Arquitectura Eagle Knights

Los programas en Open-R

Cada programa en Open-R esta formado por uno o varios objetos Open-R. Cada objeto Open-R es un único proceso single-thread, que se ejecuta concurrentemente con otro objeto 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.

HelloWorld

Dentro del paquete de ejemplos (*samples*) de Open-R se encuentra un directorio con el código del objeto Open-R \HelloWorld".

En ese directorio se encuentran tres elementos:

HelloWorld Contiene el código fuente del ejemplo.

Makefile archivo 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 componente (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 Makfile se debe indicar su localización. Por lo tanto, si compilamos, instalamos o limpiamos HelloWorld, también haremos lo mismo con PowerMonitor. El resto de las líneas de este archivo Makefile únicamente transmiten la acción a realizar a los Makfiles de los directorios que hemos colocado como componentes en la línea 1.

MS Este directorio representa la parte referente a la aplicación en la Memory Stick que se introducirá en el robot. Cuando este compilado el código, los binarios y otros archivos de configuración de la aplicación quedaran instalados en este directorio con el fin de copiarlos directamente a la Memory Stick, para introducirlo en el robot.

Hay que resaltar que la Memory Stick no estará vacía antes de instalar la aplicación, es decir, que no sólo la aplicación es lo que habrá en este dispositivo, sino datos necesarios para el funcionamiento del robot.

Entendiendo HelloWorld

En el directorio 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 archivo de compilación de este objeto Open-R.
helloWorld.ocf Este archivo 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 area de memoria para el objeto está situada en el espacio de direcciones virtuales. Si ponemos "notlb", lo pondremos en 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 Normalmente se genera automáticamente.

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

El código de HelloWorld.h se muestra a continuación:

```
1 #include <OPENR/OObject.h>
2 class HelloWorld : public OObject {
3 public:
4 HelloWorld();
```

```
5 virtual ~HelloWorld() {}
9
6 virtual OStatus DoInit (const OSystemEvent& event);
7 virtual OStatus DoStart (const OSystemEvent& event);
8 virtual OStatus DoStop (const OSystemEvent& event);
9 virtual OStatus DoDestroy(const OSystemEvent& event);
10 };
```

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 2) y tiene su constructor y destructor como cualquier objeto C++.

Como ya se ha 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:

```
1 OStatus
2 HelloWorld::DoInit(const OSystemEvent& event)
3 {
4 OSYSDEBUG((“HelloWorld::DoInit()\n”));
5 return oSUCCESS;
6 }
```

En la línea 4 encontramos una macro para imprimir mensajes de depuración por la consola. En la línea 5 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:

```
1 OStatus
2 HelloWorld::DoStart(const OSystemEvent& event)
3 {
4 OSYSDEBUG((“HelloWorld::DoStart()\n”));
5 OSYSPRINT((“!!! Hello World !!!\n”));
6 return oSUCCESS;
10
11 }
```

DoStop Es llamado al apagarse el robot:

```
1 OStatus
```

```
2 HelloWorld::DoStop(const OSystemEvent& event)
3 {
4 OSYSDEBUG((“HelloWorld::DoStop()\n”));
5 OSYSLOG1((osyslogERROR, “Bye Bye ...”));
6 return oSUCCESS;
7 }
```

En la línea 5 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] This is error!

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 implementacion es:

```
1 OStatus
2 HelloWorld::DoDestroy(const OSystemEvent& event)
3 {
4 return oSUCCESS;
5 }
```

Iniciando el Memory Stick

Partiendo de una Memory Stick vacia:

1. copiamos el sistema base de alguno de estos 3 directorios, contando con que Open-R está instalado en /usr/local/OPEN R:

/USR/LOCAl/OPEN R SDK/OPEN R/MS/BASIC/. Para trabajar sin conexión wireless.

/USR/LOCAl/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/LOCAl/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.

Para configurar la red en el aibo 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 robot, 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 valores 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

Compilando, instalando y ejecutando

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 archivos ".BIN" son realmente los ejecutables y corresponden a un objeto Open-R. El archivos "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.

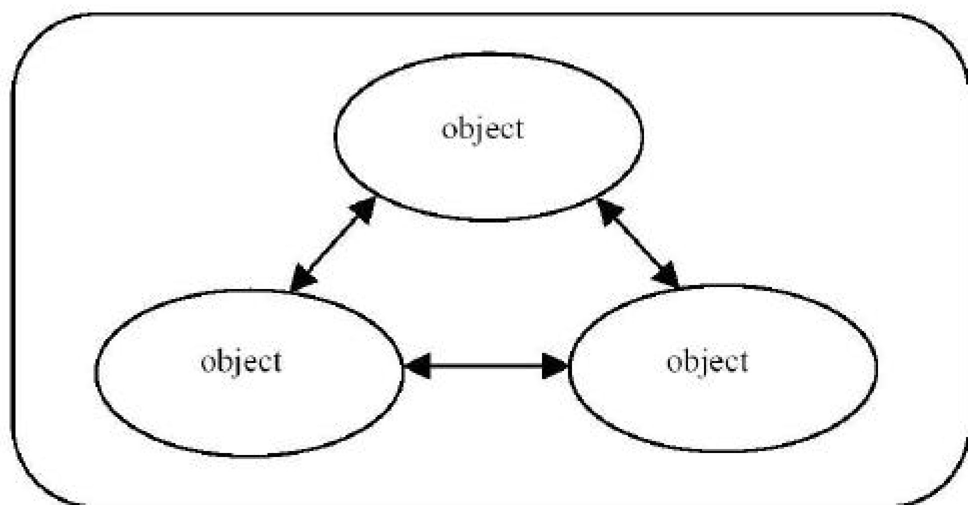
Una última advertencia es no sacar la tarjeta ni la batería durante la ejecución. Podríamos dañar el sistema.

Comunicaciones entre objetos Open-R

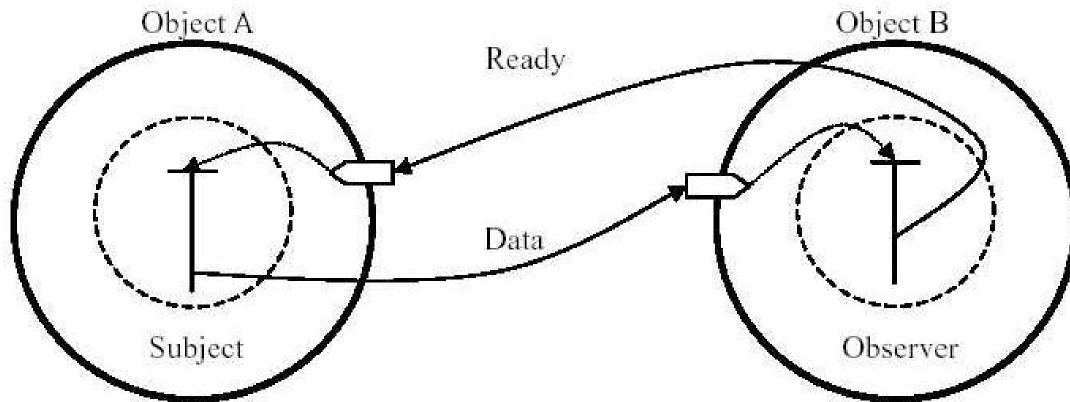
En el ejemplo anterior se desarrollo una aplicación que constaba de un único objeto Open-R. Esto no es lo habitual cuando desarrollamos nuestras aplicaciones. Lo normal es que una aplicación esté formada por un conjunto de objetos Open-R que cooperan entre sí para realizar un trabajo conjunto.

Ahora se explicará 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-thread, por lo cual, si llega un mensaje mientras se está procesando otro, Éste se encolará para ser procesado posteriormente.



Los objetos se comunican entre sí por paso de mensajes



Flujo de comunicación entre objetos

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 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.

Para ilustrar la comunicación entre objetos utilizaremos 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 al observer, y éste los imprime en la consola.

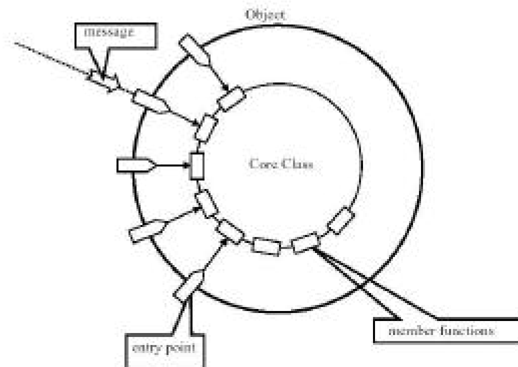
Definiendo los manejadores de los mensajes

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 que métodos la procesarán se hace en un archivo llamado **stub.cfg**. Este archivo 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.



Comunicaciones en una core class

stub.cfg del SampleSubject

SampleSubject será 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 archivo 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 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 se puede concluir que 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

Hay otro tipo de entradas que pueden aparecer en este archivo, pero que no contemplamos en éste 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.

stub.cfg del SampleObserver

SampleObserver será el objeto que hará el rol de observer en el ejemplo "Object-Comm". 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()
```

De la misma manera se puede analizar SampleSubject.

Declarando las comunicaciones en el encabezado

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 archivo de encabezados.

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 (HelloWorld), 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);
19
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
```

Iniciando y parando la comunicación

En el ejemplo anterior (HelloWorld), los métodos DoInit(), DoStart(), DoStop() y DoDestroy() estaban prácticamente vacíos y no realizaban ninguna función. Pues bien, en el ejemplo actual se usará una serie de macros para registrarse y activarse en 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;
20
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 Si queremos que un objeto Open-R se comunique con otro, deben ser incluidas en el 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.

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:

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 del documento Level2ReferenceGuide de la documentación de Open-R.

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 inicializamos la variable text con los datos que se encuentran en la zona de memoria definida por event.Data(0), y le hacemos un cast 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 a la puerta por la que hemos recibido los datos (event.ObsIndex()), que estamos preparados para recibir más datos.

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 archivo /MS/OPEN-R/MW/CONF/CONNECT.CFG de la Memory Stick, que contiene pares de subjects-observers que han de ser conectados.

Cada línea es una comunicación y deben tener el mismo tipo. Para el ejemplo, éste archivo contendrá una única línea conectando las dos puertas reales que tenemos:

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