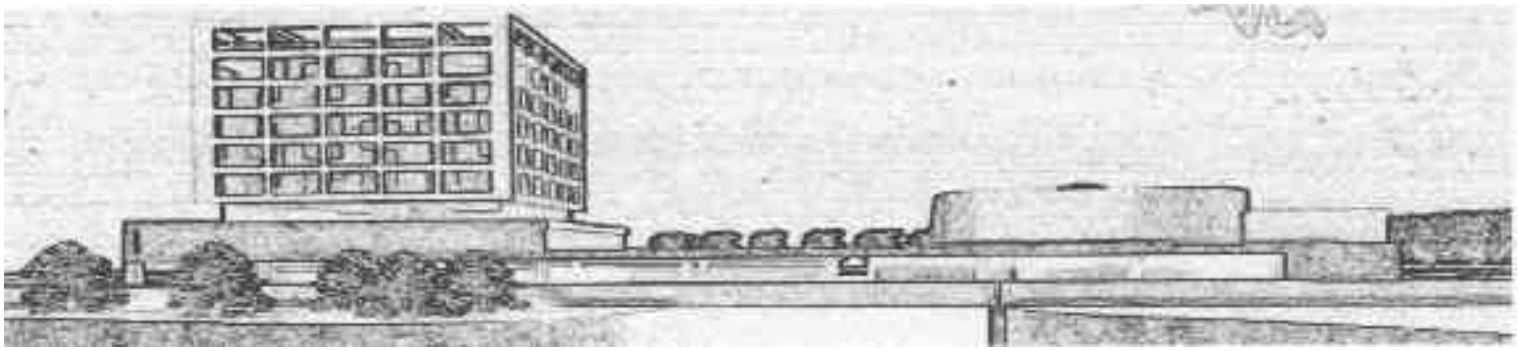
VOLUME V, NUMBER 1



REPORTS ON SYSTEMS AND COMMUNICATIONS

Open-R: un enfoque práctico

FRANCISCO MARTÍN RICO

Móstoles (Madrid), Spain, Dic. 2004
Depósito Legal: 50653-2004
ISSN: 1698-7489

Autor:
Francisco Martín Rico - fmartin@gsyc.escet.urjc.es

Índice general

Índice de figuras

Capítulo 1

Helloworld: Mi primer programa con Open-R

Como en todos los libros en los que se aprende a programar en cualquier lenguaje de programación, para realizar un "rápido desembarco" en el nuevo entorno vamos a explicar un ejemplo sencillo, que será el típico "Hello World".

Daremos por supuesto que tenemos instalado el entorno Open-R, cuyo proceso de configuración está detallado en [2].

1.1. Aplicaciones Open-R

Las aplicaciones Open-R esta formado por uno o varios objetos Open-R. Cada *objeto Open-R* es un único proceso mono-hilo, que se ejecuta concurrentemente con otros *objetos Open-R* .

Normalmente se organiza el código de los programas en distintos subdirectorios, uno para cada *objeto Open-R* . Esto hace que sea mucho más limpio el desarrollo de aplicaciones.

Dicho esto, comenzaremos paso a paso con nuestro ejemplo.

1.2. Obteniendo los fuentes de HelloWorld

En la página de Open-R podemos descargarnos una paquete llamado **samples** que contiene códigos de ejemplo. Una vez descomprimido el paquete de ejemplos, entre ellos se encuentra un directorio con el código del *objeto Open-R* "HelloWorld".

En ese directorio se encuentran tres elementos:

HelloWorld Contiene el fuente de nuestro ejemplo. Lo explicaremos con detalle más adelante.

Makefile Fichero para la compilación del código. El contenido es el siguiente:

```
1       COMPONENTS=HelloWorld ../PowerMonitor/PowerMonitor
2       INSTALLDIR=$(shell pwd)/MS
3       TARGETS=all install clean

4       .PHONY: $(TARGETS)

5       $(TARGETS):
6               for dir in $(COMPONENTS); do \
7                       (cd $$dir && $(MAKE) INSTALLDIR=$(INSTALLDIR) $@) \
8               done
```

En la línea 1 nos indica que el *objeto Open-R* **HelloWorld** va acompañado del componente **PowerMonitor** . Éste objeto siempre tiene que ser ejecutado por el robot, y por tanto tiene que formar parte de todas las aplicaciones que hagamos. Se encarga de tareas de monitorización de la batería del robot. Al igual que **HelloWorld** , **PowerMonitor** viene incluido con los ejemplos, y en este **Makefile** se debe indicar su localización. Por lo tanto, si compilamos, instalamos o limpiamos **HelloWorld** , también haremos los mismo con **PowerMonitor** .

El resto de las líneas de este fichero **Makefile** únicamente transmiten la acción a realizar a los **Makefile** de los directorios que hemos colocado como componentes en la línea 1.

MS Este directorio representa la parte referente a la aplicación del Memory Stick que se ha de introducir al robot. Cuando este compilado el código, los binarios y otros archivos de configuración de la aplicación han de ser instalados en este directorio a fin de copiarlo directamente al Memory Stick, para introducirlo en el robot.

Hay que resaltar que este Memory Stick no estará vacío antes de instalar la aplicación, es decir, que no sólo la aplicación es lo que habrá en este dispositivo, sino datos necesario para el funcionamiento del robot. La preparación de un Memory Stick desde 0 será abordada en 1.3.2.

1.3. Entendiendo HelloWorld

En el fichero de fuentes nos encontramos todo el código referente a un único *objeto Open-R* . Si visualizamos su contenido, nos encontramos con los siguientes archivos:

Makefile Fichero de compilación de este *objeto Open-R* .

helloWorld.ocf Este fichero se encarga de especificar la configuración del objeto. Tiene el siguiente formato:

object OBJECT_NAME STACK_SIZE HEAP_SIZE SCHED_PRIORITY CACHE TLB MODE

Cuya descripción es la siguiente:

OBJECT_NAME Nombre del objeto.

STACK_SIZE Tamaño de la pila. Éste tamaño no variará en tiempo de ejecución, y si nos pasamos de este tamaño el resultado no está determinado.

HEAP_SIZE Especificamos el tamaño de memoria inicial que le podemos dedicar a reservas de memoria dinámica. Si nos pasamos de esta cantidad en tiempo de ejecución, esta zona de memoria aumentará, pero esta operación será bastante costosa en tiempo.

SCHED_PRIORITY Prioridad de planificación de un proceso. Los 4 primeros bits especifican la clase de planificado; un objeto con menor clase nunca se ejecutará mientras un objeto con mayor clase se está ejecutando. La clase recomendada es 8 (128 en prioridad de planificado). Los 4 bits menores controlan el ratio de tiempo de ejecución entre objetos con la misma clase de planificado: cuanto más alto sea este número, más tiempo se ejecutará.

CACHE especificamos “cache” o “nocache” dependiendo si queremos que se usa la caché del procesador (se recomienda).

TLB especificamos “tlb” ó “notlb”. Si ponemos “tbl”, el área de memoria para el objeto está situada en el espacio de direcciones virtuales. Si ponemos “notlb”, lo pondremos en el el espacio físico de direcciones. Este valor es ignorado cuando se usa una configuración “nomemprot” (lo comentaremos más adelante). Se debe poner “tlb” si ponemos “user” en MODE.

MODE especificamos “kernel” o “user” e indica si el objeto se ejecuta en modo usuario o modo kernel. Este valor es ignorado cuando se usa una configuración “nomemprot”, ya que entonces siempre se ejecuta en modo kernel.

HelloWorldStub.cc No lo detallaremos en este capítulo. Normalmente se genera automáticamente.

HelloWorld.h y HelloWorld.cc El código de nuestro ejemplo.

1.3.1. Código del objeto

El código de **HelloWorld.h** se muestra a continuación:

```
15      #include <OPENR/OObject.h>

17      class HelloWorld : public OObject {
18      public:
19          HelloWorld();
20          virtual ~HelloWorld() {}

22          virtual OStatus DoInit   (const OSystemEvent& event);
23          virtual OStatus DoStart  (const OSystemEvent& event);
24          virtual OStatus DoStop   (const OSystemEvent& event);
25          virtual OStatus DoDestroy(const OSystemEvent& event);
26      };
```

Como vemos en el código, esto se parece bastante a un objeto C++, y realmente lo es. Todos los *objetos Open-R* heredan de **`OObject`** (línea 17) y tiene su constructor y destructor como cualquier objeto C++.

Como ya hemos comentado, la unidad mínima de ejecución es un *objeto Open-R* . A diferencia de cualquier programa en C/C++, no existe un función **`main()`** como punto de inicio a la ejecución. La ejecución está orientada a eventos, que son mandados al objeto y manejados por métodos públicos que definimos en cada objeto.

Los métodos indicados en las líneas 6-9 son los mínimos necesarios a definir en un *objeto Open-R* . Están definidos en **`OObject`** , pero han de ser redefinidos en nuestro código:

`DoInit()` Es llamado en el arranque del sistema. Debe retornar un código de error en su ejecución del tipo **`OStatus`** . Su implementación en **HelloWorld.cc** es:

```
20      OStatus
21      HelloWorld::DoInit(const OSystemEvent& event)
22      {
23          OSYSDEBUG(("HelloWorld::DoInit()\n"));
24          return oSUCCESS;
25      }
```

En la línea 23 encontramos una macro para imprimir mensajes de depuración por la consola. Usa buffering, así que puede que nos juegue alguna mala pasada si usamos trazas para detectar algún fallo de segmentación. Para que estas trazas se visualicen debemos tener definido **`OPENR_DEBUG`** . Para definirlo basta con compilar con la opción `-DOPENR_DEBUG`.

En la linea 24 devolvemos el código de error de que la ejecución se ha completado con éxito.

DoStart() Después de que **DoInit()** se haya ejecutado en todos los objetos, se ejecuta este método. Su implementación es similar en este ejemplo:

```
27      OStatus
28      HelloWorld::DoStart(const OSystemEvent& event)
29      {
30          OSYSDEBUG(("HelloWorld::DoStart()\n"));
31          OSYSPRINT(("!!! Hello World !!!\n"));
32          return oSUCCESS;
33      }
```

Podemos imaginarnos la función de la macro OSYSPRINT ¿no?. Pues sí, es una macro para imprimir por pantalla, lo que no quiere decir que no podamos usar **printf()** o **cerr()** . La ventaja es la de ser "atómico", es decir, no se mezclan los mensajes en pantalla provenientes de distintos objetos.

DoStop() Es llamado al apagarse el robot:

```
35      OStatus
36      HelloWorld::DoStop(const OSystemEvent& event)
37      {
38          OSYSDEBUG(("HelloWorld::DoStop()\n"));
39          OSYSLOG1((osyslogERROR, "Bye Bye ..."));
40          return oSUCCESS;
4°      }
```

En la línea 39 nos encontramos con una nueva macro (sí, las macros se ponen en mayúscula) OSYSLOG1. Esta macro saca por pantalla el error con su prioridad y el string que colocamos detrás. Por ejemplo, una salida podría ser:

```
[oid:80000043,prio:1]  Bye Bye ...
```

oid denota el identificador de proceso y *prio* la prioridad del error. 3 es simple información, 2 un warning y 1 un error.

DoDestroy() Es llamado al después de que todos los objetos hayan llamado a **DoStop()** . En nuestro ejemplo la implementación es:

```
43      OStatus
44      HelloWorld::DoDestroy(const OSystemEvent& event)
45      {
46         return oSUCCESS;
47      }
```

1.3.2. Iniciando el Memory Stick

Partiendo de un Memory Stick vacío:

1. copiamos el sistema base de alguno de estos 3 directorios, contando con que Open-R está instalado en **/usr/local/OPEN_R** :
   - **/USR/LOCAI/OPEN_R_SDK/OPEN_R/MS/BASIC/** . Para trabajar sin conexión wireless.
   - **/USR/LOCAI/OPEN_R_SDK/OPEN_R/MS/WLAN/** . Para trabajar con conexión wireless, pero sin obtener información desde la consola, esto es, sin ver la salida estándar.

- **/USR/LOCAI/OPEN_R_SDK/OPEN_R/MS/WCONSOLE/** . Para trabajar con conexión wireless y consola.

y luego elegir entre "memprot"(preferentemente) o "nomemprot" según si queremos protección de memoria o no.

```
cp -R /usr/local/OPEN\_R\_SDK/OPEN\_R/WCONSOLE/memprot/ /mnt/memstick
```

2. Para configurar la red en el robot únicamente es necesario editar el archivo **wlanconf.txt** que se encuentra en el directorio **/open-r/system/conf/** de MS.

   a) Cuando limpiamos el MS del perro, hay que asegurarse que el sistema base que se copia es preferentemente, para tener una consola,
   /usr/local/OPEN_R_SDK/OPEN_R/MS/WCONSOLE/nomemprot/OPEN-R/
   ó
   /usr/local/OPEN_R_SDK/OPEN_R/MS/WLAN/nomemprot/OPEN-R/
   En ambos casos activaremos la red wireless.

   b) borramos el archivo **/open-r/system/conf/wlandflt.txt** del MS, que indica la configuración por defecto de la red.

   c) Creamos `/open-r/system/conf/wlanconf.txt` en el MS. La sintaxis del archivo es *ETIQUETA=VALOR* .
   En la tabla siguiente tenemos la descripción de las etiquetas y sus valso de ejemplo para modo infraestructura y Ad-Hoc.

ETIQUETA	DESCRIPCIÓN	VALOR INFR	VALOR AD-HOC
HOSTNAME	Nombre del robot	AIBO	AIBO
ETHER_IP	Su IP	193.147.71.20	10.0.1.101
ETHER_NETMASK	La máscara de red	255.255.255.128	255.255.255.0
IP_GATEWAY	El Gateway	193.147.71.1	NADA
ESSID	El essid de la red	GSYC_WLAN_SS4	AIBONET
WEPENABLE	Encriptación 0	0 (no usar)	0
APMODE	modo	2 (auto)	2(auto)
CHANNEL	Canal(1-11)	NADA	8

1.3.3. Compilando, instalando y ejecutando

La compilación la hacemos con el compilador cruzado que es instalado con Open-R para crear ejecutables para el procesador MIPS que lleva el robot AIBO dentro.

1. Procedemos a la compilación de la aplicación:

```
~/HelloWorld$ make
~/HelloWorld$ make install
```

2. Con **`make`** compilamos tanto los fuentes de Helloworld como los de PowerMonitor, y con **`make install`** copiamos los ejecutables en el directorio **MS** .

```
~/HelloWorld$ find MS
MS/
MS/OPEN-R
MS/OPEN-R/MW
MS/OPEN-R/MW/CONF
MS/OPEN-R/MW/CONF/OBJECT.CFG
MS/OPEN-R/MW/OBJS
MS/OPEN-R/MW/OBJS/HELLO.BIN
MS/OPEN-R/MW/OBJS/POWERMON.BIN
```

Los ficheros **.BIN** son realmente los ejecutables y corresponden a un *objeto Open-R* . El fichero **OBJECT.CFG** contiene únicamente los **.BIN** que se deberán cargar al iniciar el robot.

3. Copiamos este directorio al Memory Stick

```
~/HelloWorld$cp -R MS/ /mnt/memstick
```

4. Una vez que lo tenemos todo copiado al Memory Stick, lo introducimos al robot y pulsamos el botón de arranque. Haciendo telnet al puerto 59000 podremos obtener la salida de los objetos por pantalla.

   Una última advertencia es no sacar la tarjeta ni la batería durante la ejecución. Podríamos dañar el sistema.

1.4. Ejercicio: Compilación e instalación de Helloworld en el robot AIBO

1.4.1. Enunciado

En este ejercicio, aprenderemos como compilar y ejecutar el programa ejemplo "HelloWord"que se encuentra entre los ejemplos de código que proporciona OPEN-R[1]. Este programa está preparado para imprimir el mensaje "HelloWord.[en] la consola Wireless.

Los pasos a seguir para la realización de este ejercicio, los encontramos en [2]., en el aparatado 3 titulado "Building and runnig HelloWord"

Por último, modifica el programa para que ponga el mensaje “Grupo de robótica: Hola” al encenderse el robot, y el mensaje “Grupo de robótica: Adiós” antes de apagarlo.

1.4.2. Cuestiones

Las siguientes cuestiones deben ser respondidas buscando en la documentación:

1. Antes de realizar la compilación ¿Para qué sirven cada uno de los archivos que existen en el directorio de fuentes?
2. ¿Qué clase de compilador usamos para construir los ejecutables?
3. Diferencia entre wlanconf.txt y wlandflt.txt ¿Para qué sirven?
4. ¿A qué puertos te puedes conectar al robot para obtener información?
5. ¿Qué función tiene el fichero **object.cfg** ?
6. ¿Por qué no hay una función *main()*, como es normal en un programa C/C++?

[1] http://openr.aibo.com/

Capítulo 2

Comunicaciones entre objetos Open-R

En el capítulo anterior desarrollamos una aplicación que constaba de un único *objeto Open-R* . Esto no es lo habitual cuando desarrollamos nuestras aplicaciones. Lo normal es que una aplicación esté formada por un conjunto de *objetos Open-R* que cooperan entre sí para realizar un trabajo conjunto.

En el presente capítulo vamos a explicar cómo los *objetos Open-R* se comunican entre ellos y cual es el mecanismo por el que realizan tal función.

Lo primero que debemos dejar claro es que el mecanismo es el de *paso de mensajes* . Este mensaje, que es comunicado de un objeto a otro, contiene los datos y un identificador. Cuando este mensaje llegue a su destino, el identificador seleccionará qué método del objeto destino se encargará de procesar este mensaje.

Recordemos que cada objeto tiene una sola hebra de ejecución, esto es, es *single-threaded* , por lo cual, si llega un mensaje mientras se está procesando otro, éste se encolará para ser procesado posteriormente.

2.1. Sujetos y observadores

Cuando se realiza una comunicación entre *objetos Open-R* , cada uno asume un rol que puede ser de *sujeto (subject)* u *observador (observer)* . El *subject* es quien produce los datos y el *observer* , que puede ser uno o varios, consume los datos.

Para que un *subject* pueda mandar datos a uno o varios *observers* , al menos uno de ellos debe notificar estar preparado para recibirlos. Esto lo hace mandándole un *ReadyEvent* por medio de un **`ASSERT_READY`** al *subject* y su función es indicarle al *subject* que está

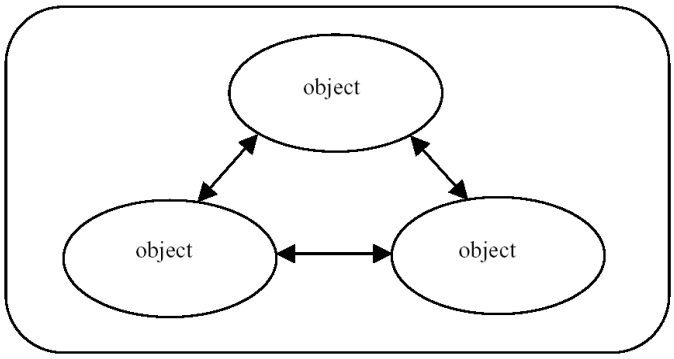


Figura 2.1: Los objetos se comunican entre sí por paso de mensajes

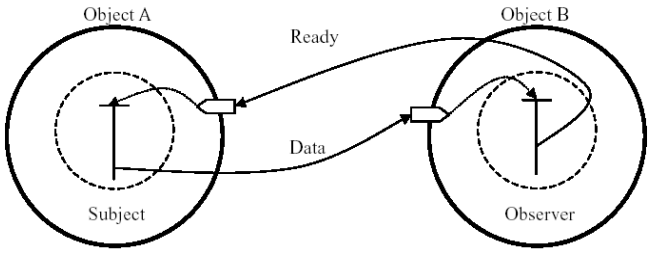


Figura 2.2: Flujo de comunicación entre objetos

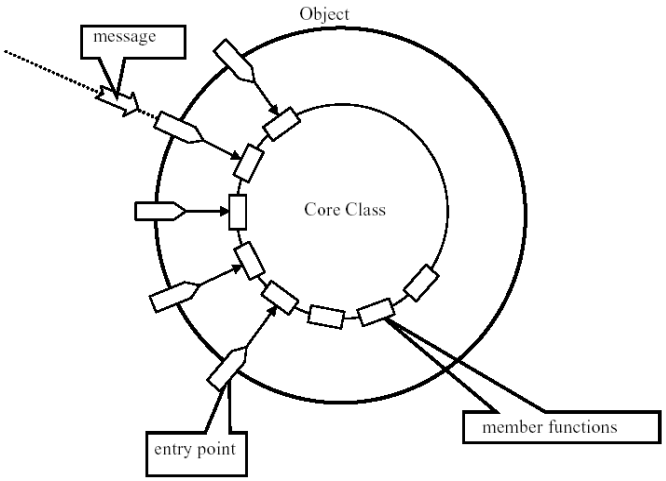


Figura 2.3: Comunicaciones en una *core class*

preparado para recibir los datos. En ese momento el *subject* manda un *NotifyEvent* con los datos a los *observers* (si hubiera más de uno). Si en algún momento el observer no desea recibir datos, manda un **`DEASSERT_READY`** al *subject* indicándole que no está preparado.

Usando estos mensajes también podremos sincronizar los distintos *objetos Open-R* a modo de *rendezvous* .

Para ilustrar la comunicación entre objetos vamos a usar el ejemplo ObjectComm que proporciona Open-R entre los programas de ejemplo. En este ejemplo dos objetos se comunican entre sí. El *subject* manda al *observer* dos mensajes, y éste los imprime en la consola.

2.2. Definiendo los manejadores de los mensajes

En Open-R una *core class* es una clase C++ que representa un objeto, y ha de ser única por cada *objeto Open-R* . Estos objetos tienen una serie de puertas (*entry points*) a las que llegan los mensajes provenientes de otros objetos. Así, un objeto puede verse como una "caja negra" donde lo único que se ve desde el exterior son estas puertas. Los identificadores que viajan en los mensajes lo que seleccionan son estas puertas, que son lo único que ven.

Dentro de cada objeto debe definirse qué método debe procesar los mensajes que llegan a cada puerta.

La declaración de estas puertas y qué métodos la procesarán se hace en un fichero llamado **stub.cfg** . Este fichero es necesario definirlo para cada objeto y será único. Durante el proceso

de compilación, el comando “stubgen2” se encargará de crear automáticamente los fuentes con los stubs necesarios para el procesamiento de los mensajes.

2.2.1. stub.cfg del SampleSubject

SampleSubject sera el objeto que hará el rol de *subject* en el ejemplo “ObjectComm”. Lo único que deberá hacer es emitir por una de sus puertas un mensaje.

```
1       ObjectName : SampleSubject
2       NumOfOSubject   : 1
3       NumOfOObserver  : 1
4       Service : "SampleSubject.SendString.char.S", null, Ready()
5       Service : "SampleSubject.DummyObserver.DoNotConnect.O", null, null
```

La sintaxis de este fichero ha de ser fija y guardar un orden: primero el nombre del objeto (debe coincidir con el que se le asigno en el .ocf), luego indicamos los subject y observers (siempre ha de haber al menos uno de cada, aunque sean tontos, como en el observer de la linea 5 de este ejemplo) y finalizamos con los servicios, que se corresponden con cada *entry point* .

Cada servicio tiene una sintaxis fija para definir cada puerta:

```
Service: Nombre_objeto.Nombre_puerta.Tipo_datos.Función, Método_1, Método_2
```

Nombre_objeto Nombre del objeto definido en la línea 1.

Nombre_puerta Identificamos el entry point. Ha de ser único en este objeto.

Tipo_datos El tipo de los datos que se enviarán.

Función Indicamos si esta puerta va a ser de *(S)ubject* o de *(O)bserver* .

Método_1 Método que manejará el resultado de la conexión. De no usarse de pondrá “null”. Es normal que no se use esta funcionalidad.

Método_2 Puede ser “null” si no es usado, y dependiendo de su función:

observers Este método es llamado cuando cuando un mensaje es recibido desde un subject.

subjects Este método es usado cuando recibimos de un *observer* un **`ASSERT_READY`** o un **`DEASSERT_READY`**

Dicho esto podemos concluir que nuestro SampleObserver va a tener una puerta real (la otra es una puerta falsa que nunca usaremos y la definimos porque debe haber al menos una de cada función) de tipo

Hay otro tipo de entradas que pueden aparecer en este fichero, pero que no contemplamos en nuestro ejemplo. Éstas son las entradas extras, y son especificadas si tenemos otro tipo de *entry points* que no son las propias de la comunicación entre objetos ordinaria. Suelen usarse para las comunicaciones TCP/IP, por lo que serán abordadas en el capítulo 6, cuando expliquemos este tipo de conexiones.

2.2.2. stub.cfg del SampleObserver

SampleObserver sera el objeto que hará el rol de observer en el ejemplo "ObjectComm". Lo único que deberá hacer es emitir **ASSERT_READY** cuando procese un mensaje, para indicar que ya está preparado para recibir otro mensaje:

```
1       ObjectName : SampleObserver
2       NumOfOSubject   : 1
3       NumOfOObserver  : 1
4       Service : "SampleObserver.DummySubject.DoNotConnect.S", null, null
5       Service : "SampleObserver.ReceiveString.char.O", null, Notify()
```

Con la explicación del SampleSubject deberemos ser capaces de entender este fichero.

2.3. Declarando las comunicaciones en la cabecera

Una vez que tenemos definidas las puertas al *objeto Open-R* y los métodos del objeto que van a procesar los mensajes que llegan por ellas, debemos declarar los métodos, así como los *observers* y *subjects* que tiene cada objeto, en el fichero de cabecera.

En el caso de **SampleSubject.h** :

```
1       #ifndef SampleSubject_h_DEFINED
2       #define SampleSubject_h_DEFINED

3       #include <OPENR/OObject.h>
4       #include <OPENR/OSubject.h>
5       #include <OPENR/OObserver.h>
6       #include "def.h"

7       class SampleSubject : public OObject {
8       public:
9           SampleSubject();
10          virtual ~SampleSubject() {}

11          OSubject*   subject[numOfSubject];
12          OObserver*  observer[numOfObserver];

13          virtual OStatus DoInit   (const OSystemEvent& event);
14          virtual OStatus DoStart  (const OSystemEvent& event);
15          virtual OStatus DoStop   (const OSystemEvent& event);
16          virtual OStatus DoDestroy(const OSystemEvent& event);

17          void Ready(const OReadyEvent& event);
18      };

19      #endif // SampleSubject_h_DEFINED
```

Si lo comparamos con el ejemplo de *objeto Open-R* sencillo del capítulo anterior, notamos ciertas diferencias:

- Incluimos las líneas 3 y 4, necesarias para los tipos de datos y funciones que usaremos en la comunicación entre objetos.
- En la línea 11 y 12 declaramos un array de **OSubject** y **OObserver** , que contendrán los datos de los *subjects* y *observers* de este objeto. El número de éstos será de **numOfSubject** , que está declarado en def.h, que se generará automáticamente a partir del stub.cfg.
- En la línea 17 declaramos el método que procesará los **ASSERT_READY** o **DEASSERT_READY** provenientes del observer. Como vemos, su argumento es un **OReadyEvent** , que contendrá el mensaje.

El caso de **SampleObserver.h** es similar, pero con el método que procesará los mensajes del *subject* , en la línea 17:

```
1       #ifndef SampleObserver_h_DEFINED
2       #define SampleObserver_h_DEFINED

3       #include <OPENR/OObject.h>
4       #include <OPENR/OSubject.h>
5       #include <OPENR/OObserver.h>
6       #include "def.h"

7       class SampleObserver : public OObject {
8       public:
9           SampleObserver();
10          virtual ~SampleObserver() {}

11          OSubject*    subject[numOfSubject];
12          OObserver*   observer[numOfObserver];

13          virtual OStatus DoInit   (const OSystemEvent& event);
14          virtual OStatus DoStart  (const OSystemEvent& event);
15          virtual OStatus DoStop   (const OSystemEvent& event);
16          virtual OStatus DoDestroy(const OSystemEvent& event);

17          void Notify(const ONotifyEvent& event);
18      };

19      #endif // SampleObserver_h_DEFINED
```

2.4. Iniciando y parando la comunicación

En el ejemplo del capítulo 1, los métodos **`DoInit()`** , **`DoStart()`** , **`DoStop()`** y **`DoDestroy()`** estaban prácticamente vacíos y no realizaban ninguna función. Pues bien, en nuestro ejemplo actual usará una serie de macros para registrarse y activarse de cara a otros objetos. En **SampleSubject.cc** :

```
8       OStatus
9       SampleSubject::DoInit(const OSystemEvent& event)
10      {
11          NEW_ALL_SUBJECT_AND_OBSERVER;
12          REGISTER_ALL_ENTRY;
13          SET_ALL_READY_AND_NOTIFY_ENTRY;
14          return oSUCCESS;
15      }

16      OStatus
17      SampleSubject::DoStart(const OSystemEvent& event)
18      {
19          ENABLE_ALL_SUBJECT;
20          ASSERT_READY_TO_ALL_OBSERVER;
21          return oSUCCESS;
22      }

23      OStatus
24      SampleSubject::DoStop(const OSystemEvent& event)
25      {
26          DISABLE_ALL_SUBJECT;
27          DEASSERT_READY_TO_ALL_OBSERVER;
28          return oSUCCESS;
29      }

30      OStatus
31      SampleSubject::DoDestroy(const OSystemEvent& event)
32      {
33          DELETE_ALL_SUBJECT_AND_OBSERVER;
34          return oSUCCESS;
```

```
35      }
```

y en **SampleObserver.cc** :

```
8       OStatus
9       SampleObserver::DoInit(const OSystemEvent& event)
10      {
11          NEW_ALL_SUBJECT_AND_OBSERVER;
12          REGISTER_ALL_ENTRY;
13          SET_ALL_READY_AND_NOTIFY_ENTRY;
14          return oSUCCESS;
15      }

16      OStatus
17      SampleObserver::DoStart(const OSystemEvent& event)
18      {
19          ENABLE_ALL_SUBJECT;
20          ASSERT_READY_TO_ALL_OBSERVER;
21          return oSUCCESS;
22      }

23      OStatus
24      SampleObserver::DoStop(const OSystemEvent& event)
25      {
26          DISABLE_ALL_SUBJECT;
27          DEASSERT_READY_TO_ALL_OBSERVER;
28          return oSUCCESS;
29      }

30      OStatus
31      SampleObserver::DoDestroy(const OSystemEvent& event)
32      {
33          DELETE_ALL_SUBJECT_AND_OBSERVER;
34          return oSUCCESS;
35      }
```

Las líneas en mayúsculas son macros destinadas a facilitar el ciclo de vida de los los *subjects* y *observers* . Realmente ellas hablan por sí mismas y no consideramos necesario comentarlas. Si queremos que un *objeto Open-R* se comunique con otro, han de ser incluidas en nuestro código.

No todas son necesarias. Por ejemplo, si no quisiéramos que los objetos emitieran un **`ASSERT_READY`** al inicio, no incluiremos la línea 20 en nuestro código.

2.5. Enviando y recibiendo datos

Ya tenemos todo listo para la comunicación, pero hace falta ahora comunicarse, es decir, los métodos **`Ready()`** y **`Notify()`** de nuestro ejemplo:

2.5.1. Envío de datos

Recordemos que al iniciarse ambos *objetos Open-R* , en la línea 20 mandamos un **`ASSERT_READY`** , con lo cual, tal mensaje llegará al objeto SampleSubject y activará el método **`Ready()`** , como se especificó en el **stub.cfg** de este objeto. La implementación en **SampleSubject.cc** de este método es el siguiente:

```
37      void
38      SampleSubject::Ready(const OReadyEvent& event)
39      {
40          OSYSPRINT(("SampleSubject::Ready() : %s\n",
```

```
41                    event.IsAssert() ? "ASSERT READY" : "DEASSERT READY"));

42          static int counter = 0;
43          char str[32];

44          if (counter == 0) {

45              strcpy(str, "!!! Hello world !!!");
46              subject[sbjSendString]->SetData(str, sizeof(str));
47              subject[sbjSendString]->NotifyObservers();

48          } else if (counter == 1) {

49              strcpy(str, "!!! Hello world again !!!");
50              subject[sbjSendString]->SetData(str, sizeof(str));
51              subject[sbjSendString]->NotifyObservers();

52          }

53          counter++;
54      }
```

- En la línea 40 y 41 obtenemos de la variable event, correspondiente al mensaje **`ASSERT_READY`** , si es un **`ASSERT_READY`** o un **`DEASSERT_READY`** .
- En la línea 46 y 50 , establecemos los datos a mandar por la puerta correspondiente al subject numerado **`sbjSendString`** . Este valor está definido en def.h (generado automáticamente) y el nombre de esta variable está formada por el nombre de la puerta definida en el **stub.cfg** , anteponiendo "sbj" si es subject o "obs" si es observer. En resumen, escribimos un string en la zona de datos de la puerta de salida SendString. Al método SetData hay que pasarle como argumentos los datos propiamente dichos, y el tamaño de éstos.
- En la línea 47 y 51 mandamos los datos que hay en esa puerta a todos los observadores conectados a esta puerta mediante el método **`NotifyObservers()`** . Podríamos haber el método **`NotifyObserver()`** para enviárselo únicamente al observador que quisiéramos. Para ver cómo usar esta función y como seleccionar el observador destino, consultar la página 18 de [3].

2.5.2. Recepción de datos

Cuando recibimos un mensaje, definimos en el stub.cfg del SampleObserver que debía ser procesado por el método **`Notify()`** :

```
37      void
38      SampleObserver::Notify(const ONotifyEvent& event)
39      {
40          const char* text = (const char *)event.Data(0);
41          OSYSPRINT(("SampleObserver::Notify() \%s\n", text));
42          observer[event.ObsIndex()]->AssertReady();
43      }
```

- En la línea 40 iniciamos la variable text con los datos que se encuentran en la zona de memoria definida por **`event.Data(0)`** , y le hacemos un casting a **`(const char *)`** . Estos son los datos que el subject envió y han llegado a la puerta que es manejada por este método. La impresión de este string la realizamos por la línea 41.
- En la línea 42, ya procesados los datos, le indicamos a los observadores atados (esto lo comentaremos a continuación) a la puerta por la que hemos recibido los datos (event.ObsIndex()), que estamos preparados para recibir más datos.

2.6. Comunicando objetos

Ya tenemos los objetos perfectamente implementados, pero ¿como indicamos qué puertas de un objeto deben ir conectadas a qué puertas de otro objeto. Para realizar esta labor debemos editar el fichero **/MS/OPEN-R/MW/CONF/CONNECT.CFG** del Memory Stick, que contiene pares de subjects-observers que han de ser conectados.

Cada línea ha de ser una comunicación y deben tener el mismo tipo. Para nuestro ejemplo, éste fichero contendrá una única línea conectando las dos puertas reales que tenemos:

```
1       SampleSubject.SendString.char.S SampleObserver.ReceiveString.char.O
```

2.7. Ejercicio: Comunicación entre objetos Open-R

2.7.1. Enunciado

En el ejercicio anterior aprendimos a hacer un único *objeto Open-R* que mostraba un mensaje por la consola. En este ejercicio haremos tres objetos Open-R independientes, pero que se han de comunicar entre sí.

Debemos hacer 3 objetos Open-R: `Pepe, Paco, Quique`. Lo único que han de hacer es "saludarse" de modo que uno mande un mensaje al otro (o a los otros) y este le responda **en orden**, mostrando lo que está haciendo por la consola. Un ejemplo sería:

```
(Mandamos un mensaje del Objeto Pepe a Paco)
Pepe: Hola Paco ¿Que tal?

(Respondemos mandando un mensaje del Objeto Paco a Pepe)
Paco: Hola, Bien ¿y tú?

(Cuando nos responda, mandamos un mensaje del Objeto Pepe a Quique)
Pepe: Bien. Hola Quique ¿Que tal?

(Cuando nos responda, mandamos un mensaje a los otros dos)
Quique: Muy bien.¿Nos vamos de fiesta?

En estos dos mensajes el orden no importa:

(Respondemos a Quique)
Pepe: Ok.
Paco: Vale.
```

2.7.2. Cuestiones

Las siguientes cuestiones deben ser respondidas buscando en la documentación:

1. ¿Qué función tiene el archivo stub.cfg y qué significa)
2. ¿Qué función tiene el archivo **connect.cfg** que existe en el Memory Stick y qué significa?
3. ¿Cómo hacemos para enviar a todos el mensaje o solo a la persona que nos ha enviado algo?
4. ¿Qué diferencia existe entre un "Entry Point" y la función miembro que maneja un mensaje"?
5. ¿Qué tipos de datos se pueden mandar en un mensaje Open-R?

Capítulo 3

Sensores

Hasta este momento hemos programado el robot de la misma manera que si programáramos una computadora personal o un dispositivo normal de sobremesa. Lo hemos hecho de una manera *virtual* y sin usar las propiedades que se le suponen a un robot, como son la percepción del entorno y la actuación.

El objeto *OVirtualRobotComm* contiene un subject que manda mensajes con la información de los sensores y otro subject con los datos de la cámara. En este capítulo aprenderemos a obtener la información de los sensores. La obtención de imágenes de la cámara será abordado en capítulos posteriores.

Para ilustrar este capítulo usaremos el ejemplo `SensorObserver7` que viene incluido con los programas de ejemplo del ERS-7. Nótese que antes de este capítulo no habíamos hecho referencia a ningún modelo en particular del robot aibo, pero la lectura el acceso a los recursos del robot, tanto actuadores como sensores, es ligeramente diferente. En primer lugar tienen sensores y actuadores diferentes, por ejemplo, el ers7 tiene 3 sensores de infrarrojos, por uno del ers210. En segundo lugar, los identificadores de los sensores y actuadores varían de un modelo a otro. Para obtener los correctos para cada modelo, referirse a los documentos [4] [5] .

3.1. Seleccionando el subject

La puerta de salida de *OVirtualRobotComm* que manda información de los sensores se llama `Sensor` y manda mensajes de tipo `OSensorFrameVectorData`. Así, si queremos obtener la información de los sensores, debemos referirnos en el fichero **connect.cfg** a este subject de la siguiente forma:

```
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S
```

3.2. El formato de datos `OSensorFrameVectorData`

`OSensorFrameVectorData` es una estructura que contiene información de **todos** los sensores del robot. Esto quiere decir que si necesitamos la información de un sensor, debemos obtener esta estructura de `OVirtualRobotComm` y seleccionar la información del sensor que os interese.

Como vemos en la figura 3.1, la estructura `OSensorFrameVectorData` contiene tres miembros:

- **`vectorInfo`** Es un `ODataVectorInfo`.
- **array de `OSensorFrameInfo`**

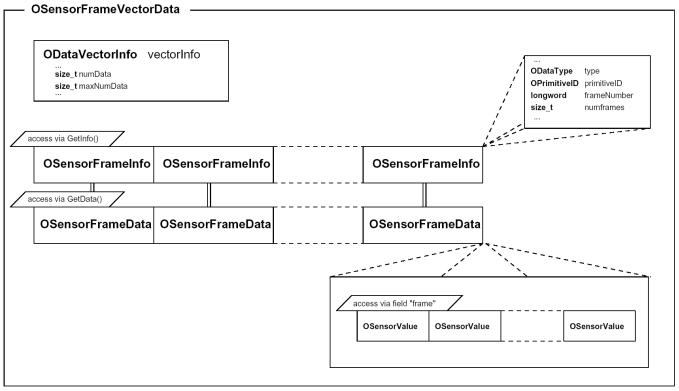


Figura 3.1: Formato de datos OSensorFrameVectorData

- **array de `OSensorFrameData`**

Los dos array tienen reservada memoria que corresponden al número de `vectorInfo.maxNumData` celdas, aunque su tamaño real es de `vectorInfo.NumData` celdas.

Cada celda de `OSensorFrameInfo` tiene su correspondiente celda en `OSensorFrameData` con el mismo índice. Cada una de estas celdas contienen información de un único sensor durante los últimos *frames* (cada *frames* puede considerarse como un *latido* representa tiempo en el hardware del aibo).

3.2.1. `OSensorFrameInfo`

Tiene dos miembros describiendo el tipo de sensor y su ID (`primitiveID`), y otros dos miembros describiendo el número del primer *frame* (`framenumber`) y el número de *frames* válidos (`numframes`).

Su valor es obtenido mediante la función `GetInfo(int index)`.

3.2.2. `OSensorFrameData`

`OSensorFrameData` es una estructura cuyo único miembro es un array de `OSensorValue` cuyo tamaño es `osensorframeMAX_FRAMES` y su contenido en los primeros `numframes` es el valor del sensor en cada uno de esos *frames*. La primera celda está numerada a `framenumber`, la segunda con `framenumber + 1` y así sucesivamente.

Su valor es obtenido mediante la función `GetData(int index)`. Esta función devuelve una estructura genérica de datos `OSensorFrameVectorData` para los valores de los sensores. Se le realizará un *casting* al tipo de datos correcto dependiendo del sensor que estemos leyendo (`OJointValue` para articulaciones, `OSwitchStatus` para switches...).

3.3. Seleccionando el sensor

En aibo, los sensores y articulaciones son llamadas *primitivas*. Cada *primitiva* puede ser referenciada por un cadena de texto (*localizador de primitivas*) que podemos encontrar en la documentación del modelo que estemos usando. A partir de esta *primitiva* y usando la función estática `OPENR::OpenPrimitive`, obtenemos su `primitiveID`. Para obtener el índice de la celda de `OSensorFrameInfo` y su correspondiente `OSensorFrameData` del sensor del que queremos extraer la información podemos recorrer el array y comparar el

primitiveID obtenido con OPENR::OpenPrimitive y el que está en el campo de ID del OSensorFrameInfo de cada sensor. El índice obtenido en el recorrido de este array debería ser almacenado para no tener que repetir el procedimiento cada vez que se quiera extraer información.

En Resumen, los pasos necesarios para conseguir el índice de un sensor son:

1. Conseguir la dirección de un sensor usando los *localizadores de primitivas*.
2. Usar OPENR::OpenPrimitive para obtener el ID de la *primitiva*.
3. Comparar este ID con los que tiene el array OSensorFrameInfo para obtener el correcto índice.
4. Guardar este índice en un array.

3.4. Ejemplo de lectura de sensores

Como comentamos anteriormente, vamos a usar el ejemplo SensorObserver7 que viene incluido con los programas de ejemplo del ers7.

3.4.1. SensorObserver7.h

La cabecera de la clase SensorObserver7 es la siguiente:

```
1       //
2       // Copyright 2003 Sony Corporation
3       //
4       // Permission to use, copy, modify, and redistribute this software for
5       // non-commercial use is hereby granted.
6       //
7       // This software is provided "as is" without warranty of any kind,
8       // either expressed or implied, including but not limited to the
9       // implied warranties of fitness for a particular purpose.
10      //
11
12      #ifndef SensorObserver7_h_DEFINED
13      #define SensorObserver7_h_DEFINED
14
15      #include <OPENR/OObject.h>
16      #include <OPENR/OSubject.h>
17      #include <OPENR/OObserver.h>
18      #include "def.h"
19
20      const int NUM_ERS7_SENSORS = 34;
21
22      const int ACC_Y         =  0;
23      const int ACC_X         =  1;
24      const int ACC_Z         =  2;
25      const int BODY_PSD      =  3;
26      const int WLAN_SW       =  4;
27      const int BACK_SW_R     =  5;
28      const int BACK_SW_M     =  6;
29      const int BACK_SW_F     =  7;
30      const int HEAD_SENSOR   =  8;
31      const int CHIN_SW       =  9;
32      const int HEAD_PSD_NEAR = 10;
33      const int HEAD_PSD_FAR  = 11;
34      const int HEAD_TILT1    = 12;
35      const int HEAD_PAN      = 13;
36      const int HEAD_TILT2    = 14;
37      const int MOUTH         = 15;
38      const int RFLEG_J1      = 16;
39      const int RFLEG_J2      = 17;
```

```
40      const int RFLEG_J3      = 18;
41      const int RFLEG_SW      = 19;
42      const int LFLEG_J1      = 20;
43      const int LFLEG_J2      = 21;
44      const int LFLEG_J3      = 22;
45      const int LFLEG_SW      = 23;
46      const int RRLEG_J1      = 24;
47      const int RRLEG_J2      = 25;
48      const int RRLEG_J3      = 26;
49      const int RRLEG_SW      = 27;
50      const int LRLEG_J1      = 28;
51      const int LRLEG_J2      = 29;
52      const int LRLEG_J3      = 30;
53      const int LRLEG_SW      = 31;
54      const int TAIL_TILT     = 32;
55      const int TAIL_PAN      = 33;
56
57      static const char* const ERS7_SENSOR_LOCATOR[] = {
58          // BODY
59          "PRM:/a1-Sensor:a1",                // ACCELEROMETER Y
60          "PRM:/a2-Sensor:a2",                // ACCELEROMETER X
61          "PRM:/a3-Sensor:a3",                // ACCELEROMETER Z
62          "PRM:/p1-Sensor:p1",                // BODY PSD
63          "PRM:/b1-Sensor:b1",                // WIRELESS LAN SWITCH
64          "PRM:/t2-Sensor:t2",                // BACK SENSOR (REAR)
65          "PRM:/t3-Sensor:t3",                // BACK SENSOR (MIDDLE)
66          "PRM:/t4-Sensor:t4",                // BACK SENSOR (FRONT)
67
68          // HEAD
69          "PRM:/r1/c1/c2/c3/t1-Sensor:t1",    // HEAD SENSOR
70          "PRM:/r1/c1/c2/c3/c4/s5-Sensor:s5", // CHIN SWITCH
71          "PRM:/r1/c1/c2/c3/p1-Sensor:p1",    // HEAD PSD (NEAR)
72          "PRM:/r1/c1/c2/c3/p2-Sensor:p2",    // HEAD PSD (FAR)
73          "PRM:/r1/c1-Joint2:11",             // HEAD TILT1
74          "PRM:/r1/c1/c2-Joint2:12",          // HEAD PAN
75          "PRM:/r1/c1/c2/c3-Joint2:13",       // HEAD TILT2
76          "PRM:/r1/c1/c2/c3/c4-Joint2:14",    // MOUTH
77
78          // RFLEG (Right Front Leg)
79          "PRM:/r4/c1-Joint2:41",             // RFLEG J1
80          "PRM:/r4/c1/c2-Joint2:42",          // RFLEG J2
81          "PRM:/r4/c1/c2/c3-Joint2:43",       // RFLEG J3
82          "PRM:/r4/c1/c2/c3/c4-Sensor:44",    // RFLEG SW
83
84          // LFLEG (Left Front Leg)
85          "PRM:/r2/c1-Joint2:21",             // LFLEG J1
86          "PRM:/r2/c1/c2-Joint2:22",          // LFLEG J2
87          "PRM:/r2/c1/c2/c3-Joint2:23",       // LFLEG J3
88          "PRM:/r2/c1/c2/c3/c4-Sensor:24",    // LFLEG SW
89
90          // RRLEG (Right Rear Leg)
91          "PRM:/r5/c1-Joint2:51",             // RRLEG J1
92          "PRM:/r5/c1/c2-Joint2:52",          // RRLEG J2
93          "PRM:/r5/c1/c2/c3-Joint2:53",       // RRLEG J3
94          "PRM:/r5/c1/c2/c3/c4-Sensor:54",    // RRLEG SW
95
96          // LRLEG (Left Rear Leg)
97          "PRM:/r3/c1-Joint2:31",             // LRLEG J1
98          "PRM:/r3/c1/c2-Joint2:32",          // LRLEG J2
99          "PRM:/r3/c1/c2/c3-Joint2:33",       // LRLEG J3
100         "PRM:/r3/c1/c2/c3/c4-Sensor:34",    // LRLEG SW
101
102         // TAIL CPC
103         "PRM:/r6/c1-Joint2:61",             // TAIL TILT
104         "PRM:/r6/c2-Joint2:62"              // TAIL PAN
105     };
106
107     class SensorObserver7 : public OObject {
108     public:
109         SensorObserver7();
110         virtual ~SensorObserver7() {}
111
112         OSubject*    subject[numOfSubject];
113         OObserver*   observer[numOfObserver];
114
```

```
115         virtual OStatus DoInit   (const OSystemEvent& event);
116         virtual OStatus DoStart  (const OSystemEvent& event);
117         virtual OStatus DoStop   (const OSystemEvent& event);
118         virtual OStatus DoDestroy(const OSystemEvent& event);
119
120         void NotifyERS7(const ONotifyEvent& event);
121
122     private:
123         void InitERS7SensorIndex(OSensorFrameVectorData* sensorVec);
124         void PrintERS7Sensor(OSensorFrameVectorData* sensorVec);
125         void PrintSensorValue(OSensorFrameVectorData* sensorVec, int index);
126         void PrintJointValue(OSensorFrameVectorData* sensorVec, int index);
127         void PrintSeparator();
128         void WaitReturnKey();
129
130         bool initSensorIndex;
131         int  ers7idx[NUM_ERS7_SENSORS];
132     };
133
134     #endif // SensorObserver7_h_DEFINED
```

- En la línea 20 se guarda el número de *primitivas* en la constante `NUM_ERS7_SENSORS`.
- En las líneas 22-55 se definen las constantes que indican la posición en el array `ers7idx` (ver debajo) que contiene el índice de las *primitivas*.
- En la línea 57-105 se definen los *localizadores de primitivas* para cada sensor.
- En la línea 130 se declara una varible que almacenará si ya se han iniciado los índices de los sensores.
- En la línea 131 se declara el array de enteros `ers7idx` que almacenarán los índices de las *primitivas*.

3.4.2. SensorObserver7.cc

La definición de la clase `SensorObserver7` es la siguiente:

```
1       //
2       // Copyright 2003 Sony Corporation
3       //
4       // Permission to use, copy, modify, and redistribute this software for
5       // non-commercial use is hereby granted.
6       //
7       // This software is provided "as is" without warranty of any kind,
8       // either expressed or implied, including but not limited to the
9       // implied warranties of fitness for a particular purpose.
10      //
11
12      #include <OPENR/ODataFormats.h>
13      #include <OPENR/OPENRAPI.h>
14      #include <OPENR/OSyslog.h>
15      #include <OPENR/core_macro.h>
16      #include "SensorObserver7.h"
17
18      SensorObserver7::SensorObserver7() : initSensorIndex(false)
19      {
20          for (int i = 0; i < NUM_ERS7_SENSORS; i++) ers7idx[i] = -1;
21      }
22
23      OStatus
24      SensorObserver7::DoInit(const OSystemEvent& event)
25      {
26          NEW_ALL_SUBJECT_AND_OBSERVER;
27          REGISTER_ALL_ENTRY;
28          SET_ALL_READY_AND_NOTIFY_ENTRY;
29          OPENR::SetMotorPower(opowerON);
```

```
30          return oSUCCESS;
31      }
32
33      OStatus
34      SensorObserver7::DoStart(const OSystemEvent& event)
35      {
36          ENABLE_ALL_SUBJECT;
37          ASSERT_READY_TO_ALL_OBSERVER;
38          return oSUCCESS;
39      }
40
41      OStatus
42      SensorObserver7::DoStop(const OSystemEvent& event)
43      {
44          DISABLE_ALL_SUBJECT;
45          DEASSERT_READY_TO_ALL_OBSERVER;
46          return oSUCCESS;
47      }
48
49      OStatus
50      SensorObserver7::DoDestroy(const OSystemEvent& event)
51      {
52          DELETE_ALL_SUBJECT_AND_OBSERVER;
53          return oSUCCESS;
54      }
55
56      void
57      SensorObserver7::NotifyERS7(const ONotifyEvent& event)
58      {
59          OSensorFrameVectorData* sensorVec = (OSensorFrameVectorData*)event.Data(0);
60
61          if (initSensorIndex == false) {
62              InitERS7SensorIndex(sensorVec);
63              initSensorIndex = true;
64          }
65
66          OSYSPRINT(("ERS-7 numData %d frameNumber %d\n",
67                     sensorVec->vectorInfo.numData, sensorVec->info[0].frameNumber));
68
69          PrintERS7Sensor(sensorVec);
70          WaitReturnKey();
71
72          observer[event.ObsIndex()]->AssertReady();
73      }
74
75      void
76      SensorObserver7::InitERS7SensorIndex(OSensorFrameVectorData* sensorVec)
77      {
78          OStatus result;
79          OPrimitiveID sensorID;
80
81          for (int i = 0; i < NUM_ERS7_SENSORS; i++) {
82
83              result = OPENR::OpenPrimitive(ERS7_SENSOR_LOCATOR[i], &sensorID);
84              if (result != oSUCCESS) {
85                  OSYSLOG1((osyslogERROR, "%s : %s %d",
86                            "SensorObserver7::InitERS7SensorIndex()",
87                            "OPENR::OpenPrimitive() FAILED", result));
88                  continue;
89              }
90
91              for (int j = 0; j < sensorVec->vectorInfo.numData; j++) {
92                  OSensorFrameInfo* info = sensorVec->GetInfo(j);
93                  if (info->primitiveID == sensorID) {
94                      ers7idx[i] = j;
95                      OSYSPRINT(("[%2d] %s\n",
96                                 ers7idx[i], ERS7_SENSOR_LOCATOR[i]));
97                      break;
98                  }
99              }
100         }
101     }
102
103     void
104     SensorObserver7::PrintERS7Sensor(OSensorFrameVectorData* sensorVec)
```

```
105     {
106         PrintSeparator();
107 
108         //
109         // BODY
110         //
111         OSYSPRINT(("ACC X      | "));
112         PrintSensorValue(sensorVec, ers7idx[ACC_X]);
113 
114         OSYSPRINT(("ACC Y      | "));
115         PrintSensorValue(sensorVec, ers7idx[ACC_Y]);
116 
117         OSYSPRINT(("ACC Z      | "));
118         PrintSensorValue(sensorVec, ers7idx[ACC_Z]);
119 
120         OSYSPRINT(("BODY PSD   | "));
121         PrintSensorValue(sensorVec, ers7idx[BODY_PSD]);
122 
123         OSYSPRINT(("WLAN SW    | "));
124         PrintSensorValue(sensorVec, ers7idx[WLAN_SW]);
125 
126         OSYSPRINT(("BACK SW F  | "));
127         PrintSensorValue(sensorVec, ers7idx[BACK_SW_F]);
128 
129         OSYSPRINT(("BACK SW M  | "));
130         PrintSensorValue(sensorVec, ers7idx[BACK_SW_M]);
131 
132         OSYSPRINT(("BACK SW R  | "));
133         PrintSensorValue(sensorVec, ers7idx[BACK_SW_R]);
134 
135         //
136         // HEAD
137         //
138         OSYSPRINT(("HEAD SENSOR| "));
139         PrintSensorValue(sensorVec, ers7idx[HEAD_SENSOR]);
140 
141         OSYSPRINT(("CHIN SW    | "));
142         PrintSensorValue(sensorVec, ers7idx[CHIN_SW]);
143 
144         OSYSPRINT(("PSD NEAR   | "));
145         PrintSensorValue(sensorVec, ers7idx[HEAD_PSD_NEAR]);
146 
147         OSYSPRINT(("PSD FAR    | "));
148         PrintSensorValue(sensorVec, ers7idx[HEAD_PSD_FAR]);
149 
150         OSYSPRINT(("HEAD TILT1 | "));
151         PrintJointValue(sensorVec, ers7idx[HEAD_TILT1]);
152 
153         OSYSPRINT(("HEAD PAN   | "));
154         PrintJointValue(sensorVec, ers7idx[HEAD_PAN]);
155 
156         OSYSPRINT(("HEAD TILT2 | "));
157         PrintJointValue(sensorVec, ers7idx[HEAD_TILT2]);
158 
159         OSYSPRINT(("MOUTH      | "));
160         PrintJointValue(sensorVec, ers7idx[MOUTH]);
161 
162         //
163         // RFLEG
164         //
165         OSYSPRINT(("RFLEG J1   | "));
166         PrintJointValue(sensorVec, ers7idx[RFLEG_J1]);
167 
168         OSYSPRINT(("RFLEG J2   | "));
169         PrintJointValue(sensorVec, ers7idx[RFLEG_J2]);
170 
171         OSYSPRINT(("RFLEG J3   | "));
172         PrintJointValue(sensorVec, ers7idx[RFLEG_J3]);
173 
174         OSYSPRINT(("RFLEG SW   | "));
175         PrintSensorValue(sensorVec, ers7idx[RFLEG_SW]);
176 
177         //
178         // LFLEG
179         //
```

```
180         OSYSPRINT(("LFLEG J1   | "));
181         PrintJointValue(sensorVec, ers7idx[LFLEG_J1]);
182
183         OSYSPRINT(("LFLEG J2   | "));
184         PrintJointValue(sensorVec, ers7idx[LFLEG_J2]);
185
186         OSYSPRINT(("LFLEG J3   | "));
187         PrintJointValue(sensorVec, ers7idx[LFLEG_J3]);
188
189         OSYSPRINT(("LFLEG SW   | "));
190         PrintSensorValue(sensorVec, ers7idx[LFLEG_SW]);
191
192         //
193         // RRLEG
194         //
195         OSYSPRINT(("RRLEG J1   | "));
196         PrintJointValue(sensorVec, ers7idx[RRLEG_J1]);
197
198         OSYSPRINT(("RRLEG J2   | "));
199         PrintJointValue(sensorVec, ers7idx[RRLEG_J2]);
200
201         OSYSPRINT(("RRLEG J3   | "));
202         PrintJointValue(sensorVec, ers7idx[RRLEG_J3]);
203
204         OSYSPRINT(("RRLEG SW   | "));
205         PrintSensorValue(sensorVec, ers7idx[RRLEG_SW]);
206
207         //
208         // LRLEG
209         //
210         OSYSPRINT(("LRLEG J1   | "));
211         PrintJointValue(sensorVec, ers7idx[LRLEG_J1]);
212
213         OSYSPRINT(("LRLEG J2   | "));
214         PrintJointValue(sensorVec, ers7idx[LRLEG_J2]);
215
216         OSYSPRINT(("LRLEG J3   | "));
217         PrintJointValue(sensorVec, ers7idx[LRLEG_J3]);
218
219         OSYSPRINT(("LRLEG SW   | "));
220         PrintSensorValue(sensorVec, ers7idx[LRLEG_SW]);
221
222         //
223         // TAIL
224         //
225         OSYSPRINT(("TAIL TILT  | "));
226         PrintJointValue(sensorVec, ers7idx[TAIL_TILT]);
227
228         OSYSPRINT(("TAIL PAN   | "));
229         PrintJointValue(sensorVec, ers7idx[TAIL_PAN]);
230     }
231
232     void
233     SensorObserver7::PrintSensorValue(OSensorFrameVectorData* sensorVec, int index)
234     {
235         if (index == -1) {
236             OSYSPRINT(("[%d] INVALID INDEX\n", index));
237             PrintSeparator();
238             return;
239         }
240
241         OSensorFrameData* data = sensorVec->GetData(index);
242         OSYSPRINT(("[%2d] val    %d %d %d %d\n",
243                    index,
244                    data->frame[0].value, data->frame[1].value,
245                    data->frame[2].value, data->frame[3].value));
246
247         OSYSPRINT(("           | "));
248         OSYSPRINT(("     sig    %d %d %d %d\n",
249                    data->frame[0].signal, data->frame[1].signal,
250                    data->frame[2].signal, data->frame[3].signal));
251
252         PrintSeparator();
253     }
254
```

```
255     void
256     SensorObserver7::PrintJointValue(OSensorFrameVectorData* sensorVec, int index)
257     {
258         if (index == -1) {
259             OSYSPRINT(("[%d] INVALID INDEX\n", index));
260             PrintSeparator();
261             return;
262         }
263
264         OSensorFrameData* data = sensorVec->GetData(index);
265         OJointValue* jval = (OJointValue*)data->frame;
266
267         OSYSPRINT(("[%2d] val    %d %d %d %d\n",
268                    index,
269                    jval[0].value, jval[1].value,
270                    jval[2].value, jval[3].value));
271
272         OSYSPRINT(("           | "));
273         OSYSPRINT(("     sig    %d %d %d %d\n",
274                    jval[0].signal, jval[1].signal,
275                    jval[2].signal, jval[3].signal));
276
277         OSYSPRINT(("           | "));
278         OSYSPRINT(("     pwm    %d %d %d %d\n",
279                    jval[0].pwmDuty, jval[1].pwmDuty,
280                    jval[2].pwmDuty, jval[3].pwmDuty));
281
282         OSYSPRINT(("           | "));
283         OSYSPRINT(("     refval %d %d %d %d\n",
284                    jval[0].refValue, jval[1].refValue,
285                    jval[2].refValue, jval[3].refValue));
286
287         OSYSPRINT(("           | "));
288         OSYSPRINT(("     refsig %d %d %d %d\n",
289                    jval[0].refSignal, jval[1].refSignal,
290                    jval[2].refSignal, jval[3].refSignal));
291
292         PrintSeparator();
293     }
294
295     void
296     SensorObserver7::PrintSeparator()
297     {
298         OSYSPRINT(("-----------+"));
299         OSYSPRINT(("------------------------------------------------------------\n"));
300     }
301
302     #ifdef WAIT_RETURN_KEY
303     #include <stdio.h> // for getchar()
304     #endif
305
306     void
307     SensorObserver7::WaitReturnKey()
308     {
309     #ifdef WAIT_RETURN_KEY
310         OSYSPRINT(("Hit return key> "));
311         char c = getchar();
312     #endif
313     }
```

- En la línea 20 iniciamos el array `ers7idx`.
- En la función `NotifyERS7` (línea 57) recogemos la información de los sensores. Si no se han iniciado los sensores (`initSensorIndex == false`), llamamos a `InitERS7SensorInd` para iniciarlos.
- La función `InitERS7SensorIndex` se define en la línea 76 y realiza las siguientes operaciones para cada sensor:
    - En la línea 83 usamos `OPENR::OpenPrimitive` para obtener el ID del sensor

especificado por su *localizador de primitiva*.

- En el bucle de la línea 91 comparamos este ID del sensor con los del array `OSensorFrameInfo` de la estructura `vectorInfo` del `OSensorFrameVectorData` recibido por primera vez. Cuando coincidan, se almacena este índice en la variable `ers7idx` (linea 94).

Una vez completado este proceso para cada sensor, ya podremos referirnos a cada sensor por su índice sin tener que realizar de nuevo la comparación.

- Cuando queramos imprimir el valor de un sensor, por ejemplo, el de aceleración en el eje x (línea 112), usaremos el método `GetData` (línea 241) con el índice del sensor para obtener el `OSensorFrameData` que contiene el valor buscado. Accediendo al *frame* del que deseemos la información y a su campo `value` (línea 244) obtendremos el valor del sensor.

- Si lo que queremos en saber el valor de una articulación el proceso es semejante, pero haciendo un casting (línea 265) del array de `OSensorFrameData` a `OJointValue`.

3.5. Ejercicio: Lectura de los valores de los sensores

3.5.1. Enunciado

En este ejercicio, vamos a establecer la comunicación entre uno de nuestros objetos de usuario y el objeto de aplicación OVirtualRobotComm para obtener los valores de los sensores de aibo en un momento dado, sacando dichos valores por la consola.

La realización de este ejercicio se realizará basándonos en el ejemplo SensorObserver7 provisto por OPEN-R. Los sensores cuyos valores sacaremos por consola serán:

- los valores de todos los joints correspondientes a la pierna posterior izquierda
- los valores de los sensores de toque o táctiles de la espalda de aibo
- el valor del sensor de toque de la planta del pie delantero derecho
- el valor del sensor infrarrojo situado en el pecho o de detección de bordes
- el valor del sensor infrarrojo situado en la cabeza (far)
- el valor del sensor de aceleración para X e Y
- los valores de los joints de la cabeza

3.5.2. Cuestiones

Las siguientes cuestiones deben ser respondidas buscando en la documentación y durante la ejecución de esta práctica:

1. ¿Que leen los infrarrojos situado en la cabeza de aibo?
2. ¿Cuales son los servicios provistos por OVirtualRobotComm ?
3. ¿Que servicios provee OVirtualRobotAudioComm?

4. ¿El objeto SensorObserver7, cuantos Subjects tiene y cuantos Observers?. Explicar el fichero stub.cfg

5. ¿Cómo se realiza la lectura de los valores de sensores y joints?

6. ¿Que son las primitivas? ¿para que sirven?

Capítulo 4

Actuadores

En este capítulo se mostrará como mandar comandos al robot para controlar sus articulaciones y los dispositivos de audio y vídeo (cámara).

El método es muy parecido al que se usa para obtener la información de los sensores.

Para ilustrar este capítulo usaremos el ejemplo `MovingLegs7` que viene incluido con los programas de ejemplo del ers7. Se ha modificado ligeramente este ejemplo para que se ejecute sólo. En el código original, se ejecutaba junto con el resto de los objetos del ejemplo BallTrackingHead7. Algunas descripciones más detalladas de este ejemplo se pueden encontrar en [7].

4.1. Seleccionando el subject

La puerta de entrada de *OVirtualRobotComm* que recibe comandos a las articulaciones o a los LEDs se llama *Effector* y manda mensajes de tipo **`OCommandVectorData`** . Al lugar donde debemos mandar los comandos lo especificamos en el fichero **connect.cfg** como:

```
OVirtualRobotComm.Effector.OCommandVectorData.O
```

4.2. El formato de datos `OCommandVectorData`

`OCommadVectorData` es una estructura que contiene comandos para los actuadores del AIBO.

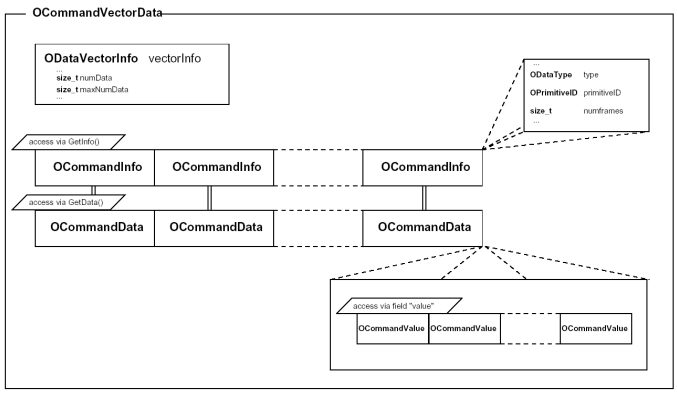


Figura 4.1: Formato de datos OCommandVectorData

Como vemos en la figura 3.1, la estructura **OCommandVectorData** contiene tres miembros:

- **vectorInfo** Es un **ODataVectorInfo** .
- **array de OCommandInfo**
- **array de OCommadData**

Los dos array tienen reservada memoria que corresponden al número de **vectorInfo.maxNumData** celdas, aunque su tamaño real es de **vectorInfo.NumData** celdas.

Cada celda de **OCommandInfo** tiene su correspondiente celda en **OCommandData** con el mismo índice. Cada una de estas celdas contienen la información y los datos para un actuador en los próximos *frames*.

4.2.1. OCommadInfo

Tiene dos miembros describiendo el tipo de actuador y su ID (**primitiveID**), y el número *frames* que vamos a comandar (**numframes**).

Su valor es obtenido mediante la función **GetInfo(int index)** .

4.2.2. OCommandData

OCommandData es una estructura cuyo único miembro es un array de **OCommandValue** , que es el tipo general de los comandos a los actuadores. En realidad, a **OCommandValue** debe hacérsele un "casting" dependiendo del tipo de actuador que vaya a ser comandado. Por ejemplo, **OCommandValue2** es el tipo para las articulaciones. Más adelante se describirán los distintos tipos dependiendo del actuador.

El tamaño del array **OCommandValue** es **ocommandMAX_FRAMES** y su contenido en los primeros **numframes** es el valor del actuador en cada uno de esos *frames*

Su valor es obtenido mediante la función **GetData(int index)** .

4.3. Inicialización de los actuadores

Como comentamos antes, el ejemplo a usar es MovingLegs7 que viene incluido con los programas de ejemplo del ers7.

4.3.1. Conseguir los ID de primitivas

Al igual que los sensores, cada actuador tiene un localizador de primitiva. Esta dirección ha de ser convertida a un **OPrimitiveID** mediante **OPENR::OpenPrimitive()** .

De **MovingLegs7.h** :

```
33      static const char* const JOINT_LOCATOR[] = {
34          "PRM:/r4/c1-Joint2:41",       // RFLEG J1 (Right Front Leg)
35          "PRM:/r4/c1/c2-Joint2:42",    // RFLEG J2
36          "PRM:/r4/c1/c2/c3-Joint2:43", // RFLEG J3
37
38          "PRM:/r2/c1-Joint2:21",       // LFLEG J1 (Left Front Leg)
39          "PRM:/r2/c1/c2-Joint2:22",    // LFLEG J2
40          "PRM:/r2/c1/c2/c3-Joint2:23", // LFLEG J3
41
42          "PRM:/r5/c1-Joint2:51",       // RRLEG J1 (Right Rear Leg)
```

```
43          "PRM:/r5/c1/c2-Joint2:52",    // RRLEG J2
44          "PRM:/r5/c1/c2/c3-Joint2:53", // RRLEG J3
45
46          "PRM:/r3/c1-Joint2:31",       // LRLEG J1 (Left Rear Leg)
47          "PRM:/r3/c1/c2-Joint2:32",    // LRLEG J2
48          "PRM:/r3/c1/c2/c3-Joint2:33"  // LRLEG J3
49      };
...
113         static const size_t NUM_JOINTS         = 12;
...
136         OPrimitiveID      jointID[NUM_JOINTS];
```

Estos localizadores corresponden a los 3 de cada una de las 4 patas.
En **MovingLegs7.cc** :

```
122     void
123     MovingLegs7::OpenPrimitives()
124     {
125         for (int i = 0; i < NUM_JOINTS; i++) {
126             OStatus result = OPENR::OpenPrimitive(JOINT_LOCATOR[i], &jointID[i]);
127             if (result != oSUCCESS) {
128                 OSYSLOG1((osyslogERROR, "%s : %s %d",
129                           "MovingLegs7::DoInit()",
130                           "OPENR::OpenPrimitive() FAILED", result));
131             }
132         }
133     }
```

4.3.2. Configurar las ganancias de las articulaciones

Para poder usar los actuadores, debemos configurar las ganacias (ganancias P, I y D) y los desplazamientos (desplazamientos P, I y D) de los motores.

Para configurarlo usamos **`OPENR::SetJointGain()`** . En **MovingLegs.h** :

```
116         static const word   J1_PGAIN = 0x0010;
117         static const word   J1_IGAIN = 0x0004;
118         static const word   J1_DGAIN = 0x0001;
119
120         static const word   J2_PGAIN = 0x000a;
121         static const word   J2_IGAIN = 0x0004;
122         static const word   J2_DGAIN = 0x0001;
123
124         static const word   J3_PGAIN = 0x0010;
125         static const word   J3_IGAIN = 0x0004;
126         static const word   J3_DGAIN = 0x0001;
127
128         static const word   PSHIFT   = 0x000e;
129         static const word   ISHIFT   = 0x0002;
130         static const word   DSHIFT   = 0x000f;
```

En **MovingLegs7.cc** :

```
167     void
168     MovingLegs7::SetJointGain()
169     {
170         for (int i = 0; i < 4; i++) {
171
172             int j1 = 3 * i;
173             int j2 = 3 * i + 1;
174             int j3 = 3 * i + 2;
175
176             OPENR::EnableJointGain(jointID[j1]);
```

```
177             OPENR::SetJointGain(jointID[j1],
178                                 J1_PGAIN, J1_IGAIN, J1_DGAIN,
179                                 PSHIFT, ISHIFT, DSHIFT);
180
181             OPENR::EnableJointGain(jointID[j2]);
182             OPENR::SetJointGain(jointID[j2],
183                                 J2_PGAIN, J2_IGAIN, J2_DGAIN,
184                                 PSHIFT, ISHIFT, DSHIFT);
185
186             OPENR::EnableJointGain(jointID[j3]);
187             OPENR::SetJointGain(jointID[j3],
188                                 J3_PGAIN, J3_IGAIN, J3_DGAIN,
189                                 PSHIFT, ISHIFT, DSHIFT);
190         }
191     }
```

4.3.3. Calibrar las articulaciones

En el arranque del AIBO, puede existir un pequeño desajuste entre la posición real de las articulaciones y la obtenida por el sensor. El programa deberá leer las articulaciones con **`OPENR::GetJointValue(OPrimitiveID primitiveID, *OJointValue value)`** y poniendo a continuación la articulación a este valor. En **MovingLegs7.cc** :

```
193     MovingResult
194     MovingLegs7::AdjustDiffJointValue()
195     {
196         OJointValue current[NUM_JOINTS];
197
198         for (int i = 0; i < NUM_JOINTS; i++) {
199             OJointValue current;
200             OPENR::GetJointValue(jointID[i], &current);
201             SetJointValue(region[0], i,
202                           degrees(current.value/1000000.0),
203                           degrees(current.value/1000000.0));
204         }
205
206         subject[sbjMove]->SetData(region[0]);
207         subject[sbjMove]->NotifyObservers();
208
209         return MOVING_FINISH;
210     }
```

En la línea 201 nos encontramos con la función **`SetJointValue()`** , que es una función definida por el usuario y será explicada más adelante. Lo que hace principalmente en mandarle una posición a un actuador, en este caso a una articulación.

4.3.4. RCRegions

En todos los ejemplos de Sony se usa un buffer para mandar los comandos a los actuadores. si no se usara este buffer, podríamos mandar los comandos al robot y esperar un mensaje *Assert Ready* para mandar el siguiente comando. Esto, aunque es correcto, podría hacernos perder mucho tiempo entre que nos devuelven el *Assert Ready* y volvemos a enviar el siguiente comando. También existe la limitación de que el tamaño del mensaje para los actuadores sería mayor del permitido para un mensaje normal Open-R.

El método usado es crear un buffer en memoria con los comandos para los actuadores. Iríamos rellenando el buffer mientras que existiera espacio libre, y el robot iría consumiendo por otra parte los comandos de este buffer.

Open-R ofrece la clase **RCRegion** que puede acceder a la memoria compartida, y guarda un contador que indica el número de objetos que están accediendo a esta región de memoria. También aporta cerrojo mutex para esta región de memoria.

Veamos en nuestro ejemplo como inicializamos este buffer. En **MovingLegs7.h** :

```
137         RCRegion*         region[NUM_COMMAND_VECTOR];
```

y en **MovingLegs7.cc** :

```
135     void
136     MovingLegs7::NewCommandVectorData()
137     {
138         OStatus result;
139         MemoryRegionID      cmdVecDataID;
140         OCommandVectorData* cmdVecData;
141         OCommandInfo*       info;
142
143         for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
144
145             result = OPENR::NewCommandVectorData(NUM_JOINTS,
146                                                  &cmdVecDataID, &cmdVecData);
147             if (result != oSUCCESS) {
148                 OSYSLOG1((osyslogERROR, "%s : %s %d",
149                           "MovingLegs7::NewCommandVectorData()",
150                           "OPENR::NewCommandVectorData() FAILED", result));
151             }
152
153             region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID,
154                                      cmdVecData->vectorInfo.offset,
155                                      (void*)cmdVecData,
156                                      cmdVecData->vectorInfo.totalSize);
157
158             cmdVecData->SetNumData(NUM_JOINTS);
159
160             for (int j = 0; j < NUM_JOINTS; j++) {
161                 info = cmdVecData->GetInfo(j);
162                 info->Set(odataJOINT_COMMAND2, jointID[j], ocommandMAX_FRAMES);
163             }
164         }
165     }
```

Se crea un buffer de tamaño **NUM_COMMAND_VECTOR** llamado **region** , y cada elemento del buffer tiene un **CommandVectorData** . La forma de crear cada elemento del buffer es la siguiente:

- En la línea 145 se crea un **CommandVectorData** que controla **NUM_JOINTS** actuadores. Este **CommandVectorData** es representado por su **OCommandVectorData** y su **OCommandInfo** .
- En la línea 153 se crea un nuevo elemento del buffer con el `CommandVectorData` que acabamos de crear. La información que necesitamos para crearlo está en el mismo `CommandVectorData`.
- De la línea 157-165 se configura el **CommandVectorData** que se encuentra en el buffer.

4.4. Dándole valor a un actuador

Una vez creado el buffer de **RCRegion** s, ya es posible darle valor a las articulaciones. Vemos el código de la función **MoveToBroadBase()** para ilustrarlo. De **MovingLegs7.h** :

```
51      const double BROADBASE_ANGLE[] = {
52          120,  // RFLEG J1
53          90,   // RFLEG J2
54          30,   // RFLEG J3
55
56          120,  // LFLEG J1
57          90,   // LFLEG J2
58          30,   // LFLEG J3
59
60          -120, // RRLEG J1
61          70,   // RRLEG J2
62          30,   // RRLEG J3
63
64          -120, // LRLEG J1
65          70,   // LRLEG J2
66          30    // LRLEG J3
67      };

....

132         static const int BROADBASE_MAX_COUNTER = 24; // 128ms * 24 = 3072ms
```

Esto define los ángulos a los que se deben poner cada una de las articulaciones de las patas. En **MovingLegs7.cc** :

```
212     MovingResult
213     MovingLegs7::MoveToBroadBase()
214     {
215         static int counter = -1;
216         static double start[NUM_JOINTS];
217         static double delta[NUM_JOINTS];
218         double ndiv = (double)BROADBASE_MAX_COUNTER;
219
220         if (counter == -1) {
221
222             for (int i = 0; i < NUM_JOINTS; i++) {
223                 OJointValue current;
224                 OPENR::GetJointValue(jointID[i], &current);
225                 start[i] = degrees(current.value/1000000.0);
226                 delta[i] = (BROADBASE_ANGLE[i] - start[i]) / ndiv;
227             }
228
229             counter = 0;
230
231             RCRegion* rgn = FindFreeRegion();
232             for (int i = 0; i < NUM_JOINTS; i++) {
233                 SetJointValue(rgn, i, start[i], start[i] + delta[i]);
234                 start[i] += delta[i];
235             }
236
237             subject[sbjMove]->SetData(rgn);
238             counter ++;
239         }
240
241         RCRegion* rgn = FindFreeRegion();
242         for (int i = 0; i < NUM_JOINTS; i++) {
243             SetJointValue(rgn, i, start[i], start[i] + delta[i]);
244             start[i] += delta[i];
245         }
246
247         subject[sbjMove]->SetData(rgn);
248         subject[sbjMove]->NotifyObservers();
```

```
249
250         counter++;
251         return (counter == BROADBASE_MAX_COUNTER) ? MOVING_FINISH : MOVING_CONT;
252     }

.....

285     RCRegion*
286     MovingLegs7::FindFreeRegion()
287     {
288         for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
289             if (region[i]->NumberOfReference() == 1) return region[i];
290         }
291
292         return 0;
293     }
294
295     void
296     MovingLegs7::SetJointValue(RCRegion* rgn, int idx, double start, double end)
297     {
298         OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();
299
300         OCommandInfo* info = cmdVecData->GetInfo(idx);
301         info->Set(odataJOINT_COMMAND2, jointID[idx], ocommandMAX_FRAMES);
302
303         OCommandData* data = cmdVecData->GetData(idx);
304         OJointCommandValue2* jval = (OJointCommandValue2*)data->value;
305
306         double delta = end - start;
307         for (int i = 0; i < ocommandMAX_FRAMES; i++) {
308             double dval = start + (delta * i) / (double)ocommandMAX_FRAMES;
309             jval[i].value = oradians(dval);
310         }
311     }
```

En este ejemplo, se lleva un contador para diferenciar la primera vez que se le manda un movimiento al robot, de las siguiente veces. Dada la posición inicial y la final, se definen la posición de las articulaciones en cada uno de los **`BROADBASE_MAX_COUNTER`** pasos en los que vamos a dividir el movimiento, para hacerlo con suavidad en el tiempo que hemos elegido (128ms que contiene un **`OCommandVectorData`** por 24 pasos = 3072ms.).

En cada uno de los pasos:

- Lo primero que hacemos (línea 231 y 241) es buscar una posición del buffer donde podemos escribir el movimiento. Esto lo hacemos por la función **`FindFreeRegion`** , que devuelve la primera posición que solo está referenciada por él. Es decir, que o ya ha sido consumida por el robot, o aún no se ha usado.

- Una vez obtenido una referencia a dónde debemos escribir, en la linea 233 y 243, por cada articulación, le damos el valor que hemos calculado según es paso en el movimiento en el que estamos.

- En la función **`SetJointValue`** volvemos a dividir los ángulos a mover entre **`ocommandMAX_FRAM`** , para definir la posición en cada *frame*. Escribimos en la posición del buffer **`rgn`** , los valores de las articulaciones, asignando en el campo **`value`** de cada una, el valor adecuado.

4.5. Poniendo todo junto

Pero, ¿como se junta todo esto?. Aún no hemos contado como se pone todo esto en marcha. Pongámoslo todo junto y analicemos la estructura general:

4.5.1. MovinLegs7.h completo

```
1       //
2       // Copyright 2003 Sony Corporation
3       //
4       // Permission to use, copy, modify, and redistribute this software for
5       // non-commercial use is hereby granted.
6       //
7       // This software is provided "as is" without warranty of any kind,
8       // either expressed or implied, including but not limited to the
9       // implied warranties of fitness for a particular purpose.
10      //
11
12      #ifndef MovingLegs7_h_DEFINED
13      #define MovingLegs7_h_DEFINED
14
15      #include <OPENR/OObject.h>
16      #include <OPENR/OSubject.h>
17      #include <OPENR/OObserver.h>
18      #include "def.h"
19
20      enum MovingLegs7State {
21          MLS_IDLE,
22          MLS_START,
23          MLS_ADJUSTING_DIFF_JOINT_VALUE,
24          MLS_MOVING_TO_BROADBASE,
25          MLS_MOVING_TO_SLEEPING
26      };
27
28      enum MovingResult {
29          MOVING_CONT,
30          MOVING_FINISH
31      };
32
33      static const char* const JOINT_LOCATOR[] = {
34          "PRM:/r4/c1-Joint2:41",       // RFLEG J1 (Right Front Leg)
35          "PRM:/r4/c1/c2-Joint2:42",    // RFLEG J2
36          "PRM:/r4/c1/c2/c3-Joint2:43", // RFLEG J3
37
38          "PRM:/r2/c1-Joint2:21",       // LFLEG J1 (Left Front Leg)
39          "PRM:/r2/c1/c2-Joint2:22",    // LFLEG J2
40          "PRM:/r2/c1/c2/c3-Joint2:23", // LFLEG J3
41
42          "PRM:/r5/c1-Joint2:51",       // RRLEG J1 (Right Rear Leg)
43          "PRM:/r5/c1/c2-Joint2:52",    // RRLEG J2
44          "PRM:/r5/c1/c2/c3-Joint2:53", // RRLEG J3
45
46          "PRM:/r3/c1-Joint2:31",       // LRLEG J1 (Left Rear Leg)
47          "PRM:/r3/c1/c2-Joint2:32",    // LRLEG J2
48          "PRM:/r3/c1/c2/c3-Joint2:33"  // LRLEG J3
49      };
50
51      const double BROADBASE_ANGLE[] = {
52          120,  // RFLEG J1
53          90,   // RFLEG J2
54          30,   // RFLEG J3
55
56          120,  // LFLEG J1
57          90,   // LFLEG J2
58          30,   // LFLEG J3
59
60          -120, // RRLEG J1
61          70,   // RRLEG J2
62          30,   // RRLEG J3
63
64          -120, // LRLEG J1
65          70,   // LRLEG J2
66          30    // LRLEG J3
67      };
68
69      const double SLEEPING_ANGLE[] = {
70          59,   // RFLEG J1
71          0,    // RFLEG J2
72          30,   // RFLEG J3
```

```
73
74          59,   // LFLEG J1
75          0,    // LFLEG J2
76          30,   // LFLEG J3
77
78          -119, // RRLEG J1
79          4,    // RRLEG J2
80          122,  // RRLEG J3
81
82          -119, // LRLEG J1
83          4,    // LRLEG J2
84          122   // LRLEG J3
85      };
86
87      class MovingLegs7 : public OObject {
88      public:
89          MovingLegs7();
90          virtual ~MovingLegs7() {}
91
92          OSubject*  subject[numOfSubject];
93          OObserver* observer[numOfObserver];
94
95          virtual OStatus DoInit   (const OSystemEvent& event);
96          virtual OStatus DoStart  (const OSystemEvent& event);
97          virtual OStatus DoStop   (const OSystemEvent& event);
98          virtual OStatus DoDestroy(const OSystemEvent& event);
99
100         void Ready(const OReadyEvent& event);
101
102     private:
103         void          OpenPrimitives();
104         void          NewCommandVectorData();
105         void          SetJointGain();
106         MovingResult  AdjustDiffJointValue();
107         MovingResult  MoveToBroadBase();
108         MovingResult  MoveToSleeping();
109
110         RCRegion* FindFreeRegion();
111         void SetJointValue(RCRegion* rgn, int idx, double start, double end);
112
113         static const size_t NUM_JOINTS         = 12;
114         static const size_t NUM_COMMAND_VECTOR = 2;
115
116         static const word   J1_PGAIN = 0x0010;
117         static const word   J1_IGAIN = 0x0004;
118         static const word   J1_DGAIN = 0x0001;
119
120         static const word   J2_PGAIN = 0x000a;
121         static const word   J2_IGAIN = 0x0004;
122         static const word   J2_DGAIN = 0x0001;
123
124         static const word   J3_PGAIN = 0x0010;
125         static const word   J3_IGAIN = 0x0004;
126         static const word   J3_DGAIN = 0x0001;
127
128         static const word   PSHIFT   = 0x000e;
129         static const word   ISHIFT   = 0x0002;
130         static const word   DSHIFT   = 0x000f;
131
132         static const int BROADBASE_MAX_COUNTER = 24; // 128ms * 24 = 3072ms
133         static const int SLEEPING_MAX_COUNTER  = 24; // 128ms * 24 = 3072ms
134
135         MovingLegs7State  movingLegsState;
136         OPrimitiveID      jointID[NUM_JOINTS];
137         RCRegion*         region[NUM_COMMAND_VECTOR];
138     };
139
140     #endif // MovingLegs7_h_DEFINED
```

Lo principal es que el proceso de este objeto se divide en 5 estados consecutivos (línea 20) que indica lo que hay que hacer en cada momento.

4.5.2. MovinLegs7.cc completo

```
1       //
2       // Copyright 2003 Sony Corporation
3       //
4       // Permission to use, copy, modify, and redistribute this software for
5       // non-commercial use is hereby granted.
6       //
7       // This software is provided "as is" without warranty of any kind,
8       // either expressed or implied, including but not limited to the
9       // implied warranties of fitness for a particular purpose.
10      //
11
12      #include <math.h>
13      #include <OPENR/OPENRAPI.h>
14      #include <OPENR/OUnits.h>
15      #include <OPENR/OSyslog.h>
16      #include <OPENR/core_macro.h>
17      #include "MovingLegs7.h"
18
19      MovingLegs7::MovingLegs7() : movingLegsState(MLS_IDLE)
20      {
21      }
22
23      OStatus
24      MovingLegs7::DoInit(const OSystemEvent& event)
25      {
26          OSYSDEBUG(("MovingLegs7::DoInit()\n"));
27
28          NEW_ALL_SUBJECT_AND_OBSERVER;
29          REGISTER_ALL_ENTRY;
30          SET_ALL_READY_AND_NOTIFY_ENTRY;
31
32          OpenPrimitives();
33          NewCommandVectorData();
34
35          //
36          OPENR::SetMotorPower(opowerON);// is executed in blinkingLED.
37          // So, it isn't necessary here.
38          //
39
40          return oSUCCESS;
41      }
42
43      OStatus
44      MovingLegs7::DoStart(const OSystemEvent& event)
45      {
46          OSYSDEBUG(("MovingLegs7::DoStart()\n"));
47
48          if (subject[sbjMove]->IsReady() == true) {
49              AdjustDiffJointValue();
50              movingLegsState = MLS_ADJUSTING_DIFF_JOINT_VALUE;
51          } else {
52              movingLegsState = MLS_START;
53          }
54
55          ENABLE_ALL_SUBJECT;
56          ASSERT_READY_TO_ALL_OBSERVER;
57
58          return oSUCCESS;
59      }
60
61      OStatus
62      MovingLegs7::DoStop(const OSystemEvent& event)
63      {
64          OSYSDEBUG(("MovingLegs7::DoStop()\n"));
65
66          movingLegsState = MLS_IDLE;
67
68          DISABLE_ALL_SUBJECT;
69          DEASSERT_READY_TO_ALL_OBSERVER;
70
71          return oSUCCESS;
72      }
```

```
73
74      OStatus
75      MovingLegs7::DoDestroy(const OSystemEvent& event)
76      {
77          DELETE_ALL_SUBJECT_AND_OBSERVER;
78          return oSUCCESS;
79      }
80
81      void
82      MovingLegs7::Ready(const OReadyEvent& event)
83      {
84          OSYSDEBUG(("MovingLegs7::Ready()\n"));
85
86          if (movingLegsState == MLS_IDLE) {
87
88              OSYSDEBUG(("MLS_IDLE\n"));
89              ; // do nothing
90
91          } else if (movingLegsState == MLS_START) {
92
93              OSYSDEBUG(("MLS_START\n"));
94              AdjustDiffJointValue();
95              movingLegsState = MLS_ADJUSTING_DIFF_JOINT_VALUE;
96
97          } else if (movingLegsState == MLS_ADJUSTING_DIFF_JOINT_VALUE) {
98
99              OSYSDEBUG(("MLS_ADJUSTING_DIFF_JOINT_VALUE\n"));
100             SetJointGain();
101             MovingResult r = MoveToBroadBase();
102             movingLegsState = MLS_MOVING_TO_BROADBASE;
103
104         } else if (movingLegsState == MLS_MOVING_TO_BROADBASE) {
105
106             OSYSDEBUG(("MLS_MOVING_TO_BROADBASE\n"));
107             MovingResult r = MoveToBroadBase();
108             if (r == MOVING_FINISH) {
109                 movingLegsState = MLS_MOVING_TO_SLEEPING;
110             }
111
112         } else if (movingLegsState == MLS_MOVING_TO_SLEEPING) {
113
114             OSYSDEBUG(("MLS_MOVING_TO_SLEEPING\n"));
115             MovingResult r = MoveToSleeping();
116             if (r == MOVING_FINISH) {
117                 movingLegsState = MLS_IDLE;
118             }
119         }
120     }
121
122     void
123     MovingLegs7::OpenPrimitives()
124     {
125         for (int i = 0; i < NUM_JOINTS; i++) {
126             OStatus result = OPENR::OpenPrimitive(JOINT_LOCATOR[i], &jointID[i]);
127             if (result != oSUCCESS) {
128                 OSYSLOG1((osyslogERROR, "%s : %s %d",
129                           "MovingLegs7::DoInit()",
130                           "OPENR::OpenPrimitive() FAILED", result));
131             }
132         }
133     }
134
135     void
136     MovingLegs7::NewCommandVectorData()
137     {
138         OStatus result;
139         MemoryRegionID      cmdVecDataID;
140         OCommandVectorData* cmdVecData;
141         OCommandInfo*       info;
142
143         for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
144
145             result = OPENR::NewCommandVectorData(NUM_JOINTS,
146                                                  &cmdVecDataID, &cmdVecData);
147             if (result != oSUCCESS) {
```

```
148                 OSYSLOG1((osyslogERROR, "%s : %s %d",
149                           "MovingLegs7::NewCommandVectorData()",
150                           "OPENR::NewCommandVectorData() FAILED", result));
151             }
152 
153             region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID,
154                                      cmdVecData->vectorInfo.offset,
155                                      (void*)cmdVecData,
156                                      cmdVecData->vectorInfo.totalSize);
157 
158             cmdVecData->SetNumData(NUM_JOINTS);
159 
160             for (int j = 0; j < NUM_JOINTS; j++) {
161                 info = cmdVecData->GetInfo(j);
162                 info->Set(odataJOINT_COMMAND2, jointID[j], ocommandMAX_FRAMES);
163             }
164         }
165     }
166 
167     void
168     MovingLegs7::SetJointGain()
169     {
170         for (int i = 0; i < 4; i++) {
171 
172             int j1 = 3 * i;
173             int j2 = 3 * i + 1;
174             int j3 = 3 * i + 2;
175 
176             OPENR::EnableJointGain(jointID[j1]);
177             OPENR::SetJointGain(jointID[j1],
178                                 J1_PGAIN, J1_IGAIN, J1_DGAIN,
179                                 PSHIFT, ISHIFT, DSHIFT);
180 
181             OPENR::EnableJointGain(jointID[j2]);
182             OPENR::SetJointGain(jointID[j2],
183                                 J2_PGAIN, J2_IGAIN, J2_DGAIN,
184                                 PSHIFT, ISHIFT, DSHIFT);
185 
186             OPENR::EnableJointGain(jointID[j3]);
187             OPENR::SetJointGain(jointID[j3],
188                                 J3_PGAIN, J3_IGAIN, J3_DGAIN,
189                                 PSHIFT, ISHIFT, DSHIFT);
190         }
191     }
192 
193     MovingResult
194     MovingLegs7::AdjustDiffJointValue()
195     {
196         OJointValue current[NUM_JOINTS];
197 
198         for (int i = 0; i < NUM_JOINTS; i++) {
199             OJointValue current;
200             OPENR::GetJointValue(jointID[i], &current);
201             SetJointValue(region[0], i,
202                           degrees(current.value/1000000.0),
203                           degrees(current.value/1000000.0));
204         }
205 
206         subject[sbjMove]->SetData(region[0]);
207         subject[sbjMove]->NotifyObservers();
208 
209         return MOVING_FINISH;
210     }
211 
212     MovingResult
213     MovingLegs7::MoveToBroadBase()
214     {
215         static int counter = -1;
216         static double start[NUM_JOINTS];
217         static double delta[NUM_JOINTS];
218         double ndiv = (double)BROADBASE_MAX_COUNTER;
219 
220         if (counter == -1) {
221 
222             for (int i = 0; i < NUM_JOINTS; i++) {
```

```
223                 OJointValue current;
224                 OPENR::GetJointValue(jointID[i], &current);
225                 start[i] = degrees(current.value/1000000.0);
226                 delta[i] = (BROADBASE_ANGLE[i] - start[i]) / ndiv;
227             }
228 
229             counter = 0;
230 
231             RCRegion* rgn = FindFreeRegion();
232             for (int i = 0; i < NUM_JOINTS; i++) {
233                 SetJointValue(rgn, i, start[i], start[i] + delta[i]);
234                 start[i] += delta[i];
235             }
236 
237             subject[sbjMove]->SetData(rgn);
238             counter ++;
239         }
240 
241         RCRegion* rgn = FindFreeRegion();
242         for (int i = 0; i < NUM_JOINTS; i++) {
243             SetJointValue(rgn, i, start[i], start[i] + delta[i]);
244             start[i] += delta[i];
245         }
246 
247         subject[sbjMove]->SetData(rgn);
248         subject[sbjMove]->NotifyObservers();
249 
250         counter++;
251         return (counter == BROADBASE_MAX_COUNTER) ? MOVING_FINISH : MOVING_CONT;
252     }
253 
254     MovingResult
255     MovingLegs7::MoveToSleeping()
256     {
257         static int counter = -1;
258         static double start[NUM_JOINTS];
259         static double delta[NUM_JOINTS];
260         double ndiv = (double)SLEEPING_MAX_COUNTER;
261 
262         if (counter == -1) {
263 
264             for (int i = 0; i < NUM_JOINTS; i++) {
265                 start[i] = BROADBASE_ANGLE[i];
266                 delta[i] = (SLEEPING_ANGLE[i] - start[i]) / ndiv;
267             }
268 
269             counter = 0;
270         }
271 
272         RCRegion* rgn = FindFreeRegion();
273         for (int i = 0; i < NUM_JOINTS; i++) {
274             SetJointValue(rgn, i, start[i], start[i] + delta[i]);
275             start[i] += delta[i];
276         }
277 
278         subject[sbjMove]->SetData(rgn);
279         subject[sbjMove]->NotifyObservers();
280 
281         counter++;
282         return (counter == SLEEPING_MAX_COUNTER) ? MOVING_FINISH : MOVING_CONT;
283     }
284 
285     RCRegion*
286     MovingLegs7::FindFreeRegion()
287     {
288         for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
289             if (region[i]->NumberOfReference() == 1) return region[i];
290         }
291 
292         return 0;
293     }
294 
295     void
296     MovingLegs7::SetJointValue(RCRegion* rgn, int idx, double start, double end)
297     {
```

```
298         OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();
299
300         OCommandInfo* info = cmdVecData->GetInfo(idx);
301         info->Set(odataJOINT_COMMAND2, jointID[idx], ocommandMAX_FRAMES);
302
303         OCommandData* data = cmdVecData->GetData(idx);
304         OJointCommandValue2* jval = (OJointCommandValue2*)data->value;
305
306         double delta = end - start;
307         for (int i = 0; i < ocommandMAX_FRAMES; i++) {
308             double dval = start + (delta * i) / (double)ocommandMAX_FRAMES;
309             jval[i].value = oradians(dval);
310         }
311     }
```

- En la función **`DoInit()`** iniciamos las primitivas y creamos el buffer (linea 31-32). No se nos olvide encender los motores (línea 36). Si no, el robot no se moverá.
- En la función **`DoStart()`** calibramos las articulaciones si el robot está preparado.
- En la función **`Ready()`** controlamos los cambios de estado del objeto.

4.6. Ejercicio: Movimiento de actuadores

4.6.1. Enunciado

Consistirá en mover todas las articulaciones de una pata cualquiera del robot. Una detrás de otra, se moverá cada una de las articulaciones a lo largo de todos los valores posibles para cada una de ellas, en el orden deseado por el alumno.

4.6.2. Cuestiones

Las siguientes cuestiones deben ser respondidas buscando en la documentación y durante la ejecución de esta práctica:

1. ¿Que es una RCRegion y para qué piensas que se puede usar?
2. ¿Cúales son los valores máximos de las articulaciones?
3. ¿Como conseguir un movimiento suave?
4. ¿Cómo es en este caso la comunicación con OVirtualRobotComm?. Explicar el fichero stub.cfg
5. ¿A través de que estructura de datos estableces los valores deseados para los joints? ¿Que campos contiene?
6. ¿La estructura de las respuesta de la anterior pregunta... es siempre igual independientemente del actuador cuyo valor queramos modificar?. Razona tu respuesta.

Capítulo 5

Visión en AIBO

En el capítulo 3 aprendimos la manera de obtener la información de los sensores del robot. El sensor más complejo del aibo, la cámara, no fue descrito en ese capitulo debido a la complejidad de su manejo y a la riqueza de información que aporta.

En este capítulo aprenderemos a obtener la imagen de la cámara en varios tamaños , su segmentación por hardware y software y varios ejemplos de su tratamiento. Entre esos ejemplos, usaremos la biblioteca de visión OpenCV, con la que podremos obtener sorprendentes resultados.

5.1. Configuración de la cámara

Para obtener las imágenes adecuadas, la cámara del robot ha de ser configurada según las condiciones de luz del entorno donde vaya a situarse. La función **`OPENR::ControlPrimitive()`** se puede usar para cambiar la configuración de la cámara. Para usar esta función debemos haber obtenido el identificador de la primitiva de la cámara, que podemos obtener con la función **`OPENR::OpenPrimitive()`** .

- El **balance de blancos** puede tomar los valores **`ocamparamWB_INDOOR_MODE, ocamparamWB_OUTDOOR_MODE y ocamparamWB_FL_MODE`** (para lámparas fluorescentes).

```
OPrimitiveControl_CameraParam wb(ocamparamWB_OUTDOOR_MODE);
OPENR::ControlPrimitive(cameraID, oprmreqCAM_SET_WHITE_BALANCE, &wb, sizeof(wb), 0, 0);
```

- La **ganancia** puede tomar los valores **`ocamparamGAIN_LOW, ocamparamGAIN_MID, ocamparamGAIN_HIGH`**

```
OPrimitiveControl_CameraParam gain(ocamparamGAIN_MID);
OPENR::ControlPrimitive(cameraID, oprmreqCAM_SET_GAIN, &gain, sizeof(gain), 0, 0);
```

- La **velocidad del obturador** puede tomar los valores **`ocamparamSHUTTER_SLOW, ocamparamSHUTTER_MID, ocamparamSHUTTER_FAST`** .

```
OPrimitiveControl_CameraParam shutter(ocamparamSHUTTER_FAST) ;
OPENR::ControlPrimitive(cameraID, oprmreqCAM_SET_GAIN, &shutter, sizeof(shutter), 0, 0);
```

Ya que la cámara no es muy sensible, los valores recomendados para la mayor parte de los entornos interiores son la velocidad del obturador baja y la ganancia alta.

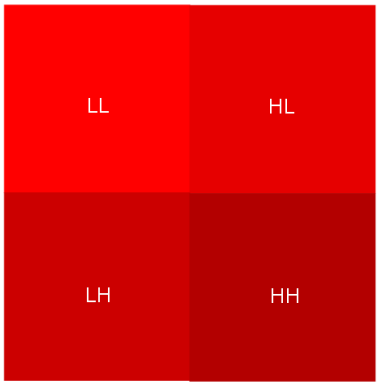


Figura 5.1: Cada 4 píxeles comparten las mismas características de color, pero no las de luminosidad

5.2. Principios básicos

La información de la cámara se suministra a través de *capas*. Existen 4 capas: 3 para obtener las imágenes en color en diferentes resoluciones, y una última que permite la segmentación de la imagen basada en color mediante una tabla configurable previamente (*detección de color*.

Las imágenes en color se obtienen en formato *YCrCb*. Cada pixel tiene tres componentes:

Y luminancia.

Cr Componente rojo - luminancia.

Cb Componente azul - luminancia.

Este formato tiene mucho que ver en cómo el ojo humano capta el color y las formas. El ojo es mucho más sensible a los cambios de luminosidad que a los de color, por lo que cada pixel contiene información de la componente **Y**, pero cada conjunto de $2x2$ píxeles comparten los componentes de color, como muestra la figura 5.1

5.3. Obtención de información de la cámara

OVirtualRobotComm manda imágenes de la cámara a través de su puerta **`FbkImageSensor`** , y el tipo de datos que usa para el envío se llama **`OFbkImageVectorData`** . Este sujeto puede ser referenciado en **connect.cfg** usando la siguiente línea:

`OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S`

5.3.1. El tipo de datos `OFbkImageVectorData`

Cada **`OFbkImageVectorData`** (Figura 5.2) contiene varias imágenes. Tiene tres miembros: **`vectorInfo`** que es un **`ODataVectorInfo`** , un array de **`OFbkImageInfo`** y un array de bytes. Estos array tienen 4 posiciones, correspondientes a cada una de las capas mencionadas en el capítulo anterior:

ofbkimageLAYER_H Imagen en color a alta resolución.

ofbkimageLAYER_M Imagen en color a media resolución (la mitad de resolución de *ofbkimageLAYER_H*).

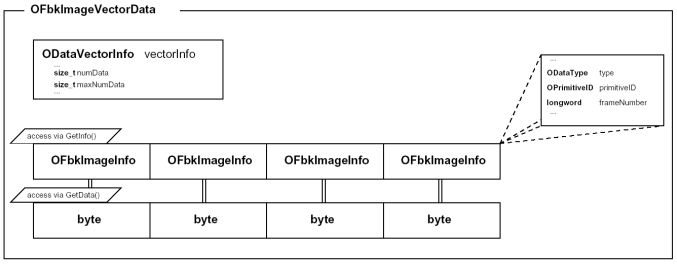


Figura 5.2: Formato de datos OFbkImageVectorData

ofbkimageLAYER_L Imagen en color a baja resolución (la mitad de resolución de *ofbkimageLAYER_M*).

ofbkimageLAYER_C Imagen de *detección de color.*

Para acceder a las capas, usamos las funciones **`GetInfo()`** y **`GetData()`** . Por ejemplo, para acceder a la información de la capa *ofbkimageLAYER_M* usaremos **`GetInfo(ofbkimageLAYER_M)`** , y para acceder a sus datos usaremos **`GetData(ofbkimageLAYER_M)`**

5.4. Accediendo a las bandas de color

Una vez seleccionada la capa a la que queremos acceder, debemos obtener información de las bandas de color. La clase C++ que maneja los datos de las imágenes se llama **`OFbkImage`** . Para crear una instancia de **`OFbkImage`** necesitamos punteros a la información y a los datos de una capa, que hemos obtenido previamente con las funciones **`GetInfo()`** y **`GetData()`** . También necesitamos especificar la banda de color (como explicamos en la sección 5.2) a la que deseamos acceder (**`ofbkimageBAND_Y`** para la banda Y, **`ofbkimageBAND_Cr`** para la banda Cr, **`ofbkimageBAND_Cb`** para la banda Cb y ofbkimageBAND_CDT si estamos usando la imagen de *detección de color*).

La clase **`OFbkImage`** tiene los siguientes métodos:

- **`bool IsValid()`** : Devolverá **`true`** si **`OFbkImage`** es válido.
- **`byte* Pointer()`** : Devuelve un puntero al inicio de los datos de la imagen.
- **`int Width()`** : Devuelve el ancho de la imagen.
- **`int Height()`** : Devuelve la altura de la imagen.
- **`byte Pixel(int x, int y)`** : Devuelve el valor de banda del pixel (x, y) de la imagen.
- **`intSkip()`** : Devuelve el número de bytes necesarios para saltarse una línea entera en una imagen
- **`byte ColorFrequency(OCdtChannel chan)`** : Devuelve el número de píxeles (divididos por 16) detectados en la banda **`chan`** de la capa de *detección de color* (se explicará más adelante).

Ejemplo: Acceso a un pixel de una imagen en color

En el siguiente ejemplo mostraremos como acceder al pixel $(30, 32)$ de una imagen de resolución media:

```
1       OStatus
2       Sensor::NotifyCamara(const ONotifyEvent& event)
3       {
4         OFbkImageVectorData* fbkImageVectorData =
                (OFbkImageVectorData*)event.Data(0);
5
6         byte pixelYCrCb[3];

7         OFbkImageInfo *info =
                fbkImageVectorData->GetInfo(ofbkimageLAYER_M);
8         byte *data =
                fbkImageVectorData->GetData(ofbkimageLAYER_M);

9         OFbkImage yImage(info, data, ofbkimageBAND_Y);
10        OFbkImage crImage(info, data, ofbkimageBAND_Cr);
11        OFbkImage cbImage(info, data, ofbkimageBAND_Cb);

12        pixelYCrCb[0] = yImage.Pixel(30, 32);
13        pixelYCrCb[1] = crImage.Pixel(30, 32);
14        pixelYCrCb[2] = cbImage.Pixel(30, 32);

15        return oSUCCESS;

16      }
```

En el código mostrado vemos la función que maneja los datos que provienen de la cámara por medio de un mensaje *Notify*.

- En la línea 4 creamos e inicializamos la variable **`fbkImageVectorData`** de la manera habitual en los manejadores de *Notify*.
- En la línea 6 creamos el array que va a contener los 3 valores YCrCb del píxeles. Cada valor es un byte.
- en las líneas 7 y 8 obtenemos la información y los datos correspondientes a la capa de color en resolución media (**`ofbkimageLAYER_M`**)
- En las lineas 9, 10 y 11 obtenemos las imágenes, por separado, correspondientes a las tres bandas de color **`ofbkimageBAND_Y ofbkimageBAND_Cr ofbkimageBAND_Cb`** .
- En las líneas 12, 13 y 14 se usa la función **`Pixel()`** para acceder al pixel del que deseamos obtener su valor.

Ejemplo: Acceso a un pixel de una imagen de *detección de color*

En este ejemplo accedemos al pixel 30, 32, pero en la capa de *detección de color*:

```
1       void
2       Image::NotifyImage ( const ONotifyEvent& event ) {

3       OFbkImageVectorData  imageVec = (OFbkImageVectorData ) event .Data (0) ;
4       OFbkImageInfo   info = imageVec >Get Info (ofbkimageLAYER_C) ;
5       byte   data = imageVec >GetData(ofbkimageLAYER_C) ;

6       OFbkImage cdtImage ( info , data , ofbkimageBAND_CDT ) ;

7       byte valor;
8       valor = cdtImage.Pixel(30, 32);
```

```
9       observer [ event . ObsIndex()] >AssertReady ( ) ;
10      }
```

5.4.1. Obteniendo una imagen completa a resolución máxima

Anteriormente se han obtenido los valores *YCrCb* de los pixeles con la función **`Pixel()`** accediendo a las imágenes de las bandas **`ofbkimageBAND_Y, ofbkimageBAND_Cr y ofbkimageBAND_Cb`** . Si quisiéramos obtener una imagen en alta resolución pronto nos daríamos cuenta de que esta resolución es exactamente la mitad de la que las especificaciones del robot describen. La razón de esto es porque realmente estamos accediendo a uno de cada cuatro píxeles en la imagen, y estos tres píxeles (figura 5.1) que faltan han de ser calculados. Estos 4 píxeles, que forman una matriz de $2x2$, comparten las componentes *CrCb*, pero su valor Y es diferente, por las razones expuestas anteriormente cuando se describió el formato *YCrCb*.

El acceso a los componentes de las imágenes se hace mediante bandas. Ya comentamos la existencia de las bandas **`ofbkimageBAND_Y, ofbkimageBAND_Cr y ofbkimageBAND_Cb`** , pero aparte de estas existen 3 más que son **`ofbkimageBAND_Y_LH, ofbkimageBAND_Y_HL y ofbkimageBAND_Y_HH`** . Realmente la banda **`ofbkimageBAND_Y`** es la banda **`ofbkimageBAND_Y`** , que corresponde a la intensidad del componente superior izquierda en la figura 5.1, y es a lo que hemos estado accediendo hasta ahora.

Normalmente la resolución de las imágenes que solo contemplan las capas **`ofbkimageBAND_Y, ofbkimageBAND_Cr y ofbkimageBAND_Cb`** son suficientes para la mayor parte de las aplicaciones y son las que son comúnmente usadas, pero existe un método (con la carga computacional que supone) para obtener imágenes a la mayor resolución posible. Este método está expuesto en el siguiente ejemplo:

Ejemplo: Obtención de una imagen a resolución máxima

```
1           byte ClipRange(int val) {
2               if (val < 0)        { return 0;         }
3                else if (val > 255) { return 255;       }
4                else                { return (byte)val; }
5            }

6          void
7          ReconstructAndConvertYCbCr(OFbkImageVectorData* imageVec,
                                      byte* image,
                                      int* width, int* height)
8          {
9             OFbkImageInfo* info = imageVec->GetInfo(ofbkimageLAYER_H);
10            byte*          data = imageVec->GetData(ofbkimageLAYER_H);

11            OFbkImage yLLImg(info, data, ofbkimageBAND_Y); // Y_LL
12            OFbkImage yLHImg(info, data, ofbkimageBAND_Y_LH);
13            OFbkImage yHLImg(info, data, ofbkimageBAND_Y_HL);
14            OFbkImage yHHImg(info, data, ofbkimageBAND_Y_HH);

15            OFbkImage crImg(info, data, ofbkimageBAND_Cr);
16            OFbkImage cbImg(info, data, ofbkimageBAND_Cb);

17            int w = yLLImg.Width();
18            int h = yLLImg.Height();

19            for (int y = 0; y < h; y++) {

20                for (int x = 0; x < w; x++) {
21                    //
22                    // yLH, yHL, yHH : offset binary [0, 255] -> signed int [-128, 127]
23                    //
24                    int yLL = (int)yLLImg.Pixel(x, y);
25                    int yLH = (int)yLHImg.Pixel(x, y) - 128;
```

```
26                  int yHL = (int)yHLImg.Pixel(x, y) - 128;
27                  int yHH = (int)yHHImg.Pixel(x, y) - 128;

28                  int a = yLL + yLH + yHL + yHH; // ypix11
29                  int b = 2 * (yLL + yLH);       // ypix11 + ypix01
30                  int c = 2 * (yLL + yHL);       // ypix11 + ypix10
31                  int d = 2 * (yLL + yHH);       // ypix11 + ypix00

32                  byte ypix00 = ClipRange(d - a);
33                  byte ypix10 = ClipRange(c - a);
34                  byte ypix01 = ClipRange(b - a);
35                  byte ypix11 = ClipRange(a);

36                  byte cb = cbImg.Pixel(x, y);
37                  byte cr = crImg.Pixel(x, y);

38                  PutYCbCrPixel(image, 2*w, 2*x,   2*y,   ypix00, cb, cr);
39                  PutYCbCrPixel(image, 2*w, 2*x+1, 2*y,   ypix10, cb, cr);
40                  PutYCbCrPixel(image, 2*w, 2*x,   2*y+1, ypix01, cb, cr);
41                  PutYCbCrPixel(image, 2*w, 2*x+1, 2*y+1, ypix11, cb, cr);
42              }

43          }

44          *width  = 2 * w;
45          *height = 2 * h;
46      }
```

- La función **`ClipRange()`** definida en las líneas 1-5 se usa para "ajustar" los valores enteros para que quepan en un byte.

- En la línea 7 se declara la función **`ReconstructAndConvertYCbCr()`** que toma como argumentos un **`OFbkImageVectorData`** y devuelve una imagen en el array cuyo espacio ha sido previamente reservado **`image`** y la anchura y altura de la imagen.

- En las líneas 9 y 10 se obtiene la imagen en alta resolución.

- En las líneas 11 a la 16 se obtienen las bandas necesarias para formar la imagen.

- En el bucle de las líneas 19 al 42 se obtiene el valor de cada uni de los píxeles, que son copiados a **`image`** con la función **`PutYCbCrPixel`** (líneas 38 a 41) que toma la coordenada del pixel y su valor *YCrCb*.

- En las líneas 24 a 35 está la clave de la obtención de los píxeles restantes de la matriz $2x2$ que faltaban por averiguar. De estas operaciones se deduce que la luminancia de los píxeles LH, HL y HH está relacionado con el valor del pixel LL.

5.5. *detección de color*

Aibo tiene un algoritmo de segmentación de color implementado en Open-R y del que podemos hacer uso una vez configurado. Este algoritmo es muy rápido, por estar codificado en el hardware.

Esta detección es configurable y puede detectar simultáneamente 8 color distintos (canales). Cada canal se referencia como **`ocdtCHANNEL0, ocdtCHANNEL1, ..., ocdtCHANNEL7`** y es completamente programable. Con la información del pixel y usando máscaras, podemos saber si un pixel esta "encendido" en cierto canal, lo que significará que pertenece al color configurado en ese canal.

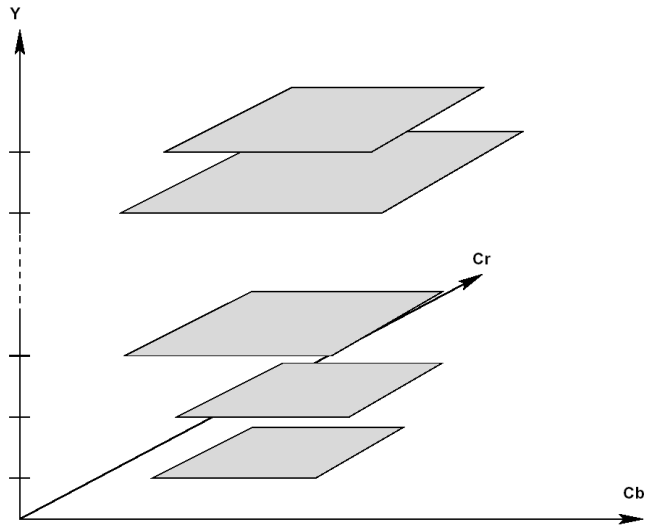


Figura 5.3: Representación en 3D de los rectángulos en el espacio (Y, Cr, Cb)

Ejemplo: Averiguando si un pixel pertenece al quinto canal

```
1       //fbkIVD es un OFbkImageVectorData
2       // Obtenemos la banda de color de la capa de detección de color
3
4       OFbkImage channel(fbkIVD->GetInfo(ofbkimageLAYER_C) ,
        fbkIVD->GetData(ofbkimageLAYER_C) ,
        ofbkimageBAND_CDT) ;

5       // Miramos si el píxel (10,10) está encendido en el canal 5
6       if (( channel.Pixel(10 ,10) & 0x10) == 0x01) { //El píxel está encendido en el cana 0x10 (5)
                //....
7               }
```

5.5.1. Configurando un canal de color

Un canal de color en YCrCb está formado por 32 rectángulos *CrCb*, cada uno de ellos definido para un valor de *Y*. En la figura 5.3 se puede apreciar una representación 3D de este concepto.

Durante el proceso de detección el hardware del robot toma el componente *Y* de cada pixel y comprueba si el correspondiente valor de *CrCb* cae dentro del rectángulo que es definido para este valor de *Y* en la descripción de color de cada canal de la tabla de detección, y detecta a qué canal este pixel pertenece.

El método para configurar un canal consiste en configurar un tabla comunicárselo a *FbkImageSensor*. Los pasos se especifican a continuación, y se relacionan con el ejemplo que aparece debajo, donde se configura un canal para la detección de la pelota rosa:

1. Se crea un **`OCdtVectorData`** en memoria compartida con **`OPENR::NewCdtVectorData()`** (línea 6-9). **`OCdtVectorData`** es una estructura de datos que contiene una tabla de detección de color o "Cdt" (Figura 5.4). Tiene dos miembros: Un **`ODataVectorData`** y 8 **`OCdtInfo`** . Un **`OCdtInfo`** contiene la información de un color del "Cdt". Se puede acceder a cada **`OCdtInfo`** por medio de la función **`GetInfo`** de la estructura **`ODataVectorData`** . Cada uno de los ocho canales está representado por un **`OCdtInfo`** .

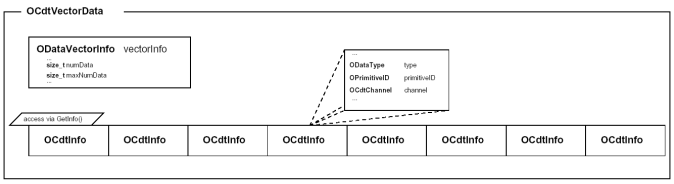


Figura 5.4: Estructura de datos **`OCdtVectorData`**

2. Se indica la cantidad de canales que se usarán con **`SetNumData()`** (línea 14)
3. Se inicializa un **`OCdtInfo`** con la función **`Init()`** antes de usarla (línea 16).
4. Se configura cada **`OCdtInfo`** con la función **`Set()`** . A esta función se le da el valor de *Y* y el rectángulo de *CrCb* (líneas 21-52).
5. Se manda el **`ODataVectorData`** al *FbkImageSensor* llamando a **`OPENR::SetCdtVectorData(`** (línea 53).
6. Liberamos la memoria compartida con **`OPENR::DeleteCdtVectorData()`** (línea 59).

Ejemplo: Configurando un canal para detectar el color rosa de la pelota del aibo

```
1       #define BALL_CDT_CHAN 0

2       void
        BallTrackingHead7::SetCdtVectorDataOfPinkBall()
3       {
4           OSYSDEBUG(("BallTrackingHead7::SetCdtVectorDataOfPinkBall()\n"));

5           OStatus result;
6           MemoryRegionID  cdtVecID;
7           OCdtVectorData* cdtVec;
8           OCdtInfo*       cdt;

9           result = OPENR::NewCdtVectorData(&cdtVecID, &cdtVec);
10          if (result != oSUCCESS) {
11              OSYSLOG1((osyslogERROR, "%s : %s %d",
                          "ImageObserver::SetCdtVectorDataOfPinkBall()",
                          "OPENR::NewCdtVectorData() FAILED", result));
12              return;
13          }

14          cdtVec->SetNumData(1);

15          cdt = cdtVec->GetInfo(0);
16          cdt->Init(fbkID, BALL_CDT_CHAN);

17          //
18          // cdt->Set(Y_segment, Cr_max,  Cr_min, Cb_max, Cb_min)
19          //
21          cdt->Set( 0, 230, 150, 190, 120);
22          cdt->Set( 1, 230, 150, 190, 120);
23          cdt->Set( 2, 230, 150, 190, 120);
24          cdt->Set( 3, 230, 150, 190, 120);
25          cdt->Set( 4, 230, 150, 190, 120);
26          cdt->Set( 5, 230, 150, 190, 120);
27          cdt->Set( 6, 230, 150, 190, 120);
28          cdt->Set( 7, 230, 150, 190, 120);
29          cdt->Set( 8, 230, 150, 190, 120);
30          cdt->Set( 9, 230, 150, 190, 120);
31          cdt->Set(10, 230, 150, 190, 120);
32          cdt->Set(11, 230, 150, 190, 120);
33          cdt->Set(12, 230, 150, 190, 120);
```

```
24          cdt->Set(13, 230, 150, 190, 120);
35          cdt->Set(14, 230, 150, 190, 120);
36          cdt->Set(15, 230, 150, 190, 120);
37          cdt->Set(16, 230, 150, 190, 120);
38          cdt->Set(17, 230, 150, 190, 120);
39          cdt->Set(18, 230, 150, 190, 120);
40          cdt->Set(19, 230, 150, 190, 120);
41          cdt->Set(20, 230, 160, 190, 120);
42          cdt->Set(21, 230, 160, 190, 120);
43          cdt->Set(22, 230, 160, 190, 120);
44          cdt->Set(23, 230, 160, 190, 120);
45          cdt->Set(24, 230, 160, 190, 120);
46          cdt->Set(25, 230, 160, 190, 120);
47          cdt->Set(26, 230, 160, 190, 120);
48          cdt->Set(27, 230, 160, 190, 120);
49          cdt->Set(28, 230, 160, 190, 120);
50          cdt->Set(29, 230, 160, 190, 120);
51          cdt->Set(30, 230, 160, 190, 120);
52          cdt->Set(31, 230, 160, 190, 120);

53          result = OPENR::SetCdtVectorData(cdtVecID);
54          if (result != oSUCCESS) {
55              OSYSLOG1((osyslogERROR, "%s : %s %d",
56                        "ImageObserver::SetCdtVectorDataOfPinkBall()",
57                        "OPENR::SetCdtVectorData() FAILED", result));
58          }

59          result = OPENR::DeleteCdtVectorData(cdtVecID);
60          if (result != oSUCCESS) {
61              OSYSLOG1((osyslogERROR, "%s : %s %d",
62                        "ImageObserver::SetCdtVectorDataOfPinkBall()",
63                        "OPENR::DeleteCdtVectorData() FAILED", result));
64          }
65      }
```

5.6. OpenCV en el aibo

OpenCV es una librería de tratamiento de imágenes muy potente que desarrolla Intel y que ha sido portada al robot aibo. Gracias a esta librería podemos aplicar algoritmos extremadamente complejos a las imágenes obtenidas por la cámara del robot.

La librería OpenCV permite almacenar las imágenes en objetos C++ **`CvAiboImage`** . Este objeto al crearse se le indica en su constructor el número de canales que almacenará, dependiendo si la imagen es en blanco y negro (1 canal) o en color (3 canales).

```
CvAiboImage imageGray(1); // 1 channel - intensity only
CvAiboImage imageColor(3); // YCrCb
CvAiboImage image16s(1, IPL_DEPTH_16S); // short values
```

Una vez creado un objeto **`CvAiboImage`** , podemos rellenarlo con una imagen proveniente del **`FbkImageSensor`** mediante el evento de imagen directamente.

```
imageColor.GetFromFbkImageEvent(event); // gets RGB image
```

Cuando tenemos la imagen dentro de un objeto **`CvAiboImage`** , podemos aplicarle cualquiera de las numerosas funciones que la librería OpenCV posee. En el ejemplo que se muestra a continuación se puede observar como se obtienen las líneas de fuga de una imagen, como se salva como BMP una imagen en color, y como aplicarle un filtro Canny para detectar los bordes en la imagen. Algunos resultados de este ejemplo se pueden ver en las figuras 5.5.

Ejemplo: Uso de OpenCV en el aibo

Este ejemplo está dividido en bloques que realizan tareas distintas sobre la imagen:

- El bloque primero (líneas 5-23) obtiene las líneas que hay en la imagen usando la función **`cvHoughLinesP()`** , que implementa el filtro de Hough probabilístico.
- El bloque segundo (líneas 25-35) salva en el memory stick una imagen en color como un ficheroBMP con el método **`SaveAsBmp()`** .
- El bloque tercero (líneas 37-47) aplica un filtro Canny a la imagen, implementado en la función **`cvCanny()`** , y posteriormente la salva en el memory stick como un fichero BMP.
- El bloque cuarto (líneas 48-61) aplica un filtro laplaciano a la imagen, implementado en la función **`cvLaplace()`** , y posteriormente la salva en el memory stick como un fichero BMP.

```
1  void CVTest::NotifyAiboImage(const ONotifyEvent& event)
2  {
3    char fileName[64];
4
5    //BLOCK: imagen de lineas
6    {
7      CvAiboImage imageColor(1);
8
9      imageColor.GetFromFbkImageEvent(event); // gets RGB image
10
11      float rho = 10.0f, theta = 0.1f;
12      int threshold = 10;
13      int lineLength = 10, lineGap = 2;
14      int nlines = 10;
15      int* lines = new int[4*nlines];
16
17      cvHoughLinesP(&imageColor, rho, theta, threshold, lineLength, lineGap, lines, nlines);
18
19      for (int i=0;i<40;i=i++)
20        {
21          OSYSDEBUG(("%d \n",lines[i]));
22        };
23    }
24
25    //BLOCK: color image
26    {
27      CvAiboImage imageColor(3);
28      imageColor.GetFromFbkImageEvent(event); // gets RGB image
29
30      sprintf(fileName, "/MS/IMAGE.BMP");
31      if (imageColor.SaveAsBmp(fileName))
32        printf("saved original image as %s\n", fileName);
33      else
34        printf("ERROR: SaveAsBmp failed (%s)\n", fileName);
35    }
36
37    //BLOCK: Canny edge detection
38    {
39      CvAiboImage imageCanny(1); // 1 channel - intensity only
40
41      cvCanny(&imageY, &imageCanny, 4, 8, 3); // REVIEW: parameters?
42      sprintf(fileName, "/MS/CANNY.BMP");
43      if (imageCanny.SaveAsBmp(fileName))
44        printf("saved Canny edges as B&W %s\n", fileName);
45      else
46        printf("ERROR: SaveAsBmp failed (%s)\n", fileName);
47    }
48    //BLOCK: Laplacian
49    {
50      CvAiboImage imageLaplacian16s(1, IPL_DEPTH_16S); // short values
51      cvLaplace(&imageY, &imageLaplacian16s, 3);
```

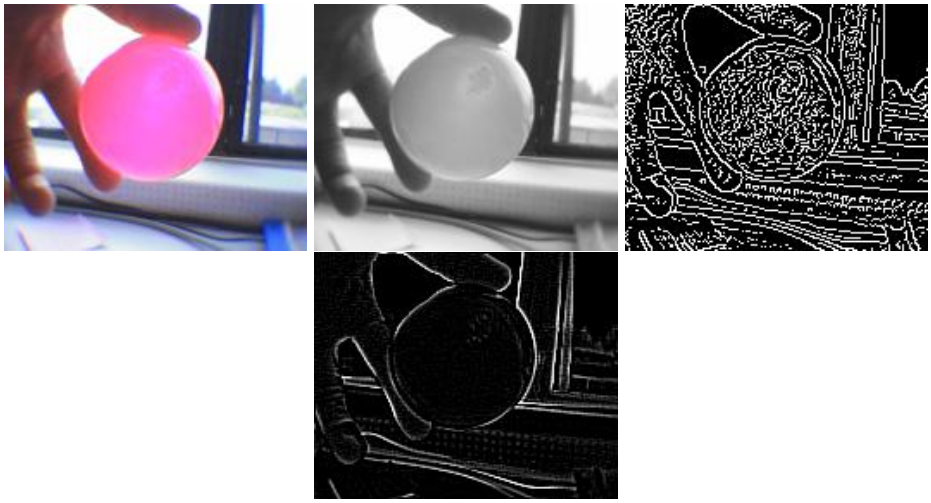


Figura 5.5: Imagen salvada en rgb, en escala de grises, filtrada con un filtro de Canny y filtrada con un filtro Laplaciano

```
52
53      CvAiboImage imageOut(1); // B&W image - looks cool
54      cvConvertScale(&imageLaplacian16s, &imageOut);
55
56      sprintf(fileName, "/MS/LAPLA.BMP");
57      if (imageOut.SaveAsBmp(fileName))
58        printf("saved laplace convolution as B&W %s\n", fileName);
59      else
60        printf("ERROR: SaveAsBmp failed (%s)\n", fileName);
61    }
62    observer[event.ObsIndex()]->AssertReady(event.SenderID());
63  }
```

Capítulo 6

Programación de red con Open-R

Open-R implementa la pila de protocolos IPv4 por medio de OPEN-R Networking Toolkit (ANT). IPv4 es un protocolo par enviar y recibir información en tre host en Internet. La pila de protocolos en Open-R ofrece varios protocolos:

IP (Internet Protocol) es un protocolo orientado a datos usado tanto por la fuente como por el destino para la comunicación de datos a través de una red de paquetes conmutados. Los datos en una red basada en IP son enviados en bloques conocidos como paquetes o datagramas (en el protocolo IP estos términos se suelen usar indistintamente). IP provee un servicio de datagramas no confiable (también llamado del mejor esfuerzo); por ejemplo, no garantiza nada sobre la recepción del paquete. El paquete podría llegar dañado, en otro orden con respecto a otros paquetes, duplicado o simplemente no llegar. Si se necesita confiabilidad, esta es proporcionada por los protocolos de la capa de transporte.

TCP (Transmission Control Protocol) El protocolo TCP de la capa de transporte es un servicio orientado a conexión y la unidad de datos que envía o recibe el protocolo TCP es conocido con el nombre de segmento TCP. La función protocolo TCP consiste en ofrecer un servicio de envío y recepción de datos orientado a conexión que sea seguro y que goce de los siguientes mecanismos: Multiplexamiento, conexiones, Fiabilidad y Control de flujo y congestión.

UDP (User Datagram Protocol) Protocolo del nivel de transporte basado en el intercambio de datagramas. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera. Se utiliza cuando se necesita transmitir voz o vídeo y resulta más importante transmitir con velocidad que garantizar el hecho de que lleguen absolutamente todos los bytes.

DNS (Domain Name System) Un DNS (Domain Name System) es un conjunto de protocolos y servicios (base de datos distribuida) que permiten a los usuarios utilizar nombres en vez de tener que recordar direcciones IP numéricas. Ésta es ciertamente la función más conocida de los protocolos DNS: la asignación de nombres a direcciones IP.

DHCP (Dynamic Host Configuration Protocol) Es un protocolo de red en el que un servidor provee los parámetros de configuración a las computadoras conectadas a la red informática que los requieran (máscara, puerta de enlace y otros) y también incluye un mecanismo de asignación de direcciones de IP).

6.1. Conceptos básicos para usar la pila de protocolos en Open-R

Cualquiera de los objetos Open-R que se desarrollen dentro del robot[1] puede usar los servicios que ofrece la pila de protocolos IPv4 [1]. Los objetos usan la pila mediante el método habitual de paso de mensajes y mediante un método especial para comunicarse con la pila.

Los pasos para que un objeto pueda usar la pila de protocolos [6] son los siguientes:

- Crear un *endpoint* por el que comunicarse.
- Reservar un buffer de memoria compartida para intercambiar datos entre el objeto y la pila de protocolos.
- Requerir un servicio de red a la pila de protocolos.

6.1.1. Creación de enpoints

Cuando un objeto quiere crear un endpoint, manda un `antEnvCreateEndpointMsg` al IPStack. Este mensaje contiene la siguiente información:

```
antEnvCreateEndpointMsg createMsg(
        EndpointType_TCP,
        4 * PACKETSIZE
);
```

Protocolo Protocolo a implementar por el enpoint. Los siguientes endpoints están disponibles:

- EndpointType_TCP
- EndpointType_UDP
- EndpointType_DNS
- EndpointType_IP

El tamaño de la memoria destinada al banco de SDU[2] que debe ser asignada al endpoint. El banco de SDU debe tener al menos el tamaño del máximo paquete que se pretende mandar, si bien es altamente recomendable que sea mayor.

Una vez creado el mensaje `antEnvCreateEndpointMsg`, el mensaje se manda al IPStack por medio del método `Call()`, que es heredado de `antEnvMsg` y es llamado especificándole una referencia al IPStack (`antStackRef`. Este método manda un mensaje especial a la pila de protocolo de manera síncrona, quedándose boqueado hasta recibir respuesta del IPStack.

```
createMsg.Call(
        IPStackRef,
        sizeof(antEnvCreateEndpointMsg)
);
```

Cuando la respuesta es recibida, el objeto almacena la referencia al endpoint recién creado en su campo `moduleRef`.

```
endpoint = createMsg.moduleRef;
```

[1] Nótese que esto no está disponible para objetos que usen Remote Processing

[2] Service data unit. Es el contenedor básico de datos en el entorno ANT. Un SDU es un puntero a una serie de celdas de datos en el banco SDU. Los SDUs llevan los datos que están siendo mandados sobre la red. A estos se les llama unidades de datos del protocolo (PDUs) en el estandar OSI

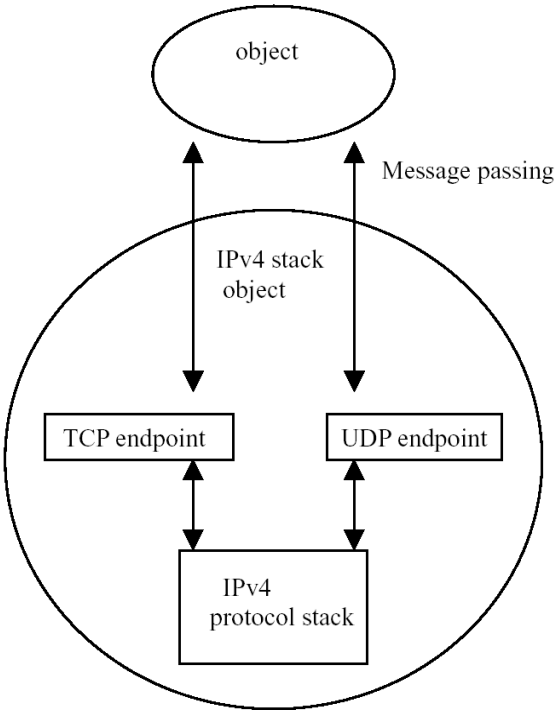


Figura 6.1: Los objetos crean endpoint dentro del objeto ANT

6.1.2. Creación de los buffer de memoria compartida

Los buffer de memoria compartida son implementados por la estructura `antSharedBuffer`. Estos buffer son zonas de memoria compartida que son mapeados dentro del espacio de direcciones del objeto de usuario y de la pila de protocolos. Cuando intercambiamos datos entre nuestro objeto y la pila, especificamos un puntero a este buffer y un offset que `antSharedBuffer` mapea de un espacio de direcciones al otro de manera transparente.

Se pueden adoptar políticas muy diferentes con relación al número de buffer que tenemos reservados a la vez. Podemos tener un solo buffer para enviar y recibir, o uno o varios buffer para enviar y otros tantos para recibir. El proceso de asignación y liberación de memoria son implementadas por las rutinas de Open-R de administración de memoria compartida. Estas rutinas con muy costosas, por lo que se recomienda que se reutilicen las zonas de memoria compartida todo lo posible. Típicamente una de las soluciones más aceptadas es la de tener un buffer para recibir y otro para enviar, e ir reutilizándolo constantemente. Claro está que esta no es la solución estándar, y puede que las necesidades de diseño de nuestra aplicación dicten que otra solución sea más óptima.

Existe una restricción a la hora de fijar el tamaño de los buffer compartidos a múltiplos del tamaño de la página en Open-R (4 KB), de modo que siempre que fijemos el tamaño de los buffer, se redondeará al próximo múltiplo de 4 KB.

Los pasos para la reserva de estos buffer son los siguientes:

1. Creación de un mensaje `antEnvCreateSharedBufferMsg` para solicitar la creación del buffer.

```
antEnvCreateSharedBufferMsg bufferMsg(PACKETSIZE);
```

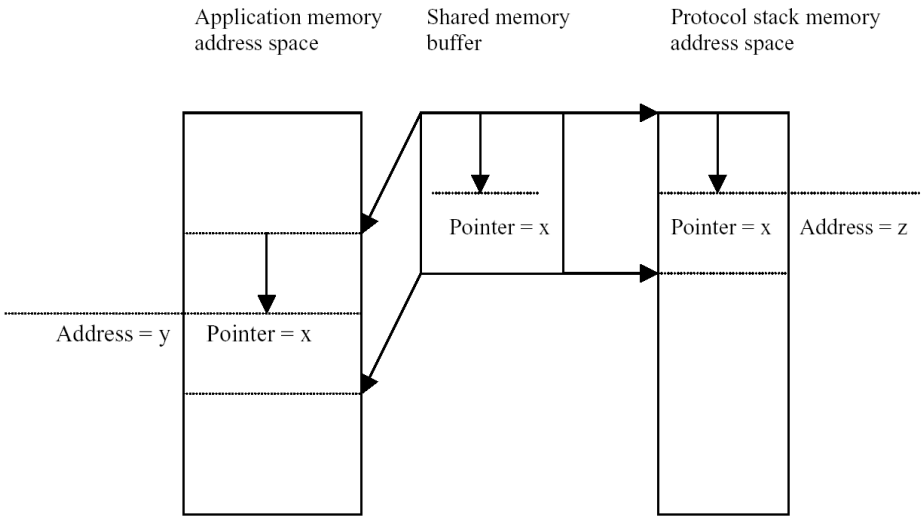


Figura 6.2: El buffer de memoria compartida es mapeada en los espacios de direcciones del objeto de usuario y de la pila de protocolos

Los objetos de usuario intercambian sólo un paquete cada vez con la pila de protocolos, con lo que `PACKETSIZE` ha de ser fijado al tamaño máximo de paquete que vayamos a enviar o recibir por el objeto.

2. Una vez creado el mensaje `antEnvCreateSharedBufferMsg`, el mensaje se manda asíncronamente al IPStack nuevamente por medio del método `Call()`, que ya se comentó que heredaba de `antEnvMsg`.

```
bufferMsg.Call(
        IPStackRef,
        sizeof(antEnvCreateSharedBufferMsg)
);
```

3. El objeto recibe la respuesta, que es una referencia al recién creado buffer de memoria compartida.

```
sendBuffer = bufferMsg.buffer;
```

4. El objeto mapea el buffer de memoria compartida a su espacio de direcciones.

```
sendBuffer.Map()
```

6.1.3. Solicitud de servicios de red

Para solicitar servicios de red, un objeto debe crear un mensaje con la solicitud y mandarlo al IPStack.

```
TCPEndpointConnectMsg connectMsg(
        endpoint,
        0, 0,
        "193.74.243.95", 7
);

connectMsg.Call(
        IPStackRef,
        sizeof(TCPEndpointConnectMsg)
);
```

En el ejemplo anterior se solicita un servicio TCP por medio de un mensaje `TCPEndpointConnectMsg` en el quele especificamos el enpoint a usar, la dirección y puerto origen (en este caso a 0 por retornar los valores cuando la conexión se establezca) y destino.

6.2. Conectándose con TCP

En esta sección mostraremos con el ejemplo *Echoserver*, incluido en los ejemplos de Open-R, como enviar y recibir datos por el protocolo TCP. Este objeto recibirá un mensaje por TCP y contestará exactamente con el mismo mensaje al que se lo mandó.

Iniciando la conexión

Los pasos iniciar las conexiones son:

1. En **Echoserver.h:50** declaramos la variable privada `ipstackRef` de tipo `antStackRef`. Esta variable contendrá una referencia al IPStack. Es inicializada en la función `DoStart()` (**Echoserver.h:37**).

```
37       ipstackRef = antStackRef("IPStack");
```

2. En En **Echoserver.h:51** declaramos el array `connection` de tipo `TCPConnection`. Cada una de las `ECHOSERVER_CONNECTION_MAX` posiciones de este array contendrá una conexión donde podrá atender una conexión a la vez. La estructura `TCPConnection` ls definimos (y podemos modificar) en **TCPConnection.h:27**.

```
17      enum ConnectionState {
18          CONNECTION_CLOSED,
19          CONNECTION_CONNECTING,
20          CONNECTION_CONNECTED,
21          CONNECTION_LISTENING,
22          CONNECTION_SENDING,
23          CONNECTION_RECEIVING,
24          CONNECTION_CLOSING
25      };

27      struct TCPConnection {
28          antModuleRef     endpoint;
29          ConnectionState  state;

31          // send buffer
32          antSharedBuffer  sendBuffer;
33          byte*            sendData;
34          int              sendSize;

36          // receive buffer
37          antSharedBuffer  recvBuffer;
38          byte*            recvData;
39          int              recvSize;
40      };
```

3. En la misma función `DoStart()` hacemos llamadas a la función `InitTCPConnection()` (**Echoserver.cc:40**) por cada una de las `ECHOSERVER_CONNECTION_MAX` conexiones que aceptará nuestro servidor.

```
39          for (int index = 0; index < ECHOSERVER_CONNECTION_MAX; index++) {
40              OStatus result = InitTCPConnection(index);
41              if (result != oSUCCESS) return oFAIL;
42          }
```

4. En la función `InitTCPConnection()` ((**Echoserver.cc:244**) creamos, para cada una de las conexiones, un buffer de envio `sendBufferMsg` y otro de recepción `recvBufferMsg`.

```
253         antEnvCreateSharedBufferMsg sendBufferMsg(ECHOSERVER_BUFFER_SIZE);
255         sendBufferMsg.Call(ipstackRef, sizeof(sendBufferMsg));

272         antEnvCreateSharedBufferMsg recvBufferMsg(ECHOSERVER_BUFFER_SIZE);
274         recvBufferMsg.Call(ipstackRef, sizeof(recvBufferMsg));
```

Después de crear cada buffer de memoria compartida, mapeamos esta memoria en la memoria del objeto *Echoserver* y del IPStack.

```
264          connection[index].sendBuffer = sendBufferMsg.buffer;
265          connection[index].sendBuffer.Map();
266          connection[index].sendData
267              = (byte*)(connection[index].sendBuffer.GetAddress());

283          connection[index].recvBuffer = recvBufferMsg.buffer;
284          connection[index].recvBuffer.Map();
285          connection[index].recvData
286              = (byte*)(connection[index].recvBuffer.GetAddress());
```

5. Una vez reservados los buffers para cada conexión, procedemos a crear un endpoint con un servicio asociado a él. En la función `DoStart()` llamamos a la función `Listen()` para ponernos a escuchar en un en el endpoint, y en ese momento es cuando solicitamos el servicio de red para él, en este caso `EndpointType_TCP`.

```
72          antEnvCreateEndpointMsg tcpCreateMsg(EndpointType_TCP,
73                                               ECHOSERVER_BUFFER_SIZE * 2);
74          tcpCreateMsg.Call(ipstackRef, sizeof(tcpCreateMsg));
82          connection[index].endpoint = tcpCreateMsg.moduleRef;
```

`ECHOSERVER_BUFFER_SIZE` está ajustada en **EchoServerConfig.h:16** a 512 bytes.

Ciclo de uso de las conexiones

Una vez creadas las conexiones podemos proceder a usarlas para recibir los mensajes y contestarlos. Las comunicaciones de red en Open-R son asíncronas, de tal manera que si decidimos ponernos, por ejemplo, a escuhar esperando un mensaje no bloqueamos el flujo de ejecución de ese objeto. A la hora de hacer la petición de escuchar le indicamos que cuando haya "oido" algo, el evento procedente de ANT con la información necesaria sobre esa escucha sea atendida por una función que maneje ese evento.

Los manejadores de estos eventos has de ser declarados en el fichero **stub.cfg** como *Extra Entries*.

```
12      ObjectName : EchoServer
13      NumOfOSubject   : 1
14      NumOfOObserver  : 1
15      Service : "EchoServer.DummySubject.DoNotConnect.S", null, null
16      Service : "EchoServer.DummyObserver.DoNotConnect.O", null, null

18      Extra : ListenCont()
19      Extra : SendCont()
20      Extra : ReceiveCont()
21      Extra : CloseCont()
```

De igual manera, cuando queramos mandar, recibir o cerrar un conexión, el resultado de esa acción deberá ser atendida por sus manejadores correspondientes.

Listen Cuando deseamos ponernos a escuchar en un conexión debemos indicarlo justo después de crear un endpoint asociado a una conexión, a la vez que le indicamos que cuando se conecten al enpoint algo, debe ser atendido ese evento por un de los *Extra Entries* definidos en **stub.cfg** (línea 92 del siguiente código).

```
62      OStatus
63      EchoServer::Listen(int index)
64      {

87          TCPEndpointListenMsg listenMsg(connection[index].endpoint,
88                                         IP_ADDR_ANY, ECHOSERVER_PORT);
89          listenMsg.continuation = (void*)index;

91          listenMsg.Send(ipstackRef, myOID_,
92                         Extra_Entry[entryListenCont], sizeof(listenMsg));

96          return oSUCCESS;
97      }
```

En el método `ListenCont` se decide qué hacer cuando se conecten al endpoint, en este caso hacemos una llamada al método `Receive` (línea 119) para pedirle que reciba los datos cuando le lleguen por la conexión que se ha establecido.

```
99      void
100     EchoServer::ListenCont(ANTENVMSG msg)
101     {

104         TCPEndpointListenMsg* listenMsg
105             = (TCPEndpointListenMsg*)antEnvMsg::Receive(msg);
106         int index = (int)listenMsg->continuation;

118         SetReceiveData(index);
119         Receive(index);
120     }
```

Receive En el momento que ya es establecida la conexión, en el método `Receive` se pide a la pila de comunicaciones que cuando reciba datos sean manejados por el *Extra Entrie* `ReceiveCont` (línea 182).

```
168     OStatus
169     EchoServer::Receive(int index)
170     {

176         TCPEndpointReceiveMsg receiveMsg(connection[index].endpoint,
177                                          connection[index].recvData,
178                                          1, connection[index].recvSize);
179         receiveMsg.continuation = (void*)index;

181         receiveMsg.Send(ipstackRef, myOID_,
182                         Extra_Entry[entryReceiveCont], sizeof(receiveMsg));

184         return oSUCCESS;
185     }
```

En el método `ReceiveCont` ya podemos analizar el mensaje que nos ha llegado y procesarlo. En el caso de nuestro ejemplo, lo que vamos a hacer es imprimirlo y mandárselo (línea 208) de vuelta a quien nos lo ha enviado:

```
187     void
188     EchoServer::ReceiveCont(ANTENVMSG msg)
189     {
191
192         TCPEndpointReceiveMsg* receiveMsg
193             = (TCPEndpointReceiveMsg*)antEnvMsg::Receive(msg);
194         int index = (int)(receiveMsg->continuation);

204         OSYSPRINT(("recvData : \%s", connection[index].recvData));

206         connection[index].sendSize = receiveMsg->sizeMin;
207         SetSendData(index);
208         Send(index);
209     }
```

```
291     void
292     EchoServer::SetSendData(int index)
293     {
294         memcpy(connection[index].sendData,
294                connection[index].recvData, connection[index].sendSize);
295     }
```

Send Como ya viene siendo habitual, para enviar datos por un endpoint, pedimos a la pila de comunicaciones que lo envie y que el resultado lo maneje el *Extra Entrie* `SendCont` (línea 136).

```
122     OStatus
123     EchoServer::Send(int index)
124     {

130         TCPEndpointSendMsg sendMsg(connection[index].endpoint,
131                                    connection[index].sendData,
132                                    connection[index].sendSize);
133         sendMsg.continuation = (void*)index;

135         sendMsg.Send(ipstackRef, myOID_,
136                      Extra_Entry[entrySendCont],
137                      sizeof(TCPEndpointSendMsg));

139         connection[index].state = CONNECTION_SENDING;
140         connection[index].sendSize = 0;
141         return oSUCCESS;
142     }
```

En el manejador `SendCont` recibimos un evento de la pila de comunicaciones indicando si el envio se ha producido de manera correcta, en cuyo caso llamamos al método `Receive` para recibir más datos, o de manera errónea, cerrando entonces la conexión con el método `Close`.

```
144     void
145     EchoServer::SendCont(ANTENVMSG msg)
146     {

149         TCPEndpointSendMsg* sendMsg = (TCPEndpointSendMsg*)antEnvMsg::Receive(msg);
150         int index = (int)(sendMsg->continuation);

152         if (sendMsg->error != TCP_SUCCESS) {
156             Close(index);
157             return;
158         }

160         OSYSPRINT(("sendData : \%s", connection[index].sendData));

164         SetReceiveData(index);
165         Receive(index);
166     }
```

Close En el método `Close` pedimos a la pila de comunicaciones que cierre la conexión. Cuando reciba el evento con el resultado del cierre, pedirá que se ponga a aceptar nuevas peticiones en el endpoint, invocando al método `Listen`.

```
211     OStatus
212     EchoServer::Close(int index)
213     {
219         TCPEndpointCloseMsg closeMsg(connection[index].endpoint);
210         closeMsg.continuation = (void*)index;
```

```
212         closeMsg.Send(ipstackRef, myOID_,
213                       Extra_Entry[entryCloseCont], sizeof(closeMsg));

217         return oSUCCESS;
218     }
```

```
210     void
211     EchoServer::CloseCont(ANTENVMSG msg)
212     {
215         TCPEndpointCloseMsg* closeMsg
216             = (TCPEndpointCloseMsg*)antEnvMsg::Receive(msg);
217         int index = (int)(closeMsg->continuation);

210         Listen(index);
211     }
```

Capítulo 7

Remote Processing

Remote processing es un entorno de desarrollo que aporta Open-R y permite la ejecución de parte o de la totalidad de aplicaciones destinadas para el AIBO en una máquina que no es el AIBO, típicamente un PC. De esta manera una aplicación escrita en Open-R puede estar formada por objetos que son ejecutados en el robot y otros objetos que son ejecutados en un host, ambos conectados entre sí mediante una conexión inalámbrica.

En la figura 7.1 se muestra un esquema del funcionamiento de este mecanismo. En la parte izquierda del esquema nos encontramos con una serie de objetos Open-R que se encuentran en ejecución dentro del robot. Estos objetos se comunican con otra serie de objetos, situadas en la parte derecha del esquema, que se ejecutan en una máquina Linux como procesos UNIX. Para que este esquema de ejecución pueda darse nos encontramos en la parte central del esquema con el elemento que hace posible esta comunicación: TCPGateway.

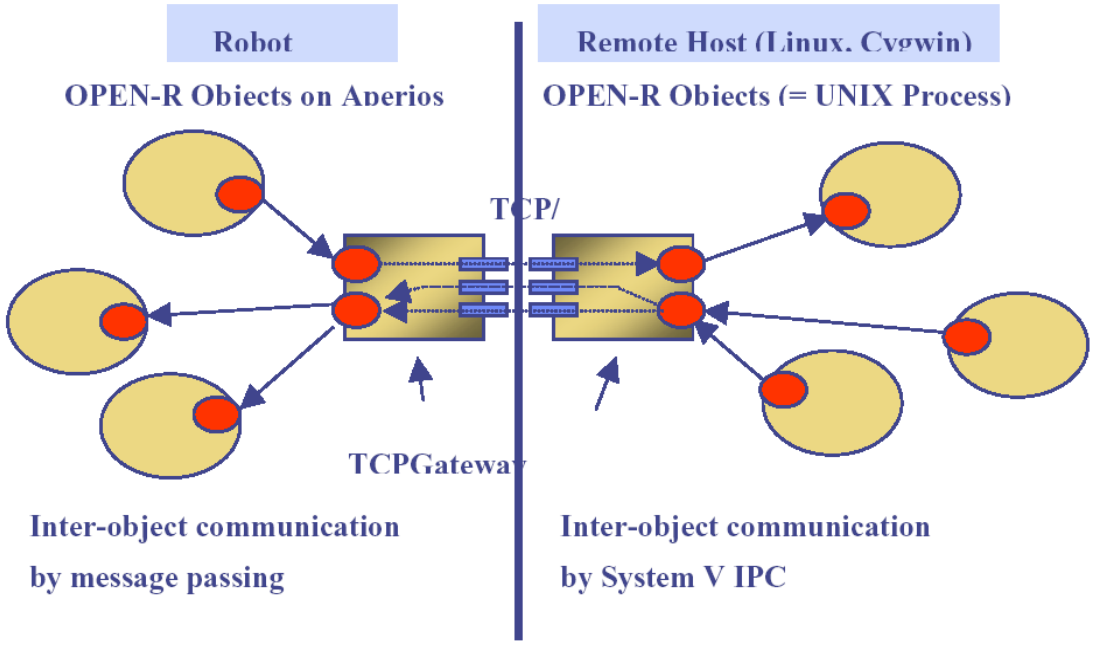


Figura 7.1: Comunicación entre objetos con Open-R

A TCPGateway le llegan todas las comunicaciones de un lado de la aplicación (del lado del robot o del lado del PC) que han de llegar al otro lado. Por cada conexión unidireccional entre cada par de objetos, TCPGateway mantiene una conexión TCP en un puerto diferente.

Cuando le llega un mensaje, lo aplana y lo envía por TCP al otro lado, donde la otra parte del TCPGateway desaplanará el mensaje y lo entregará al objeto que corresponda.

Pero, ¿para qué podríamos necesitar de este mecanismo? ¿que ventajas me aporta?. Si una persona está programando un robot ¿que le podría llevar a querer ejecutar parte o la totalidad de una aplicación en un PC?. La respuesta a estas preguntas se encuentran relacionadas con la potencia de desarrollo que nos da un PC con respecto al AIBO, que no deja de ser un robot cuyo interfaz está muy limitado. Las principales ventajas son las siguientes:

- Los objetos que se ejecutan en el PC pueden reconectarse a los que se ejecutan en el AIBO y viceversa, lo que permite que el desarrollo de una aplicación se acelera. Imagináos que tuviérais una aplicación formada por tres objetos. Para la fase de desarrollo, ejecutáis dos de ellos en el robot y el tercero, que es donde se está centrando la mayor parte del esfuerzo de desarrollo, en el PC. Remote Processing permite que, sin reiniciar en ningún momento el robot, podáis cambiar y recompilar el objeto que ejecuta en el PC cuantas veces queráis, y que éste se pueda integrar con los objetos ejecutando en el robot sin ningún problema, manteniendo todas las comunicaciones con ellos.

- Al ejecutarse los objetos Open-R en el PC como procesos UNIX, podemos usar depuradores(p.e. gdb) para analizar su ejecución, visualizando el valor de variables, poniendo breakpoins o analizando fallos de segmentanción, por ejemplo. Todo ello sin cambiar el código fuente en ningún momento.

- Permite el uso de recursos del PC para desarrollar aplicaciones más ricas. Podríamos,por ejemplo, crear un interfaz gráfica con GTK que estuviera en el código de un objeto Open-R para representar parte de su información o controlarlo.

Todo esto nos indica que tenemos una poderosa herramienta de desarrollo, pero esto también tiene sus limitaciones:

- Solo pueden ejecutarse de esta manera los sistemas de objetos que han sido configurados como “nomemprot”, así que al crear el memory stick, hay que prepararlo con:

```
/usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/WCONSOLE/nomemprot/OPEN-R
```

- la configuración del **OBJECT.CFG** y **CONNECT.CFG** basada en el modelo del robot vía [RobotDesign] no está soportada. Por ejemplo, esto no está soportado:

```
#
# OBJECT.CFG
#
[ERS-210]
/MS/OPEN-R/MW/OBJS/ERS-210.BIN

[ERS-220]
/MS/OPEN-R/MW/OBJS/ERS-220.BIN
```

- Parte del API de Open-R no está soportado al ejecutarse en un PC:

  - OPENR::ControlPrimitive()
  - OPENR::NewSoundVectorData()
  - OPENR::DeleteSoundVectorData()
  - OPENR::NewCdtVectorData()

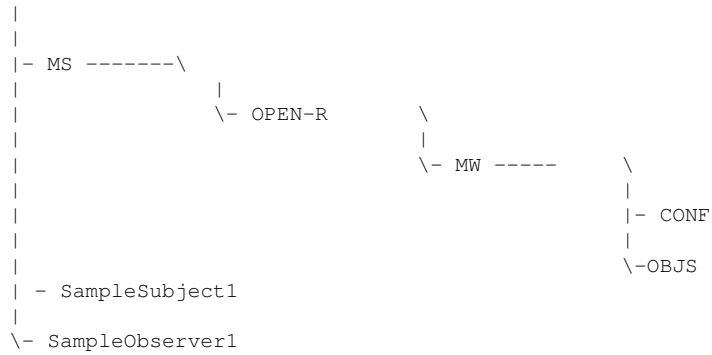
- OPENR::DeleteCdtVectorData()
- OPENR::SetCdtVectorData()
- OPENR::Shutdown()
- OPENR::ObservePowerStatus()
- OPENR::UnobservePowerStatus()
- OPENR::FindDesignData()
- OPENR::DeleteDesignData()
- OPENR::Fatal()

- La clase ANT (Aperios Network Toolkit), que implementa la pila de protocolos de IP, no está soportada al ejecutarse en un PC.
- Las partes internas de Open-R están construidas sobre UNIX, por lo que Remote Processing no está disponible en MacOS ni en ninguna arquitectura que no sea x86, principalmente debido a que la arquitectura "Indian" es diferente.

Se pueden desarrollar aplicaciones con Remote Processing , aún con estas restricciones, si se diseñan adecuadamente las aplicaciones de manera que la parte de red y la parte del API que no pueda ser ejecutado en el PC esté siempre en los objetos se ejecutan en el robot.

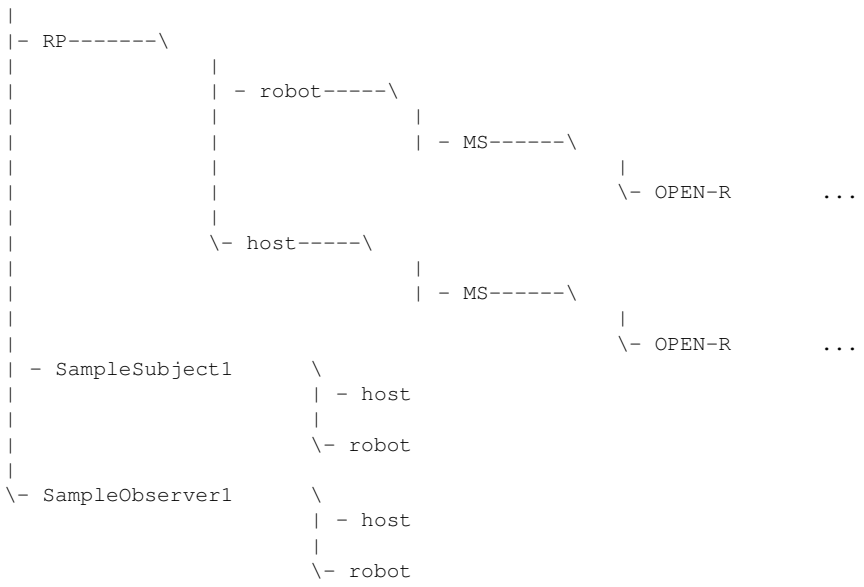
7.1. Creación de una aplicación con TCP Gateway

Cuando se desea desarrollar una aplicación en la que se puedan dividir arbitrariamente los objetos entre la parte del robot y la del PC, se han de seguir ciertas pautas para que el desarrollo se haga de manera sencilla. Para esta explicación partamos de un ejemplo típico de una aplicación Open-R con una distribución de directorios como la siguiente:



7.1.1. Creación de la estructura de directorios

Una buena separación en directorios de las partes relativas a la ejecución en el PC o en el robot es crucial para obtener un sistema limpio y mantenible. Nuestro objetivo será crear, dentro de cada directorio de fuentes (**SampleSubject1** y **SampleObserver1**) dos directorios para distintas compilaciones. El contenido los memory stick, ya sea para ejecutarse en el robot (**RP/robot**) o para la ejecución en el host (**RP/host**) deben ser creados. Remote Processing también necesita un directorio **MS** con los mismos contenidos que los que tendría la parte para el robot para ejecución en el PC. La estructura debería ser algo como esto:



7.1.2. Compilando de manera separada

Los fuentes se mantendrían en **SampleSubject** 1 y **SampleObserver1** , pero los binarios se construirían en los directorios **host** y **robot** , donde habría un **Makefile** diferente para ambos con las partes relativas a las opciones de compilación para el PC o el robot. Si observamos un listado del contenido de los fuentes del objeto SampleObserver1, tenemos:

```
paco@trantran:ObjectComm-multi $ find SampleObserver1

SampleObserver1/host
SampleObserver1/host/Makefile

SampleObserver1/robot
SampleObserver1/robot/Makefile

SampleObserver1/Makefile.common
SampleObserver1/SampleObserver.cc
SampleObserver1/SampleObserver.h
SampleObserver1/sampleObserver.ocf
SampleObserver1/stub.cfg
```

Como se puede apreciar, se han creado dos directorios, que lo único que contiene es un **Makefile** . También observamos que en el directorio donde están las fuentes hemos creado un fichero **Makefile.common** que contiene las reglas para construir los binarios, pero sin definir aún ni las librerías, ni la ruta de las cabeceras, ni el compilador a usar, ni el comando para copiar el binario una vez construido:

```
paco@trantran:ObjectComm-multi $ cd SampleObserver1/
paco@trantran:ObjectComm-multi/SampleObserver1 $ cat Makefile.common
#
# Copyright 2002,2004 Sony Corporation
#
# Permission to use, copy, modify, and redistribute this software for
# non-commercial use is hereby granted.
#
# This software is provided "as is" without warranty of any kind,
# either expressed or implied, including but not limited to the
# implied warranties of fitness for a particular purpose.
#
```

```
# NOTE: This makefile is included by the makefiles in the
#       subdirectories.

.PHONY: all install clean

all: sampleObserver1.bin

%.o: %.cc
        $(CXX) $(CXXFLAGS) -o $@ -c $^

SampleObserverStub.cc: stub.cfg
        $(STUBGEN) $^

sampleObserver1.bin: SampleObserverStub.o SampleObserver.o $(OCF)
        $(MKBIN) $(MKBINFLAGS) -o $@ $^ $(LIBS)
        $(STRIP) $@

install: sampleObserver1.bin
        $(CP) $^ $(INSTALLDIR)/OPEN-R/MW/OBJS/OBS1.BIN

clean:
        rm -f *.o *.bin *.elf *.snap.cc
        rm -f SampleObserverStub.h SampleObserverStub.cc def.h entry.h
        rm -f $(INSTALLDIR)/OPEN-R/MW/OBJS/OBS1.BIN
```

El truco consiste en que los valores de las variables que quedan por definir son las que marcarán si se construye el objeto para ser ejecutado en el robot o en un PC. En caso de ser construido para el robot, nos encontramos con los valores habituales en la construcción de objetos Open-R convencionales:

```
paco@trantran:ObjectComm-multi/SampleObserver1 $ cat robot/Makefile
#
# Copyright 2002 Sony Corporation
#
# Permission to use, copy, modify, and redistribute this software for
# non-commercial use is hereby granted.
#
# This software is provided "as is" without warranty of any kind,
# either expressed or implied, including but not limited to the
# implied warranties of fitness for a particular purpose.
#

OPENRSDK_ROOT?=/usr/local/OPEN_R_SDK
CP=$(OPENRSDK_ROOT)/OPEN_R/bin/gzcp
INSTALLDIR=../../RP/robot/MS
CXX=$(OPENRSDK_ROOT)/bin/mipsel-linux-g++
STRIP=$(OPENRSDK_ROOT)/bin/mipsel-linux-strip
MKBIN=$(OPENRSDK_ROOT)/OPEN_R/bin/mkbin
STUBGEN=$(OPENRSDK_ROOT)/OPEN_R/bin/stubgen2
MKBINFLAGS=-p $(OPENRSDK_ROOT)
LIBS=-lObjectComm -lOPENR
CXXFLAGS= \
        -O2 \
        -g \
        -I. \
        -I.. \
        -I$(OPENRSDK_ROOT)/OPEN_R/include/R4000 \
        -I$(OPENRSDK_ROOT)/OPEN_R/include
OCF=sampleObserver.ocf

VPATH=..
include ../Makefile.common
paco@trantran:ObjectComm-multi/SampleObserver1 $
```

De esta manera, si se ejecuta “make” dentro del directorio **robot** , los fuentes generados a partir de **stub.cfg** y los binarios se constuirán en este directorio, y los binarios se colocarán al instalarlos en el directorio **RP/robot/MS** del raiz del ejemplo.

Si queremos que la compilación se produzca para ejecutarse en el PC, en el directorio **host** debemos llamar a "make", que realizará la compilación, pero con los compiladores, librerías y cabeceras convencionales propias de desarrollos en un PC:

```
paco@trantran:ObjectComm-multi/SampleObserver1 $ cat host/Makefile
#
# Copyright 2002 Sony Corporation
#
# Permission to use, copy, modify, and redistribute this software for
# non-commercial use is hereby granted.
#
# This software is provided "as is" without warranty of any kind,
# either expressed or implied, including but not limited to the
# implied warranties of fitness for a particular purpose.
#

OPENRSDK_ROOT?=/usr/local/OPEN_R_SDK
CP=cp
INSTALLDIR=../../RP/host/MS
STRIP=strip
MKBIN=g++
STUBGEN=$(OPENRSDK_ROOT)/RP_OPEN_R/bin/rp-openr-stubgen2
MKBINFLAGS=
OPENRCONFIG=$(OPENRSDK_ROOT)/RP_OPEN_R/bin/rp-openr-config
LIBS=`$(OPENRCONFIG) --libs`
CXXFLAGS= \
        -O2 \
        -g \
        -I. \
        -I.. \
        `$(OPENRCONFIG) --cflags`
OCF=

VPATH=..
include ../Makefile.common
```

Al instalar los objetos generados por este binario, estos se colocarán en el directorio **RP/host/MS** del raiz del ejemplo.

Después de esta fase de preparación disponemos de un método para generar de manera sencilla los binarios para el robot o para el pc, e instalarlos en su lugar adecuado.

7.1.3. Configuración de TCP Gateway

Ya tenemos los binarios necesarios para la ejecución en el PC y en robot, y cada uno en su directorio. Lo que nos falta es activar el objeto TCP Gateway y configurarlo para conectar la parte del robot y la del PC. Para lo primero debemos indicar a Open-R que ejecute el objeto TCP Gateway **TCPGW.BIN** en el robot:

```
paco@trantran:ObjectComm-multi $ cat RP/robot/MS/OPEN-R/MW/CONF/OBJECT.CFG

/MS/OPEN-R/SYSTEM/OBJS/TCPGW.BIN

/MS/OPEN-R/MW/OBJS/POWERMON.BIN
/MS/OPEN-R/MW/OBJS/SBJ1.BIN
/MS/OPEN-R/MW/OBJS/SBJ2.BIN
```

Ahora tenemos un objeto TCPGW en el robot que actuará como proxy para todas las comunicaciones con los objetos situados en el PC. Para cada conexión, el objeto TCPGW debe usar una puerta diferente y una conexión diferente, tal y como muestra la figura 7.2.

Por el contrario, un mal diseño sería el mostrado en la figura 7.3. Cada puerta de TCPGW usa un puerto diferente, y se envía por TCP por una puerta diferente a la parte del PC, donde

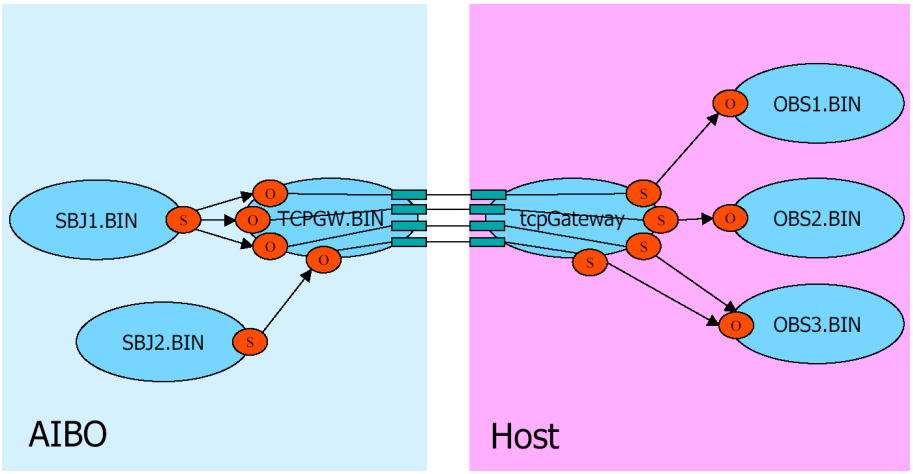


Figura 7.2: Esquema de comunicación correcta

será repartido al objeto correspondiente. ¿Como se configuran las conexiones y se establece la correspondencia de las conexiones a través de TCPGW?. Una serie archivos deben ser configurados.

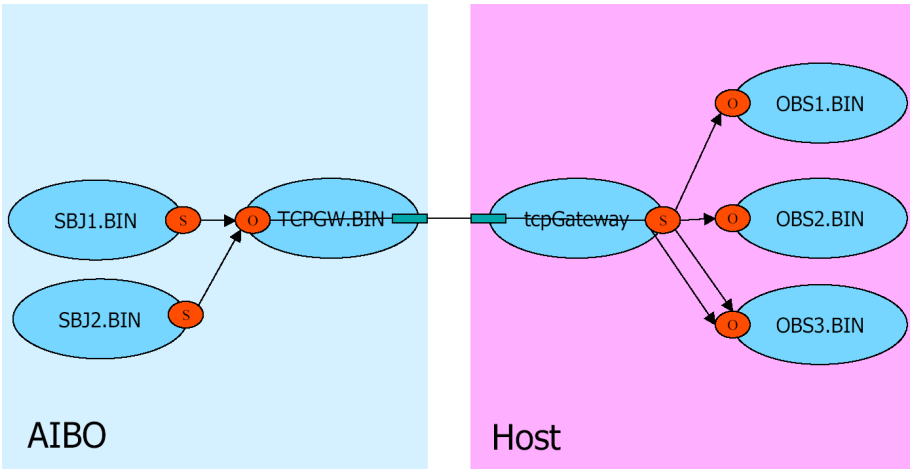


Figura 7.3: Esquema de comunicación erroneo

- La conexión entre los objetos del robot y el objeto TCPGW se hacen mediante el fichero de configuración **CONNECT.CFG** :

```
paco@trantran:ObjectComm-multi $ cat RP/robot/MS/OPEN-R/MW/CONF/CONNECT.CFG

SampleSubject1.SendString.char.S TCPGateway.ReceiveString1.char.O
```

- El objeto TCPGW ha de ser configurado indicándole las puertas que tiene (que se han usado en el punto anterior) y los puertos que usará para la comunicación TCP. Una conexión extra (TCPGateway.Proxy.AperiosMessage.P) ha de ser configurada adicionalmente:

```
paco@trantran:ObjectComm-multi $ cat RP/robot/MS/OPEN-R/MW/CONF/ROBOTGW.CFG

TCPGateway.Proxy.AperiosMessage.P 59001
TCPGateway.ReceiveString1.char.O  59002
```

- En el PC debemos configurar las conexiones desde los puertos a los objetos correspondientes. La direccion IP del robot ha de ser especificada en este punto, ya que sin ella no sabríamos a dónde conectarnos. El PC es el cliente de la conexión, y el robot el servidor al que se conectan:

```
paco@trantran:ObjectComm-multi $ cat RP/host/MS/OPEN-R/MW/CONF/HOSTGW.CFG

!ROBOT_PROXY                  59001 192.168.2.10
TCPGateway.SendString1.char.S 59002 192.168.2.10
```

7.1.4. Ejecución de la aplicación

Para la ejecución de una aplicación debemos colocarnos en el directorio **RP/host** y ejecutar:

```
paco@trantran:ObjectComm-multi/RP2/host $ /usr/local/OPEN_R_SDK/RP_OPEN_R/bin/start-rp-openr

[pid:13924,msqid:65538,oid:0x00010002] oserviceManager
[pid:13925,msqid:98307,oid:0x00018003] tcpGateway
[pid:13926,msqid:131076,oid:0x00020004] MS/OPEN-R/MW/OBJS/OBS1.BIN

....
```

"start-rp-openr" es un script que arranca los servicios necesarios para la comunicación TCP Gateway en el PC. Lee el directorio **MS** que encuentra en el directorio donde se ejecuta, y actúa como si la ejecución se realizara en el PC. Aparte de arrancar los servicios necesarios para la comunicación como procesos, los configura leyendo del archivo **CONNECT.CFG** y **HOSTGW.CFG** . Una vez levantados los servicios y preparadas las comunicaciones, solo queda arrancar los objetos Open-R que son indicados en el archivo **OBJECT.CFG** . Este proceso es muy similar al que ocurre en el robot al arrancarse.Por otra parte, el robot será encendido, y los objetos se conectarán entre ellos de manera transparente, continuando normalmente con la aplicación.

Una ventaja que se comentó antes, es la de que en cualquier momento cualquiera de las partes se puede parar sin que esto suponga que la otra parte deba finalizar su ejecución. Esto podría ser útil para que los objetos que son temporalmente el foco de nuestro desarrollo sean ejecutados en el PC, y se paren, modifiquen y vuelvan a ser ejecutados sin que la parte de robot sea consciente de esto. De esta manera el procesos de desarrollo se puede acelerar.

7.2. Depurar un proceso Open-R ejecutándose en el PC con Gdb

GDB es el debugger (depurador) de GNU. Es un potente depurador que permite "ver"que esta sucediendo dentro de programas. Soporta varios lenguajes, aunque originalmente estaba diseñado para C/C++ fundamentalmente. El uso de Gdb va más allá de los objetivos de este documento y no lo describiremos aquí. Entre las características que posee, podemos citar:

- Debugging de programas complejos con multiples archivos.
- Capacidad para detener el programa o ejecutar un comando en un punto especifico (breakpoints), según una condición (watchpoints) o al llegar un signal (catchpoints).
- Capacidad para mostrar valores de expresiones cuando el programa se detiene automaticamente (displays).
- Es posible examinar la memoria y/o variables de diversas formas y tipos, incluyendo estructuras, arrays y objetos.
- Es posible igualmente cambiar los valores de las variables para estudiar el comportamiento del programa sin necesidad de recompilar.
- Posibilidad de realizar debugging a programas en ejecucion (procesos).
- Posibilidad de realizar debugging a programas que han finalizado.

Como vemos, Gdb es una herramienta que nos puede ayudar en la tearea de desarrollar aplicaciones. Desgraciadamente no podemos usarlo directamente en el robot, cosa que personalmente he echado mucho de menos al desarrollar en esta plataforma. Afortunadamente, como ya alguno se habrá imaginado, ahora no estamos en las mismas condiciones. Ahora podemos ejecutar objetos en un PC como procesos UNIX, tenemos el fuente y el lenguaje está soportado por Gdb ¿a que esperamos para meternos en el interior de nuestros programas?.

Aunque las posibilidades que nos da Gdb son muy amplias, es complicado arrancar los objetos Open-R en el PC con Gdb. Son los servicios de Remote Processing los que se encargan de leer los ficheros de configuración y cargar unos binarios u otros. La mejor opción que tenemos es la de arrancar los procesos y unirnos a ellos en tiempo de ejecución.

Para ilustrar este proceso vamos a usar el ejemplo “ObjectComm-multi “ que se encuentra dentro de los ejemplos que Open-R proporciona. Este ejemplo permite ser aplicado de varias maneras, según la distribución de objetos que queremos. Usaremos la que se encuentra en el directorio **RP2** .

Compilación e instalación El directorio **RP2/host** contiene un **Makefile** que compila los fuentes de los objetos que se ejecutan en el PC en su directorio **host** y los instala en **RP2/host/MS** . De igual manera, El directorio **RP2/robot** contiene un **Makefile** que compila los fuentes de los objetos que se ejecutan en el robot en su directorio **host** y los instala en **RP2/host/MS** , desde donde los copiaremos al Memory Stick.

Un detalle importante que debemos tener en cuenta son las opciones de compilación y el proceso por el que pasan los binarios hasta que son instalados en sus directorios:

1. Para poder depurar los objetos, debemos compilar siempre con la opción “-g”. Este opción genera información de depuración en el formato nativo del sistema operativo que Gdb puede usar, aunque es probable que si se usa otro depurador, esto haga que no se ejecute correctamente.
2. Existe un comando llamado “strip” que en los ejemplos que Open-R, y en este en concreto, se usa justo después de la compilación antes de copiar los binarios. Este comando elimina todos los símbolos de un binario, que son útiles en el proceso de depurado. Se debe evitar, modificado el archivo **Makefile** correspondiente, que este comando se ejecute.

Ejecución y depuración Lo primero que haremos es colocarnos en el directorio **RP2/host** , desde el que ejecutaremos:

```
paco@trantran:ObjectComm-multi/RP2/host $ start-rp-openr

[pid:13286,msqid:294914,oid:0x00048002] oserviceManager
[pid:13291,msqid:327683,oid:0x00050003] tcpGateway
[pid:13292,msqid:425990,oid:0x00068006] MS/OPEN-R/MW/OBJS/OBS1.BIN
[pid:13293,msqid:360452,oid:0x00058004] MS/OPEN-R/MW/OBJS/OBS2.BIN
[pid:13294,msqid:393221,oid:0x00060005] MS/OPEN-R/MW/OBJS/OBS3.BIN
```

Los objetos anteriores se quedan esperando a que el robot se encienda y empieze la comunicación. En ese momento podemos engancharnos al proceso del objeto **OBS1.BIN** :

```
paco@trantran:ObjectComm-multi/RP2/host $ ps aux

USER       PID COMMAND

paco     13639  /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/RegistryManager/registryManager
        /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/OObjectManager/oobjectManager
        /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/OServiceManager/oserviceManager
        /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/TCPGateway/tcpGateway
paco     /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/OObjectManager/oobjectManager 13639
        /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/OServiceManager/oserviceManager
        /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/TCPGateway/tcpGateway
paco     13648  /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/OServiceManager/
        oserviceManager 13639
paco     13651  /usr/local/OPEN_R_SDK/RP_OPEN_R/lib/linux-i386-gcc3/TCPGateway/tcpGateway 13639
paco     13652  MS/OPEN-R/MW/OBJS/OBS1.BIN 13639
paco     13653  MS/OPEN-R/MW/OBJS/OBS2.BIN 13639
paco     13654  MS/OPEN-R/MW/OBJS/OBS3.BIN 13639

paco@trantran:ObjectComm-multi/RP2/host $ cd MS/OPEN-R/MW/OBJS
paco@trantran:ObjectComm-multi/RP2/host $ gdb OBS1.BIN 13652

GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
        "/lib/tls/i686/cmov/libthread_db.so.1".

Attaching to program: /home/paco/labor/universidad/investigacion/aibo/tmp
        /ObjectComm-multi/RP2/host/MS/OPEN-R/MW/OBJS/OBS1.BIN, process 13652
'system-supplied DSO at 0xffffe000' has disappeared; keeping its symbols.
Reading symbols from /usr/lib/libstdc++.so.5...done.
Loaded symbols for /usr/lib/libstdc++.so.5
Reading symbols from /lib/tls/i686/cmov/libm.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libm.so.6
Reading symbols from /lib/libgcc_s.so.1...done.
Loaded symbols for /lib/libgcc_s.so.1
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in __kernel_vsyscall ()
(gdb) list
105         }
106
107         return 1;
108     }
109
110
111     int
```

```
112     main(int argc, char* argv[])
113     {
114         strcpy(myname, argv[0]);
```

Ya estamos dentro del proceso, vemos que la ejecución se encuentra en la función main() del objeto. En este experimento vamos a establecer un breakpoint en la línea 55 de SampleObserver.cc:

```
52      void
53      SampleObserver::Notify(const ONotifyEvent& event)
54      {
55          const char* text = (const char *)event.Data(0);
56          OSYSPRINT(("SampleObserver1::Notify() %s\n", text));
57      }
```

Con lo que usamos la instrucción de Gdb “break”, que sitúa un breakpoint en la línea que le indiquemos. En este caso la depuración la haremos paso a paso, pero puede servirnos para aclararnos cuando llegamos al punto deseado. En este punto podemos encender el robot y continuar con la aplicación. Todos los objetos en el PC responderán a las llamadas de los objetos en el robot, salvo el que estamos depurando, que se encuentra parado al estar depurándolo. Aún así, un mensaje le ha llegado, y terminará desencadenando las acciones necesarias para tratar el mensaje, lo que podemos ir trazando poco a poco hasta que llegue a la función que lo trata, que es donde hemos puesto el breakpoint:

```
(gdb) break /home/paco/labor/universidad/investigacion/aibo/tmp/ObjectComm-multi/
        SampleObserver1/SampleObserver.cc:55
Breakpoint 1 at 0x80498b1: file ../SampleObserver.cc, line 55.
(gdb) n
92          OServiceEntry() : selector(UNDEF) { oid.MakeInvalid(); }

...

Breakpoint 1, SampleObserver::Notify (this=0x80584d0, event=@0x0) at ../SampleObserver.cc:55
55          const char* text = (const char *)event.Data(0);
(gdb) list
50      }
51
52      void
53      SampleObserver::Notify(const ONotifyEvent& event)
54      {
55          const char* text = (const char *)event.Data(0);
56          OSYSPRINT(("SampleObserver1::Notify() %s\n", text));
57      }
(gdb) p text
$1 = 0x0
(gdb) s
54      {
(gdb) s
55          const char* text = (const char *)event.Data(0);
(gdb) s
56          OSYSPRINT(("SampleObserver1::Notify() %s\n", text));
(gdb) p text
$3 = 0xb7fe8010 "!!! Hello world !!!"
```

Hemos llegado a la función y hemos inspeccionado el contenido de las variables, en este caso el texto enviado desde un objeto en el robot.

7.3. Crear una interfaz GTK+ a un proceso Open-R ejecutándose en el PC

En esta sección vamos a mostrar como se puede usar cualquier recurso del ordenador para enriquecer los objetos que son ejecutados en el PC. Lo que vamos a hacer es modificar el ejemplo "HelloWorld-stubgen" que se encuentra dentro de los ejemplos que Open-R proporciona para que genere una ventana gráfica con una caja de texto y un botón. El ejemplo"HelloWorld-stubgen" ya está preparado para ser ejecutado en el PC, con lo que los cambios serán mínimos.

GTK+ is un conjunto de herramientas para crear interfaces gráficos de usuario. Gtk+ ofrece un completo conjunto de widgets que usaremos para crear los interfaces. GTK+ es software libre y parte del proyecto GNU y está licenciado bajo GNU LGPL, con lo que podemos usarla libremente. El uso de GTK+ está más allá de los objetivos de este documento, y la documentación puede ser encontrada fácilmente en la web.

La primera cosa que necesita una aplicación GTK+ es que la ejecución de la aplicación, después de la inicialización de la interfaz, quede en manos del bucle de control de GTK, con la llamada a la función "gtk_main()", que implementa este bucle. Esta llamada ha de ser realizada sobre en la función "main()" del objeto Open-R, ya que es el único punto de entrada inicial. Esta función "main()" se encuentra en el fichero **HelloWorldStub.cc** , que es generado automáticamente a partir del fichero **stub.cfg** y que se supone que en condiciones normales no debería ser nunca modificado. Originalmente es así:

```
28      int
29      check_message_queue()
30      {
31          HelloWorld& Self = *pSelf;
32
33          if ( msgrcv(msqid, (msgbuf*)&currentMsg, currentMsg.Size(), msgType, 0) > 0 ) {
34              void* pMsg = &currentMsg.data;

...

110     int
111     main(int argc, char* argv[])
112     {
113         strcpy(myname, argv[0]);
114         SetRegistryManagerKey(argv[1]);
115
116         // Buffer Creation
117         key_t msgkey = (key_t)getpid();
118         msqid = msgget(msgkey, IPC_CREAT|IPC_EXCL|0666);
119         if (msqid < 0){
120             cerr << "!!! ERROR Message queue creation failed in " << myname;
121             ::perror("msgget()");
122             exit(1);
123         }
124
125     #if 0
126         cout << "starting " << myname
127              << ", OID = " << msqid << " (0x" << hex << msqid << dec << ")" << endl;
128     #endif
129
130         // Signal Setting
131         signal(SIGINT, Terminate);
132         signal(SIGSEGV, Terminate);
133         signal(SIGTERM, Terminate);
134
135         // Create OPEN-R Object
136         pSelf = new HelloWorld();
137
138         // Main Routine
139         while (check_message_queue())
140             ;
141
```

```
142     return 0;
143     }
```

El primer problema es el de que la función “check_message_queue()” ha de ser llamada constantemente para atender los mensajes que recibe el objeto. Esto se puede solucionar mediante la función de GTK+ “g_idle_add()”, que indica a GTK+ que, estando en el bucle principal que “gtk_main()” produce, si no hay eventos que atender, que llame a otra función.

El último problema al que tenemos que enfrentarnos en esta parte es la de que dentro de la función “check_message_queue()”, en la línea 33 existe una llamada a “msgrcv()”, que tal y como está es bloqueante, con lo que se quedaría siempre esperando un mensaje y el bucle de GTK+ quedaría estancado. Lo que debemos hacer es hacer que esa llamada no sea bloqueante, y que si no hay mensajes en la cola que continúe la ejecución. Después de la adaptación del código para usar GTK+, éste presenta este aspecto:

```
...
17      #include <gtk/gtk.h>
...

29      gint
30      check_message_queue(gpointer ptr)
31      {
32          HelloWorld& Self = *pSelf;
33
34
35          if ( msgrcv(msqid, (msgbuf*)&currentMsg, currentMsg.Size(), msgType, IPC_NOWAIT) > 0 ) {
36              void* pMsg = &currentMsg.data;

...

111     int
112     main(int argc, char* argv[])
113     {
114
115          // GTK+ setting
116         gtk_set_locale();
117         gtk_init(&argc, &argv);
118
119         strcpy(myname, argv[0]);
120         SetRegistryManagerKey(argv[1]);
121
122         // Buffer Creation
123         key_t msgkey = (key_t)getpid();
124         msqid = msgget(msgkey, IPC_CREAT|IPC_EXCL|0666);
125         if (msqid < 0){
126             cerr << "!!! ERROR Message queue creation failed in " << myname;
127             ::perror("msgget()");
128             exit(1);
129         }
130
131     #if 0
132         cout << "starting " << myname
133              << ", OID = " << msqid << " (0x" << hex << msqid << dec << ")" << endl;
134     #endif
135
136         // Signal Setting
137         signal(SIGINT, Terminate);
138         signal(SIGSEGV, Terminate);
139         signal(SIGTERM, Terminate);
140
141         // Create OPEN-R Object
142         pSelf = new HelloWorld();
143
144         // Main Routine
145         g_idle_add(check_message_queue, NULL);
146
147         gtk_main();
148
```

```
149         return 0;
150     }
```

Una vez solucionado esto ya podemos crear una ventana en el objeto Open-R y establecer callbacks si así lo deseamos:

```
En HelloWorld.h

...

19      #include <gtk/gtk.h>
20
21      #define GLADE_HOOKUP_OBJECT(component,widget,name) \
22        g_object_set_data_full (G_OBJECT (component), name, \
23          gtk_widget_ref (widget), (GDestroyNotify) gtk_widget_unref)
24
25      #define GLADE_HOOKUP_OBJECT_NO_REF(component,widget,name) \
26        g_object_set_data (G_OBJECT (component), name, widget)
27
28      class HelloWorld : public OObject {

...

41          GtkWidget *window1;
42          GtkWidget* create_window1 (void);
43
44      };
45
46      #endif // HelloWorld_h_DEFINED


Y en HelloWorld.cc

14      #include <gtk/gtk.h>
15
16      HelloWorld::HelloWorld ()
17      {
18        window1 = create_window1 ();
19        gtk_widget_show (window1);
20
21
22          OSYSDEBUG(("HelloWorld::HelloWorld()\n"));
23      }

...

55      // GTK Methods
56      void
57      on_button1_clicked (GtkButton *button, gpointer user_data)
58      {
59
60      }
61
62      GtkWidget*
63      HelloWorld::create_window1 (void)
64      {
65        GtkWidget *window1;
66        GtkWidget *vbox1;
67        GtkWidget *entry1;
68        GtkWidget *button1;
69
70        window1 = gtk_window_new (GTK_WINDOW_TOPLEVEL);
71        gtk_window_set_title (GTK_WINDOW (window1), "window1");
72
73        vbox1 = gtk_vbox_new (FALSE, 0);
74        gtk_widget_show (vbox1);
75        gtk_container_add (GTK_CONTAINER (window1), vbox1);
76
77        entry1 = gtk_entry_new ();
78        gtk_widget_show (entry1);
79        gtk_box_pack_start (GTK_BOX (vbox1), entry1, FALSE, FALSE, 0);
```

```
80
81        button1 = gtk_button_new_with_mnemonic ("button1");
82        gtk_widget_show (button1);
83        gtk_box_pack_start (GTK_BOX (vbox1), button1, FALSE, FALSE, 0);
84
85        g_signal_connect ((gpointer) button1, "clicked",
86                          G_CALLBACK (on_button1_clicked),
87                          NULL);
88
89        /* Store pointers to all widgets, for use by lookup_widget(). */
90        GLADE_HOOKUP_OBJECT_NO_REF (window1, window1, "window1");
91        GLADE_HOOKUP_OBJECT (window1, vbox1, "vbox1");
92        GLADE_HOOKUP_OBJECT (window1, entry1, "entry1");
93        GLADE_HOOKUP_OBJECT (window1, button1, "button1");
94
95        return window1;
96      }
```

Al ejecutar este objeto en el PC nos generará una bonita ventana desde donde podremos controlar el objeto Open-R y visualizar la información que queramos.

Bibliografía

[1] Sony Corporation. *OPEN-R SDK: Internet Protocol Vesion 4*.

[2] Sony Corporation. *OPEN-R SDK: Instalation Guide*. Sony Corporation, 2003.

[3] Sony Corporation. *OPEN-R SDK: Level 2 Reference Guide*. Sony Corporation, 2003.

[4] Sony Corporation. *OPEN-R SDK: Model Information for ERS-210*. Sony Corporation, 2003.

[5] Sony Corporation. *OPEN-R SDK: Model Information for ERS-7*. Sony Corporation, 2003.

[6] Sony Corporation. *OPEN-R SDK: Programmer's Guide*. Sony Corporation, 2003.

[7] Françoss Serra. *Aibo Programming using OPEN-R SDK. Tutorial*. 2003.