romeolaser.cpp
romeolaserthread.cpp
Matemáticas
matematicas.h
matematicas.cpp
Cálculos
calculos.h
calculos.cpp

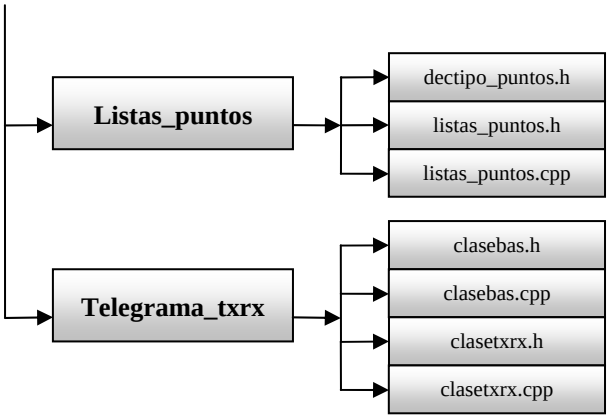


Figura 3.1-2: Esquema del Hardware Abstraction Module

A continuación se describirán con detalle todos los elementos que aparecen en el anterior esquema y que forman parte del HAM.

3.1.1. Drivers

En esta carpeta se encuentran los *drivers* que se encargan de la comunicación de las tarjetas DCX-PC100 (*./romeo/ril/HAM/drivers/dcx)* y AX5411 (*./romeo/ril/HAM/ drivers/ax5411)* con el PC, así como sus correspondientes ficheros de instrucciones.

Los *drivers* o controladores de los dispositivos se encuentran en el nivel más bajo de abstracción. Un *driver* es un elemento software que permite el acceso a un dispositivo para enviar o recibir información hacia o desde el mismo. Por tanto, se encarga de la comunicación a bajo nivel de un dispositivo con un PC.

Cuando se realiza la programación de los *drivers*, es imprescindible conocer bien tanto la interfaz de los dispositivos, esto es, la forma en que se comunica con el PC, como el sistema operativo con el que se esté trabajando, ya que los *drivers* de un mismo dispositivo serán distintos según el sistema operativo para el que se programen.

Al estar el ROMEO-4R basado en el sistema operativo *Linux*, en su distribución *Debian*, se han implementado los *drivers* para dicho sistema, variando a lo largo del tiempo sus distintas versiones, para someterse a las distintas actualizaciones del sistema, por lo que se han ido realizando las modificaciones necesarias en el código para adaptarlas a estas nuevas versiones.

Cabe indicar la necesidad de utilizar los *drivers* que manejan el puerto serie, pero al estar estos implementados dentro del propio código fuente del sistema operativo, no será necesario que se haga.

En la carpeta principal del HAM, existe un programa ejecutable (*./romeo/ril/ HAM/iniciar_drivers*) que se encarga de cargar los *drivers* en el ROMEO-4R. Así, una vez programados los mismos, se obtienen los ficheros objeto (.o) mediante la compilación de los ficheros fuente (.c), cargando después dichos módulos compilados en el núcleo mediante la herramienta *insmod*.

Para una información más extensa y detallada sobre la implementación de los drivers del ROMEO-4R, ver el Proyecto Fin de Carrera de Rafael Martín de Agar Tirado "*Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control*".

3.1.2. Programa principal

El programa principal de ejecución del *Hardware Abstraction Module* (*./romeo/ril/HAM/main_HAM.cpp*) es el encargado, básicamente, de lanzar el hilo de comunicaciones y los hilos de procesamiento de los dispositivos, y de permanecer a la espera de que se salga del mismo pulsando '*Ctrl+C*' para finalizar los hilos.

En él, también se crean dos variables globales muy importantes:

- Variable global *end* : indica el final del programa al resto de los hilos.
- Variable global *initial_time_ref* : indica la referencia de tiempo inicial.

Dispone de un manejador, *handler_controlC*, al que una vez que le llega la señal *SIGINT* (provocada cuando el usuario pulsa '*Ctrl+C*') activa la variable global *end*, que se encarga de finalizar tanto los hilos de comunicaciones y de procesamiento, como el propio programa principal.

Para que la señal *SIGINT* pueda ser tratada por el manejador, se debe modificar la máscara del proceso, la cual se refiere a las señales que pueden o no interrumpir el funcionamiento de un hilo, desbloqueando dicha señal en la misma.

El código del programa principal tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto de la sección crítica, que se estudiará en apartados posteriores.
- Crea el hilo de comunicaciones, el hilo de control, y los hilos de procesamiento de los dispositivos.
- Crea las variables globales, anteriormente comentadas.
- Crea y configura el manejador para la señal de finalización del programa.
- Lanza el programa *main*.
- Crea el objeto de lectura del fichero de configuración y el de la clase que contiene las funciones del programa *main*, cuya única función será la de establecer el tiempo inicial de referencia.

- Configura la señal *SIGINT*, desbloqueándola en la máscara del proceso para que sea tratada por el manejador.
- Comprueba que se ha llamado correctamente al programa: *./ROMEO_HAM [robot id]*.
- Realiza la lectura del fichero de configuración, donde se verá que dispositivos están en uso y cuales no.
- Coloca la referencia de tiempo inicial.
- Lanza el hilo de comunicaciones, el hilo de control, y aquellos hilos correspondientes a los dispositivos que se van a usar.
- Mientras se están ejecutando los hilos, el programa *main* permanece a la espera de que se pulse '*Ctrl+C*' para finalizar el programa.
- Una vez se active la variable *end*, se finalizarán los hilos y terminará el programa principal.

3.1.3. Ficheros principales del HAM

Dentro de la carpeta principal del HAM, se encuentra su fichero de configuración y los ficheros que contienen las clases correspondientes a la lectura del mismo, y al establecimiento de la referencia de tiempo inicial.

A continuación se realiza una descripción general de dichos ficheros.

3.1.3.1. Fichero de configuración del HAM

El fichero de configuración del HAM (*./romeo/ril/HAM/romeoHAM.conf*) contiene información sobre los dispositivos del ROMEO-4R que se van a usar (1) o que no (0). Estos dispositivos serán: GPS, giróscopo, sonares y láser.

Esta información permitirá crear únicamente los hilos correspondientes a los dispositivos en uso, en el programa *main*, evitando así crear hilos de dispositivos que no se usarán y que ocasionarían un consumo de recursos innecesario.

3.1.3.2. Ficheros de lectura del fichero de configuración del HAM

Para poder leer la información correspondiente al fichero de configuración del HAM, existe un fichero de lectura del mismo (*./romeo/ril/HAM/romeoHAM readfileconf.cpp*) que contiene las funciones correspondientes a la clase creada para dicha tarea, llamada *CRomeoHAMReadFileConf*.

La **clase *CRomeoHAMReadFileConf*** está formada por las siguientes funciones:

- **read_HAM_file_conf**: lee y procesa el archivo de configuración del ROMEO-4R en el que se especifican los dispositivos que se van a usar.

  Parámetros:

  - char *name: nombre del fichero de configuración.

  Devuelve:

  - int: *SUCCESS* si no hay errores o *ERROR* en caso contrario.

- **gps_use**: comprueba si se usa el GPS.

  Parámetros:

  - void.

  Devuelve:

  - int: *YES* si se usa el GPS o *NO* en caso contrario.

- **gyro_use**: comprueba si se usa el giróscopo.

  Parámetros:

  - void.

  Devuelve:

  - int: *YES* si se usa el giróscopo o *NO* en caso contrario.

- **sonar_use**: comprueba si se usan los sonares.

  Parámetros:

  - void.

  Devuelve:

  - int: *YES* si se usan los sonares o *NO* en caso contrario.

- **laser_use**: comprueba si se usa el láser.

  Parámetros:

  - void.

Devuelve:

- int: *YES* si se usa el láser o *NO* en caso contrario.

El programa *main* llamará a estas funciones, que indicarán si se activa el hilo del dispositivo correspondiente.

3.1.3.3. Ficheros con funciones para el programa *main*

Es necesario el establecimiento de una referencia de tiempo inicial que permita a los distintos hilos controlar el momento en el cual le llegan los datos correspondientes. Esta referencia se establece al comienzo de cada hilo, y marcará a su vez el tiempo global de uso del dispositivo.

Esto se implementa mediante una clase, llamada *CRomeoHamMainFunctions*, (*./romeo/ril/HAM/romeoHAMmainfunctions.h*) que proporcionará al programa *main* una referencia de tiempo inicial, la cual será almacenada en una variable global de dicho programa, llamada *initial_time_ref*, mediante la función *set_initial_time_ref()* (*./romeo/ril/HAM/romeoHAMmainfunctions.cpp*), de forma que la misma esté disponible para todos los hilos.

La **clase *CRomeoHamMainFunctions*** está formada por la siguiente función:

- **set_initial_time_ref**: establece la referencia de tiempo inicial para los dispositivos del ROMEO-4R.

  Parámetros:

  - double *init_time_ref: variable en la que se almacenará la referencia de tiempo inicial.

  Devuelve:

  - void.

3.1.4. Source

Siguiendo un orden en la arquitectura del ROMEO-4R, existirá siempre una carpeta llamada *Source* dentro de cada módulo. Esta carpeta contendrá los ficheros de comunicaciones, así como los de la sección crítica, además de contener las carpetas correspondientes a los hilos de procesamiento, que en el caso del HAM, serán los dispositivos del ROMEO-4R (*dcx*, *ax5411*, *gps*, *gyro*, *sonar* y *laser*) y el puerto serie.

A continuación se verá, con más detalle, todo el contenido de esta carpeta.

3.1.4.1. Comunicaciones

Las comunicaciones, tal y como se comentó en el Capítulo 2, se realizan empleando el protocolo BBCS (*Black Board Communication System*), el cual gestiona las comunicaciones de un proceso con el resto de procesos situados dentro de la misma capa o en capas adyacentes, incluyendo por tanto que dichos procesos se encuentren en máquinas distintas.

Cuando se establecen las comunicaciones, cada nodo ve al resto de nodos de la misma forma, independientemente de la máquina en la que se ubique o la forma de comunicarse. Como su nombre indica, el protocolo se asemeja a una pizarra, esto es, básicamente, 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. De esta forma, se simplifica en gran medida tanto la transmisión de datos como su estructuración.

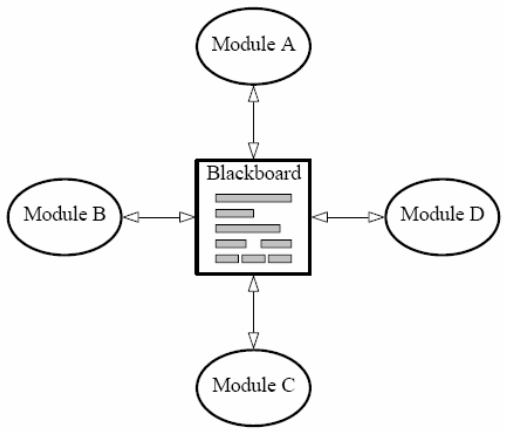


Figura 3.1.4.1-1: Sistema de comunicaciones BBCS

Para una mayor información acerca del protocolo de comunicaciones BBCS que aquí se emplea se recomienda la lectura del documento basado en el proyecto *COMETS* "*Functional Specification of the Real-Time Communications API: Real-time coordination and control of multiple heterogeneous unmanned aerial vehicles*".

En la arquitectura 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. El protocolo de comunicaciones usado, BBCS (*./BBCS*), se podría cambiar de forma sencilla, al comunicarse los hilos a partir de la sección crítica y ser el hilo de comunicaciones totalmente independiente del resto del sistema, exceptuando la clase *CRomeoCommunication_XXX*, donde *XXX* hace referencia al módulo para el cual se implementan las comunicaciones.

Para los módulos con los que actualmente cuenta la RIL, las clases que se usan para las comunicaciones, heredan de la clase *CBBCSFunctions* (*./BBCS/header/ BBCS.h*). Aunque lo que se está indicando es común a todos los módulos, ya que todos implementan sus comunicaciones de la misma forma, este apartado se centrará en los archivos y funciones correspondientes a la clase que se usará en el HAM.

En el Capítulo 2, ya se comentó la topología de las comunicaciones que se ha adoptado en esta arquitectura. Ahora se verán con más detalle los conceptos de conexiones, puertos y slots.

- ***Conexiones*:** el fichero general de conexiones se encuentra en *./robot_architecture/relay_node/NetConnections.conf*, a partir del cual se creará el fichero de conexiones para el HAM (*connectionsHAM_X.conf*, donde *X* indica el identificador del robot correspondiente) utilizando la clase *RN_connections*, que a su vez utiliza la clase *Connections_conf*, la cual contiene las funciones necesarias para escribir una conexión en el formato adecuado a partir de los parámetros de ésta.

- ***Puertos*:** todos los puertos del sistema, tanto locales como remotos, utilizados para las conexiones, se encuentran definidos en el fichero *./robot_architecture/comms/ NetPorts.h*, siendo los específicos del HAM, los indicados en la siguiente tabla:

Nombre del Puerto	Nº del Puerto
HAM_ROMEO_LOCAL_PORT	2800
HAM_ROMEO_REMOTE_PORT	2850

Tabla 3.1.4.1-1: Puertos usados por el Hardware Abstraction Module

- ***Slots*:** los slots se clasifican en seguros (lo importante es que el dato llegue a su destino) o no seguros (lo importante es que el dato sea lo más reciente posible). Así, los datos que marcan la trayectoria de un camino a seguir se corresponderán con el primer caso, mientras que los datos que envían los sensores formarán parte del segundo. Todos los slots necesarios para los módulos del ROMEO-4R vienen definidos en *./romeo/comms/ ROMEO_IMI_slots.h* y se encontrarán dentro del intervalo [4000, 4999], donde el número de cada slot también guardará relación con el identificador del robot. Para el HAM los slots serán los mostrados en el siguiente cuadro, donde la '*i*' hace referencia al identificador (ID) del robot, la columna *Input/Output* indica si son slots de entrada o salida para el módulo en particular y la columna *Secure* indica si es seguro (Sí/1) o si no lo es (No/0):

Nombre del Slot	Nº del Slot	Input / Output	Secure
SPEED_SLOT (i)	4050+ i	Input	No
CURV_SLOT (i)	4100 +i	Input	No
DCX_DATA_SLOT (i)	4150 +i	Output	No
GYRO_DATA_SLOT (i)	4200 +i	Output	No
GPS_DATA_SLOT (i)	4250 +i	Output	No
SONAR_DATA_SLOT (i)	4300 +i	Output	No
LASER_DATA_SLOT (i)	4350 +i	Output	No

Tabla 3.1.4.1-2: Slots usados para las comunicaciones del HAM

De esta forma, el hilo de comunicaciones en este módulo estará formado por los siguientes archivos:

- *romeoHAMcommunication.cpp* y *romeoHAMcommunication.h*: implementan la clase que se encarga de las comunicaciones con el resto de módulos. Esta clase implementa una interfaz que permite abstraer las comunicaciones de su implementación, por lo cual si en el futuro se quiere cambiar a otro protocolo de comunicaciones sólo habría que cambiar la implementación de esta clase, y nada

más, ya que se mantiene la interfaz. En ella, se encuentran funciones de envío de datos (que comprueban si existen datos en la sección crítica para, en caso afirmativo, colocarlos en el slot correspondiente para ser enviados por la red) y de recepción de datos (que comprueban que no exista error en las comunicaciones, para escribir los datos que se encuentran en el slot correspondiente, en la sección crítica).

La **clase *CRomeoCommunication_HAM*** está formada por las siguientes funciones:

- **Init**: inicializa las comunicaciones y añade los slots necesarios. Además, calcula el ancho de banda máximo para cada slot, teniendo en cuenta el ancho de banda máximo del canal y el número de slots antes comentado. Además de dichos slots, existirán dos slots más para las propias comunicaciones internas del BBCS.

  Parámetros:

  - void.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendDcxData**: envía los datos de la DCX al *Romeo Status Module*.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendGpsData**: envía los datos del GPS al *Romeo Status Module*.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendGyroData**: envía los datos del giróscopo al *Romeo Status Module*.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendSonarData**: envía los datos de los sonares al *Romeo Status Module*.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendLaserData**: envía los datos del láser al *Romeo Status Module*.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveRefSpeedForDcx**: recibe la referencia de velocidad del *Path Follower Module*. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveRefCurvForDcx**: recibe la referencia de curvatura del *Path Follower Module*. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_HAM *RomeoHamCS: objeto de la sección crítica correspondiente al HAM.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

Por último se debe indicar que las variables de la clase son inicializadas en el constructor de la misma.

- *romeoHAMcommunicationthread.cpp*: implementa la ejecución del hilo que se encarga de las comunicaciones en el HAM.

  El código del hilo tiene el siguiente orden:

  - Añade las cabeceras necesarias.
  - Crea un objeto de la sección crítica del HAM (se estudiará en el siguiente apartado).
  - Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
  - Lanza el hilo de comunicaciones del HAM.
  - Crea las variables y objetos locales necesarios, tanto para trabajar con el BBCS como para crear el archivo de conexiones correspondiente.
  - Añade los slots.
  - Crea el archivo de conexiones *connectionsHAM_X.conf*, donde *X* indica el identificador del robot correspondiente.
  - Añade la conexión al *Relay Node* del robot, indicando tanto la dirección IP del mismo, como los puertos local y remoto del HAM.
  - Crea las conexiones usando el archivo de parámetros creado anteriormente.
  - Comienza el bucle del que se saldrá cuando se active la variable *end*.
  - Se sincronizan los datos y se espera un período de tiempo.

- Se realiza una prueba del canal de comunicaciones que indica si el canal está activo o inactivo.

- Se reciben las referencias de velocidad y curvatura del *Path Follower Module*.

- Se envían los datos de los dispositivos al *Romeo Status Module*.

- Una vez se sale del bucle, mediante la activación de la variable *end*, se finalizan las comunicaciones del BBCS y se eliminan los canales correspondientes.

Como en la mayoría de los hilos, cualquier error que se produzca en el mismo será indicado por pantalla.

3.1.4.2. Sección crítica

La comunicación entre hilos pertenecientes a un mismo módulo se realiza mediante una sección crítica, o como en este caso de varias secciones críticas, aunque siempre todas englobadas dentro de un objeto de la clase *CRomeoCriticalSection_XXX* (donde *XXX* hace referencia al módulo en que se encuentra), que heredará de forma pública de la clase *CCriticlaSection*. Cada sección crítica implementa un mutex, esto es, un ente software que permite proteger una región de memoria compartida. De esta forma, cuando un hilo hace un *Lock* (lo bloquea) para acceder a una variable de la sección crítica, ningún otro hilo podrá acceder a la misma hasta que el hilo que entró en la sección crítica haga un *Unlock* (lo desbloquea). Esto permite que las variables dentro de dicha región sean siempre coherentes a pesar de que varios hilos tengan acceso a ellas.

Como se vio en el capítulo anterior, cada hilo de procesamiento se comunicará con el hilo de comunicaciones y con los demás hilos de procesamiento (en caso de que proceda) a partir de una sección crítica distinta. Así, la información recibida procedente del resto de módulos de la arquitectura a través del hilo de comunicaciones, es escrita por éste en la sección crítica para que los hilos de procesamiento que necesiten dicha información puedan utilizarla, y viceversa.

La clase que implementa la sección crítica del HAM, llamada *CRomeoCriticalSection_HAM* (*./romeo/ril/HAM/source/romeoHAMcriticalsection.h*), se encargada de las comunicaciones de los hilos de los dispositivos con el hilo de comunicaciones. En ella están las funciones *Set*, que permiten colocar en la sección crítica los datos de los dispositivos, y las funciones *Get*, que permiten obtener dichos datos de la misma, las cuales vienen descritas en *./romeo/ril/HAM/source/ romeoHAMcriticalsection.cpp.*

Por tanto, la **clase *CRomeoCriticalSection_HAM*** estará formada por las siguientes funciones:

- **SetDcxData**: coloca en la sección crítica los datos de la tarjeta DCX.

  Parámetros:

  - DCX_DATA *dcx_data: variable que contiene una estructura de datos de la DCX.

  Devuelve:

  - void.

- **GetDcxData**: obtiene de la sección crítica los datos de la tarjeta DCX.

  Parámetros:

  - DCX_DATA *dcx_data: variable que contiene una estructura de datos de la DCX.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetGpsData**: coloca en la sección crítica los datos del GPS.

  Parámetros:

  - GPS_DATA *gps_data: variable que contiene una estructura de datos del GPS.

  Devuelve:

  - void.

- **GetGpsData**: obtiene de la sección crítica los datos del GPS.

  Parámetros:

  - GPS_DATA *gps_data: variable que contiene una estructura de datos del GPS.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetGyroData**: coloca en la sección crítica los datos del giróscopo.

  Parámetros:

  - GYRO_DATA *gyro_data: variable que contiene una estructura de datos del giróscopo.

Devuelve:

- void.

- **GetGyroData**: obtiene de la sección crítica los datos del giróscopo.

  Parámetros:

  - GYRO_DATA *gyro_data: variable que contiene una estructura de datos del giróscopo.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetSonarData**: coloca en la sección crítica los datos de los sonares.

  Parámetros:

  - SONAR_DATA *sonar_data: variable que contiene una estructura de datos de los sonares.

  Devuelve:

  - void.

- **GetSonarData**: obtiene de la sección crítica los datos de los sonares.

  Parámetros:

  - SONAR_DATA *sonar_data: variable que contiene una estructura de datos de los sonares.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetLaserData**: coloca en la sección crítica los datos del láser.

  Parámetros:

  - LASER_DATA *laser_data: variable que contiene una estructura de datos del láser.

  Devuelve:

  - void.

- **GetLaserData**: obtiene de la sección crítica los datos del láser.

  Parámetros:

  - LASER_DATA *laser_data: variable que contiene una estructura de datos del láser.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetRefSpeedForDcx**: coloca en la sección crítica la referencia de velocidad para la DCX.

  Parámetros:

  - double *ref_speed: variable que contiene la referencia de velocidad para la DCX.

  Devuelve:

  - void.

- **GetRefSpeedForDcx**: obtiene de la sección crítica la referencia de velocidad para la DCX.

  Parámetros:

  - double *ref_speed: variable que contiene la referencia de velocidad para la DCX.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetRefCurvForDcx**: coloca en la sección crítica la referencia de curvatura para la DCX.

  Parámetros:

  - double *ref_curv: variable que contiene la referencia de curvatura para la DCX.

  Devuelve:

  - void.

- **GetRefCurvForDcx**: obtiene de la sección crítica la referencia de curvatura para la DCX.

  Parámetros:

  - double *ref_curv: variable que contiene la referencia de curvatura para la DCX.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

Por último, se debe indicar que las variables de la sección crítica deben ser inicializadas con un valor por defecto para evitar errores de lectura en caso de que los dispositivos a los que corresponden dichas variables no se encuentren en uso.

3.1.4.3. Control

Este hilo se encarga de transmitir las referencias de velocidad y curvatura, procedentes del *Path Follower Module*, a la tarjeta DCX para que mueva el vehículo según dichas referencias.

El código del hilo de control tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
- Crea un objeto de la clase *CRomeoDcx*, que será global y externo debido a su utilización tanto por este hilo como por el de la DCX.
- Lanza el hilo de control del vehículo *ControlThread*.
- Crea un objeto de la clase de los temporizadores *CRomeoTimer* y crea un temporizador con un período de *T_CONTROL*. Los temporizadores serán estudiados en el apartado *"3.5. Temporizadores"*.
- Comienza un bucle del que se saldrá cuando se active la variable *end*.
- Recibe las referencias de curvatura y velocidad procedentes del *Path Follower Module* y se las asigna a la tarjeta DCX.
- Espera el vencimiento del temporizador para una nueva recepción de dichas referencias.
- Una vez se sale del bucle, mediante la activación de la variable *end*, se elimina el temporizador y se termina el hilo.

De esta forma, este hilo se encargará del control del vehículo propiamente dicho.

3.1.4.4. Tarjeta DCX-PC100

La tarjeta *DCX-PC100*, de la compañía *Precision MicroControl Corporation*, es un controlador de movimiento multi-eje modular, que se compone de 1 á 9 placas de circuito ensambladas de forma conjunta para conformar un único dispositivo. La placa principal del circuito puede incluir, a su vez, hasta 8 módulos, y va insertada en una de las ranuras disponibles en el PC.

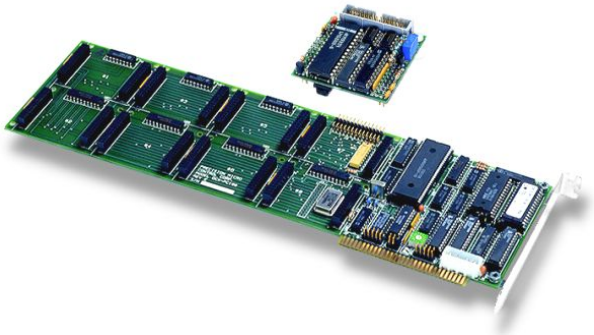


Figura 3.1.4.4-1: Tarjeta DCX-PC100

Figura 3.1.4.4-2: Módulo DCX-MC200

En el ROMEO-4R, esta tarjeta se encarga del control de bajo nivel, siendo el PC el encargado de controlarla, mandando los comandos necesarios y recibiendo las respuestas de los mismos. Para el control de los motores se hace uso de 2 módulos *MC200* que lleva instalados la tarjeta: uno para el control del motor de tracción y otro para el de dirección. También se hace cargo de leer toda la información sensorial necesaria para llevar a cabo el control de estos motores, principalmente la lectura de los codificadores incrementales instalados en cada uno de ellos. Antes de comenzar el control es necesario realizar una configuración previa de los parámetros relacionados con los controladores *PID* que incluye cada módulo *MC200*.

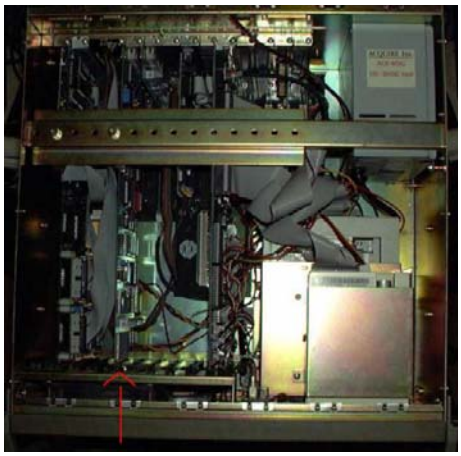


Figura 3.1.4.4-3: Tarjeta DCX-PC100 insertada en el PC del ROMEO-4R

La **clase *DCX*** (definida en *./romeo/ril/HAM/source/dcx/dcx.h*), que se encarga de las tareas más básicas para el manejo de la tarjeta y que será utilizada por la clase *CRomeoDcx*, está formada por las siguientes funciones:

- **open_dcx**: se encarga de abrir el fichero de dispositivo de la tarjeta (*/dev/dcx*) para comenzar la comunicación con la misma.

- **close_dcx**: se encarga de cerrar el fichero de dispositivo de la tarjeta (*/dev/dcx*) para finalizar la comunicación con la misma.

- **load_macro**: carga el fichero de configuración de la DCX, necesario para la inicialización de la tarjeta.

- **write_dcx**: envía un comando a la tarjeta. Gracias a ella, será posible enviar diferentes comandos de manera que la tarjeta realice diversas operaciones de control sobre los motores del ROMEO-4R.

- **read_dcx**: lee la respuesta ante algún comando enviado previamente a la tarjeta. Gracias a ella, será posible estimar diversos parámetros del ROMEO-4R, como velocidad y curvatura, a partir de las medidas obtenidas por los *encoders* instalados en cada motor.

- **send_reset_dcx**: se encarga de hacer un reset a la tarjeta, mediante el envío del comando *"RT"* a la misma.

- **odometria_dcx**: realiza una estimación de la posición y orientación actual a partir de la posición y orientación anterior, las diferencias de tiempo, la velocidad y la curvatura.

- **command_2_code**: función privada de la clase que se encarga de convertir un comando de la DCX (dado como cadena de caracteres) en su correspondiente código numérico.

El fichero que contiene los comandos que se enviarán a la DCX durante la fase de inicialización del programa principal, *./romeo/ril/HAM/source/dcx/inicial.mcr*, tiene la función de configurar adecuadamente la acción de control sobre cada uno de los motores. Así, se configurarán los diversos parámetros de los controladores *PID* de cada motor, la acción que se debe llevar a cabo cuando se alcanza el final de carrera para el caso del motor de dirección, la aceleración y desaceleración del motor de tracción, y la velocidad máxima, entre otros muchos.

La **clase *CRomeoDcx*** (definida en *./romeo/ril/HAM/source/dcx/romeodcx.h*), que hereda los métodos de la clase *DCX*, se encarga de proporcionar la interfaz que se utilizará en el hilo correspondiente a esta tarjeta (*DcxThread*). Esta interfaz está formada por las siguientes funciones:

- **init_dcx**: se encarga de abrir el fichero de dispositivo de la tarjeta (*/dev/dcx*) para comenzar la comunicación con la misma y así proceder con su inicialización.

- **end_dcx**: se encarga de cerrar el fichero de dispositivo de la tarjeta (*/dev/dcx*) para finalizar la comunicación con la misma.

- **set_speed**: establece una consiga de velocidad (en m/s) para el vehículo.

- **set_curv**: establece una consiga de curvatura (en m^{-1}) para el vehículo.

- **get_ref_speed**: devuelve la última consigna de velocidad establecida.

- **get_ref_curv**: devuelve la última consigna de curvatura establecida.

- **centrar_volante_modo_posicion**: se encarga de centrar las ruedas del ROMEO-4R, independientemente del estado inicial en que se encuentren.

- **update_dcx**: se encarga de la actualización de la información procedente de la DCX. Hace una lectura de las magnitudes que es posible leer directamente de la misma para, a partir de ellas, realizar estimaciones de otros parámetros:

    - *Estimación de la velocidad*: se realizan dos medidas consecutivas del *encoder* del motor de tracción, separadas por una cantidad de tiempo (pasada como parámetro), se calcula la diferencia de pulsos medidos en cada caso, se realiza la conversión de pulsos a metros, y dividiendo por el tiempo transcurrido, se obtiene la estimación deseada.

    - *Estimación de la curvatura*: se lee el *encoder* del motor de dirección y se realiza la conversión de pulsos a curvatura.

  Las estimaciones de velocidad y curvatura, junto con la marca de tiempo de las medidas (obtenida mediante la función *get_relative_time()*), se almacenan en una cola circular de estructuras de datos de la DCX (*DCX_DATA*), de forma que se podrá tener acceso tanto a la última estimación realizada como a estimaciones anteriores. Una vez almacenadas las nuevas estimaciones se activa el flag que indica la existencia de nuevos datos procedentes de la DCX (*new_dcx_data_flag*).

- **get_dcx_data**: devuelve una estructura *DCX_DATA*, con estimaciones calculadas a partir de medidas de la DCX. Si el parámetro que se le pasa es un cero (0) devolverá la última estimación, si es un uno (1) devolverá la anterior, y así sucesivamente.

- **new_dcx_data**: se encarga de comprobar la existencia de nuevas estimaciones de la DCX desde la última vez que se llamó a esta función.

- **ReceiveSpeedForDcx**: recibe a través de la sección crítica la referencia de velocidad, procedente del *Path Follower Module*, que debe aplicar a la DCX mediante la función *set_speed()*.

- **ReceiveCurvForDcx**: recibe a través de la sección crítica la referencia de curvatura, procedente del *Path Follower Module*, que debe aplicar a la DCX mediante la función *set_curv()*.

- **SendDcxData**: en caso de que existan nuevas estimaciones de la DCX (que comprueba mediante la función *new_dcx_data()*), coloca en la sección crítica la última estimación obtenida (*get_dcx_data(0)*) para que sea enviada al *Romeo Status Module*.

- **read_encoder**: función privada de la clase que se encarga de la lectura de los *encoders* de los motores de tracción y dirección.

- **fin_carrera_direc**: función privada de la clase que detecta el fin de carrera derecho.

- **curv_a_pulsos**: función privada de la clase que se encarga de la conversión de curvatura a pulsos de *encoder*.

- **pulsos_a_curv**: función privada de la clase que se encarga de la conversión de pulsos de *encoder* a curvatura.

- **get_relative_time**: función privada de la clase que devuelve el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

En el constructor de la clase se realiza la inicialización de las variables de la misma.

Las magnitudes geométricas y constructivas del ROMEO-4R, así como los prototipos de funciones y estructuras de datos referentes a la DCX, se encuentran definidas en *./romeo/ril/comms/param_romeodcx.h.* La estructura para albergar los datos de la DCX es la siguiente:

```
typedef struct
{
  double dcx_speed;
  double dcx_curv;
  double time;

} DCX_DATA;
```

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información de la tarjeta, vienen reflejadas en el Proyecto Fin de Carrera de Rafael Martín de Agar Tirado "*Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control*".

La implementación del hilo que se encarga de manejar la DCX (*DcxThread*), se encuentra en *./romeo/ril/HAM/source/dcx/romeodcxthread.cpp*, y consta del siguiente código:

- Añade las cabeceras necesarias.
- Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
- Crea un objeto de la clase *CRomeoDcx (RomeoDcx)*, que es global y externo debido a que es utilizado tanto por este hilo como por el hilo de control (*ControlThread*).
- Lanza el hilo de la DCX (*DcxThread*).
- Abre el fichero de dispositivo de la tarjeta (*/dev/dcx*) para comenzar la inicialización de la misma. En caso de producirse un error, activa la variable *end*, finalizando así el hilo y el programa principal.
- Comienza el bucle del que se saldrá cuando se active la variable *end*.
- Se realiza la lectura y actualización de las estimaciones de la DCX, mediante la función *update_dcx()*, avisando en caso de producirse un error.
- Se envía la estructura de datos obtenida (*DCX_DATA*) a la red para que la reciba el *Romeo Status Module*.
- Una vez se active la variable *end*, finaliza la comunicación con la DCX, terminando así el hilo.

3.1.4.5. Tarjeta AX5411

La tarjeta *AX5411*, de la compañía *Axiom Technology Co.*, está diseñada para permitir la adquisición de datos a alta velocidad. Se inserta en una de las ranuras disponibles del PC, siendo éste el encargado controlarla, mandando los comandos necesarios y recibiendo las respuestas de los mismos. El convertidor A/D permite velocidades de hasta 60 KHz, velocidad a la que también se realiza la transferencia de datos a memoria mediante DMA. Además la tarjeta permite la configuración de varios parámetros como las ganancias y la escala (que se ha configurado a ±10 V).

Por otra parte la tarjeta posee 16 canales analógicos de entrada con conversores A/D de 12 bits y 2 canales analógicos de salida en forma de tensión, cada uno con su convertidor D/A de 12 bits. Estos canales de salida pueden configurarse de forma independiente en el rango de 0 a 5 V o de 0 a 10 V. La tarjeta también tiene 24 canales de entrada y otros 24 de salida digitales con compatibilidad TTL.

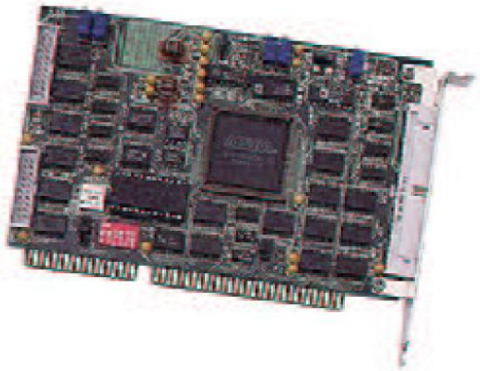


Figura 3.1.4.5-1: Tarjeta AX5411

En el ROMEO-4R, la tarjeta se utiliza para la lectura de varios sensores analógicos, como son los sonares (los cuales también se disparan a través de ella) y el potenciómetro encargado de la detección del ángulo de giro del remolque.

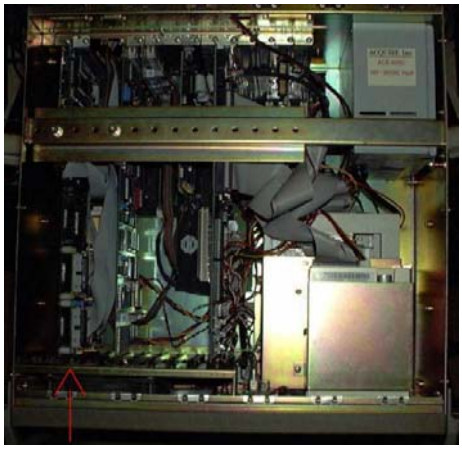


Figura 3.1.4.5-2: Tarjeta AX5411 insertada en el PC del ROMEO-4R

La **clase *AX5411***, definida en *./romeo/ril/HAM/source/ax5411/ax5411.h*, está formada por las siguientes funciones:

- **open_ax5411**: se encarga de abrir el fichero de dispositivo de la tarjeta (*/dev/ax5411*) para comenzar la comunicación con la misma.
- **close_ax5411**: se encarga de cerrar el fichero de dispositivo de la tarjeta (*/dev/ax5411*) para finalizar la comunicación con la misma.
- **read_ax5411**: función privada de la clase que realiza una lectura de todas las entradas digitales y analógicas programadas.
- **write_ax5411**: función privada de la clase que escribe en todas las salidas digitales y analógicas programadas.

- **update_all_input_ax5411**: realiza una lectura de todas las entradas digitales y analógicas programadas llamando a la función *read_ax5411()*. Es necesario realizar este paso previo antes de llamar a las siguientes funciones.

- **read_analog_input_ax5411**: lee una de las entradas analógicas, que se le pasa como parámetro.

- **read_digital_input_ax5411**: lee una de las entradas digitales, que se le pasa como parámetro.

- **write_analog_output_ax5411**: escribe un valor en una de las salidas analógicas, pasándose ambos como parámetros.

- **write_digital_output_ax5411**: escribe un valor en una de las salidas digitales, pasándose ambos como parámetros.

- **set_range_ax5411**: establece el rango de conversión analógico/digital, indicando los canales inicial y final de conversión, pasados como parámetros.

- **set_gain_ax5411**: establece la ganancia de la tarjeta, que se pasa como parámetro.

- **get_relative_time**: devuelve el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*)

Todas las declaraciones de la tarjeta se encuentran en el archivo *./romeo/ril/ HAM/source/ax5411/param_ax5411.h*.

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información de la tarjeta, vienen reflejadas en el Proyecto Fin de Carrera de Rafael Martín de Agar Tirado "*Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control*".

3.1.4.6. Puerto serie

Debido a la existencia de varios periféricos que se comunican con el PC mediante una interfaz serie RS-232, se crea una clase que esté asociada al puerto serie, dando lugar a una interfaz que permite trabajar con este dispositivo.

Por tanto, se ha implementado una clase para el puerto serie, de forma que aquellas clases correspondientes a los dispositivos que se comuniquen mediante interfaz serie, heredarán las funciones de esta clase para facilitar la comunicación.

La **clase *Puerto_serie***, definida en *./romeo/ril/HAM/source/puerto_serie/ serial.h*, está formada por las siguientes funciones:

- **init_puerto_serie**: se encarga de la inicialización del puerto serie.
- **close_puerto_serie**: cierra el puerto serie, restableciendo previamente la configuración inicial que existía antes de la inicialización.
- **leer_caracter_puerto_serie**: lee un único carácter del puerto serie, tanto para el láser, como para el giróscopo y GPS.
- **escribir_caracter_puerto_serie**: escribe un único carácter en el puerto serie, tanto para el láser, como para el giróscopo y GPS.
- **enviar_cadena_puerto_serie**: envía un conjunto de caracteres al puerto serie, tanto para el láser, como para el giróscopo y GPS.
- **recibir_cadena_puerto_serie**: recibe un conjunto de caracteres por el puerto serie, tanto para el láser, como para el giróscopo y GPS.
- **posicionar_principio_puerto_serie**: posiciona al principio del puerto serie.

El constructor de la clase se encarga de asignar la velocidad del puerto serie, por defecto 9.600 baudios, y el puerto que se debe abrir, por defecto */dev/gps*.

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información del puerto serie, vienen reflejadas en el Proyecto Fin de Carrera de Rafael Martín de Agar Tirado "*Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control*".

3.1.4.7. GPS

El sistema de posicionamiento global (*Global Positioning System* o GPS), de la compañía *NovAtel*, consiste en un sistema de navegación por satélite (mediante la red de satélites *NAVSTAR*) que proporciona una cobertura mundial, con información acerca de la posición, velocidad y tiempo.

Figura 3.1.4.7-1: Satélite NAVSTAR

Se ha visto como mediante los codificadores ópticos, el vehículo puede conocer tanto el giro, como el desplazamiento y la velocidad del mismo. Sin embargo, esta forma de medir el desplazamiento tiene el problema de la acumulación de errores, por lo que se hace necesaria la instalación de un sistema GPS.

El sistema GPS está constituido por tres segmentos:

- *Segmento espacial*: está formado por una constelación de 24 satélites, en seis planos separados 55 grados con periodos orbitales de 12 horas a una altura aproximada de 20.200 Km. Esto proporciona a un receptor GPS de 6 a 12 satélites a la vista en cualquier punto de la superficie del planeta en cualquier instante. Cada satélite envía una señal característica que lo identifica, en la que va incluida información sobre su posición, estado, tiempo, parámetros orbitales y otros datos.

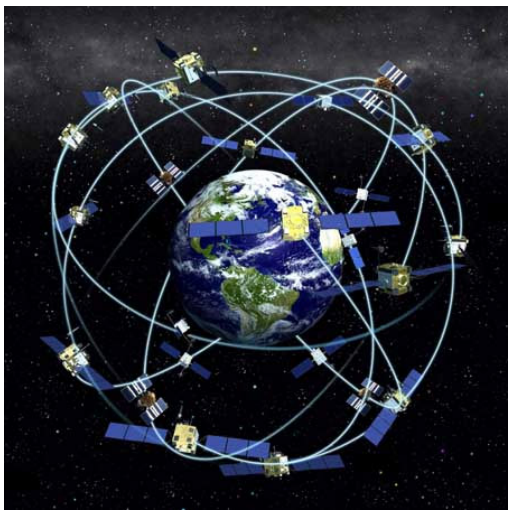


Figura 3.1.4.7-2: Red de satélites NAVSTAR

- *Segmento de control*: está formado por una estación maestra, cinco estaciones monitor que supervisan los satélites gracias a las señales difundidas por éstos (posición, movimiento orbital y temporización) y tres estaciones para el envío de datos a los satélites. Los datos de las estaciones monitor se envían a la estación maestra donde se recalculan parámetros orbitales, y se vuelven a enviar a los satélites a través de las estaciones de envío. Los satélites difunden los nuevos datos recibidos en lugar de los antiguos.

- *Segmento de usuario*: está formado por los equipos receptores de las señales de los satélites. Estos equipos deben procesar simultáneamente las señales de un mínimo de cuatro satélites para obtener medidas precisas.

El receptor GPS mide el tiempo que tarda la señal del satélite en llegar desde éste hasta el receptor. La distancia a la que se halla el equipo del satélite se calcula conociendo el tiempo en que la señal sale del satélite, el tiempo en que es recibida y la velocidad de propagación de la señal. Si el receptor tuviera un reloj perfecto,

sincronizado de forma perfecta con los de los satélites, bastarían tres medidas para determinar las tres coordenadas de la posición. Sin embargo, un reloj de elevada precisión es muy costoso, por lo que se emplea un cuarto satélite para determinar el error en el reloj del receptor o *"bias" B.* El receptor mide el tiempo que tarda en llegar la información GPS de cada uno de los cuatro satélites y obtiene las medidas denominadas *pseudorangos* (PR_1, PR_2, PR_3, PR_4) multiplicando por la velocidad de la luz (c). Así, se plantean cuatro ecuaciones con cuatro incógnitas:

$$(X_i - U_x)^2 + (Y_i - U_y)^2 + (Z_i - U_z)^2 = (PR_i - Bc)^2 \;\; ; \;\; i = 1...4$$

donde (X_i, Y_i, Z_i) son las posiciones de cada uno de los satélites, transmitidas de forma codificada a una frecuencia de 50 Hz por los mismos satélites, y las cuatro incógnitas son (U_x, U_y, U_z, B).

Debido a las numerosas fuentes de error que afectan al GPS, la precisión del sistema está entre 50 y 100 metros. Esta precisión puede aumentarse de forma importante (entre 1 y 5 metros, y hasta pocos centímetros) utilizando un sistema diferencial (DGPS). En el DGPS las ecuaciones de recepción funcionan a pares, constando cada pareja de una estación base y otra remota (siendo posible conectar varias estaciones remotas a una estación base). Para que el funcionamiento en modo diferencial sea efectivo, es necesario que las estaciones base y remota reciban simultáneamente las señales de los mismos satélites, de forma que si estas estaciones están suficientemente próximas (separadas menos de 50 Km) los errores se consideran iguales, eliminándose mediante correcciones diferenciales.

Para conocer en mayor profundidad el sistema GPS se recomienda la lectura del libro *"Robótica: manipuladores y robots móviles"* (Aníbal Ollero), del cual ha sido tomada esta información.

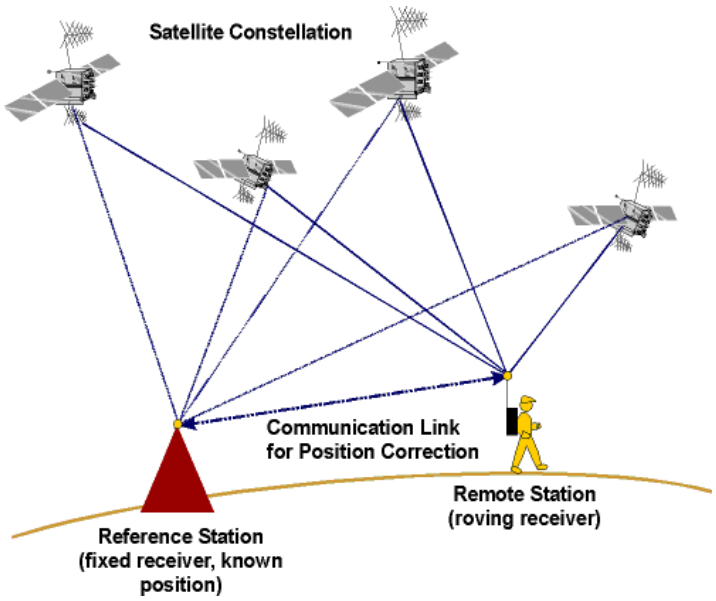


Figura 3.1.4.7-3: Sistema DGPS

Así, el ROMEO-4R cuenta con dos GPS, uno montado en el vehículo (estación remota) y otro montado en una posición conocida (estación base), comunicándose ambos mediante radio-módems, con una velocidad máxima de envío de paquetes de 5 Hz. El GPS se conecta al PC a través del puerto serie, y va instalado en la intersección del techo con la vertical trazada a partir del centro del eje trasero de las ruedas.

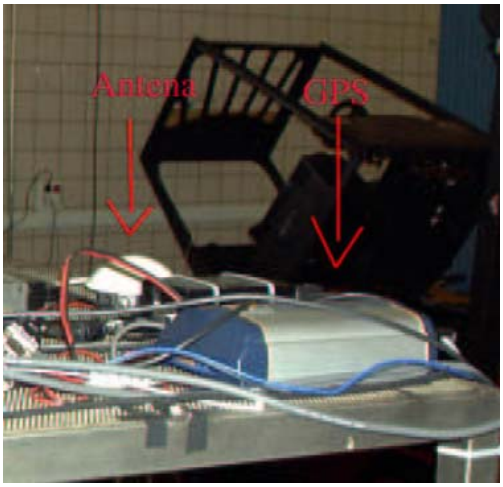


Figura 3.1.4.7-4: Sistema GPS instalado en el ROMEO-4R

La **clase *Gps*** (definida en *./romeo/ril/HAM/source/gps/gps.h*), que se encarga de las tareas más básicas para el manejo del GPS y que será utilizada por la clase *CRomeoGps*, hereda los métodos de la clase *Puerto_serie*, y está formada por las siguientes funciones:

- **init_gps**: se encarga de la inicialización del puerto serie, haciendo uso de los métodos de la clase *Puerto_serie* para configurarlo.

- **start_gps**: inicia el envío de paquetes P20A desde el GPS a la velocidad deseada.

- **stop_gps**: detiene el envío de paquetes desde el GPS.

- **enviar_cadena_gps**: función privada de la clase que se encarga de enviar una cadena de caracteres al GPS.

- **procesar_caracter_gps**: función privada de la clase que se encarga de procesar un mensaje del GPS. Para ello, va analizando uno a uno los caracteres que lo componen, a la vez que va calculando el *checksum*. Una vez recibe el *checksum* del GPS, lo compara con el que ha calculado para comprobar que el mensaje se ha recibido correctamente.

- **char_2_dec**: función privada de la clase que se encarga de la conversión de caracteres hexadecimales a números con base decimal.

- **checksum**: función privada de la clase que calcula el *checksum* de dos bytes mediante una operación *XOR*.

- **hacer_lectura_gps**: realiza la lectura de un mensaje completo del GPS. Para ello, va leyendo los caracteres que se reciben por el puerto serie (*leer_caracter_puerto_serie()*) al que se conecta el GPS, y los va procesando (calculando el *checksum*) mediante la función *procesar_caracter_gps()* para, en caso de que no existan errores, descomponer los campos del mismo, y guardarlos en la estructura correspondiente, mediante la función *procesar_mensaje_p20a()*.

- **procesar_mensaje_p20a**: función privada de la clase que se encarga de procesar un mensaje P20A, guardando cada campo en su lugar correspondiente dentro de la estructura *gps_mensaje_p20a*.

- **get_coord_utm_gps**: devuelve las coordenadas UTM a partir de la latitud y longitud obtenidas en la última medida recibida del GPS.

- **calc_relative_time**: función privada de la clase que devuelve el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

- **get_relative_time**: devuelve el instante de tiempo actual calculado mediante la función *calc_relative_time()*.

- **get_dif_mode_gps**: devuelve el campo *DIF_MODE* (que indica si se usa o no el GPS en modo diferencial) del último mensaje recibido.

- **odometria_gps**: realiza una estimación de la posición y orientación actual a partir de la posición y orientación anterior, las diferencias de tiempo, la velocidad y la curvatura.

El constructor de la clase se encarga de asignar el puerto que se debe abrir, el cual viene indicado en el fichero *./romeo/ril/HAM/source/gps/gps_config.h,* donde se define el nombre del puerto (*GPS_PORT*) en el que se encuentra dispositivo que maneja el GPS, que es */dev/gps*. Esto se hace así para permitir el cambio del puerto sin tener que tratar con el código de la clase.

Las funciones que se encargan de trabajar con las coordenadas UTM vienen definidas en *./romeo/ril/HAM/source/gps/utm.h*, y son las siguientes:

- **LLtoUTM**: convierte las coordenadas dadas por la pareja [Latitud, Longitud] a las coordenadas UTM, formadas por el par [UTMNorthing, UTMEasting].

- **UTMLetterDesignator**: proporciona la letra UTM correcta para una latitud dada, avisando en caso de que la misma se encuentre fuera de los límites UTM.

En el fichero *./romeo/ril/HAM/source/gps/utm_constants.h* aparecen definidas las estructuras y constantes necesarias para trabajar con las coordenadas UTM.

La **clase *CRomeoGps*** (definida en *./romeo/ril/HAM/source/gps/romeogps.h*), que hereda los métodos de la clase *Gps*, se encarga de proporcionar la interfaz que se utilizará en el hilo correspondiente al GPS (*GpsThread*). Esta interfaz está formada por las siguientes funciones:

- **Init_gps**: se encarga de llamar a las funciones *init_gps()* y *start_gps()* para realizar la inicialización del GPS.

- **end_gps**: se encarga de llamar a la función *stop_gps()* para finalizar el uso del GPS.

- **update_gps**: se encarga de la actualización de la información procedente del GPS. Hace una lectura de las magnitudes que es posible leer directamente para, a partir de ellas, realizar la estimación de velocidad y curvatura (mediante las funciones *estimate_gps_speed()* y *estimate_gps_curv()*), las cuales, junto con la marca de tiempo de las medidas, obtenida mediante la función *get_relative_time()*, se almacenan en una cola circular de estructuras de datos del GPS (*GPS_DATA*), de forma que se podrá tener acceso tanto a la última estimación realizada como a estimaciones anteriores. Una vez almacenadas las nuevas estimaciones se activa el flag que indica la existencia de nuevos datos procedentes del GPS (*new_gps_data_flag*).

- **get_gps_data**: devuelve una estructura *GPS_DATA*, con estimaciones calculadas a partir de medidas del GPS. Si el parámetro que se pasa es un cero (0) devolverá la última estimación, si es un uno (1) devolverá la anterior, y así sucesivamente.

- **new_gps_data**: se encarga de comprobar la existencia de nuevas estimaciones del GPS desde la última vez que se llamó a esta función.

- **SendGpsData**: en caso de que existan nuevas estimaciones del GPS (que comprueba mediante la función *new_gps_data()*), coloca en la sección crítica la última estimación obtenida (*get_gps_data(0)*) para que sea enviada al *Romeo Status Module*.

- **estimate_gps_orientation**: función privada de la clase que realiza una estimación de la orientación (en radianes) del vehículo a partir de los datos obtenidos del GPS, calculando el ángulo de la recta que une los dos últimos puntos obtenidos del GPS.

- **estimate_gps_speed**: función privada de la clase que realiza una estimación de la velocidad del vehículo a partir de los datos obtenidos del GPS, es decir, a partir de dos puntos de la trayectoria y los tiempos en que se pasan por cada uno de ellos.

En el constructor de la clase se realiza la inicialización de las variables de la misma.

La estructura para albergar los datos del GPS se encuentra definida en *./romeo/ril/comms/param_romeogps.h*, y es la siguiente:

```
typedef struct
{
  double x_utm;
  double y_utm;
  double time;
  double gps_speed;
  double gps_orientation;
  int dif_mode;
  int valid_data;

} GPS_DATA;
```

donde *valid_data* indica si se usa el GPS (1) o no (0).

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información del GPS, vienen reflejadas en el Proyecto Fin de Carrera de Rafael Martín de Agar Tirado "*Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control*".

La implementación del hilo que se encarga de manejar el GPS (*GpsThread*), se encuentra en *./romeo/ril/HAM/source/gps/romeogpsthread.cpp*, y consta del siguiente código:

- Añade las cabeceras necesarias.
- Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
- Lanza el hilo del GPS (*GpsThread*).
- Crea un objeto de la clase *CRomeoGps (RomeoGps)*.
- Abre el fichero de dispositivo */dev/gps*, para comenzar la inicialización del GPS. En caso de producirse un error, activa la variable *end*, finalizando así el hilo y el programa principal.
- Comienza el bucle del que se saldrá cuando se active la variable *end*.
- Se realiza la lectura y actualización de las estimaciones del GPS, mediante la función *update_gps()*, avisando en caso de producirse un error.
- Se envía la estructura de datos obtenida (*GPS_DATA*) a la red para que la reciba el *Romeo Status Module*.
- Una vez se active la variable *end*, se finaliza la comunicación con el GPS, terminando así el hilo.

3.1.4.8. Giróscopo

El giróscopo *Autogyro Navigator Plus*, de la compañía *KVH Industries*, realiza lecturas de la velocidad angular de rotación de la plataforma sobre la que se instala, que en nuestro caso será el ROMEO-4R, y mediante la integración de las mismas en el tiempo, se puede conseguir el giro del vehículo, y por tanto la orientación del mismo. Se trata de un interferómetro de fibra óptica de un solo eje, adecuado para sistemas de navegación terrestre.

Figura 3.1.4.8-1: Giróscopo Autogyro Navigator Plus

El giróscopo instalado en el ROMEO-4R, al igual que ocurriera con el GPS, se comunica con el PC a través del puerto serie, mediante el protocolo RS-232, a una velocidad de 9.600 baudios, y se encuentra situado en la intersección del techo con la vertical trazada a partir del centro del eje trasero de las ruedas.

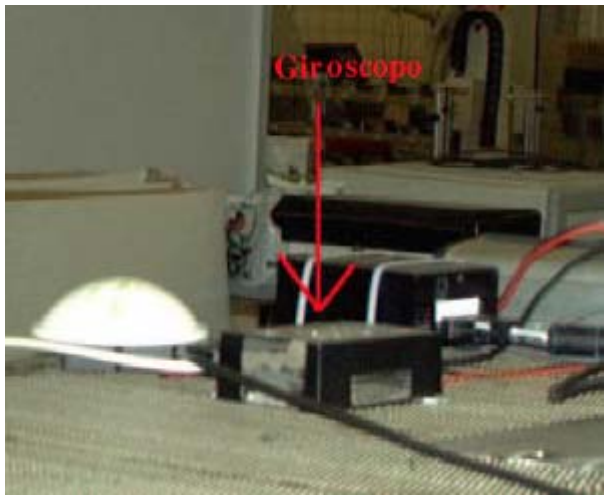


Figura 3.1.4.8-2: Giróscopo instalado en el ROMEO-4R

La **clase *Gyro*** (definida en *./romeo/ril/HAM/source/gyro/gyro.h*), que se encarga de las tareas más básicas para el manejo del giróscopo y que será utilizada por la clase *CRomeoGyro*, hereda los métodos de la clase *Puerto_serie*, y está formada por las siguientes funciones:

- **init_gyro**: se encarga de la inicialización del puerto serie, haciendo uso de los métodos de la clase *Puerto_serie* para configurarlo.

- **procesar_caracter_gyro**: función privada de la clase que se encarga de procesar un mensaje del giróscopo. Para ello, va analizando uno a uno los caracteres que lo componen, a la vez que va calculando el *checksum*.

- **hacer_lectura_gyro**: realiza la lectura de un mensaje completo del giróscopo. Para ello, va leyendo los caracteres que se reciben por el puerto serie (*leer_caracter_puerto_serie()*) al que se conecta el giróscopo, y los va procesando (calculando el *checksum*) mediante la función *procesar_ caracter_gyro()*.

- **get_rate_gyro**: devuelve el valor de la velocidad de giro (*angle_rate*) leído directamente del giróscopo.

- **get_temp_gyro**: devuelve el valor de la temperatura del giróscopo (*temp*) leído directamente de él.

- **get_correcion_temp_gyro**: devuelve el valor corregido de la temperatura del giróscopo. Debido a que el error en las medidas del giróscopo depende mucho de la temperatura de éste, se ha estimado el error mediante la implementación de la recta de regresión obtenida a partir de 250.000 medidas (*angle_rate, temp*) obtenidas con el giróscopo parado durante 4 horas.

- **calc_relative_time**: función privada de la clase que devuelve el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

- **get_relative_time**: devuelve el instante de tiempo actual calculado mediante la función *calc_relative_time()*.

- **odometria_gyro**: realiza una estimación de la posición y orientación actual a partir de la posición y orientación anterior, las diferencias de tiempo, la velocidad y la curvatura.

El constructor de la clase se encarga de asignar el puerto que se debe abrir, el cual viene indicado en el fichero *./romeo/ril/HAM/source/gyro/gyro_config.h*, donde se define el nombre del puerto (*GYRO_PORT*) en el que se encuentra dispositivo que maneja el giróscopo, que es */dev/gyro*. Esto se hace así para permitir el cambio del puerto sin tener que tratar con el código de la clase.

La **clase *CRomeoGyro*** (definida en *./romeo/ril/HAM/source/gyro/romeogyro.h*) que hereda los métodos de la clase *Gyro*, se encarga de proporcionar la interfaz que se utilizará en el hilo correspondiente al giróscopo (*GyroThread*). Esta interfaz está formada por las siguientes funciones:

- **Init_gyro**: se encarga de llamar a la funciones *init_gyro()* para realizar la inicialización del giróscopo.

- **update_gyro**: se encarga de la actualización de la información procedente del giróscopo. Hace una lectura de las magnitudes que es posible leer directamente para, a partir de ellas, realizar la estimación de la curvatura (mediante la función *estimate_gyro_curv()*), además de realizar la oportuna corrección de la temperatura (mediante la función *get_correcion_temp_gyro()*). Estos datos junto con la marca de tiempo de las medidas, obtenida mediante la función *get_relative_time()*, se almacenan en una cola circular de estructuras de datos del giróscopo (*GYRO_DATA*), de forma que se podrá tener acceso tanto a la última estimación realizada como a estimaciones anteriores. Una vez almacenadas las nuevas estimaciones se activa el flag que indica la existencia de nuevos datos procedentes del giróscopo (*new_gyro_data_flag*).

- **get_gyro_data**: devuelve una estructura *GYRO_DATA*, con estimaciones calculadas a partir de medidas del giróscopo. Si el parámetro que se pasa es cero (0) devolverá la última estimación, si es un uno (1) devolverá la anterior, y así sucesivamente.

- **new_gyro_data**: se encarga de comprobar la existencia de nuevas estimaciones del giróscopo desde la última vez que se llamó a esta función.

- **SendGyroData**: en caso de que existan nuevas estimaciones del giróscopo (que comprueba mediante la función *new_gyro_data()*), coloca en la sección crítica la última estimación obtenida (*get_gyro_data(0)*) para que sea enviada al *Romeo Status Module*.

- **estimate_gyro_orientation**: función privada de la clase que realiza una estimación de la orientación del vehículo a partir de la orientación anterior, con su respectiva marca de tiempo, y de la nueva medida de la velocidad de giro, también con su respectiva marca de tiempo.

En el constructor de la clase se realiza la inicialización de las variables de la misma.

La estructura para albergar los datos del giróscopo se encuentra definida en *./romeo/ril/comms/param_romeogyro.h*, y es la siguiente:

```
typedef struct
{
  double angle_rate;
  double temp;
  double time;
  double gyro_orientation;
  int valid_data;

} GYRO_DATA;
```

donde *valid_data* indica si se usa el giróscopo (1) o no (0).

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información del giróscopo, vienen reflejadas en el Proyecto Fin de Carrera de Rafael Martín de Agar Tirado "*Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control*".

La implementación del hilo que se encarga de manejar el giróscopo (*GyroThread*), se encuentra en *./romeo/ril/HAM/source/gyro/romeogyrothread.cpp,* y consta del siguiente código:

- Añade las cabeceras necesarias.
- Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
- Lanza el hilo del giróscopo (*GyroThread*).
- Crea un objeto de la clase *CRomeoGyro (RomeoGyro)*.
- Abre el fichero de dispositivo, */dev/gyro*, para comenzar la inicialización del giróscopo. En caso de producirse un error, activa la variable *end*, finalizando así el hilo y el programa principal.
- Comienza el bucle del que se saldrá cuando se active la variable *end*.
- Se realiza la lectura y actualización de las estimaciones del giróscopo, mediante la función *update_gyro()*, avisando en caso de producirse un error.
- Se envía la estructura de datos obtenida (*GYRO_DATA*) a la red para que la reciba el *Romeo Status Module*.
- Hace una pausa de 30 milisegundos que permita que le lleguen nuevos datos del giróscopo por el puerto serie (se reciben nuevos datos cada 0.1 segundos), evitando así un uso excesivo de CPU. Esto no ha sido necesario en los hilos correspondientes al GPS y DCX, debido a que en dichos hilos, las funciones *update_gps()* y *update_dcx()* realizaban esta operación en su interior.
- Una vez se active la variable *end*, se finaliza la comunicación con el giróscopo, terminando así el hilo.

3.1.4.9. Sónar

Los sonares *BERO*, de la compañía *Siemens*, son sensores de proximidad basados en ultrasonidos, que son exactamente igual a los sonidos que el ser humano oye normalmente, salvo que tienen una frecuencia mayor (40 KHz) que la audible por el oído humano (16 - 20 KHz).

Figura 3.1.4.9-1: Sonar BERO de Siemens

El funcionamiento básico de los sensores de ultrasonidos como medidores de distancia se muestra de una manera muy clara en la siguiente figura, donde se tiene un transmisor que emite un pulso de ultrasonido el cual rebota sobre un determinado objeto, de forma que la reflexión de ese pulso es detectada por un receptor, el cual la amplificará para compensar los efectos de atenuación del sonido en el aire.

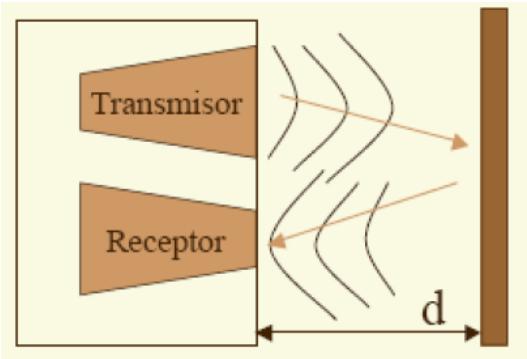


Figura 3.1.4.9-2: Funcionamiento básico de un sonar

La mayoría de los sensores de ultrasonido se basan en la emisión de un pulso de ultrasonido cuyo lóbulo, o campo de acción, es de forma cónica. Midiendo el tiempo que transcurre entre la emisión del sonido y la percepción del eco, se puede establecer la distancia a la que se encuentra el obstáculo que ha producido la reflexión de la onda sonora, mediante la fórmula:

$$d = \frac{1}{2} V \cdot t$$

donde V es la velocidad del sonido en el aire y t es el tiempo transcurrido entre la emisión y la recepción del pulso.

A pesar de que su funcionamiento parece muy sencillo, existen factores que influyen de una forma determinante en las medidas realizadas. Por tanto, es necesario un conocimiento de las diversas fuentes de incertidumbre que afectan a las medidas para poder tratarlas de forma adecuada, minimizando su efecto en el conocimiento del entorno que se desea adquirir. Entre los diversos factores que alteran las lecturas que se realizan con los sensores de ultrasonido cabe destacar:

- El campo de actuación del pulso que se emite desde un transductor de ultrasonido tiene forma cónica. El eco que se recibe como respuesta a la reflexión del sonido indica la presencia del objeto más cercano que se encuentra dentro del cono acústico, y no especifica en ningún momento la localización angular del mismo. Aunque la máxima probabilidad es que el objeto detectado esté sobre el eje central del cono acústico, la probabilidad de que el eco se haya producido por un objeto presente en la periferia del eje central no es en absoluto despreciable y ha de ser tenida en cuenta y tratada convenientemente.

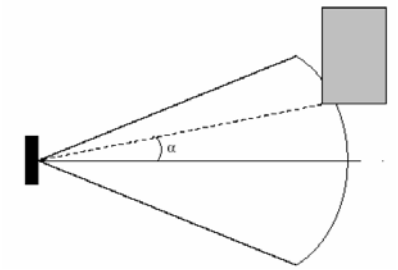


Figura 3.1.4.9-3: Incertidumbre angular en la medida de un ultrasonido

- La cantidad de energía acústica reflejada por el obstáculo depende en gran medida de la estructura de su superficie.

- En los sensores de ultrasonido más comunes (como es el caso de los sonares *BERO*) se utiliza el mismo transductor como emisor y receptor. Tras la emisión del ultrasonido se espera un determinado tiempo a que las vibraciones en el sensor desaparezcan y esté preparado para recibir el eco producido por el obstáculo. Esto implica que existe una distancia mínima *d*, proporcional al tiempo de relajación del transductor, a partir de la cual el sensor mide con precisión. Por lo general, todos los objetos que se encuentren por debajo de esta distancia, *d*, serán interpretados por el sistema como que están a una distancia igual a la distancia mínima. Por otra parte, el sonar tiene que esperar un tiempo máximo para enviar al transductor un nuevo tren de ondas. Este tiempo se corresponde con el tiempo que tardaría la onda ultrasónica en llegar a la distancia máxima de detección y volver al transductor. Sin embargo, en el caso de que se produzca detección, no es necesario esperar ningún tiempo máximo, procediéndose con el disparo de un nuevo tren de ondas.

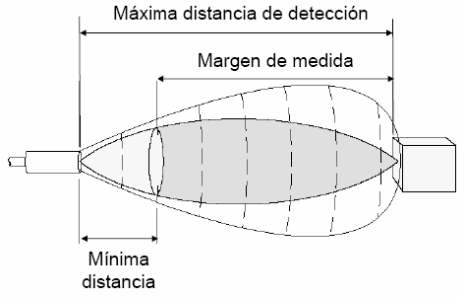


Figura 3.1.4.9-4: Distancias medibles por los sonares

- Los factores ambientales tienen una gran repercusión sobre las medidas. Si las ondas de ultrasonido se mueven por un medio material como el aire, la densidad de éste depende de la temperatura, influyendo este factor sobre la velocidad de propagación de la onda según la expresión:

$$V_S = V_{SO}\sqrt{1+\frac{T}{273}}$$

siendo *Vso* la velocidad de propagación de la onda sonora a 0 ºC, y *T* la temperatura absoluta (grados Kelvin).

- Un factor de error muy común es el conocido como falsos ecos. Estos falsos ecos se pueden producir por razones diferentes. Puede darse el caso en que la onda emitida por el transductor se refleje varias veces en diversas superficies antes de que vuelva a incidir en el transductor (si es que incide). Este fenómeno, conocido como reflexiones múltiples, implica que la lectura del sensor evidencia la presencia de un obstáculo a una distancia proporcional al tiempo transcurrido en el viaje de la onda, es decir, una distancia mucho mayor de la que se encuentra en realidad el obstáculo más cercano, que pudo provocar la primera reflexión de la onda. Otra fuente más común de falsos ecos, conocida como *crosstalk*, se produce cuando se emplea un cinturón de ultrasonidos donde una serie de sensores están trabajando al mismo tiempo. En este caso puede ocurrir (y ocurre con una frecuencia relativamente alta) que un sensor emita un pulso y sea recibido por otro sensor que estuviese esperando el eco del pulso que él había enviado con anterioridad (o viceversa).

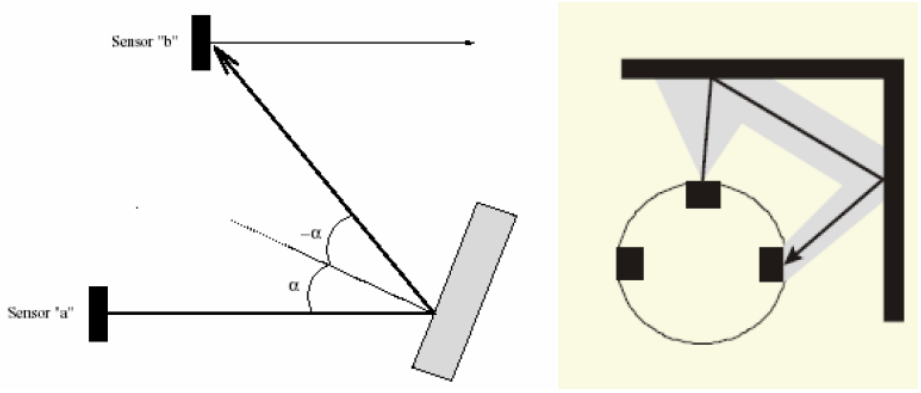


Figura 3.1.4.9-5: Diferentes casos de falsos ecos

- Las ondas de ultrasonido obedecen a las leyes de reflexión de las ondas, por lo que una onda de ultrasonido tiene el mismo ángulo de incidencia y reflexión respecto a la normal a la superficie. Esto implica que si la orientación relativa de la superficie reflectora con respecto al eje del sensor de ultrasonido es mayor que un cierto umbral, el sensor nunca reciba el pulso de sonido que emitió.

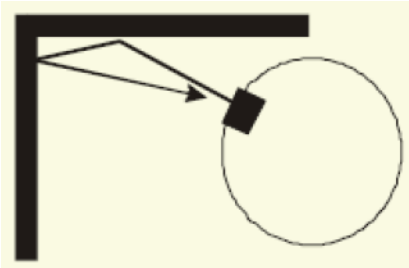


Figura 3.1.4.9-6: Problema en la reflexión de las ondas

Los sonares instalados en ROMEO-4R son analógicos, y se clasifican en dos tipos, de corto alcance (menor tamaño) y largo alcance (mayor tamaño):

Tipo de sonar analógico	Nº de sonares	Zona de detección
CORTO ALCANCE	4	40 – 300 cm
LARGO ALCANCE	6	60 – 600 cm

Tabla 3.1.4.9-1: Tipos de sonar usados en el ROMEO-4R

La disposición y numeración de los 10 sonares instalados en el ROMEO-4R es la siguiente:

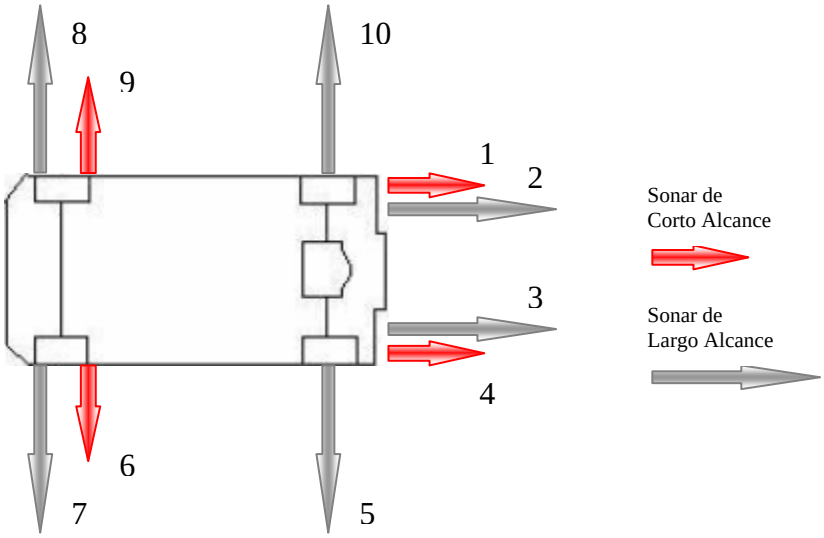


Figura 3.1.4.9-7: Ubicación y numeración de los sonares en el ROMEO-4R

Figura 3.1.4.9-8: Sonares instalados en la parte delantera del ROMEO-4R

El fabricante de los sonares, *Siemens*, proporciona un software (*SONPROG*) para la programación de los sonares *BERO*, a través del puerto RS-232 del ordenador, que permite comprobar y variar todos los parámetros de éstos.

Como se ha comentado antes, uno de los problemas que se tienen a la hora de utilizar los sonares son las interferencias mutuas que se producen cuando se disparan a la vez sonares próximos entre sí, por lo que para corregir este problema es necesario algún tipo de mecanismo de sincronismo que evite el disparo simultáneo de dichos sonares. Dos métodos basados en el hardware de éstos son los siguientes:

1. Mediante una opción que traen los sonares *BERO*, que tras programarlos y conectar las señales de sincronismo de éstos, se consigue que al activar el grupo de sonares, éstos se disparen de forma secuencial, y por lo tanto disminuya la interferencia mutua.

2. Mediante la caja de los sonares *BERO*, se pueden sincronizar los sonares en cuatro grupos a partir de la conexión o desconexión de una serie de *jumpers* que se encuentran en dicha caja.

No obstante, estos dos métodos tienen el problema de que son poco flexibles y dinámicos, es decir, que no se pueden cambiar las estrategias de sincronización en tiempo real, y además para realizarlo hay que reprogramar los sonares y cambiar su cableado o cambiar los *jumpers* de la caja de los sonares. Por lo tanto con estos métodos sólo se pueden realizar estrategias de sincronismo estáticas.

De esta forma, para la obtención de dicha flexibilidad y dinamismo se realiza la sincronización por software, dividiendo los sonares en grupos, donde aquellos que pertenezcan a un mismo grupo se activan de forma simultánea mientras que el resto permanecen desactivados. Después de transcurrir el tiempo necesario para la detección de un objeto en el peor de los casos, se desactivan los sonares de ese grupo y se continúa con el siguiente.

La forma más simple de crear inicialmente los grupos de sincronización, se basa en el uso del fichero de configuración *./romeo/ril/HAM/source/sonar/sonar.conf*, cuya parte explicativa se muestra a continuación:

```
/*********************************************************************

Ejemplo del fichero "sonar.conf":

4 0         //Número de grupos de sonares
1 4 7 0     //Sonares que forman cada grupo
2 5 8 0
3 6 9 0
10 0
0

Como se puede ver en la primera línea se indica el número de grupos
que va haber en total.

En las siguientes líneas se especifican los sonares que se quieren
activar en cada turno (o grupo), y siempre se debe terminar la línea
con el número 0.

Además cuando se han especificado todos los grupos, se debe poner un
único 0 en la última línea.

Si se quiere que todos los sonares se activen a la vez, es decir, como
si sólo hubiera un único turno (o grupo), se debe poner en el número
de grupos un 0, de esta forma se ejecutará el código que había antes
(para mantener compatibilidades). También se puede poner un único
grupo con todos los sonares en ese grupo, pero eso no asegura que los
algoritmos anteriores a esta modificación vayan a funcionar bien.

AVISO: no se puede poner un número mayor de grupos que el número de
sonares especificado en la variable NUM_SONAR.

*********************************************************************/
```

Mediante este fichero de configuración se consigue la inicialización de los grupos de sincronización sin escribir código alguno, lo cual permite configurar los grupos de los sonares sin necesidad de conocer el código.

Posteriormente, se pueden modificar dichos grupos utilizando las funciones implementadas en la clase que se detalla a continuación.

La **clase *CRomeoSonar*** (*./romeo/ril/HAM/source/sonar/romeosonar.h*) que hereda los métodos de la clase *AX5411*, se encarga de proporcionar la interfaz que se utilizará en el hilo correspondiente a los sonares (*SonarThread*). Esta interfaz está formada por las siguientes funciones:

- **Init_sonar**: se encarga de la inicialización de los sonares. Para ello, llama a las funciones correspondientes de la clase *AX5411*, abriendo el fichero de dispositivo la de la tarjeta (*/dev/ax5411*) para comenzar la comunicación con la misma, establece el rango de entradas para la conversión A/D, y establece la ganancia, para luego desactivar todos los sonares y realizar la lectura del archivo de configuración de los mismos.

- **finaliza_sonares**: se encarga de desactivar todos los sonares, llamando a la función *deact_sonar()*.

- **update_sonar**: se encarga de la actualización de la información procedente de los sonares. Estos datos junto con la marca de tiempo de las medidas, obtenida mediante la función *get_relative_time()*, se almacenan en una cola circular de estructuras de datos de los sonares (*SONAR_DATA*), de forma que se podrá tener acceso tanto a la última estimación realizada como a estimaciones anteriores. Una vez almacenadas las nuevas estimaciones se activa el flag que indica la existencia de nuevos datos procedentes de los sonares (*new_sonar_data_flag*).

- **get_sonar_data**: devuelve una estructura *SONAR_DATA*. Si el parámetro que se pasa es cero (0) devolverá la última estimación, si es un uno (1) devolverá la anterior, y así sucesivamente.

- **new_sonar_data**: se encarga de comprobar la existencia de nuevas estimaciones de los sonares desde la última vez que se llamó a esta función.

- **SendSonarData**: en caso de que existan nuevas estimaciones de los sonares (que comprueba mediante la función *new_sonar_data()*), coloca en la sección crítica la última estimación obtenida (*get_sonar_data(0)*) para que sea enviada al *Romeo Status Module*.

- **act_sonar**: función privada de la clase que se encarga de activar un sonar, poniendo la señal de disparo del mismo a nivel alto.

- **deact_sonar**: función privada de la clase que se encarga de desactivar un sonar, poniendo la señal de disparo del mismo a nivel bajo.

- **read_sonar**: función privada de la clase que devuelve el valor obtenido (en metros) de un sonar.

- **get_sonar_time_groups**: función privada de la clase que devuelve el tiempo de adquisición de datos asociado a un grupo. El primer grupo es el número 1.

- **copy_cadena_sonares**: función privada de la clase que copia, a partir de la estructura donde se encuentran los datos obtenidos del fichero de configuración, los distintos grupos de sincronización en la tabla *cadena_sonares*. En el caso de que los grupos o sonares no estén dentro de un rango válido, o que el número de grupos no coincida con el especificado, se finalizará el programa y se sacará por pantalla el mensaje de error correspondiente. .

- **erase_all_groups**: función privada de la clase que borra todos los grupos de sincronización de los sonares. Hay que tener en cuenta que si después de ejecutar esta función no se crea ningún grupo, todos los sonares se activarán a la vez (como si se pusiera un cero (0) en el fichero de configuración).

- **erase_group**: función privada de la clase que borra el grupo de sincronización especificado. El primer grupo es el número 1.

- **create_group**: función privada de la clase que crea un grupo de sincronización, el cual es añadido al final.

- **put_sonar_in_group**: función privada de la clase que añade un sonar al final del grupo de sincronización especificado. El primer grupo es el número 1.

- **erase_sonar_group**: función privada de la clase que elimina un sonar del grupo de sincronización especificado. El primer grupo es el número 1.

- **read_sonar_file_conf**: función privada de la clase que carga el fichero de configuración de los sonares.

En el constructor de la clase se realiza la inicialización de las variables de la misma.

La estructura para albergar los datos de los sonares se encuentra definida en *./romeo/ril/comms/param_romeosonar.h*, y es la siguiente:

```
typedef struct
{
  double sonar[NUM_SONAR];
  double beta_remolque;
  double time;
  double time_groups [NUM_SONAR];

} SONAR_DATA


typedef struct
{
  int active_sonars[NUM_SONAR+1][NUM_SONAR+1];
  int num_groups;

} CONF_SONAR_DATA;
```

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información de los sonares, vienen reflejadas en el Proyecto Fin de Carrera de Antidio Viguria Jiménez "*Sistema de sensores de ultrasonidos para navegación autónoma*".

La implementación del hilo que se encarga de manejar los sonares (*SonarThread*), se encuentra en *./romeo/ril/HAM/source/sonar/romeosonarthread.cpp*, y consta del siguiente código:

- Añade las cabeceras necesarias.

- Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
- Lanza el hilo de los sonares (*SonarThread*).
- Crea un objeto de la clase *CRomeoSonar (RomeoSonar)*.
- Se realiza la inicialización de los sonares, llamando a la función *Init_sonar()*. En caso de producirse un error, activa la variable *end*, finalizando así el hilo y el programa principal.
- Comienza el bucle del que se saldrá cuando se active la variable *end*.
- Se realiza la lectura y actualización de las estimaciones de los sonares, mediante la función *update_sonar()*, avisando en caso de producirse un error.
- Se envía la estructura de datos obtenida (*SONAR_DATA*) a la red para que la reciba el *Romeo Status Module*.
- Una vez se active la variable *end*, se desactivan los sonares, finalizando así el hilo.

3.1.4.10. Láser

El láser *LMS-220*, de la compañía *SICK Optic Electronic*, es un láser escáner bidimensional (*2D*) de medida de distancias de no-contacto. Es un láser bidimensional que mide distancias en el plano horizontal de hasta 150 metros, realizando un barrido de derecha a izquierda en un rango de 100º ó 180º, con un ángulo de resolución de 0.25º, 0.5º ó 1º.

Figura 3.1.4.10-1: Láser LMS-220

Como se ha comentado, el láser realiza las medidas en el sentido contrario a las agujas del reloj, por lo que el sistema de referencia que se le ha asociado al mismo es el siguiente:

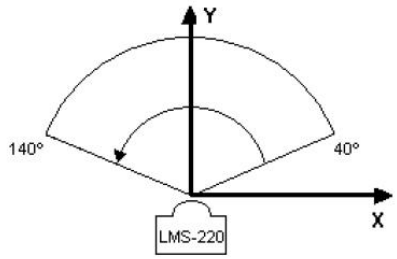


Figura 3.1.4.10-2: Sistema de Referencia para el Láser LMS-220

El láser *LMS-220* instalado en el ROMEO-4R, al igual que ocurriera con el giróscopo y el GPS, se comunica con el PC a través del puerto serie, mediante el protocolo RS-232, a una velocidad configurable, que por defecto es de 9.600 baudios, y se encuentra situado en la vertical trazada a partir del centro del eje delantero de las ruedas.

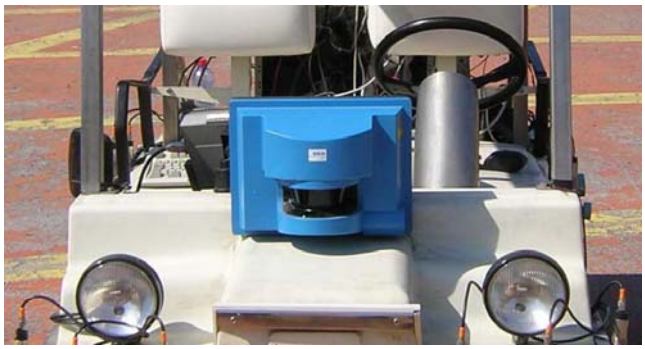


Figura 3.1.4.10-3: Láser LMS-220 instalado en el ROMEO-4R

La comunicación con el láser se lleva a cabo a través de alguno de los puertos series del PC mediante el envío de *telegramas*. Cada uno de estos telegramas tendrá lo que se llamará un *comando*, que indica al láser lo que debe hacer y al PC lo que hay en el campo de datos del telegrama. Los comandos se encuentran definidos en el archivo *./romeo/ril/HAM/source/laser/comandos.h*.

La **clase *CTlgramb*** (definida en *./romeo/ril/HAM/source/laser/telegrama_txrx/clasebas.h*), contiene las funciones para la creación y acceso a los telegramas, y es utilizada como base por la clase *CTlgramtxrx*. Cada telegrama contiene una serie de campos (*STX, ADR, Length, CMD, datos, byte de estado y CRC*), y es esta clase la encargada de introducir en el telegrama dichos campos, y de leerlos en caso de ser necesario. Los telegramas son almacenados en una estructura *T_Trama*, teniendo esta clase una estructura de este tipo llamada *telegram*, cuya composición es la siguiente:

```
typedef struct T_Trama
  {

       unsigned char Dir;     //Dirección
       unsigned int Long;     //Longitud= Información + Datos
       unsigned char Comando; //Comando
       unsigned char * Pqt;   //Paquete= Datos que Transporta la Trama
       unsigned int Crc_16;   //Código corrector de error

   } telegram;
```

Las funciones que componen esta clase son las siguientes:

- **ReserMem**: función privada de la clase que reserva memoria para el campo *Pqt* de la estructura *telegram*. Si no existe espacio suficiente de memoria, el programa finaliza llamando a la función *ErrorMem()*.

- **ErrorMem**: función privada de la clase que imprime por pantalla un mensaje de error cuando no hay suficiente para memoria para el campo *Pqt* en *telegram*.

- **ConstruirCrc**: se encarga de construir el código de redundancia cíclico (*CRC*). Para la construcción del *CRC* de 16 bits se tienen en cuenta los bytes de dirección y también los datos (*Pqt*).

- **AsignarCampos**: se encarga de asignar los campos al telegrama que se va a enviar. Esta función crea completamente el telegrama, estando el campo de la longitud formado por la suma de los bytes de *Pqt*, más 1 byte con el comando.

- **InsertData**: introduce un dato de tipo *unsigned char* (1 byte) en *telegram.Pqt*, en la posición indicada por parámetro. En caso de que la posición indicada (índice) sea negativa, no se introduce nada.

- **InsertDataInt**: introduce un dato de tipo *unsigned int* (2 bytes) en *telegram.Pqt*, en la posición indicada por parámetro.

- **CargarComando**: inserta un comando en el telegrama.

- **LeerComando**: lee un comando del telegrama.

- **LeerCrc**: lee el código de redundancia cíclico (*CRC*) del telegrama.

- **CargarLong**: inserta la longitud (2 bytes) en el telegrama.

- **CargarDir**: inserta la dirección (1 byte) en el telegrama.

- **LeerDato**: lee un dato (1 byte) del *telegram.Pqt* situado en la posición que se le pasa por parámetro.

- **LeerDatoInt**: lee un dato tipo *unsigned int* (2 bytes) del *telegram.Pqt* situado en la posición que se le pasa por parámetro.

- **CargarDat**: introduce un dato de tipo *unsigned char* (1 byte) en *telegram.Pqt*, en la posición indicada por parámetro..

- **LeerLongitud**: lee la longitud del telegrama.

- **CargarCrc**: inserta el código de redundancia cíclico *CRC* (2 bytes) en el telegrama.

La **clase *CTlgramtxrx*** (definida en *./romeo/ril/HAM/source/laser/telegrama_txrx/clasetxrx.h*), que se encarga de la transmisión y recepción de los telegramas entre el PC y el láser, y que será utilizada por la clase *Laser*, hereda los métodos de la clase *Puerto_serie*, para poder usar las funciones correspondientes al envío y recepción de caracteres y cadenas a través de él. Además, contiene dos variables públicas de la clase *CTlgramb* que permiten el almacenamiento y tratamiento de los telegramas de transmisión (*tlgmtx*) y recepción (*tlgmrx*). Las funciones que componen esta clase son las siguientes:

- **EnviarDtg**: función que se encarga de enviar una cadena de datos al puerto serie. Introduce dichos datos en una variable (*Buffer_Tx*) de esta clase, tipo *T_Buffer*, para posteriormente enviarlos almacenados en el telegrama de transmisión (*tlgmtx*).

- **Verificar**: función privada de la clase que comprueba la existencia de errores en la trama serie recibida, indicándo el tipo de error en caso de existir. Recorre el telegrama recibido comprobando que los datos son correctos, es decir, que el orden de llegada es el adecuado, además de introducirlos en *BufferRx*. Los posibles errores se enumeran a continuación:

Errores	Valor	Definición
E_Correcto	0	La última operación se realizo con éxito
E_Fisico	1	Error al recibir un dato (nivel Físico)
E_Ocupado	2	Maquina de Transmisión Ocupada
E_Formato	3	La Trama tiene un formato "incorrecto"
E_Ruptura	4	Parte de la Trama llega fuera de tiempo
E_Medio	5	Destino desconectado o Medio "Roto"
E_Verificacion	6	Suma de Verificación "Equivocada"
E_Secuencia	7	Numero de Orden de la Trama "Equivocado"
E_Protocolo	8	Se "viola" alguna norma de procedimiento
E_Negativo	9	El Destinatario recibió "Mal" mi mensaje

Tabla 3.1.4.10-1: Tipos de errores en el nivel de enlace e inferiores

- **RecibirDtg**: función que se encarga de la recepción de una cadena de datos procedente del puerto serie. Recibe dichos datos y los mete en un *buffer* de 1024 posiciones (*Buffer_aux*), de forma que una vez compruebe que todo es correcto y los meta en otro *buffer* idéntico (*Buffer_Rx*), mediante la función *Verificar()*, los introduzca en el telegrama de recepción, comprobando también que el *CRC* es correcto. Al principio de esta función se espera un tiempo necesario para que el láser pueda procesar el telegrama que se le ha enviado mediante la función *EnviarDtg()*. Si no se espera dicho tiempo, la comunicación no se puede llevar a cabo.

La **clase *Laser*** (definida en *./romeo/ril/HAM/source/laser/laser.h*), que se encarga de las tareas más básicas para el manejo del láser y que será utilizada por la clase *CRomeoLaser*, hereda los métodos de la clase *CTlgramtxrx* (la cual, a su vez, hereda los métodos de la clase *Puerto_serie)*, y está formada por las siguientes funciones:

- **LsrInit_and_Reset**: función privada de la clase que realiza el reseteo software del escáner. Su efecto es el mismo que el de un reseteo hardware.

- **LsrChange_Modo**: función privada de la clase empleada para cambiar el modo de funcionamiento del láser.

- **LsrObtener_Medidas**: función privada de la clase que se encarga de la obtención de medidas del láser.

- **LsrObt_Medidas_Medias**: función privada de la clase que devuelve el valor medio de las medidas en cada punto.

- **LsrPet_Estado**: función privada de la clase que devuelve información sobre el funcionamiento actual del escáner (baudios, ángulo de exploración, resolución angular, modo de funcionamiento, campos definidos, posibilidad de sensor defectuoso, etc.) y datos sobre el código de producción, versión del software, etc.

- **LsrPet_Error**: función privada de la clase que proporciona el error que se ha producido.

- **LsrPet_Type**: función privada de la clase que devuelve la identificación del producto. Cuando termina, en *telegram.Pqt* se encuentra disponible la información (velocidad, ángulo de exploración y resolución angular).

- **LsrChange_Variant**: función privada de la clase que se emplea para cambiar el ángulo de exploración y la resolución angular.

- **LsrRead_Config**: función privada de la clase que lee las unidades de medida usadas, el ángulo de resolución, el ángulo de exploración, etc.

- **LsrSet_Config**: función privada de la clase que pone una nueva configuración al láser.

- **LsrPet_Campos**: función privada de la clase que pide información sobre los campos programados en el escáner.

- **LsrGrupoAct_Campos**: función privada de la clase que solicita la activación de un determinado grupo de campos.

- **LsrConfig_Campos**: función privada de la clase que configura un campo del escáner (rectangular, semicircular o recto).

- **Cambiar_Modo_Laser**: función privada de la clase que cambia el modo de funcionamiento del láser, asegurándose de que así se hace. Puede ser que *LsrChange_Mode()* no de error (responda correctamente), pero aún así no cambie el modo de funcionamiento. Esta función se encarga de asegurar que cambia el modo.

- **Cambiar_Variables_Laser**: función privada de la clase que cambia las variables del láser, asegurándose de que así se hace. Puede ser que *LsrChange_Variant()* no de error (responda correctamente), pero aún así no cambien las variables de ángulo de resolución y escaneo. Esta función se encarga de asegurar que cambian las variables.

- **Obtener_Velocidad_Comunicacion_Laser**: función privada de la clase que devuelve la velocidad de comunicación del láser.

- **calc_time_mark_laser**: función privada de la clase que devuelve el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

- **init_laser**: se encarga de la inicialización del láser. Abre el puerto serie y configura el láser de modo que la comunicación se pueda llevar a cabo correctamente. Nada más entrar, el láser está configurado por defecto a 9.600 baudios, pero si posteriormente se cambia la velocidad, o si da algún error en la inicialización, no se sabrá a que velocidad está. De ahí que exista una variable *vel_com_laser* que se actualiza cada vez que se cambie de velocidad el funcionamiento del láser.

- **start_laser**: se encarga de comenzar la comunicación con el láser. Abre el puerto serie para comenzar la comunicación con el láser y así poder realizar más medidas. El láser ya está configurado y funcionando a una velocidad concreta. Si da algún error, se aborta el programa y se cierra el puerto serie.

- **stop_laser**: corta la comunicación con el láser. Cierra el puerto serie para terminar la comunicación con el láser y que no se puedan realizar más medidas.

- **Obtener_Unidades_Medida_Laser**: devuelve las unidades de medida.

- **Comprobar_Estado_Laser**: comprueba el estado del láser.

- **Obtener_Modo_Laser**: devuelve el modo de funcionamiento del láser.

- **Obtener_Medidas_Laser**: devuelve las medidas del láser (en metros).

- **get_time_mark_laser**: devuelve el instante de tiempo actual calculado mediante la función *calc_time_mark_laser()*.

- **Obtener_Tipo_Laser:** obtiene el tipo del láser. Devuelve en la cadena *tipo* (de 30 elementos) el tipo del láser, que viene a ser algo como esto: `"LMS210;20203;V01.01a".`

- **Obtener_Configuracion_Angulo_Laser:** obtiene el ángulo de exploración y el ángulo de resolución del láser.

- **Leer_Configuracion_Angulo_Laser:** lee el ángulo de exploración y el ángulo de resolución del láser.

- **Leer_Velocidad_Comunicacion_Laser:** lee la velocidad de comunicación del láser.

- **Obtener_Campos_Programados:** lee los campos programados en el láser y sus respectivos datos. Escribe por pantalla el mensaje recibido.

El constructor de la clase se encarga de asignar el puerto que se debe abrir, y la velocidad de transmisión del láser, el cual viene indicado en el fichero *./romeo/ril/HAM/ source/laser/laser_config.h*, donde se define el nombre del puerto (*LASER_PORT*) en el que se encuentra el dispositivo que maneja el láser, que es */dev/laser*, y la velocidad de transmisión por defecto, que es de 9.600 baudios. Esto se hace así para permitir el cambio del puerto sin tener que tratar con el código de la clase.

Para la realización de los distintos cálculos sobre el conjunto de puntos obtenidos por el láser, se encuentra la carpeta *./romeo/ril/HAM/source/laser/ calculos*, la cual contiene los archivos con las funciones correspondientes (*calculos.cpp* y *calculos.h*), mientras que en la carpeta *./romeo/ril/HAM/source/laser/matematicas* se encuentran los ficheros (*matematicas.cpp* y *matematicas.h*) que contienen las funciones matemáticas utilizadas por los archivos anteriores.

En la carpeta *./romeo/ril/HAM/source/laser/listas_puntos* están ubicados los ficheros que contienen las funciones necesarias para el manejo de listas de puntos (*listas_puntos.cpp* y *listas_puntos.h*), y con el fichero *dectipo_puntos.h*, que contiene los tipos de las variables que se usan en las listas de puntos:

```
typedef struct coordenada
  {
        double x,y,radio,angulo;

  } tipo_coordenada;

typedef struct conj_punto
 {
        struct conj_punto *ant;
        struct conj_punto *sig;
        tipo_coordenada punto;
        int angulo_y_resolucion;
        int analizado;

} tipo_conj_punto;
```

donde la variable *angulo_y_resolucion* (que refleja el ángulo de exploración o barrido, en grados, y la resolución, que indica cada cuantos grados se toma una medida) puede tomar los siguientes valores:

Valor de la variable *angulo_y_resolucion*	**Ángulo (grados)**	**Resolución (grados)**
1	100	1
2	100	0.5
3	100	0.25
4	180	0.5

Tabla 3.1.4.10-2: Valores de la variable angulo_y_resolucion

La **clase *CRomeoLaser*** (definida en *./romeo/ril/HAM/source/laser/romeolaser.h*) que hereda los métodos de la clase *Laser*, se encarga de proporcionar la interfaz que se utilizará en el hilo correspondiente al láser (*LaserThread*). Esta interfaz está formada por las siguientes funciones:

- **Init_laser**: se encarga de llamar a la función *init_laser()* para realizar la inicialización del láser.

- **end_gps**: se encarga de llamar a la función *stop_laser()* para finalizar el uso del laser.

- **update_laser**: se encarga de la actualización de la información procedente del láser. Estos datos junto con la marca de tiempo de las medidas, obtenida mediante la función *get_time_mark_laser()*, se almacenan en una cola circular de estructuras de datos del láser (*LASER_DATA*), de forma que se podrá tener acceso tanto a la última estimación realizada como a estimaciones anteriores. Una vez almacenadas las nuevas estimaciones se activa el flag que indica la existencia de nuevos datos procedentes del láser (*new_laser_data_flag*).

- **get_laser_data**: devuelve una estructura *LASER_DATA*. Si el parámetro que se pasa es cero (0) devolverá la última estimación, si es un uno (1) devolverá la anterior, y así sucesivamente.

- **new_laser_data**: se encarga de comprobar la existencia de nuevas estimaciones del láser desde la última vez que se llamó a esta función.

- **modify_laser_data**: modifica uno de los datos de la tabla *laser_data*, donde están almacenadas las medidas del láser, sustituyéndola por otra que se le pasa como parámetro. Si el parámetro que se pasa es cero (0) modificará la última estimación, si es un uno (1) modificará, y así sucesivamente.

- **SendLaserData**: en caso de que existan nuevas estimaciones del láser (que comprueba mediante la función *new_laser_data()*), coloca en la sección crítica la última estimación obtenida (*get_laser_data(0)*) para que sea enviada a la red.

- **Filtrado_ruido_laser**: función privada de la clase que se encarga del filtrado de ruido del láser. Se le pasa una lista de puntos y devuelve la misma lista de puntos modificada.

En el constructor de la clase se realiza la inicialización de las variables de la misma.

La estructura para albergar los datos del láser se encuentra definida en *./romeo/ril/comms/param_romeolaser.h*, y es la siguiente:

```
typedef struct
{
        tipo_conj_punto *C;
        double time;

} LASER_DATA;
```

Tanto la definición de la clase y sus funciones correspondientes, como una mayor información del láser, vienen reflejadas en el Proyecto Fin de Carrera de David Gómez Esteban "*Seguimiento de objetos móviles y evitación de obstáculos en ROMEO-4R*".

La implementación del hilo que se encarga de manejar el láser (*LaserThread*), se encuentra en *./romeo/ril/HAM/source/laser/romeolaserthread.cpp*, y consta del siguiente código:

- Añade las cabeceras necesarias.
- Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
- Lanza el hilo del láser (*LaserThread*).
- Crea un objeto de la clase *CRomeoLaser (RomeoLaser)*.
- Se realiza la inicialización del láser, llamando a la función *Init_laser()*. En caso de producirse un error, activa la variable *end*, finalizando así el hilo y el programa principal.
- Comienza el bucle del que se saldrá cuando se active la variable *end*.
- Se realiza la lectura y actualización de las estimaciones del láser, mediante la función *update_laser()*, avisando en caso de producirse un error.
- Se envía la estructura de datos obtenida (*LASER_DATA*) a la red.
- Una vez se active la variable *end*, se finaliza la comunicación con el láser, terminando así el hilo.

3.2. Romeo Status Module

El *Romeo Status Module* es el módulo encargado de calcular el estado del ROMEO-4R, recibiendo la información procedente de los sensores del HAM y enviando la misma a la MML y al *Path Follower Module*.

La información del *Romeo Status Module* es enviada a la MML para que su módulo correspondiente (*Robot Status Module*) transforme el estado particular del ROMEO-4R a un estado estándar común para todos los robots, el cual es transmitido por el sistema de comunicaciones al resto del sistema. Además, la información de este módulo también es enviada al *Path Follower Module* para que calcule, a partir de ella, la referencia de velocidad y curvatura que posteriormente enviará al HAM para cerrar así el bucle de control.

Como ya ocurriera en el módulo anterior (HAM), cuenta con su sistema de comunicaciones BBCS y con una sección crítica.

Para la compilación de este módulo, existe el fichero *Makefile* (*./romeo/ril/ romeo_status_module/Makefile*), que contiene una lista de todos los archivos objeto del *Romeo Status Module*, así como una relación de dependencias de éstos con los archivos fuente, de manera que se puede ahorrar tiempo de compilación gracias a la utilización del comando *make*.

Esto proporcionará un programa ejecutable, al que se llamará mediante el siguiente comando:

>> ./ROMEO_RSM [robot id]

donde *[robot id]* es el identificador del robot con el que se está trabajando

Este módulo interactúa por una parte con el *Hardware Abstraction Module*, teniendo como entrada la información procedente de los dispositivos, y por otra parte envía a la red el estado del ROMEO-4R, en su formato particular, para que sea utilizado por los módulos que lo necesiten (*Path Follower Module* y *Robot Status Module* en la MML).

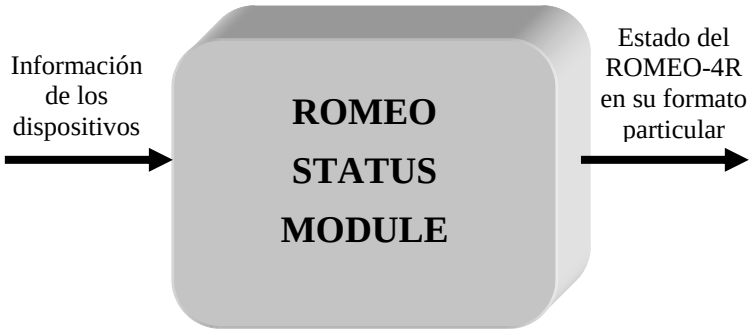


Figura 3.2-1: Entradas y salidas del Romeo Status Module

La estructura del *Romeo Status Module* viene definida por el siguiente esquema:

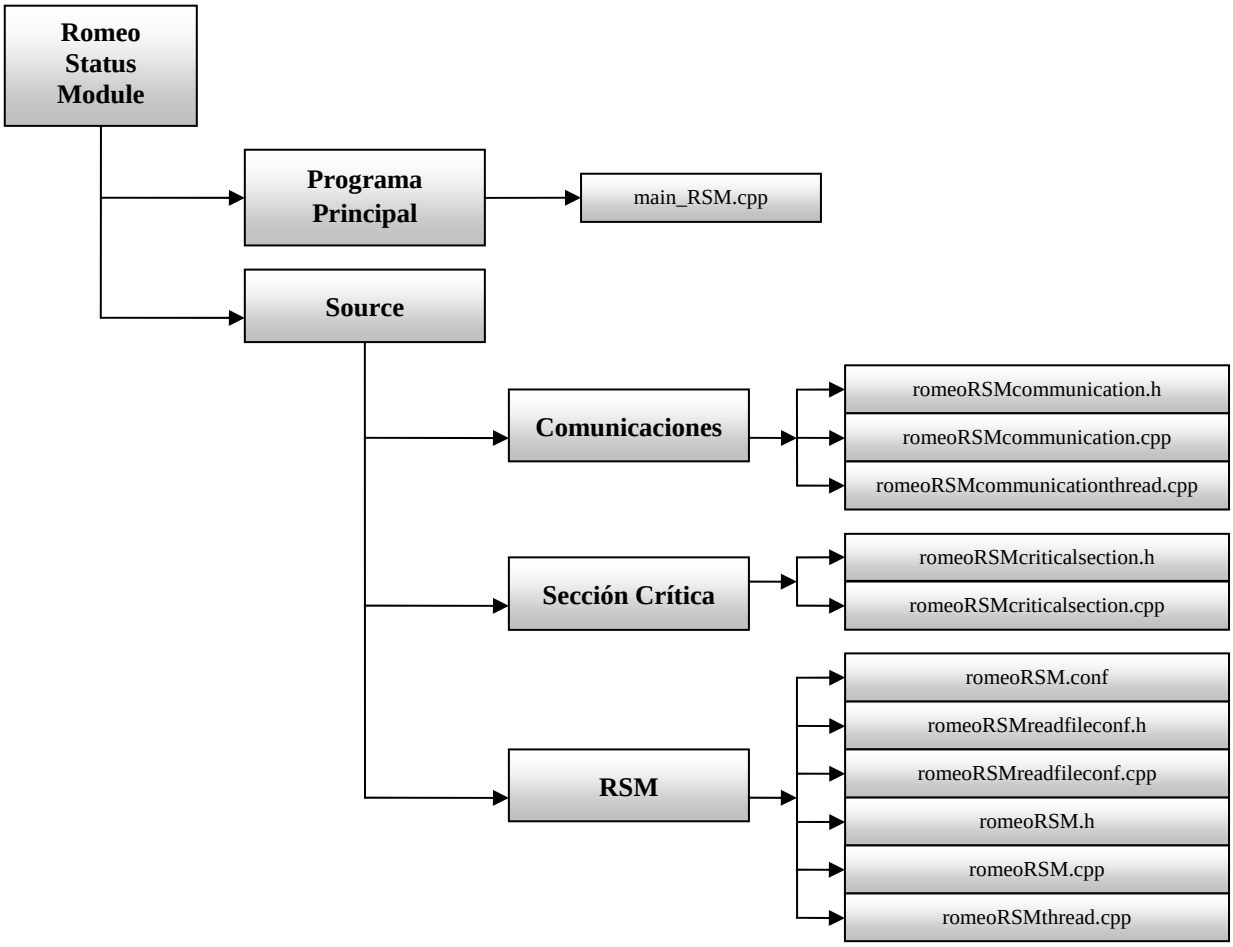


Figura 3.2-2: Esquema del Romeo Status Module

A continuación se describirán con detalle todos los elementos que aparecen en el anterior esquema y que forman parte del *Romeo Status Module*.

3.2.1. Programa principal

El programa principal de ejecución del *Romeo Status Module* (*./romeo/ril/ romeo_status_module/main_RSM.cpp*) es el encargado, básicamente, de lanzar el hilo de comunicaciones y el hilo de procesamiento, y de permanecer a la espera de que se salga del mismo pulsando '*Ctrl+C*' para finalizar los hilos.

Dispone de un manejador, *handler_controlC*, al que una vez que le llega la señal *SIGINT* (provocada cuando el usuario pulsa '*Ctrl+C*') activa la variable global *end*, que se encarga de finalizar tanto los hilos de comunicaciones y de procesamiento, como el propio programa principal.

El código del programa principal tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto de la sección crítica.
- Crea el hilo de comunicaciones (*CommunicationThread_RSM*) y el de procesamiento (*RSMThread*).
- Crea la variable global *end*.
- Crea el manejador para la señal de finalización del programa.
- Lanza el programa *main*.
- Configura la señal *SIGINT*, desbloqueándola en la máscara del proceso para que sea tratada por el manejador.
- Comprueba que se ha llamado correctamente al programa: *./ROMEO_RSM [robot id]*.
- Lanza el hilo de comunicaciones y el de procesamiento.
- Mientras se están ejecutando los hilos, el programa *main* permanece a la espera de que se pulse '*Ctrl+C*' para finalizar el programa.
- Una vez se active la variable *end*, se finalizarán los hilos y después terminará el programa principal.

3.2.2. Source

Como es sabido, esta carpeta contiene tanto los ficheros de comunicaciones como los de la sección crítica, además de contener la carpeta correspondiente al hilo de procesamiento, que en el caso del *Romeo Status Module*, es el RSM.

En los próximos apartados se mostrará con más detalle todo el contenido de esta carpeta.

3.2.2.1. Comunicaciones

Las comunicaciones ya fueron explicadas en el apartado correspondiente al HAM, por lo que aquí se procederá a particularizar las mismas para el caso del *Romeo Status Module*.

Se describirán, a continuación, los conceptos correspondientes a las conexiones, puertos y slots para este módulo:

- ***Conexiones*:** el fichero general de conexiones se encuentra en *./robot_architecture/relay_node/NetConnections.conf,* a partir del cual se creará el fichero de conexiones para el RSM (*connectionsRSM_X.conf,* donde *X* indica el identificador del robot correspondiente) utilizando la clase *RN_connections* que a su vez utiliza la clase *Connections_conf* que tiene funciones para escribir una conexión en el formato adecuado a partir de los parámetros de ésta.

- ***Puertos*:** todos los puertos del sistema, tanto locales como remotos, utilizados para las conexiones, se encuentran definidos en el fichero *./robot_architecture/comms/ NetPorts.h*, siendo los específicos del RSM los indicados en la siguiente tabla:

Nombre del Puerto	Nº del Puerto
ROMEO_STATUS_MODULE_LOCAL_PORT	3000
ROMEO_STATUS_MODULE_REMOTE_PORT	3050

Tabla 3.2.2.1-1: Puertos usados por el Romeo Status Module

- ***Slots*:** los slots del RSM serán los mostrados en el siguiente cuadro, donde la '*i*' hace referencia al identificador (ID) del robot, la columna *Input/Output* indica si son slots de entrada o salida para el módulo en particular y la columna *Secure* indica si es seguro (Sí/1) o si no lo es (No/0):

Nombre del Slot	Nº del Slot	Input / Output	Secure
DCX_DATA_SLOT (i)	4150 +i	Input	No
GYRO_DATA_SLOT (i)	4200 +i	Input	No
GPS_DATA_SLOT (i)	4250 +i	Input	No
SONAR_DATA_SLOT (i)	4300 +i	Input	No
ROMEO_SLOT_SENSORS_STATE (i)	4400 +i	Output	No

Tabla 3.2.2.1-2: Slots usados para las comunicaciones del Romeo Status Module

De esta forma, el hilo de comunicaciones en este módulo está formado por los siguientes archivos:

- *romeoRSMcommunication.cpp* y *romeoRSMcommunication.h*: implementan la clase que se encarga de las comunicaciones con el resto de módulos. En ella, se encuentran funciones de envío de datos (que comprueban si existen datos en la sección crítica para, en caso afirmativo, colocarlos en el slot correspondiente para ser enviados por la red) y de recepción de datos (que comprueban que no exista error en las comunicaciones, para escribir los datos que se encuentran en el slot correspondiente, en la sección crítica).

La **clase *CRomeoCommunication_RSM*** estará formada por las siguientes funciones:

- **Init**: inicializa las comunicaciones y añade los slots necesarios. Además, calcula el ancho de banda máximo para cada slot, teniendo en cuenta el ancho de banda máximo del canal y el número de slots antes comentado.

Además de dichos slots, existirán dos slots más para las propias comunicaciones internas del BBCS.

Parámetros:

- void.

Devuelve:

- bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveDcxData**: recibe los datos de la DCX del HAM. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_RSM *RomeoRsmCS: objeto de la sección crítica correspondiente al *Romeo Status Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveGpsData**: recibe los datos del GPS del HAM. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_RSM *RomeoRsmCS: objeto de la sección crítica correspondiente al *Romeo Status Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveGyroData**: recibe los datos del giróscopo del HAM. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_RSM *RomeoRsmCS: objeto de la sección crítica correspondiente al *Romeo Status Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveSonarData**: recibe los datos de los sonares del HAM. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_RSM *RomeoRsmCS: objeto de la sección crítica correspondiente al *Romeo Status Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendRobotStateRSM**: envía el estado del ROMEO-4R, en su formato particular, a la red.

  Parámetros:

  - CRomeoCriticalSection_RSM *RomeoRsmCS: objeto de la sección crítica correspondiente al *Romeo Status Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

La única variable que aparece en esta clase, y que aún no se ha visto en apartados anteriores, es la correspondiente al estado del ROMEO-4R, *ROMEO_TELEMETRY*, donde se almacenan en una estructura los siguientes valores enviados por los dispositivos del HAM:

```
typedef struct
{
      double      speed;
      double      curv;
      double      time_speed_curv;

      double      x;
      double      y;
      double      time_x_y;

      double      orient;
      double      dorient;
      double      time_orient;

      double      beta_remolque;
      double      time_beta;

      double      global_time;

}ROMEO_TELEMETRY;
```

En ella se recogen los valores que envían los dispositivos del HAM, tales como la velocidad y curvatura enviados por la tarjeta DCX, las coordenadas UTM [x, y] enviadas por el GPS, la orientación y su derivada (velocidad angular) enviadas por el giróscopo, el ángulo del remolque (en caso de que exista), y los tiempos en que se tomaron dichos datos correspondientes a cada dispositivo, así como el tiempo en que se almacena el estado del ROMEO-4R.

Por último se debe indicar que las variables de la clase son inicializadas en el constructor de la misma.

- *romeoRSMcommunicationthread.cpp*: implementa la ejecución del hilo que se encarga de las comunicaciones en el *Romeo Status Module*.

  El código del hilo tiene el siguiente orden:

  - Añade las cabeceras necesarias.
  - Crea un objeto de la sección crítica.
  - Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
  - Lanza el hilo de comunicaciones del *Romeo Status Module* (*CommunicationThread_RSM*).
  - Crea las variables y objetos locales necesarios, tanto para trabajar con el BBCS como para crear el archivo de conexiones correspondiente.
  - Añade los slots.
  - Crea el archivo de conexiones *connectionsRSM_X.conf*, donde *X* indica el identificador del robot correspondiente.
  - Añade la conexión al *Relay Node* del robot, indicando tanto la dirección IP del mismo, como los puertos local y remoto del *Romeo Status Module*.
  - Crea las conexiones usando el archivo de parámetros creado anteriormente.
  - Comienza el bucle del que se saldrá cuando se active la variable *end*.
  - Se sincronizan los datos y se espera un período de tiempo.
  - Se realiza una prueba del canal de comunicaciones que indica si el canal está activo o inactivo.
  - Se reciben los datos de los dispositivos provenientes del *Hardware Abstraction Module*.

- Se envía el estado del ROMEO-4R a la red.

- Una vez se sale del bucle, mediante la activación de la variable *end*, se finalizan las comunicaciones del BBCS y se eliminan los canales correspondientes.

Cualquier error que se produzca en el mismo será indicado por pantalla.

3.2.2.2. Sección crítica

Como ya se ha visto, la información recibida por el resto de módulos de la arquitectura, a través del hilo de comunicaciones, es escrita por éste en la sección crítica para que los hilos de procesamiento que necesiten dicha información puedan utilizarla, y viceversa.

La clase que implementa la sección crítica en el *Romeo Status Module*, llamada *CRomeoCriticalSection_RSM* y que se encuentra en *./romeo/ril/romeo_status_module /source/romeoRSMcriticalsection.h*, es la encargada de las comunicaciones del hilo de procesamiento (*RSMThread*) con el hilo de comunicaciones (*CommunicationThread_ RSM*). En ella están las funciones *Set*, que permiten colocar en la sección crítica los datos de los dispositivos del HAM y el estado del ROMEO-4R, y las funciones *Get*, que permiten obtener dichos datos de la misma, las cuales vienen descritas en *./romeo/ril/ romeo_status_module/source/romeoRSMcriticalsection.cpp.*

La **clase *CRomeoCriticalSection_RSM*** está formada por las siguientes funciones:

- **SetDcxData**: coloca en la sección crítica los datos de la tarjeta DCX.

  Parámetros:

  - DCX_DATA *dcx_data: variable que contiene una estructura de datos de la DCX.

  Devuelve:

  - void.

- **GetDcxData**: obtiene de la sección crítica los datos de la tarjeta DCX.

  Parámetros:

  - DCX_DATA *dcx_data: variable que contiene una estructura de datos de la DCX.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetGpsData**: coloca en la sección crítica los datos del GPS.

  Parámetros:

  - GPS_DATA *gps_data: variable que contiene una estructura de datos del GPS.

  Devuelve:

  - void.

- **GetGpsData**: obtiene de la sección crítica los datos del GPS.

  Parámetros:

  - GPS_DATA *gps_data: variable que contiene una estructura de datos del GPS.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetGyroData**: coloca en la sección crítica los datos del giróscopo.

  Parámetros:

  - GYRO_DATA *gyro_data: variable que contiene una estructura de datos del giróscopo.

  Devuelve:

  - void.

- **GetGyroData**: obtiene de la sección crítica los datos del giróscopo.

  Parámetros:

  - GYRO_DATA *gyro_data: variable que contiene una estructura de datos del giróscopo.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetSonarData**: coloca en la sección crítica los datos de los sonares.

  Parámetros:

  - SONAR_DATA *sonar_data: variable que contiene una estructura de datos de los sonares.

Devuelve:

- void.

- **GetSonarData**: obtiene de la sección crítica los datos de los sonares.

  Parámetros:

  - SONAR_DATA *sonar_data: variable que contiene una estructura de datos de los sonares.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetRobotState**: coloca en la sección crítica el estado del ROMEO-4R.

  Parámetros:

  - ROMEO_TELEMETRY *robot_state: variable que contiene una estructura de datos del estado del ROMEO-4R.

  Devuelve:

  - void.

- **GetRobotState**: obtiene de la sección crítica el estado del ROMEO-4R.

  Parámetros:

  - ROMEO_TELEMETRY *robot_state: variable que contiene una estructura de datos del estado del ROMEO-4R.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

Al igual que se indicó en la sección crítica del HAM, las variables de la sección crítica del *Romeo Status Module* deben ser inicializadas con un valor por defecto para evitar errores de lectura en caso de que los dispositivos a los que corresponden dichas variables no se encuentren en uso.

3.2.2.3. RSM

Este hilo se encarga de calcular el estado del ROMEO-4R, recibiendo la información procedente de los dispositivos del HAM y enviando la misma a la MML y al *Path Follower Module*, realizando, a su vez, la monitorización de dichos dispositivos.

Los archivos que implementan lo anteriormente dicho son los siguientes:

- *romeoRSM.cpp* y *romeoRSM.h*: implementan la clase que se encarga de obtener el estado del ROMEO-4R, así como de la monitorización de los datos procedentes de los distintos dispositivos del HAM.

La **clase *CRomeoRSM*** está formada por las siguientes funciones:

- **update_romeo_state**: se encarga de convertir los datos de los dispositivos que le llegan del HAM en el estado del robot en su formato particular, actualizando el mismo y guardándolo en una cola circular. Además, realiza la monitorización de aquellos dispositivos cuyos datos se desea almacenar para su posterior estudio (mediante la función *monitorizar()*), e imprime por pantalla todos los datos de cada dispositivo, así como el estado del ROMEO-4R. Es necesario resaltar la inclusión de una parte de la función, que se encuentra inactiva de momento, que se encarga de establecer las reglas necesarias para la configuración del estado del ROMEO-4R en función de los dispositivos que estén disponibles. Existe para ello una variable *valid_data* dentro de cada estructura de datos de los dispositivos que indicará si el dispositivo en cuestión está en uso (*valid_data=1*) o si no está siendo utilizado (*valid_data=0*), en cuyo caso se rellenarán las variables del estado del ROMEO-4R con los valores apropiados de los dispositivos en uso, siguiendo las reglas que se detallen en esta función.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **get_romeo_state**: devuelve la estructura de datos del ROMEO-4R indicada por el índice que se le pasa como parámetro. Si el índice es 0, devuelve la última estructura de datos rellenada (la más reciente), si es 1 devuelve la anterior, y así sucesivamente.

  Parámetros:

  - int index: índice del array.

  Devuelve:

  - ROMEO_TELEMETRY *: puntero a estructura de datos del estado correspondiente.

- **monitorizar**: función privada de la clase que se encarga de introducir en el fichero de monitorización (*.log*) los datos procedentes de los dispositivos que se quieren monitorizar, empleando para ello un *buffer* de escritura. Los ficheros *.log* son almacenados en la carpeta *log* (*./romeo/ril/romeo_status_module /log*).

  Parámetros:

  - void.

  Devuelve:

  - void.

- **monitor_activation**: crea el fichero de monitorización (*.log*), asignándole un nombre formado por la fecha y hora en que se crea. Posteriormente abre dicho fichero e introduce las cabeceras necesarias en función de los dispositivos que se deseen monitorizar, ayudándose de un *buffer* de escritura.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **monitor_desactivation**: cierra el fichero de monitorización (*.log*), eliminando el *buffer* de escritura utilizado.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveDcxData**: obtiene de la sección crítica los datos de la DCX que le llegan de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveGpsData**: obtiene de la sección crítica los datos del GPS que le llegan de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveGyroData**: obtiene de la sección crítica los datos del giróscopo que le llegan de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveSonarData**: obtiene de la sección crítica los datos de los sonares que le llegan de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendRomeoState**: coloca en la sección crítica el estado del ROMEO-4R para que sea enviado a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **set_initial_time_ref**: función privada de la clase que se encarga de establecer la referencia de tiempo inicial para el módulo, almacenándola en la variable de la clase *initial_time_ref*.

Parámetros:

- void.

Devuelve:

- void.

➢ **get_relative_time**: función privada de la clase que se encarga de devolver el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

Parámetros:

- void.

Devuelve:

- double: variable que contiene el instante de tiempo actual referido al instante inicial.

Por último, indicar que en el constructor de la clase, *CRomeoRSM()*, se inicializan las variables de la misma, se realiza la lectura del fichero de configuración del RSM (para ello se emplea la función *read_RSM_file_conf()*, perteneciente a la clase *CRomeoRSMReadFileConf*) donde se indican los dispositivos que deben ser monitorizados, se crea el fichero de monitorización (*.log*) mediante la función *monitor_activation()* y se establece la referencia de tiempo inicial del módulo mediante la función *set_initial_time_ref()*.

- *romeoRSMthread.cpp*: implementa la ejecución del hilo que se encarga de obtener el estado del ROMEO-4R, así como de la monitorización de los datos procedentes de los distintos dispositivos del HAM.

  El código del hilo es el siguiente:

  ➢ Añade las cabeceras necesarias.

  ➢ Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.

  ➢ Lanza el hilo de procesamiento del módulo (*RSMThread*).

  ➢ Asigna el identificador del robot que se le pasa al programa *main* tras ejecutarse el módulo.

  ➢ Crea un objeto de la clase *CRomeoRSM*.

  ➢ Comienza el bucle del que se saldrá cuando se active la variable *end*.

- Se reciben los datos de los dispositivos del ROMEO-4R, procedentes de la red.

- Convierte los datos de los dispositivos en el estado del robot en su formato particular, realiza la monitorización, e imprime por pantalla todos los datos de cada dispositivo, así como el estado del ROMEO-4R.

- Se envía el estado del ROMEO-4R, en su formato particular, a la red.

- Se esperan 30 ms mediante la función *usleep()*.

- Una vez se active la variable *end*, finaliza el hilo.

En el fichero de configuración de la monitorización del RSM (*./romeo/ril/ romeo_status_module/source/RSM/romeoRSM.conf*) se indica si un dispositivo debe ser monitorizado (1) o no (0).

De la lectura del fichero de configuración se encarga la **clase *CRomeoRSMReadFileConf*** (*./romeo/ril/romeo_status_module/source/RSM/ romeoRSMreadfileconf.h*), la cual contiene las siguientes funciones:

- **romeo_read_file_conf**: lee y procesa el archivo de configuración en el que se establecen los dispositivos que se desean monitorizar. Recorre el fichero de configuración guardando los respectivos valores (0 ò 1) en las variables de la clase correspondiente a la monitorización de cada dispositivo.

  Parámetros:

  - char *name: contiene el nombre del fichero de configuración.

  Devuelve:

  - int: *SUCCESS* si no hay errores o *ERROR* en caso contrario.

- **romeo_state_monitor**: comprueba si se monitoriza el estado del ROMEO-4R.

  Parámetros:

  - void.

  Devuelve:

  - int: *YES* si se monitoriza o *NO* en caso contrario.

Para los dispositivos del ROMEO-4R (DCX, GPS, giróscopo y sonares) existirán idénticas funciones a la anterior, llamadas *dcx_monitor(), gps_monitor(), gyro_monitor()* y *sonar_monitor()*, respectivamente.

3.3. Path Follower Module

El módulo destinado a realizar el seguimiento de caminos, enviando las referencias de velocidad y curvatura al *Hardware Abstraction Module*, para que éste se lo envíe a la tarjeta controladora de los motores (DCX), a partir del estado del ROMEO-4R procedente del *Romeo Status Module* y de la trayectoria a seguir, generada por el *Trajectory Generation Module*, es *el Path Follower Module*.

Con el *Path Follower Module* se puede cerrar en un principio el bucle de control del ROMEO-4R, pudiéndose realizar desplazamientos del vehículo hacia lugares deseados, aunque como se verá en posteriores apartados, se incluirá el módulo generador de trayectorias, que permitirá realizar el seguimiento de caminos más complejos.

Este módulo, además de contar con un sistema de comunicaciones BBCS y con una sección crítica, dispone de una clase que permite manejar el estado del módulo.

La compilación de este módulo se realiza mediante el fichero *Makefile* (*./romeo/ril/path_follower/Makefile*), que contiene una lista de todos los archivos objeto del *Path Follower Module*, así como una relación de dependencias de éstos con los archivos fuente, de manera que se puede ahorrar tiempo de compilación gracias a la utilización del comando *make*. Esto proporcionará un programa ejecutable, al que se llamará mediante el siguiente comando:

>> ./ROMEO_ PATHFOLLOWER [robot id] [lookahead]

donde:

[robot id] : identificador del robot con el que se esté trabajando.
[lookahead] : parámetro necesario para el algoritmo *pure pursuit.*

A este módulo le llega, el estado del ROMEO-4R, en su formato particular, procedente del *Romeo Status Module*, el camino a seguir, procedente del *Trajectory Generation Module* y los datos de control del módulo procedentes de la MML. A su vez, envía al *Hardware Abstraction Module* las referencias de control para los actuadores, y a la MML el estado del módulo.

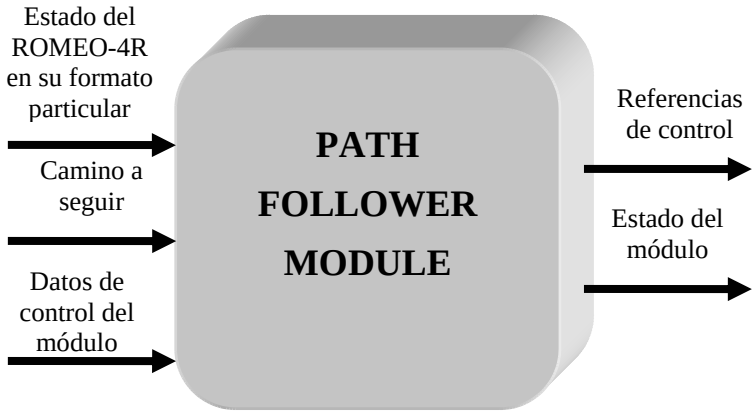


Figura 3.3-1: Entradas y salidas del Path Follower Module

La estructura del *Path Follower Module* viene definida por el siguiente esquema:

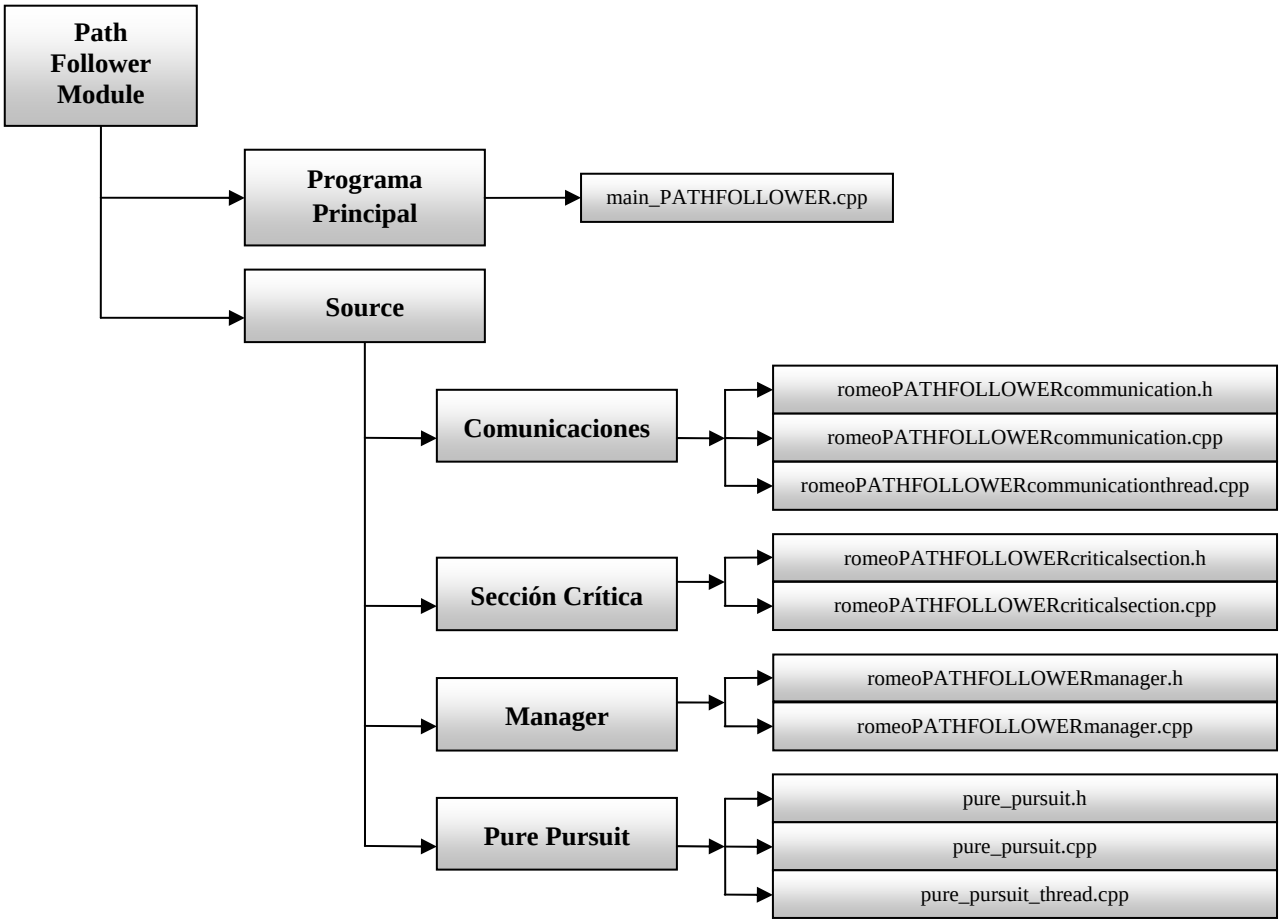


Figura 3.3-2: Esquema del Path Follower Module

A continuación se describirán con detalle todos los elementos que aparecen en el anterior esquema y que forman parte del *Path Follower Module*.

3.3.1. Programa principal

El programa encargado del lanzamiento del hilo de comunicaciones y del hilo de procesamiento, para quedar posteriormente a la espera de que se salga del mismo pulsando '*Ctrl+C*' y así finalizar los hilos, es el programa *main* (*./romeo/ril/ path_follower/main_PATHFOLLOWER.cpp*).

Dispone de un manejador, *handler_controlC*, que una vez que le llega la señal *SIGINT* (provocada cuando el usuario pulsa '*Ctrl+C*') activa la variable global *end*, que se encarga de finalizar tanto los hilos de comunicaciones y de procesamiento, como el propio programa principal.

Es importante indicar que se modifica la máscara del proceso, la cual se refiere a las señales que pueden o no interrumpir el funcionamiento de un hilo. Como se ha dicho, se desbloquea la señal *SIGINT* (generada al pulsar '*Ctrl+C*') para que pueda ser tratada por el manejador. A su vez, la máscara de proceso es heredada por los hilos que sean lanzados desde el programa *main*, por lo que se procederá a bloquear la señal *SIGRTMIN+1* en la máscara de este proceso, para desbloquearla en los hilos lanzados por el *main* que requieran del uso de temporizadores, los cuales se programan para un disparo periódico y para esperar dicha señal de forma síncrona.

El código del programa principal tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto de la sección crítica.
- Crea el hilo de comunicaciones (*CommunicationThread_ PATHFOLLOWER*) y el de procesamiento (*PurePursuitThread*).
- Crea la variable global *end*.
- Crea el manejador para la señal de finalización del programa.
- Lanza el programa *main*.
- Bloquea la señal *SIGRTMIN+1* en la máscara del proceso y desbloquea la señal *SIGINT* para que sea tratada por el manejador.
- Comprueba que se ha llamado correctamente al programa: *./ROMEO_PATHFOLLOWER [robot id] [lookahead]*.
- Coloca el valor *lookahead* en la sección crítica para que esté a disposición del algoritmo *pure pursuit*.
- Lanza el hilo de comunicaciones y el de procesamiento.
- Mientras se están ejecutando los hilos, el programa *main* permanece a la espera de que se pulse '*Ctrl+C*' para finalizar el programa.
- Una vez se active la variable *end*, se finalizarán los hilos y después terminará el programa principal.

3.3.2. Source

En común con el resto de módulos anteriormente vistos, esta carpeta contiene tanto los ficheros de comunicaciones como los de la sección crítica, además de contener la carpeta correspondiente al hilo de procesamiento *Pure Pursuit*. Sin embargo, este módulo posee, además, una clase que no se ha visto anteriormente llamada *CPathFollowerManager* que se encarga de manejar el estado del módulo.

3.3.2.1. Comunicaciones

Los conceptos de conexiones, puertos y slots, que permiten el funcionamiento del sistema de comunicaciones en el *Path Follower Module*, son descritos a continuación:

- ***Conexiones*:** a partir del fichero general de conexiones, se creará el fichero de conexiones para el *Path Follower Module* (*connectionsPATHFOLLOWER_X.conf*, donde *X* indica el identificador del robot correspondiente) utilizando la clase *RN_connections* que a su vez utiliza la clase *Connections_conf* que tiene funciones para escribir una conexión en el formato adecuado a partir de los parámetros de ésta.

- ***Puertos*:** todos los puertos del sistema, tanto locales como remotos, utilizados para las conexiones, se encuentran definidos en el fichero *./robot_architecture/comms/ NetPorts.h*, siendo los específicos del *Path Follower Module* los indicados en la siguiente tabla:

Nombre del Puerto	Nº del Puerto
PATH_FOLLOWER_ROMEO_LOCAL_PORT	2900
PATH_FOLLOWER_ROMEO_REMOTE_PORT	2950

Tabla 3.3.2.1-1: Puertos usados por el Path Follower Module

- ***Slots*:** los slots del *Path Follower Module* son los mostrados en el siguiente cuadro, donde la '*i*' hace referencia al identificador (ID) del robot, la columna *Input/Output* indica si son slots de entrada o salida para el módulo en particular y la columna *Secure* indica si es seguro (Sí/1) o si no lo es (No/0):

Nombre del Slot	Nº del Slot	Input / Output	Secure
ROMEO_SLOT_SENSORS_STATE (i)	4400 +i	Input	No
ROMEO_PATH_FOLLOWER_DATA_SLOT (i)	4450 + i	Input	Sí
ROMEO_SLOT_MODULE_CONTROL (i)	4000 +i	Input	Sí
ROMEO_PATH_FOLLOWER_MODULE_STATE_SLOT (i)	4500 +i	Output	No
SPEED_SLOT (i)	4050 +i	Output	No
CURV_SLOT (i)	4100 +i	Output	No

Tabla 3.3.2.1-2: Slots usados para las comunicaciones del Path Follower Module

Los slots que reciben información de forma segura, en este módulo son *ROMEO_SLOT_MODULE_CONTROL* y *ROMEO_PATH_FOLLOWER_DATA_SLOT*, necesitan una comprobación previa del canal de comunicaciones. Para ello, al inicializarse las comunicaciones, se sincronizan dichos slots mediante el envío de una estructura de datos de prueba que permita que los posteriores datos que se envíen a dicho slot lleguen en perfectas condiciones. Estos dos slots son seguros, a diferencia del resto, debido a la importancia de que los datos lleguen a su destino, ya que una pérdida de los mismos implicaría un gran fallo en el sistema. Esto no ocurre, por ejemplo, con los slots de envío de referencias de velocidad y curvatura al HAM, o con los datos de los dispositivos enviados por el HAM, ya que la pérdida de los mismos no sería preocupante debido a la alta frecuencia de envío de éstos, que dan lugar a que en un

tiempo insignificante, se produzca el envío de datos nuevos que dan lugar a una información más actualizada del vehículo.

De esta forma, el hilo de comunicaciones, en este módulo, está formado por los siguientes archivos:

- *romeoPATHFOLLOWERcommunication.cpp* y *romeoPATHFOLLOWER communication.h*: implementan la clase que se encarga de las comunicaciones con el resto de módulos. En ella, se encuentran funciones de envío de datos (que comprueban si existen datos en la sección crítica para, en caso afirmativo, colocarlos en el slot correspondiente para ser enviados por la red) y de recepción de datos (que comprueban que no exista error en las comunicaciones, para escribir los datos que se encuentran en el slot correspondiente, en la sección crítica).

La **clase *CRomeoCommunication_PATHFOLLOWER*** estará formada por las siguientes funciones:

- **Init**: inicializa las comunicaciones y añade los slots necesarios. Además, calcula el ancho de banda máximo para cada slot, teniendo en cuenta el ancho de banda máximo del canal y el número de slots antes comentado. Además de dichos slots, existirán dos slots más para las propias comunicaciones internas del BBCS.

  Parámetros:

  - void.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveWayPointData**: recibe los puntos de referencia para el *Path Follower Module* previa comprobación de que la estructura de datos que llega no es la utilizada para la comprobación de las comunicaciones a través de los slots seguros mediante la sincronización los mismos. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveRobotState**: recibe el estado del ROMEO-4R procedente del *Romeo Status Module*. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceivePathFollowerControlData**: recibe el dato de control para este módulo. Para ello, además de comprobar que el dato que le llega no es el de sincronización de los slots seguros, comprueba si el identificador del módulo al que va dirigido el dato de control (que se encuentra en la lista correspondiente a todos los datos de control que llegan desde la *Module Manager Layer* a los distintos módulos), se corresponde con el identificador del *Path Follower Module* (*PATH_FOLLOWER _MODULE_ID*). En caso afirmativo, coloca el dato en la sección crítica. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendRefSpeed**: envía la referencia de velocidad al *Hardware Abstraction Module*.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendRefCurv**: envía la referencia de curvatura al *Hardware Abstraction Module*.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendPathFollowerModuleState**: envía el estado del módulo a la red.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

La variable que se destaca en esta clase será la estructura *WayPointData*, que contiene los puntos de referencia para el *Path Follower Module*. Cada punto de referencia se corresponde con una estructura *waypoint*, que contiene las coordenadas [x, y] del punto, así como la velocidad con la que debe aproximarse el ROMEO-4R a dicho punto.

```
typedef struct
{
      double x;
      double y;
      double speed;

}WayPoint;
```

Los *waypoints* son almacenados en estructuras del tamaño *MAX_NUMBER_WP*, que actualmente es de 10 *waypoints*. Cuando el número de *waypoints* (indicado por la variable *length*) es superior a éste, será necesario enviar varias estructuras *WayPointData*, por lo que existen dos variables en dicha estructura que indican el inicio y el final del envío de puntos referencia. Así, estas dos variables tomarán los siguientes valores:

first_waypoint_data	last_waypoint_data	Función
1	**0**	Comienza el envío de waypoints
0	**0**	Siguen enviándose waypoints
0	**1**	Finaliza el envío de waypoints

Tabla 3.3.2.1-3: Clasificación de estructuras WayPointData

Por último, indicar que si el número de *waypoints* es inferior a *MAX_NUMBER_WP*, únicamente se enviará una estructura *WayPointData*, por lo que las variables *first_waypoint_data* y *last_waypoint_data* tomarán ambas el valor 1.

La estructura *WayPointData* viene definida por:

```
typedef struct
{
      WayPoint waypoints[MAX_NUMBER_WP];
      int length;
      int first_waypoint_data;
      int last_waypoint_data;

}WayPointData;
```

Comentar que las variables de la clase son inicializadas en el constructor de la misma.

- *romeoPATHFOLLOWERcommunicationthread.cpp*: implementa la ejecución del hilo que se encarga de las comunicaciones del *Path Follower Module* con el resto del sistema.

El código del hilo, cuya comprensión completa no será posible hasta ver los apartados posteriores, tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto a partir de la clase que implementa la sección crítica del *Path Follower Module*, *CRomeoCriticalSection_PATHFOLLOWER RomeoPathFollowerCS*.
- Crea la variable global externa *end* que fue creada en el programa principal *main*, y que indica la finalización del programa.
- Crea un objeto (*Block*), a partir de la clase *CCriticalSection*, que indica el bloqueo del hilo de funcionalidad (*PurePursuitThread*).
- Lanza el hilo de comunicaciones del módulo (*CommunicationThread_PATHFOLLOWER*).
- Crea las variables y objetos locales necesarios, tanto para trabajar con el BBCS como para crear el archivo de conexiones correspondiente.
- Inicializa las variables bloqueando el hilo de funcionalidad y activando el temporizador.

- Añade los slots.

- Crea el archivo de conexiones *connectionsPATHFOLLOWER_X.conf*, donde *X* indica el identificador del robot correspondiente.

- Añade la conexión al *Relay Node* del robot, indicando tanto la dirección IP del mismo, como los puertos local y remoto del *Path Follower Module*.

- Crea las conexiones usando el archivo de parámetros creado anteriormente.

- Inicializa el manejador del estado del módulo, colocando en la sección crítica el estado en *MODULE_ENDED*, que indica que el módulo se encuentra parado hasta que no le llegue de la *Module Manager Layer* la orden contraria.

- Bloquea el hilo de funcionalidad y para el temporizador.

- Comienza el bucle del que se saldrá cuando se active la variable *end*.

- Se sincronizan los datos y se espera un período de tiempo.

- Se realiza una prueba del canal de comunicaciones que indica si el canal está activo o inactivo.

- Se reciben los puntos de referencia para el *Path Follower Module*.

- Se recibe, del *Romeo Status Module*, el estado del ROMEO-4R.

- Se recibe el dato de control para este módulo, en caso de que se haya enviado una nueva orden de control desde la *Module Manager Layer*.

- Se ejecuta el manejador del módulo. A partir del estado del módulo y de la solicitud de la *Module Manager Layer*, la función *SetModuleAction AndUpdateModuleState()* decide si el hilo de funcionalidad tiene que ser bloqueado o desbloqueado (almacenando dicha decisión en la variable *action*), y también actualiza el estado del modulo.

- Si se ha decidido el bloqueo del hilo de funcionalidad, en cuyo caso la variable *action* toma el valor *BLOCK_MODULE_FUNCTIONALITY*, se procede a parar el temporizador y a bloquear el hilo.

- Si se ha decidido el desbloqueo del hilo de funcionalidad, la variable *action* toma el valor *UNBLOCK_MODULE_FUNCTIONALITY*, se procede a comprobar si la variable *timer* tiene un valor distinto a *CONTINUE_TIMER*, en cuyo caso, el temporizador comienza de nuevo y se desbloquea el hilo.

- Se envían las referencias de velocidad y curvatura del ROMEO-4R al *Hardware Abstraction Module*.

- Se envía el estado del módulo a la red.

- Una vez se sale del bucle, mediante la activación de la variable *end*, se finalizan las comunicaciones del BBCS y se eliminan los canales correspondientes.

Cualquier error que se produzca en el mismo será indicado por pantalla.

3.3.2.2. Sección crítica

La clase que implementa la sección crítica en el *Path Follower Module*, llamada *CRomeoCriticalSection_PATHFOLLOWER* y que se encuentra en *./romeo/ril/ path_follower/source/romeoPATHFOLLOWERcriticalsection.h*, es la encargada de las comunicaciones entre los hilos del módulo. En ella están las funciones *Set*, que permiten colocar en la sección crítica los datos que se manejan en el módulo, y las funciones *Get*, que permiten obtener dichos datos de la misma, las cuales vienen descritas en *./romeo/ril/path_follower/source/romeoPATHFOLLOWER criticalsection.cpp.*

La **clase *CRomeoCriticalSection_PATHFOLLOWER*** estará formada por las siguientes funciones:

- **SetWayPointData**: coloca en la sección crítica los puntos de referencia (*waypoints*) para el *Path Follower Module*. Para ello, comprueba si la estructura *WayPointData* que le llega es la primera, en cuyo caso se procede a borrar la lista de estructuras *WayPointData*, para posteriormente introducirla al principio de la lista.

  Parámetros:

  - WayPointData *point_data: variable que contiene una estructura que almacena los puntos de referencia.

  Devuelve:

  - void.

- **GetWayPointData**: obtiene de la sección crítica los puntos de referencia (*waypoints*) para el *Path Follower Module*. Para ello, comprueba si la longitud de la lista donde se almacenan las estructuras *WayPointData* es distinta de cero, en cuyo caso toma la última estructura *WayPointData* de la lista, borrándola de ella posteriormente.

  Parámetros:

  - WayPointData *point_data: variable que contiene una estructura que almacena los puntos de referencia.

Devuelve:

- bool: devuelve *true* una vez obtenido el dato o *false* en caso contrario.

- **CheckIfNewWayPointDataHaveArrived**: Comprueba si ha llegado una nueva estructura *WayPointData*. Para ello, comprueba si la longitud de la lista donde se almacenan dichas estructuras es distinta de cero.

  Parámetros:

  - void.

  Devuelve:

  - bool: devuelve *true* si ha llegado una nueva estructura *WayPointData* a la sección crítica o *false* en caso contrario.

- **SetRobotState**: coloca en la sección crítica el estado del ROMEO-4R.

  Parámetros:

  - ROMEO_TELEMETRY *robot_state: variable que contiene una estructura de datos del estado del ROMEO-4R.

  Devuelve:

  - void.

- **GetRobotState**: obtiene de la sección crítica el estado del ROMEO-4R.

  Parámetros:

  - ROMEO_TELEMETRY *robot_state: variable que contiene una estructura de datos del estado del ROMEO-4R.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetRefSpeed**: coloca en la sección crítica la referencia de velocidad.

  Parámetros:

  - double *ref_speed: variable que contiene la referencia de velocidad.

  Devuelve:

  - void.

- **GetRefSpeed**: obtiene de la sección crítica la referencia de velocidad.

  Parámetros:

  - double *ref_speed: variable que contiene la referencia de velocidad.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetRefCurv**: coloca en la sección crítica la referencia de curvatura.

  Parámetros:

  - double *ref_curv: variable que contiene la referencia de curvatura.

  Devuelve:

  - void.

- **GetRefCurv**: obtiene de la sección crítica la referencia de curvatura.

  Parámetros:

  - double *ref_curv: variable que contiene la referencia de curvatura.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetPathFollowerModuleState**: coloca en la sección crítica el estado del módulo.

  Parámetros:

  - ModuleState *ModuleStatus: variable que contiene el estado del módulo.

  Devuelve:

  - void.

- **GetPathFollowerModuleState**: obtiene de la sección crítica el estado del módulo.

Parámetros:

- ModuleState *ModuleStatus: variable que contiene el estado del módulo.

Devuelve:

- bool: devuelve *true* una vez obtenido el dato.

- **SetPathFollowerControlData**: coloca en la sección crítica el dato de control para este módulo, situándolo al principio de la lista.

  Parámetros:

  - ModuleControl *ControlData: variable que contiene una estructura que almacena el dato de control.

  Devuelve:

  - void.

- **GetPathFollowerControlData**: obtiene de la sección crítica el dato de control para este módulo. Para ello, comprueba si la longitud de la lista donde se almacenan los datos de control es distinta de cero, en cuyo caso toma el último dato de control de la lista, borrándolo de ella posteriormente.

  Parámetros:

  - ModuleControl *ControlData: variable que contiene una estructura que almacena el dato de control.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato o *false* en caso contrario.

- **SetPathFollowerFunctionalityState**: coloca en la sección crítica el estado del hilo de funcionalidad del módulo.

  Parámetros:

  - FUNCTIONALITY_STATE *state: variable que contiene una estructura que almacena el estado del hilo de funcionalidad del módulo.

  Devuelve:

  - void.

- **GetPathFollowerFunctionalityState**: obtiene de la sección crítica el estado del hilo de funcionalidad del módulo.

  Parámetros:

  - FUNCTIONALITY_STATE *state: variable que contiene una estructura que almacena el estado del hilo de funcionalidad del módulo.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetLookAhead**: coloca en la sección crítica el valor del parámetro *lookahead* para el algoritmo *pure pursuit*.

  Parámetros:

  - double *lookahead: variable que contiene el valor del parámetro *lookahead* para el algoritmo *pure pursuit*.

  Devuelve:

  - void.

- **GetLookAhead**: obtiene de la sección crítica el valor del parámetro *lookahead* para el algoritmo *pure pursuit.*

  Parámetros:

  - double *lookahead: variable que contiene el valor del parámetro *lookahead* para el algoritmo *pure pursuit*.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetStopFunctionality**: coloca en la sección crítica el valor de la variable *stop_functionality*, flag que indica que el vehículo se debe detener de forma segura.

  Parámetros:

  - int *stop_functionality: variable que indica que el vehículo se debe detener de forma segura.

  Devuelve:

  - void.

- **GetStopFunctionality**: obtiene de la sección crítica el valor de la variable *stop_functionality*, flag que indica que el vehículo se debe detener de forma segura.

  Parámetros:

  - int *stop_functionality: variable que indica que el vehículo se debe detener de forma segura.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetTimer**: coloca en la sección crítica el valor de la variable *timer*, flag que indica si el temporizador debe detenerse.

  Parámetros:

  - int *timer: variable que indica si el temporizador se debe detener.

  Devuelve:

  - void.

- **GetTimer**: obtiene de la sección crítica el valor de la variable *timer*, flag que indica si el temporizador debe detenerse.

  Parámetros:

  - int *timer: variable que indica si el temporizador se debe detener.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

En el constructor de la clase se inicializan las variables que se van a usar así como las listas empleadas para el almacenamiento de los *waypoints* y de los datos de control. El destructor de la clase será el encargado de eliminar la lista que contiene los datos de control para el módulo.

3.3.2.3. Manager

En este módulo se emplea una nueva clase, *CPathFollowerManager* (*./romeo/ril/path_follower/source/romeoPATHFOLLOWERmanager.h*), que permite manejar el estado del módulo. Desde la *Module Manager Layer* se enviará la señal de control al *Path Follower Module*, para que este hilo decida lo que debe realizar el módulo.

En *./robot_architecture/comms/GENERAL_IMI_data.h* se encuentran definidas todas las estructuras usadas en el control y configuración de los diferentes módulos, así como las estructuras que envían todos los módulos indicando su estado.

Las variables de la clase *CPathFollowerManager*, son las siguientes:

- *ModuleControl PathFollowerControlData*: estructura enviada desde la *Module Manager Layer* que india si el módulo debe actuar o detenerse. El módulo al cual va dirigida la orden de control viene indicado por la variable *module_id*, pudiendo tomar los siguientes valores:

module_id	Valor
HARDWARE_ABSTRACTION_MODULE_ID	0
ROMEO_STATUS_MODULE_ID	1
PATH_FOLLOWER_MODULE_ID	2
TRAJECTORY_GENERATION_MODULE_ID	3

Tabla 3.3.2.3-1: Identificadores de módulo

```
typedef enum
{
	START_MODULE,
	STOP_MODULE

}ModuleRequest;


typedef struct
{
	int working_mode;

}ModuleParams;


typedef struct
{
	int module_id;
	ModuleRequest module_request;
	ModuleParams module_params;

}ModuleControl;
```

Cabe destacar los posibles valores de la variable *module_request*, que son los siguientes:

Valor de la variable *module_request*	Descripción
START_MODULE	Módulo actuando
STOP_MODULE	Módulo parado

Tabla 3.3.2.3-2: Valores de la variable module_request

- *ModuleState PathFollowerControlData*: estructura cuya función principal es la de indicar el estado del módulo, además de los posibles errores que se produzcan.

```
typedef enum
{
      MODULE_RUNNING,
      MODULE_ENDED,
      MODULE_ABORTED,
      MODULE_WAITING_DATA_TO_START

}ModuleStatus;


typedef struct
{
      int nothing;

}SecondLevelModuleState;


typedef struct
{
      int module_id;
      int error_code;
      ModuleStatus module_status;
      SecondLevelModuleState second_level_module_status;

}ModuleState;
```

Los posibles valores de la variable *module_status* son los siguientes:

Valor de la variable *module_status*	Descripción
MODULE_RUNNING	Módulo actuando
MODULE_ENDED	Módulo parado
MODULE_ABORTED	Módulo abortado
MODULE_WAITING_DATA_TO_START	Módulo a la espera de datos para comenzar

Tabla 3.3.2.3-3: Valores de la variable module_status

- FUNCTIONALITY_STATE functionality_state: estructura que indica las posibles acciones que el manejador ejercerá sobre el hilo de funcionalidad (*PurePursuitThread*), así como los posibles errores que se produzcan.

```
typedef struct
{
      int status;
      int error_code;

}FUNCTIONALITY_STATE;
```

Los posibles valores de la variable *status* son los siguientes:

Valor de la variable *status*	Descripción
FUNCTIONALITY_RUNNING	Hilo de funcionalidad ejecutándose
FUNCTIONALITY_ENDED	Hilo de funcionalidad detenido

Tabla 3.3.2.3-4: Valores de la variable status

Actualmente, los posibles valores de la variable *error_code* son los siguientes:

Valor de la variable *error_code*	Descripción
ROMEO_FATAL_MOVEMENT_ERROR	Fallo general de movimiento
ROMEO_PATH_NOT_COMPLETE	Camino no completado

Tabla 3.3.2.3-5: Valores de la variable error_code

- *int stop_functionality*: indica si el vehículo se debe detener de forma segura.

  Los valores que puede tomar la variable son:

Valor de la variable *stop_functionality*	Descripción
STOP	El vehículo debe detenerse de forma segura.
NO_STOP	El vehículo no debe detenerse.

Tabla 3.3.2.3-6: Valores de la variable stop_functionality

Las funciones que componen la **clase *CPathFollowerManager***, son las siguientes:

- **InitManager**: se encarga de la inicialización del manejador del estado del módulo, colocando en la sección crítica dicho estado.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - void.

- **SetModuleActionAndUpdateModuleState**: esta función decide, a partir del estado del módulo y de la solicitud procedente de la *Module Manager Layer*, si el hilo de funcionalidad tiene que ser bloqueado o desbloqueado, y también actualiza el estado del módulo.

  Parámetros:

  - CRomeoCriticalSection_PATHFOLLOWER *RomeoPath FollowerCS: objeto de la sección crítica correspondiente al *Path Follower Module*.

  Devuelve:

  - int: variable que indica la acción a realizar sobre el hilo de funcionalidad, pudiendo tomar los siguientes valores:

Valor de la variable *int*	**Descripción**
BLOCK_MODULE_FUNCTIONALITY	Bloquea el hilo de funcionalidad
UNBLOCK_MODULE_FUNCTIONALITY	Desbloquea el hilo de funcionalidad
NOTHING_TO_DO	No hace nada

Tabla 3.3.2.3-7: Posibles acciones sobre el hilo de funcionalidad

Esta función tiene una estructura bastante compleja que será detallada a continuación:

1. Inicialmente el valor que devuelve es *NOTHING_TO_DO*, que indica que no se debe hacer nada sobre el hilo de funcionalidad.

2. Realiza la lectura del estado del módulo y del estado del hilo de funcionalidad.

3. Decide si bloquear o no el hilo de funcionalidad, para lo cual se contemplan tres posibles casos:

   a) Si el hilo de funcionalidad está parado (*FUNCTIONALITY_ ENDED*) y el módulo está ejecutándose (*MODULE_ RUNNING*), se detiene el módulo (*MODULE_ENDED*) y se bloquea el hilo de funcionalidad (*BLOCK_MODULE_ FUNCTIONALITY*).

   b) Si el hilo de funcionalidad presenta errores y la variable que indica que el vehículo debe detenerse (*stop_functionality*) no está activa (*NO_STOP*), se aborta la ejecución del módulo (*MODULE_ABORTED*), se indica el error que se ha producido, se detiene el vehículo de forma segura (*STOP*), y se vuelve a indicar que ya no existe error (*NO_ERROR_IMI*).

   c) Si el vehículo esta detenido (*STOP*) y el hilo de funcionalidad ha finalizado (*FUNCTIONALITY_ENDED*), se bloquea dicho hilo (*BLOCK_MODULE_FUNCTIONALITY*) y se vuelve a

desactivar la variable que indica la detención del vehículo (*NO_STOP*).

4. En caso de que llegue un nuevo dato de control del módulo (*START_ MODULE* o *STOP_MODULE*) desde la *Module Manager Layer*:

    a) Si llega *START_MODULE* pueden ocurrir los siguientes casos:

        - Si el módulo está ejecutándose (*MODULE_ RUNNING*), se indica el correspondiente error (*MODULE_RUNNING_TO_RESTART_STOP_IT_BE FORE*).

        - Si el módulo estaba detenido (*MODULE_ENDED* o *MODULE_ABORTED*) y llegan nuevos *waypoints*, se desbloquea el hilo de funcionalidad (*UNBLOCK_ MODULE_FUNCTIONALITY*) y se activa el módulo (*MODULE_RUNNING*), guardando el estado del hilo (*FUNCTIONALITY_RUNNING*). En caso de que no lleguen nuevos *waypoints* no se ejecuta ninguna acción sobre el hilo de funcionalidad (*NOTHING_ TO_DO*) y el módulo se queda a la espera de los mismos (*MODULE_WAITING_DATA_TO_START*).

    b) Si llega *STOP_MODULE* pueden ocurrir los siguientes casos:

        - Si el estado del módulo es finalizado (*MODULE_ ENDED*) o abortado (*MODULE_ABORTED*), no se realiza ninguna acción.

        - Si el módulo estaba ejecutándose (*MODULE_ RUNNING*), su estado pasará a abortado (*MODULE_ ABORTED*), y el vehículo se detendrá de forma segura (*STOP*).

5. Si el módulo se encuentra esperando nuevos *waypoints* para comenzar, una vez que estos lleguen se desbloquea el hilo de funcionalidad (*UNBLOCK_ MODULE_FUNCTIONALITY*) y se activa el módulo (*MODULE_RUNNING*), guardando el estado del hilo (*FUNCTIONALITY_RUNNING*). Mientras no lleguen los waypoints, no se actúa sobre el hilo (*NOTHING_TO_DO*).

6. Actualiza la variable que indica si el vehículo debe detenerse (*stop_functionality*) y el estado del módulo, devolviendo la acción a realizar sobre el hilo de funcionalidad.

- **TranslateErrorCode**: función privada de la clase que se encarga de traducir el código de error del hilo de funcionalidad a un código de error de módulo. Los posibles errores se encuentran definidos en *./robot_architecture/comms/GENERAL_IMI_error_code.h* (errores de módulo con un valor entre 0 y 99) y en *./romeo/comms/ROMEO_IMI_error_code.h* (errores pertenecientes al hilo de funcionalidad con un valor entre 200 y 299).

Valor de la variable *error_code*	Valor
SYNCH_CODE	-1
NO_ERROR_IMI	0
ERROR_IMPOSSIBLE_TO_STOP_MODULE	1
MODULE_RUNNING_TO_RESTART_STOP_IT_BEFORE	2
ROMEO_FATAL_MOVEMENT_ERROR	200
ROMEO_PATH_NOT_COMPLETE	201

Tabla 3.3.2.3-8: Tipos de errores

Parámetros:

- int functionality_error_code: variable que indica el tipo de error que se ha producido, y que toma los valores anteriormente descritos en la variable *error_code* perteneciente a la estructura *FUNCTIONALITY_STATE*.

Devuelve:

- int: variable que representa el error producido.

Para concluir, indicar que en el constructor de la clase se inicializarán las variables de la misma, donde el identificador del módulo será el correspondiente al *Path Follower Module* (*PATH_FOLLOWER_MODULE_ID*), no existirán errores (*NO_ERROR_IMI*), el estado del módulo será el de finalizado (*MODULE_ENDED*) y la variable que indica la detención del vehículo (*stop_functionality*) estará desactivada (*NO_STOP*).

3.3.2.4. Pure Pursuit

El método elegido para implementar el seguimiento de caminos en el ROMEO-4R es el de persecución pura (*pure pursuit*). Este método calcula la curvatura de las ruedas a partir del camino a seguir (proporcionado por el módulo de generación de trayectorias, *Trajectory Generation Module*), y de la posición y orientación del vehículo en cada instante (proporcionadas por el *Romeo Status Module*, a través de la información de los sensores enviada por el HAM), moviéndose el vehículo a velocidad constante. Este cálculo se repite cíclicamente hasta llegar al final del camino.

Para poder estudiar este método de seguimiento de caminos, considérese un sistema de referencia local asociado al movimiento del vehículo, tal y como se muestra en la Figura 3.3.2.4-1. Se supone que en el intervalo de control la curvatura es constante, describiendo el vehículo un arco de circunferencia.

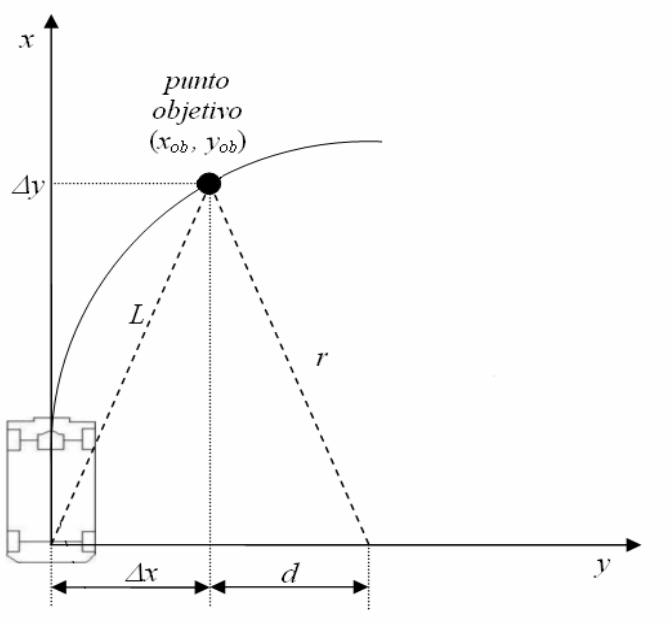


Figura 3.3.2.4-1: Seguimiento de caminos mediante persecución pura

Observando la figura anterior, se puede deducir las siguientes relaciones:

$$r = \Delta x + d$$

$$d^2 + (\Delta y)^2 = r^2$$

siendo *r* el radio de curvatura del vehículo.

Despejando *d* en la primera ecuación y sustituyendo en la segunda:

$$(r - \Delta x)^2 + (\Delta y)^2 = r^2$$

donde el radio de curvatura necesario para que el vehículo se desplace *Δx*, *Δy* es:

$$r = \frac{(\Delta x)^2 + (\Delta y)^2}{2\Delta x}$$

Por tanto, la curvatura que es necesaria suministrar al vehículo es:

$$\gamma = \frac{1}{r} = -\frac{2(\Delta x)}{L^2}$$

donde el signo viene dado por el sentido de giro necesario para alcanzar el punto objetivo de la Figura 3.3.2.4-1, *L* es la distancia a la que se encuentra el punto objetivo y *Δx* es el desplazamiento lateral.

Se observa como la ley de control de persecución pura es proporcional al error lateral (Δx) con respecto al punto objetivo, y que la ganancia varía con la inversa del cuadrado de *L*.

El algoritmo que implementa esta ley de control, determina el punto del camino que se encuentra a una distancia previamente definida *L*, *look-ahead*, y calcula el error lateral (Δx) con respecto a la posición actual del centro de guiado del vehículo.

Si el sistema de coordenadas no está ligado al movimiento del vehículo, es necesario tener en cuenta la orientación de éste para calcular el error local:

$$\Delta x = (x_{ob} - x)\cos\phi + (y_{ob} - y)sen\phi$$

Una forma de implementar lo anteriormente dicho para aplicar la ley de control consiste en obtener, en cada periodo de control, el punto objetivo del camino que esté más próximo al vehículo, y elegir el punto objetivo a una distancia fija de valor *look-ahead* sobre el camino, tomada en el sentido de avance a partir del punto del camino objetivo antes comentado. Después se calcula la distancia (*L*) existente entre el punto objetivo elegido y la posición actual del vehículo, se obtiene el error local (Δx) con respecto a la posición actual del centro de guiado del vehículo, y se aplica la ley de control que proporciona la curvatura que se debe aplicar al vehículo.

La elección del parámetro que indica la distancia al punto objetivo (*look-ahead*) es crítica para la eficiencia del controlador de persecución pura. Existen varios casos en los que la aplicación de la anterior ley de control presenta problemas:

1. Cuando el punto objetivo se encuentra muy alejado (*L* es muy grande), la actuación suministrada por la ley de control puede resultar demasiado pequeña. En este caso, debe sustituirse la distancia *L* al punto objetivo por una distancia máxima.

2. Cuando el vehículo se encuentra sobre el camino (Δx es muy pequeña), pero orientado en dirección contraria al camino, se obtendrá un valor muy pequeño de la curvatura que no permitirá girar el vehículo hasta orientarse según el camino. Si se detecta esta condición, es necesario sustituir la curvatura obtenida por otro valor mayor.

Los mejores resultados se obtienen con un ajuste automático del parámetro *look-ahead*, en función del error actual y la curvatura de camino. Esto dará lugar a un guiado eficiente a elevadas velocidades.

Los archivos que implementan el método de persecución pura (*pure pursuit*) son los siguientes:

- *pure_pursuit.cpp* y *pure_pursuit.h*: implementan la clase referente al seguimiento de caminos mediante el método de persecución pura.

La **clase *Pure_pursuit*** está formada por las siguientes funciones:

- **algoritmo_pp**: función que contiene el algoritmo principal del seguimiento de caminos mediante *pure pursuit*.

  Parámetros:

  - void.

  Devuelve:

  - void.

  El algoritmo que implementa el seguimiento de caminos mediante persecución pura es el siguiente:

  1. Se recibe de la sección crítica el valor que indica la distancia al punto objetivo (*look-ahead*).

  2. Si existe un camino a seguir y el hilo no está detenido, se procede con el resto de acciones.

  3. Se obtiene el punto objetivo que se encuentre a la distancia dada por el parámetro *look-ahead* mediante la función *elige_objetivo_ pp().*

  4. Se obtiene la distancia entre el punto actual y el punto que se encuentra a una distancia menor de *look-ahead* (punto objetivo).

  5. Si el punto se encuentra más cerca de una cierta distancia (definida mediante la macro *CERCA*) del último punto del camino que ha llegado de la sección crítica, se detiene el vehículo de forma suave mediante la función *SecurityStop(),* lo cual le permite detenerse lo más próximo posible del punto objetivo aprovechando la inercia del mismo.

  6. Si se ha conseguido que la referencia de velocidad sea nula, se pueden dar dos situaciones:

     a) No quedan más *waypoints* por llegar: se indica en el estado del hilo *FUNCTIONALITY_ENDED* y se imprime por pantalla que el camino se ha finalizado correctamente, reiniciando a su vez las variables del algoritmo mediante la función *reiniciar_pp().*

     b) Faltan waypoints por llegar: se indica el error *ROMEO_ PATH_NOT_COMPLETE* en el estado del hilo y se imprime por pantalla dicho error.

7. Mientras el vehículo se está parando, se da una referencia de curvatura nula para que no gire el volante mientras se está acercando al último punto.

8. Si el punto se encuentra más alejado del último punto del camino que ha llegado de la sección crítica que la distancia indicada por la macro *CERCA*, se procede a obtener la ley de control que proporciona la referencia de la curvatura deseada.

9. Se almacenan en las correspondientes variables de la clase las referencias de curvatura (calculada anteriormente) y de velocidad (que viene dada por la velocidad que contiene el *waypoint*) para ser enviadas al HAM.

10. Por último, se imprimen por pantalla las coordenadas del punto actual del ROMEO-4R y del punto al que debe ir, así como la referencia de curvatura y velocidad aplicada en cada momento.

- **reiniciar_pp**: función privada de la clase que se encarga de reiniciar las variables de la clase que se utilizan en el algoritmo del *pure pursuit*.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **elige_objetivo_pp**: función privada de la clase que se encarga de obtener el punto objetivo al que debe dirigirse el ROMEO-4R.

  Parámetros:

  - int objetivo_actual: índice que señala el punto actual del camino.

  Devuelve:

  - int: índice que señala el punto objetivo.

  El algoritmo de esta función es el siguiente:

  1. Se obtiene el punto del camino más cercano del punto actual mediante la función *mas_cercano_pp()*.

  2. Se comprueba si el punto del camino más cercano al actual es el último punto o no.

a) Si no lo es, se van sumando las distancias existentes entre puntos del camino consecutivos (partiendo del punto del camino más cercano al actual) hasta comprobar cual es el último punto que se encuentra a una distancia menor de *look-ahead*. Siendo este el punto objetivo elegido.

b) Si lo es, el punto objetivo elegido será el último punto del camino.

- **mas_cercano_pp**: función privada de la clase que se encarga de obtener el punto del camino más cercano al punto actual.

  Parámetros:

  - void.

  Devuelve:

  - int: devuelve el índice que señala el punto del camino más cercano al punto actual.

  El algoritmo de esta función es el siguiente:

  1. Se obtiene la distancia entre el punto actual del ROMEO-4R y el siguiente punto del camino. Si es la primera iteración, el siguiente punto del camino será el primero de éste.

  2. Si el siguiente punto del camino no es el último punto, comprueba para el resto de puntos del camino si alguno de ellos se encuentra más cerca del punto actual que el de partida, en cuyo caso ese será el punto más cercano, saltándose el anterior.

- **rotar_pp**: función privada de la clase que se encarga de pasar de ejes globales a ejes del ROMEO-4R si el sistema de coordenadas no está ligado al movimiento del vehículo.

  Parámetros:

  - double inc_x, inc_y, angulo: valores que se quieren obtener en el sistema de referencia local del ROMEO-4R, dado el ángulo de giro entre el sistema de referencia global y el anterior.

  Devuelve:

  - double: valor de entrada referido al sistema de referencia del ROMEO-4R.

- **ReceiveWayPointData**: obtiene de la sección crítica los puntos de referencia para el algoritmo de persecución pura.

  Parámetros:

  - void.

  Devuelve:

  - bool: devuelve *true* si existen *waypoints* para que el ROMEO-4R proceda a seguir dicho camino o *false* en caso contrario.

  El algoritmo de esta función es el siguiente:

  1. Si llegan estructuras *WayPointData* de la sección crítica, se pueden dar varios casos:

     a) Si es la primera estructura *WayPointData* (*first_waypoint_ data=1* y *last_waypoint_data=0*): crea la tabla dinámica *camino[],* mediante la función *malloc(),* donde se almacenan los *waypoints* (en caso de existir ya dicha tabla, la elimina primero) e introduce en dicha tabla los *waypoints* que le llegan por el canal de comunicaciones. Activa el flag *MoreWayPoints,* que indica que quedan más estructuras *WayPointData* por llegar, e indica en el estado del hilo de funcionalidad que no existen errores (*NO_ERROR_IMI*) y que éste se está ejecutando (*FUNCTIONALITY_RUNNING*).

     b) Si es la única estructura *WayPointData* (*first_waypoint_ data=1* y *last_ waypoint_data=1*): crea la tabla dinámica *camino[]* mediante la función *malloc()* (en caso de existir ya dicha tabla, la elimina primero) e introduce en dicha tabla los *waypoints* que le llegan por el canal de comunicaciones. Desactiva el flag *MoreWayPoints,* e indica en el estado del hilo de funcionalidad que no existen errores (*NO_ERROR_IMI*) y que éste se está ejecutando (*FUNCTIONALITY_RUNNING*).

     c) Si es una estructura *WayPointData* intermedia (*first_ waypoint_data=0* y *last_ waypoint_data=0*): redimensiona la tabla dinámica *camino[]* mediante la función re*alloc()* e introduce en dicha tabla los *waypoints* que le llegan por el canal de comunicaciones.

     d) Si es la última estructura *WayPointData* (*first_waypoint_ data=0* y *last_ waypoint_data=1*): redimensiona la tabla dinámica *camino[]* mediante la función re*alloc()* e introduce en dicha tabla los *waypoints* que le llegan por el canal de comunicaciones. A continuación, desactiva el flag *MoreWayPoints.*

2. Si no se ha recibido el camino completo (el flag *MoreWayPoints* está activo), se calcula la distancia total del camino, de forma que si ésta es menor que tres veces el valor del *look-ahead* no se comenzará a ejecutar el algoritmo *pure pursuit* (*algoritmo_pp()*).

- **ReceiveRobotState**: obtiene de la sección crítica el estado del ROMEO-4R que le llega de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendRefSpeed**: coloca en la sección crítica la referencia de velocidad del ROMEO-4R para que sea enviada a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendRefCurv**: coloca en la sección crítica la referencia de curvatura del ROMEO-4R para que sea enviada a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendFunctionalityStatus**: coloca en la sección crítica el estado del hilo de funcionalidad (*PurePursuitThread*) para que sea enviado a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SecurityStop**: función privada de la clase que se encarga de realizar la parada del vehículo de forma segura. Si la velocidad del ROMEO-4R es superior a 0.1 m/s, la reduce progresivamente en cantidades de 0,15 m/s hasta que ésta sea nula. En el caso de marcha atrás del vehículo, si la velocidad es menor de -0,1 m/s, se aumenta progresivamente en cantidades de 0,15 m/s hasta que ésta sea nula.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **CancelPath**: cancela el camino y realiza la parada del vehículo de forma segura. Ejecuta la función *SecurityStop()* para detener el vehículo y una vez que la referencia de velocidad sea nula y el hilo finalice (*FUNCTIONALITY_ENDED*) se imprime por pantalla un mensaje que indica que el camino ha sido abortado o que ha ocurrido un error, y se reinician las variables de la clase.

  Parámetros:

  - void.

  Devuelve:

  - void.

En el constructor de la clase se inicializan las variables de la misma, a la vez que se obtiene de la sección crítica el parámetro *look-ahead*.

- *pure_pursuit_thread.cpp*: implementa la ejecución del hilo que se encarga del seguimiento de caminos basado en el método de persecución pura.

  El código del hilo tiene el siguiente orden:

  - Añade las cabeceras necesarias.

  - Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.

  - Crea un objeto a partir de la clase que implementa la sección crítica del *Path Follower Module*, *CRomeoCriticalSection_PATHFOLLOWER RomeoPathFollowerCS*.

  - Crea un objeto (*Block*) a partir de la clase *CCriticalSection*, que indica el bloqueo del hilo de funcionalidad.

- Lanza el hilo de funcionalidad del módulo (*PurePursuitThread*).

- Crea la variable local *stop_functionality*, que indica si se debe detener (*STOP*) o no (*NO_STOP*) el hilo de funcionalidad, y *timer*, que indica si el temporizador debe comenzar (*START_TIMER*), detenerse (*STOP_TIMER*) o continuar (*CONTINUE_TIMER*), además de la variable *signal* con el valor de la señal de tiempo real.

- Crea un objeto de la clase *Pure_pursuit*.

- Crea el temporizador y le asocia la señal que se le pasa por parámetro.

- Comienza el bucle del que se saldrá cuando se active la variable *end*.

- Obtiene de la sección crítica el valor del parámetro *timer*, que le llega procedente del hilo de comunicaciones y que tomará dos posibles valores:

    1. *START_TIMER*: indica que el temporizador debe comenzar, por lo que se configura el temporizador y lo pone en marcha, programándolo para un disparo periódico y para esperar la señal de forma síncrona. El tiempo de espera será *T_CONTROL*, esto es, 50 ms. A continuación se le asigna a la variable *timer* el valor *CONTINUE_TIMER*.

    2. *STOP_TIMER*: indica que el temporizador debe detenerse, por lo que se configura el temporizador y lo pone en marcha, programándolo con un tiempo de espera de 0 ms, lo que desactivará el mismo, evitando así que este siga corriendo en caso de que se bloquee el hilo.

- Bloquea el hilo de funcionalidad.

- Si llega una estructura *WayPointData* se obtienen de la sección crítica el estado del robot y la variable *stop_functionality*. Si el valor de dicha variable es *NO_STOP* se ejecuta el algoritmo *pure pursuit* (*algoritmo_pp()*), en caso contrario se cancela el camino, parando el vehículo con seguridad (*CancelPath()*).

- Se envían las referencias de velocidad y curvatura del ROMEO-4R al *Hardware Abstraction Module*, y se coloca el estado del hilo de funcionalidad en la sección crítica.

- Desbloquea el hilo de funcionalidad.

- Espera el vencimiento del temporizador en caso de que la variable *timer* tenga el valor *CONTINUE_TIMER*.

- Una vez se sale del bucle, mediante la activación de la variable *end*, se elimina el temporizador.

3.4. Trajectory Generation Module

Anteriormente se comentó que el ROMEO-4R seguía una trayectoria que se le pasaba en forma de *waypoints* al *Path Follower Module*, y se trataba el problema del control del ROMEO-4R para que siguiera la trayectoria especificada. Aquí se estudia la generación de dichas trayectorias, que serán implementadas por el *Trajectory Generation Module*. Partiendo de una especificación del movimiento que debe seguir el vehículo, se trata de definir de forma precisa una trayectoria espacial y temporal del mismo. Este módulo, por tanto, es el encargado de generar un camino para el ROMEO-4R a partir de uno o varios puntos suministrados, siendo el *Path Follower Module* el encargado de realizar el seguimiento de dicho camino.

Al igual que se vio en el anterior módulo, éste también dispone de un sistema de comunicaciones BBCS, una sección crítica, y una clase que permite manejar el estado del módulo.

La compilación de este módulo se realiza mediante el fichero *Makefile* (*./romeo/ril/trajectory_generation/Makefile*), que contiene una lista de todos los archivos objeto de este módulo, así como una relación de dependencias de éstos con los archivos fuente, de manera que se puede ahorrar tiempo de compilación gracias a la utilización del comando *make*.

Esto proporcionará un programa ejecutable, al que se llamará mediante el siguiente comando:

>> ./ROMEO_ TRAJECTORY_GENERATION [robot id] [space]

donde:

[robot id] : identificador del robot con el que se esté trabajando.
[space]: parámetro que indica la separación existente entre dos puntos consecutivos de una trayectoria generada..

A este módulo le llega, por una parte, el estado del ROMEO-4R, en su formato particular, procedente del *Romeo Status Module*, y por otra parte, los datos de control del módulo, procedentes de la MML. A su vez, envía al *Path Follower Module* el camino a seguir, y a la MML el estado del módulo.

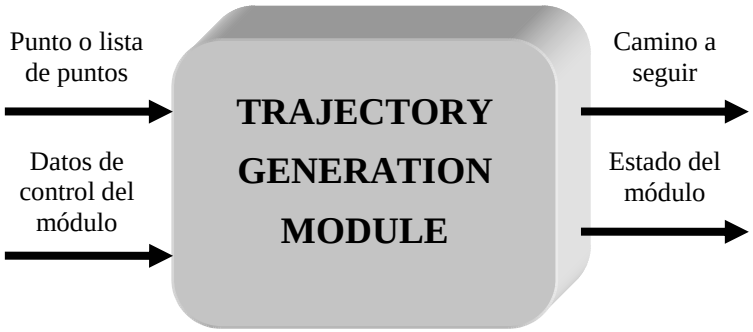


Figura 3.4-1: Entradas y salidas del Trajectory Generation Module

La estructura del *Trajectory Generation Module* viene definida por el siguiente esquema:

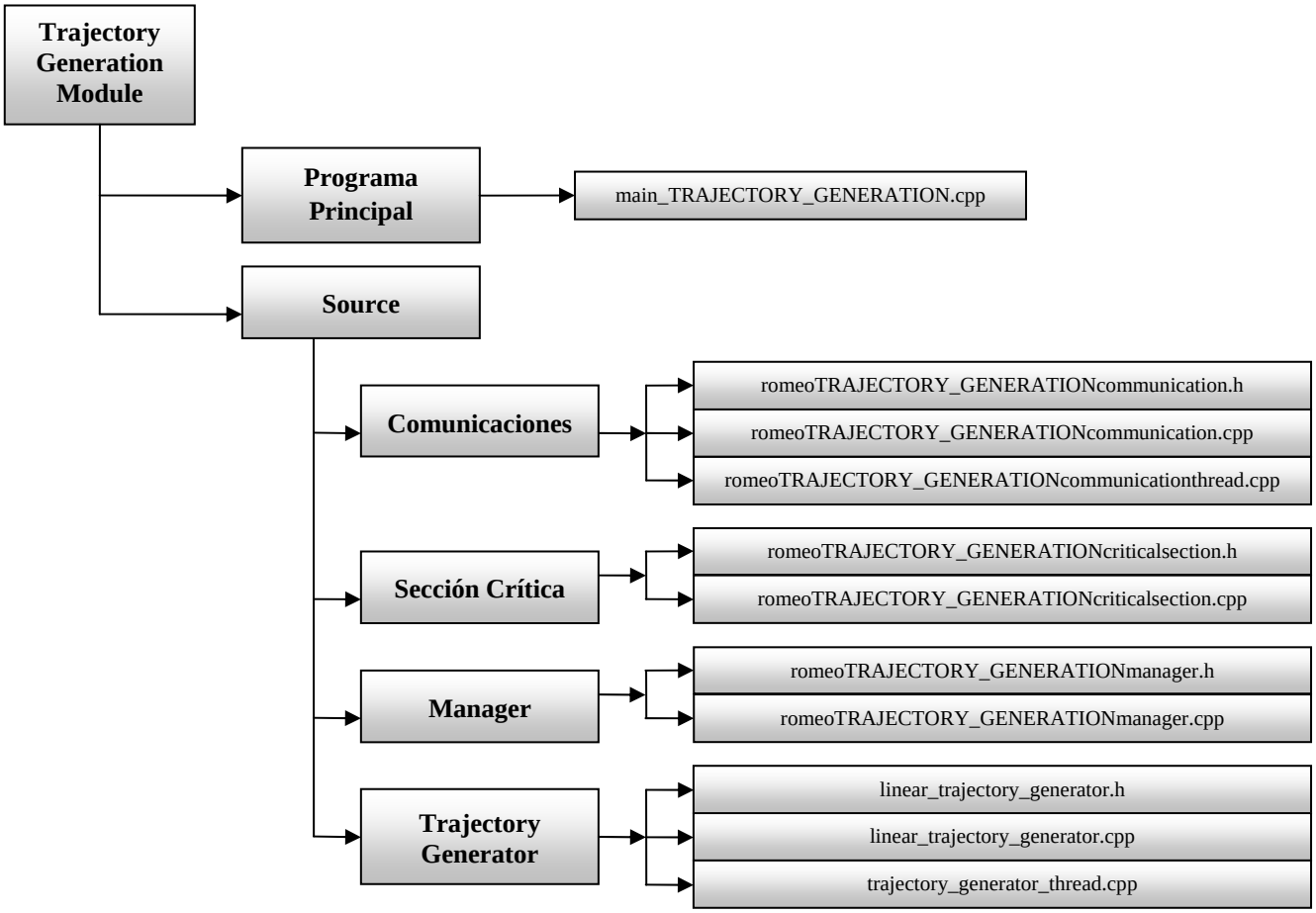


Figura 3.4-2: Esquema del Trajectory Generation Module

A continuación se describirán con detalle todos los elementos que aparecen en el anterior esquema y que forman parte del *Trajectory Generation Module*.

3.4.1. Programa principal

El programa encargado del lanzamiento del hilo de comunicaciones y del hilo de procesamiento, para quedar posteriormente a la espera de que se salga del programa pulsando '*Ctrl+C*' y así finalizar los hilos, es el programa *main* (*./romeo/ril/trajectory_generation/main_TRAJECTORY_GENERATION.cpp*).

Dispone de un manejador, *handler_controlC*, que una vez que le llega la señal *SIGINT* (provocada cuando el usuario pulsa '*Ctrl+C*') activa la variable global *end*, que se encarga de finalizar tanto los hilos de comunicaciones y de procesamiento, como el propio programa principal.

El código del programa principal tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto de la sección crítica.
- Crea el hilo de comunicaciones (*CommunicationThread_TRAJECTORY_GENERATION*) y el de procesamiento (*TrajectoryGeneratorThread*).
- Crea la variable global *end* y el manejador para la señal de finalización del programa.
- Lanza el programa *main*.
- Configura la señal *SIGINT*, desbloqueándola en la máscara del proceso para que sea tratada por el manejador.
- Comprueba que se ha llamado correctamente al programa: *./ROMEO_TRAJECTORY_GENERATION [robot id] [space]*.
- Coloca el valor *space* en la sección crítica para que esté a disposición del *TrajectoryGeneratorThread*.
- Lanza el hilo de comunicaciones y el de procesamiento.
- Mientras se están ejecutando los hilos, el programa *main* permanece a la espera de que se pulse '*Ctrl+C*' para finalizar el programa.
- Una vez se active la variable *end*, se finalizarán los hilos y después terminará el programa principal.

3.4.2. Source

Esta carpeta contiene los ficheros de comunicaciones, los de la sección crítica, la clase que se encarga de manejar el estado del módulo (*CTrajectoryGenerationManager*) y la carpeta correspondiente al hilo de procesamiento *Trajectory Generator*.

3.4.2.1. Comunicaciones

El sistema de comunicaciones en el *Trajectory Generation Module* se describe a continuación:

- ***Conexiones*:** a partir del fichero general de conexiones, se creará el fichero de conexiones para este módulo (*connectionsTRAJECTORY_GENERATION_ X.conf*, donde *X* indica el identificador del robot correspondiente) utilizando la clase *RN_connections* que a su vez utiliza la clase *Connections_conf* que tiene funciones para escribir una conexión en el formato adecuado a partir de los parámetros de ésta.

- ***Puertos*:** todos los puertos del sistema, tanto locales como remotos, utilizados para las conexiones, se encuentran definidos en el fichero *./robot_architecture/comms/ NetPorts.h*, siendo los específicos de este módulo, los indicados en la siguiente tabla:

Nombre del Puerto	Nº del Puerto
ROMEO_TRJECTORY_GENERATION_MODULE_LOCAL_PORT	3100
ROMEO_TRJECTORY_GENERATION_MODULE_REMOTE_PORT	3150

Tabla 3.4.2.1-1: Puertos usados por el Trajectory Generation Module

- ***Slots*:** los slots del *Trajectory Generation Module* son los mostrados en el siguiente cuadro, donde la '*i*' hace referencia al identificador (ID) del robot, la columna *Input/Output* indica si son slots de entrada o de salida para el módulo en particular y la columna *Secure* se indica si es seguro (Sí/1) o si no lo es (No/0):

Nombre del Slot	Nº del Slot	Input / Output	Secure
ROMEO_TRAJECTORY_GENERATION_DATA_SLOT (i)	4550 + i	Input	Sí
ROMEO_PATH_FOLLOWER_DATA_SLOT (i)	4450 + i	Input	Sí
ROMEO_SLOT_MODULE_CONTROL (i)	4000 + i	Input	Sí
ROMEO_TRAJECTORY_GENERATION_MODULE_STATE_SLOT (i)	4600 +i	Output	No

Tabla 3.4.2.1-2: Slots usados para las comunicaciones del Trajectory Generation Module

El hilo de comunicaciones en este módulo estará formado por los siguientes archivos:

- *romeoTRAJECTORY_GENERATIONcommunication.cpp* y *romeo TRAJECTORY_GENERATIONcommunication.h*: implementan la clase que se encarga de las comunicaciones con el resto de módulos. En ella, se encuentran funciones de envío de datos (que comprueban si existen datos en la sección crítica para, en caso afirmativo, colocarlos en el slot correspondiente para ser enviados por la red) y de recepción de datos (que comprueban que no exista error en las comunicaciones, para escribir los datos que se encuentran en el slot correspondiente, en la sección crítica).

La **clase *CRomeoCommunication_TRAJECTORY_GENERATION*** está formada por las siguientes funciones:

- **Init**: inicializa las comunicaciones y añade los slots necesarios. Además, calcula el ancho de banda máximo para cada slot, teniendo en cuenta el ancho de banda máximo del canal y el número de slots antes comentado. Además de dichos slots, existirán dos slots más para las propias comunicaciones internas del BBCS.

  Parámetros:

  - void.

Devuelve:

- bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveWayPointData**: recibe los puntos de referencia para el *Trajectory Generation Module* previa comprobación de que la estructura de datos que llega no es la utilizada para la comprobación de las comunicaciones a través de los slots seguros mediante la sincronización los mismos. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendWayPointData**: envía los puntos de referencia para el *Path Follower Module*. Puesto que es un slot seguro, estará constantemente enviando el dato hasta que éste se envíe correctamente. En caso de error, indica por pantalla el tipo de error producido.

  Parámetros:

  - CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

  Devuelve:

  - bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **ReceiveTrajectoryGenerationControlData**: recibe el dato de control para este módulo. Para ello, además de comprobar que el dato que le llega no es el de sincronización de los slots seguros, comprueba si el identificador del módulo al que va dirigido el dato de control (que se encuentra en la lista correspondiente a todos los datos de control enviados desde la *Module Manager Layer* a los distintos módulos), se corresponde con el identificador de módulo del *Trajectory Generation Module* (*TRAJECTORY_GENERATION_MODULE_ID*). En caso afirmativo, coloca el dato en la sección crítica. En caso de error, indica por pantalla el tipo de error producido.

Parámetros:

- CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

Devuelve:

- bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **SendTrajectoryGenerationModuleState**: envía el estado del módulo a la red.

Parámetros:

- CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

Devuelve:

- bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

- **InitCommunications**: se encarga de enviar los datos necesarios para la sincronización de los slots seguros, de forma que los primeros datos que se envíen de forma segura no se pierdan por no estar sincronizados inicialmente.

Parámetros:

- CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

Devuelve:

- bool: devuelve *true* si la función se ha ejecutado correctamente o *false* en caso contrario.

Comentar que las variables de la clase son inicializadas en el constructor de la misma.

- *romeoTRAJECTORY_GENERATIONcommunicationthread.cpp*: implementa la ejecución del hilo que se encarga de las comunicaciones de este módulo con el resto del sistema.

El código del hilo tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto a partir de la clase que implementa la sección crítica del *Trajectory Generation Module*, *CRomeoCriticalSection_TRAJECTORY_GENERATION RomeoTrajectoryGenerationCS*.
- Crea la variable global externa *end* que fue creada en el programa principal *main* y que indica la finalización del programa.
- Crea un objeto (*Block*) a partir de la clase *CCriticalSection*, que indica el bloqueo del hilo de funcionalidad (*TrajectoryGeneratorThread*).
- Lanza el hilo de comunicaciones del módulo (*CommunicationThread_TRAJECTORY_GENERATION*).
- Crea las variables y objetos locales necesarios, tanto para trabajar con el BBCS como para crear el archivo de conexiones correspondiente.
- Inicializa las variables bloqueando el hilo de funcionalidad.
- Añade los slots.
- Crea el archivo de conexiones *connectionsTRAJECTORY_GENERATION_X.conf*, donde *X* indica el identificador del robot correspondiente.
- Añade la conexión al *Relay Node* del robot, indicando tanto la dirección IP del mismo, como los puertos local y remoto del *Trajectory Generation Module*.
- Crea las conexiones usando el archivo de parámetros creado anteriormente.
- Ejecuta la función *InitCommunications()*, que se encarga de enviar los datos necesarios para la sincronización de los slots seguros, de forma que los primeros datos que se envíen de forma segura no se pierdan por no estar sincronizados inicialmente.
- Inicializa el manejador del estado del módulo, colocando en la sección crítica el estado en *MODULE_ENDED*, que indica que el módulo se encuentra parado hasta que no le llegue de la *Module Manager Layer* la orden contraria.
- Bloquea el hilo de funcionalidad.
- Comienza el bucle del que se saldrá cuando se active la variable *end*.
- Se sincronizan los datos y se espera un período de tiempo.

- Se realiza una prueba del canal de comunicaciones que indica si el canal está activo o inactivo.

- Se reciben los puntos de referencia para el *Trajectory Generation Module*.

- Se recibe el dato de control para este módulo, en caso de que desde la *Module Manager Layer* se haya enviado una nueva orden de control.

- Se ejecuta el manejador del módulo. A partir del estado del módulo y de la solicitud que llega de la *Module Manager Layer*, la función *SetModuleActionAndUpdateModuleState()* decide si el hilo de funcionalidad tiene que ser bloqueado o desbloqueado (almacenando dicha decisión en la variable *action*), y también actualiza el estado del modulo.

- Si se ha decidido el bloqueo del hilo de funcionalidad, en cuyo caso la variable *action* toma el valor *BLOCK_MODULE_FUNCTIONALITY*, se procede a bloquear el hilo.

- Si se ha decidido el desbloqueo del hilo de funcionalidad, la variable *action* toma el valor *UNBLOCK_MODULE_FUNCTIONALITY*, se procede a desbloquear el hilo.

- Se envían los puntos de referencia para el *Path Follower Module*.

- Se envía el estado del módulo a la red.

- Una vez se sale del bucle, mediante la activación de la variable *end*, se finalizan las comunicaciones del BBCS y se eliminan los canales correspondientes.

Cualquier error que se produzca en el mismo será indicado por pantalla.

3.4.2.2. Sección crítica

La clase que implementa la sección crítica en este modulo, llamada *CRomeo CriticalSection_TRAJECTORY_GENERATION* y que se encuentra en *./romeo/ril/ trajectory_generation/source/romeoTRAJECTORY_GENERATIONcriticalsection.h*, es la encargada de las comunicaciones del hilo de procesamiento (*TrajectoryGenerator Thread*) con el hilo de comunicaciones (*CommunicationThread_TRAJECTORY_ GENERATION*). En ella están las funciones *Set*, que permiten colocar en la sección crítica los datos que se manejan en el módulo, y las funciones *Get*, que permiten obtener dichos datos de la misma, las cuales vienen descritas en *./romeo/ril/trajectory_ generation/source/romeoTRAJECTORY_GENERATIONcriticalsection.cpp.*

Cabe indicar que existen dos listas que almacenan estructuras *WayPointData*, una para las que llegan al *Trajectory Generation Module* (*WayPointDataList*), y otra para las que se envían al *Path Follower Module* (*PathDataList*).

La **clase** ***CRomeoCriticalSection_TRAJECTORY_GENERATION*** está formada por las siguientes funciones:

- **SetWayPointData**: coloca en la sección crítica los puntos de referencia (*waypoints*) para el *Trajectory Generator*. Para ello, comprueba si la estructura *WayPointData* que le llega es la primera, en cuyo caso se procede a borrar la lista de estructuras *WayPointData*, para posteriormente introducirla al principio de la lista *WayPointDataList*.

  Parámetros:

  - WayPointData *point_data: variable que contiene una estructura que almacena los puntos de referencia.

  Devuelve:

  - void.

- **GetWayPointData**: obtiene de la sección crítica los puntos de referencia (*waypoints*) para el *Trajectory Generator*. Para ello, comprueba si la longitud de la lista donde se almacenan las estructuras *WayPointData* es distinta de cero, en cuyo caso toma la última estructura *WayPointData* de la lista *WayPointDataList*, borrándola de ella posteriormente.

  Parámetros:

  - WayPointData *point_data: variable que contiene una estructura que almacena los puntos de referencia.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato o *false* en caso contrario.

- **CheckIfNewWayPointDataHaveArrived**: Comprueba si han llegado nuevos puntos de referencia (*waypoints*) a la sección crítica. Para ello, comprueba si la longitud de la lista *WayPointDataList*, donde se almacenan las estructuras *WayPointData*, es distinta de cero.

  Parámetros:

  - void.

  Devuelve:

  - bool: devuelve *true* si ha llegado un nuevo punto de referencia a la sección crítica o *false* en caso contrario.

- **SetPathData**: coloca en la sección crítica el camino para el *Path Follower Module*, añadiendo la estructura *WayPointData* al principio de la lista *PathDataList*.

  Parámetros:

  - WayPointData *point_data: variable que contiene una estructura que almacena los puntos de referencia.

  Devuelve:

  - void.

- **GetPathData**: obtiene de la sección crítica el camino para el *Path Follower Module*, añadiendo la estructura *WayPointData* al principio de la lista *PathDataList*. Para ello, comprueba si la longitud de la lista es distinta de cero, en cuyo caso toma la última estructura *WayPointData* de la misma, borrándola de ella posteriormente.

  Parámetros:

  - WayPointData *point_data: variable que contiene una estructura que almacena los puntos de referencia.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetTrajectoryGenerationModuleState**: coloca en la sección crítica el estado del módulo.

  Parámetros:

  - ModuleState *ModuleStatus: variable que contiene el estado del módulo.

  Devuelve:

  - void.

- **GetTrajectoryGenerationModuleState**: obtiene de la sección crítica el estado del módulo.

  Parámetros:

  - ModuleState *ModuleStatus: variable que contiene el estado del módulo.

Devuelve:

- bool: devuelve *true* una vez obtenido el dato.

- **SetTrajectoryGenerationControlData**: coloca en la sección crítica el dato de control para este módulo. Para ello, coloca el dato de control al principio de la lista *TrajectoryGenerationControlDataList*.

  Parámetros:

  - ModuleControl *ControlData: variable que contiene una estructura que almacena el dato de control.

  Devuelve:

  - void.

- **GetTrajectoryGenerationControlData**: obtiene de la sección crítica el dato de control para este módulo. Para ello, comprueba si la longitud de la lista donde se almacenan los datos de control (*TrajectoryGeneration ControlDataList*) es distinta de cero, en cuyo caso toma el último dato de control de la lista, borrándolo de ella posteriormente.

  Parámetros:

  - ModuleControl *ControlData: variable que contiene una estructura que almacena el dato de control.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato o *false* en caso contrario.

- **SetTrajectoryGenerationFunctionalityState**: coloca en la sección crítica el estado del hilo de funcionalidad del módulo.

  Parámetros:

  - FUNCTIONALITY_STATE *state: variable que contiene una estructura que almacena el estado del hilo de funcionalidad del módulo.

  Devuelve:

  - void.

- **GetTrajectoryGenerationFunctionalityState**: obtiene de la sección crítica el estado del hilo de funcionalidad del módulo.

  Parámetros:

  - FUNCTIONALITY_STATE *state: variable que contiene una estructura que almacena el estado del hilo de funcionalidad del módulo.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetSpace**: coloca en la sección crítica el valor del parámetro *space* para el *Trajectory Generator*.

  Parámetros:

  - double *space: variable que indica la separación existente entre dos puntos consecutivos en una trayectoria generada.

  Devuelve:

  - void.

- **GetSpace**: coloca en la sección crítica el valor del parámetro *space* para el *Trajectory Generator*.

  Parámetros:

  - double *space: variable que indica la separación existente entre dos puntos consecutivos en una trayectoria generada.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

En el constructor de la clase se inicializan las variables que se van a usar, así como las listas empleadas para el almacenamiento de los *waypoints* y de los datos de control. El destructor de la clase será el encargado de eliminar la lista que contiene los datos de control para el módulo, así como las listas empleadas para el almacenamiento de los *waypoints*.

3.4.2.3. Manager

Desde la *Module Manager Layer* se enviará la señal de control al *Trajectory Generation Module*, y dependiendo de dicha señal, el manejador mediante la clase *CTrajectoryGenerationManager* (*./romeo/ril/trajectory_generation/source/romeo TRAJECTORY_GENERATIONmanager.h*), será quién decida lo que debe realizar el módulo.

En *./robot_architecture/comms/GENERAL_IMI_data.h* se encuentran definidas todas las estructuras usadas en el control y configuración de los diferentes módulos, así como las estructuras que envían todos los módulos indicando su estado.

En el apartado *"3.3.2.3. Manager"* se estudio en profundidad el *Manager* del *Path Follower Module*, por lo que se recomienda volver a dicho apartado para obtener la misma información que aquí será omitida.

Las funciones que componen la **clase *CTrajectoryGenerationManager***, son las siguientes:

- **InitManager**: se encarga de la inicialización del manejador del estado del módulo, colocando en la sección crítica dicho estado.

  Parámetros:

  - CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

  Devuelve:

  - void.

- **SetModuleActionAndUpdateModuleState**: esta función decide, a partir del estado del módulo y de la solicitud llegada desde la *Module Manager Layer*, si el hilo de funcionalidad tiene que ser bloqueado o desbloqueado, y también actualiza el estado del módulo.

  Parámetros:

  - CRomeoCriticalSection_TRAJECTORY_GENERATION *RomeoTrajectoryGenerationCS: objeto de la sección crítica correspondiente al *Trajectory Generation Module*.

  Devuelve:

  - int: variable que indica la acción a realizar sobre el hilo de funcionalidad.

Esta función tiene una estructura bastante compleja que será detallada a continuación:

1. Inicialmente el valor que devuelve es *NOTHING_TO_DO*, que indica que no se debe hacer nada sobre el hilo de funcionalidad.

2. Realiza la lectura del estado del módulo y del estado del hilo de funcionalidad.

3. Decide si bloquear o no el hilo de funcionalidad, para lo cual se contemplan dos posibles casos:

    a) Si el hilo de funcionalidad está parado (*FUNCTIONALITY_ENDED*) y el módulo está ejecutándose (*MODULE_RUNNING*), se detiene el módulo (*MODULE_ENDED*) y se bloquea el hilo de funcionalidad (*BLOCK_MODULE_FUNCTIONALITY*).

    b) Si el hilo de funcionalidad presenta errores, se aborta la ejecución del módulo (*MODULE_ABORTED*), se indica el error que se ha producido, y se vuelve a indicar que ya no existe error (*NO_ERROR_IMI*).

3. En caso de que llegue un nuevo dato de control del módulo (*START_MODULE* o *STOP_MODULE*) desde la *Module Manager Layer*:

    a) Si llega *START_MODULE* pueden ocurrir los siguientes casos:

        - Si el módulo está ejecutándose (*MODULE_RUNNING*), se indica el correspondiente error (*MODULE_RUNNING_TO_RESTART_STOP_IT_BEFORE*).

        - Si el módulo estaba detenido (*MODULE_ENDED* o *MODULE_ABORTED*) y llegan nuevos *waypoints*, se desbloquea el hilo de funcionalidad (*UNBLOCK_MODULE_FUNCTIONALITY*) y se activa el módulo (*MODULE_RUNNING*), guardando el estado del hilo (*FUNCTIONALITY_RUNNING*). En caso de que no lleguen nuevos *waypoints* no se ejecuta ninguna acción sobre el hilo de funcionalidad (*NOTHING_TO_DO*) y el módulo se queda a la espera de los mismos (*MODULE_WAITING_DATA_TO_START*).

    b) Si llega *STOP_MODULE* pueden ocurrir los siguientes casos:

        - Si el estado del módulo es finalizado (*MODULE_ENDED*) o abortado (*MODULE_ABORTED*), no se realiza ninguna acción.

- Si el módulo estaba ejecutándose (*MODULE_ RUNNING*), su estado pasará a abortado (*MODULE_ ABORTED*).

4. Actualiza el estado del módulo, devolviendo la acción a realizar sobre el hilo de funcionalidad.

- **TranslateErrorCode**: función privada de la clase que se encarga de traducir el código de error del hilo de funcionalidad a un código de error de módulo.

  Parámetros:

  - int functionality_error_code: variable que indica el tipo de error que se ha producido, y que toma los valores anteriormente descritos en la variable *error_code* perteneciente a la estructura *FUNCTIONALITY_STATE*.

  Devuelve:

  - int: variable que representa el error producido.

En el constructor de la clase se inicializarán las variables de la misma, donde el identificador del módulo será el correspondiente al *Trajectory Generation Module* (*TRAJECTORY_GENERATION_MODULE_ID*), no existirán errores (*NO_ERROR_ IMI*), y el estado del módulo será el de finalizado (*MODULE_ENDED*).

3.4.2.4. Trajectory Generator

Este hilo se encarga de generar el camino que debe seguir el ROMEO-4R. Aquí se ha optado por implementar trayectorias en línea recta, pudiéndose cambiar por trayectorias circulares con sólo modificar la función que implementa el algoritmo correspondiente (*Calculate_linear_trajectory_generator*()). Para ambos tipos de trayectorias se mantiene constante la curvatura del vehículo, siendo esta curvatura nula en el caso de la línea recta. De esta forma, si no se produjeran perturbaciones durante la ejecución de la trayectoria, no sería necesario modificar la variable de control. En la práctica siempre existirán ciertas perturbaciones, por lo que si se pretende que el vehículo siga una línea recta o circular, será necesario efectuar pequeñas correcciones en su dirección.

El problema se plantea en los extremos de los tramos rectos o los circulares. Es frecuente que el camino que describe el vehículo esté compuesto de una secuencia de tramos rectos y circulares. En el caso de uniones de tramos rectos, no existe continuidad en orientación. En el caso de uniones entre tramos rectos y circulares, puede existir continuidad en orientación pero no en curvatura.

La discontinuidad en la curvatura hace necesario un cambio brusco de la variable de control, que afecta la ejecución de la trayectoria. Este cambio es más aparente a medida que la velocidad es mayor. El problema de la generación de trayectorias debe plantearse de forma que se garantice que en toda la curva, el camino es continuo en posición, orientación y curvatura.

También conviene resaltar que, aunque no se esté considerando la evolución temporal, es necesario tener en cuenta las restricciones cinemáticas del vehículo. Si el vehículo circula con velocidades elevadas y su masa es importante, habrá que tener en cuenta las restricciones impuestas por la dinámica del mismo. Para más información véase *"Robótica: manipuladores y robots móviles"* (Aníbal Ollero).

Como se ha comentado anteriormente, este hilo implementará la generación de trayectorias mediante el uso de la línea recta, para lo cual se ha diseñado la clase *linear_trajectory_generator*, en la que se encuentran las funciones necesarias para recibir los puntos del camino a seguir, obtener las distancias y ángulos entre dos puntos consecutivos del mismo, generar los puntos intermedios separados por una distancia *space*, y que estarán en la misma línea recta, y enviar dicho camino al *Path Follower Module* para su seguimiento (ver Figura 3.4.2.4-1). Esta clase se encuentra definida en *./romeo/ril/trajectory_generation/source/TrajectoryGenerator/linear_trajectory_generator.h.*

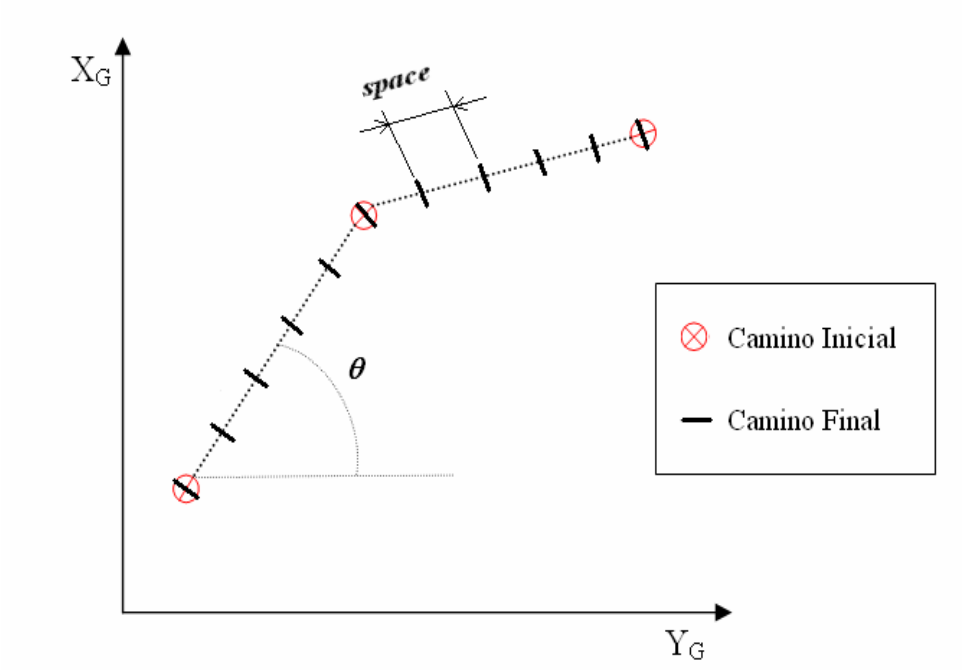


Figura 3.4.2.4-1: Generación de trayectorias en líneas rectas

Los archivos que implementan la generación de trayectorias en líneas rectas son los siguientes:

- *linear_trajectory_generator.cpp* y *linear_trajectory_generator.h*: implementan la clase referente a la generación de trayectorias en líneas rectas.

  La **clase *linear_trajectory_generator*** está formada por las siguientes funciones:

  - **ReceiveWayPointData**: recibe los puntos de referencia para el generador de trayectorias, los cuales son proporcionados por la lista de la sección crítica que contiene las estructuras *WayPointData* de entrada (*point_data_in*), es decir, los *waypoints* que una vez tratados por esta clase, darán lugar a la creación de puntos intermedios entre dos puntos consecutivos de dicha estructura, separados entre sí por una distancia *space*, los cuales serán seguidos por el vehículo realizando trayectorias en línea recta. Por tanto, si llega una nueva estructura *WayPointData*, la cual contendrá como máximo *MAX_NUMBER_WP waypoints*, ésta tendrá activadas las variables que indican que se trata de una estructura única (*first_waypoint_data* y *last_waypoint_data*), por lo que se colocará dicha estructura en una tabla de *waypoints* (*camino_inicial[]*), que contendrá el camino inicial a seguir por el vehículo, sin incluir aún los puntos intermedios anteriormente comentados, además de indicar en el estado del hilo de funcionalidad que no existen errores (*NO_ERROR_IMI*) y que el hilo esta activo (*FUNCTIONALITY_ RUNNING*).

    Parámetros:

    - void.

    Devuelve:

    - bool: devuelve *true* si el dato se ha recibido correctamente o *false* en caso contrario.

  - **Calculate_linear_trajectory_generator**: genera el camino a seguir por el ROMEO-4R (*camino_final[]*), calculando la trayectoria lineal a partir del camino inicial que le llega, y guardando dicho camino final en estructuras *WayPointData* (*point_data_out*) que serán enviadas al *Path Follower Module* para su seguimiento.

    Parámetros:

    - void.

    Devuelve:

    - void.

La estructura del algoritmo que permite el cálculo de las trayectorias es la siguiente:

1. Obtiene de la sección crítica el valor de la variable *space*, que indica la separación entre dos puntos consecutivos del camino a generar.

2. Para cada punto del camino inicial, calcula la distancia y el ángulo al siguiente punto.

3. El número de puntos intermedios entre dos puntos consecutivos del camino inicial, que darán lugar al camino final entre dichos puntos, se calcula mediante el coeficiente de la distancia existente entre ambos y la distancia de separación que se requiere entre dos puntos consecutivos del camino final (*space*).

4. Cada punto del camino final se genera por trigonometría pura a partir del punto anterior, que inicialmente siempre es el primer punto correspondiente del camino inicial, teniendo en cuenta el ángulo existente entre dos puntos consecutivos (que siempre será el mismo al calculado entre los dos puntos correspondientes del camino inicial, ya que se encontrarán en la misma línea recta) y la distancia entre ambos (*space*). La velocidad de paso en dicho punto será la misma que la del primer punto del camino inicial para el que se están obteniendo dichos puntos intermedios.

5. Los puntos intermedios generados (*camino_final[]*) se van guardando en estructuras *WayPointData* (*point_data_out*), las cuales una vez rellenas (cuando contienen un número de *MAX_NUMBER_WP waypoints*) son enviadas al *Path Follower Module*, indicando previamente si es una estructura que indica el comienzo de un nuevo camino (*first_waypoint_data=1* y *last_waypoint_data=0*) o es de transición (*first_waypoint_data=0* y *last_waypoint_data=0*).

6. El último punto del camino inicial, se introduce directamente en la última estructura que se va a enviar al *Path Follower Module*, indicando el número de *waypoints* que contiene ésta (que no tiene por que estar completa), y se le asignan los valores que indican la finalización del camino a seguir (*first_waypoint_data=0* y *last_waypoint_data=1*).

7. Una vez terminado el envío del camino completo, se indica en el estado del hilo la no existencia de errores (*NO_ERROR_IMI*) y la finalización del mismo (*FUNCTIONALITY_ENDED*).

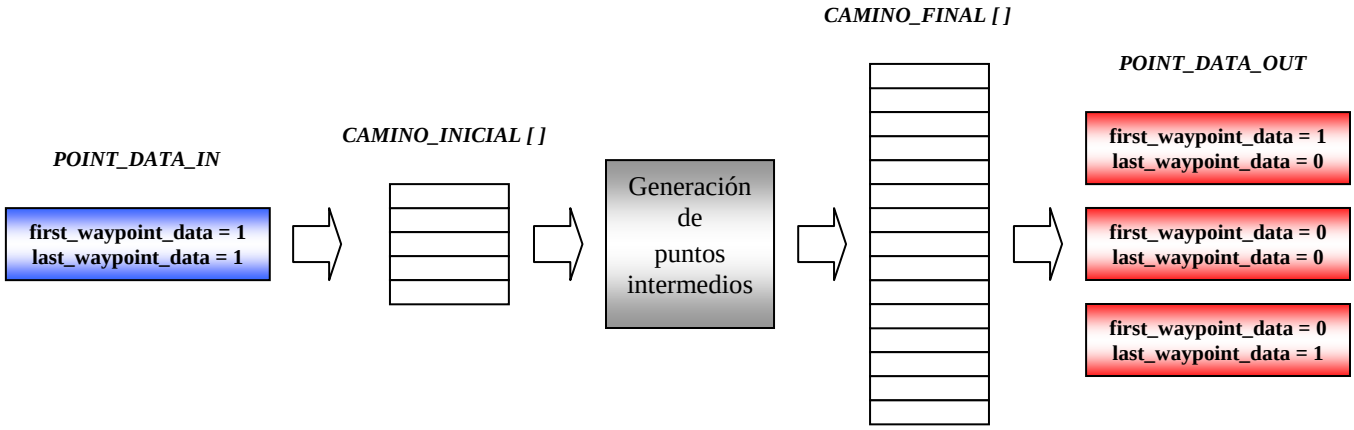


Figura 3.4.2.4-2: Esquema de generación de trayectorias

- **SendWayPointData**: coloca en una lista de la sección crítica las estructuras *WayPointData* que contienen los *waypoints* definitivos (generados mediante la función anterior) para el *Path Follower Module* (*point_data_out*), para que sean enviados a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendFunctionalityStatus**: coloca en la sección crítica el estado del hilo de funcionalidad para que sea enviado a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

Las variables de la clase son inicializadas en el constructor de la misma, donde aparecerán inicialmente con un estado del hilo que indica la no existencia de errores (*NO_ERROR_IMI*) y que se encuentra detenido (*FUNCTIONALITY_ENDED*).

- *trajectory_generator_thread.cpp*: implementa la ejecución del hilo que se encarga de la generación de trayectorias. Este hilo es independiente del tipo de trayectoria que se emplee, pudiéndose intercambiar el algoritmo que genera las trayectorias en líneas rectas con otro basado en trayectorias circulares.

  El código del hilo tiene el siguiente orden:

  - Añade las cabeceras necesarias.
  - Crea la variable global externa *end*, la cual fue creada en el programa principal *main* e indica la finalización del programa.
  - Crea un objeto (*Block*) a partir de la clase *CCriticalSection*, que implementa un mutex para el bloqueo del hilo de funcionalidad.
  - Lanza el hilo encargado de la generación de trayectorias en el módulo (*TrajectoryGeneratorThread*).
  - Crea un objeto (*LTG*) a partir de la clase *linear_trajectory_generator*, que implementa la generación de trayectorias en líneas rectas.
  - Comienza el bucle del que se saldrá cuando se active la variable *end*.
  - Se bloquea el hilo.
  - Si llega una estructura *WayPointData*, genera la trayectoria y envía el estado del hilo a la red.
  - Se desbloquea el hilo.
  - Se esperan 30 ms mediante la función *usleep()*.
  - Una vez se sale del bucle, mediante la activación de la variable *end*, se finaliza el hilo.

3.5. Temporizadores

La clase que implementa la temporización en el ROMEO-4R, *CRomeoTimer*, se encuentra en *./romeo/ril/comms/timer/timer.h*, y se encarga de marcar una cadencia en el bucle de control, evitando así que los hilos se encuentren en continua actividad, realizando un consumo de CPU innecesario.

Las funciones de la **clase *CRomeoTimer***, son las siguientes:

- **CRomeoTimer**: constructor de la clase, que se encarga crear el temporizador y de asociarle la señal que se le pasa como parámetro. Para ello, la programa como señal de tiempo real, la bloquea en la máscara de proceso, la pone como evento de temporizador y crea éste mediante la función *timer_create()*, que crea el temporizador utilizando la base de tiempo del reloj *CLOCK_REALTIME*, indicando un error en caso de no haberse creado correctamente.

  Parámetros:

  - int signal: señal que se asocia al temporizador.

  Devuelve:

  - void.

- **manejador**: función a ejecutar una vez que le llega la señal asociada al temporizador.

  Parámetros:

  - int signo , siginfo_t *datos, void *pa_na: variables que se le pasan al manejador y que actualmente no serán utilizadas, ya que el mismo solo imprimirá un mensaje por pantalla.

  Devuelve:

  - void.

- **config_timer**: función que se encarga de la configuración del temporizador, y de ponerlo en marcha. Para ello utiliza la función *timer_settime()*, que escribe el tiempo de vencimiento y arma el temporizador. Se programa para un disparo periódico y para esperar la señal de forma síncrona.

  Parámetros:

  - long int ns_cycle: tiempo de espera para el temporizador, en nanosegundos. en nuestro caso este tiempo será *T_CONTROL*, esto es, 50 ms.

  Devuelve:

  - void.

- **wait_timer**: función que espera el vencimiento del temporizador mediante la función *sigwaitinfo()*, la cual espera de forma indefinida la llegada de la señal.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **delete_timer**: función que elimina el temporizador mediante la función *timer_delete()*.

  Parámetros:

  - void.

  Devuelve:

  - void.

La implementación de un temporizador, para su uso en un hilo, es la siguiente:

```
//Crea el temporizador asignándole la señal (signal)
CRomeoTimer  Timer(signal);

//Configura el temporizador periódico de T_CONTROL y lo arma
Timer.config_timer(T_CONTROL);

while(!end)
{
      //Realiza las tareas periódicas del bucle de control
      ...
      ...
      ...

      //Espera el vencimiento del temporizador
      Timer.wait_timer();
}

//Elimina el temporizador
Timer.delete_timer();
```

Se empleará también en el simulador del *Hardware Abstraction Module* para conseguir así una simulación en tiempo real. Esto se verá en el siguiente capítulo.

3.6. Conclusiones

En el anterior capítulo se estudió la nueva arquitectura software que será utilizada para trabajar con varios robots heterogéneos. El desarrollo de la parte más abstracta de esta arquitectura, bajo la cual todos los robots son tratados de la misma forma, ya había sido desarrollado, faltando únicamente la específica al robot en cuestión sobre el que se va a trabajar, en este caso el ROMEO-4R.

En este capítulo se ha visto el principal objeto de este Proyecto Fin de Carrera, que no es otro que el desarrollo de la parte de la arquitectura específica del ROMEO-4R, la *Robot Implementation Layer* (RIL). Para describir en detalle todo lo referente a la RIL, se ha utilizado un método explicativo similar al seguido durante el desarrollo y programación de sus módulos. Así, se han descrito todos los módulos prácticamente de forma individual, aún teniendo en cuenta la similitud existente entre ellos, para permitir al lector, o futuro programador que realice mejoras en un módulo específico del la RIL, la posibilidad de conocer dicho módulo sin necesidad de recurrir a la lectura completa del Proyecto Fin de Carrera.

En el *Hardware Abstraction Module* se ha partido de la recopilación de la información de cada dispositivo del ROMEO-4R que existía de Proyectos Fin de Carrera anteriores, los cuales se basaban en la arquitectura software antigua. Por ello, se ha tomado todo lo referente a la descripción de dichos dispositivos, así como el código antiguo de los mismos, el cual, en su gran mayoría, ha sido modificado para adaptarlo a la nueva arquitectura, realizando la limpieza y depuración de mismo.

Para el resto de módulos se ha desarrollado código nuevo, ya basado en la nueva arquitectura, siguiendo una línea de programación que facilitará la adición de nuevos módulos a esta capa,

Con el desarrollo de la RIL, se ha conseguido el funcionamiento completo del ROMEO-4R bajo esta nueva arquitectura.