Contenido

[Introducción 2](#_Toc335382375)

[¿Qué es BIT? 2](#_Toc335382376)

[¿Dónde puedo aplicar BIT? 2](#_Toc335382377)

[¿Por qué necesito BIT? 3](#_Toc335382378)

[Instalación. 4](#_Toc335382379)

[El editor de mensajes. 6](#_Toc335382380)

[Uso del editor de mensajes. 6](#_Toc335382381)

[Otras consideraciones. 13](#_Toc335382382)

[Generación de código 14](#_Toc335382383)

[El generador de documentos HTML 15](#_Toc335382384)

[El generador de APIs C. 16](#_Toc335382385)

[El generador de APIs C++. 21](#_Toc335382386)

[El simulador de mensajes. 25](#_Toc335382387)

## Introducción

### ¿Qué es BIT?

Binary Interface Tools (BIT), es un conjunto de herramientas orientadas al desarrollo y prueba de mensajería binaria.

BIT entiende la mensajería binaria como un conjunto de campos de bits, de tamaño fijo, que se apilan de una manera determinada y cumplen una serie de requisitos.

BIT se compone de un editor visual, que permite definir la estructura de cada mensaje, y de una serie de generadores de código que producen librerías C/C++ para la gestión de dichos mensajes.

Además incluye la posibilidad de generar documentación en HTML a partir de su mensajería, y de producir el código fuente de un simulador que, una vez compilado, proporciona una herramienta útil para el desarrollador.

### ¿Dónde puedo aplicar BIT?

BIT es útil en aquellos casos donde se necesite definir una mensajería binaria entre dos o más dispositivos.

La mensajería debe cumplir los siguientes requisitos:

* Una mensajería se compone de mensajes independientes. Los datos contenidos dentro de un mensaje no pueden tener dependencias de los datos contenidos en otro mensaje diferente.
* Un mensaje se compone de "dataItems" o estructuras que engloban uno o más campos. Los dataItems pueden ser únicos o bien pueden ser repetitivos (arrays de estructuras). En caso de ser repetitivos, su multiplicidad puede ser fija o bien puede ser dependiente de otro campo del mismo mensaje.
* Un dataItem se compone de campos. El número de campos definidos para cada dataItem es fijo, y siempre deberán figurar en el mismo orden. Hay dos tipos de campos definibles; los "dataFields" o campos de bits y los "datablocks" o búferes de bytes. Para los datablocks se pueden definir tamaño máximo y dependencia a otros campos del mismo mensaje.
* Dentro de un mismo mensaje, no puede haber dos dataItems ni dos dataFields con el mismo nombre.

Con dichos requisitos se abarca una parte muy importante de los protocolos que habitualmente se utilizan en ingeniería. No obstante se quedan fuera las mensajerías basadas en textos ASCII y aquellas cuyos campos pueden ser opcionales (ASTERIX por ejemplo).

### ¿Por qué necesito BIT?

La definición, implementación, prueba y documentación de la mensajería entre dos dispositivos suele ser una tarea recurrente en el desarrollo de cualquier proyecto software.

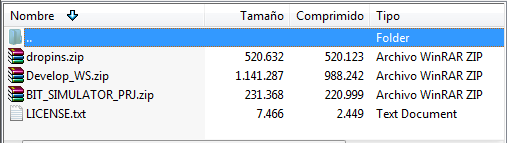
BIT pretende proporcionar un procedimiento ordenado para el desarrollo de estas tareas, agilizando el desarrollo y automatizando la generación del código fuente relativo a la gestión de mensajes.

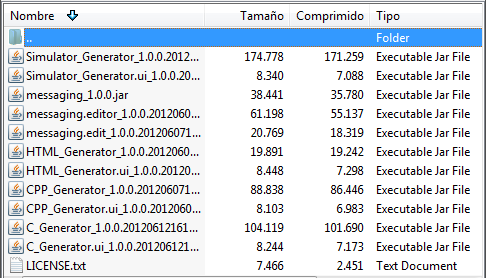
Al generar el código de forma automática, se ahorra tiempo y se minimiza la posibilidad de cometer errores de codificación.

Por otro lado, si las partes implicadas en la comunicación de mensajes utilizan librerías generadas a partir de una misma mensajería editada con BIT, se evitan muchos de los problemas que pueden aparecer durante la integración del sistema.

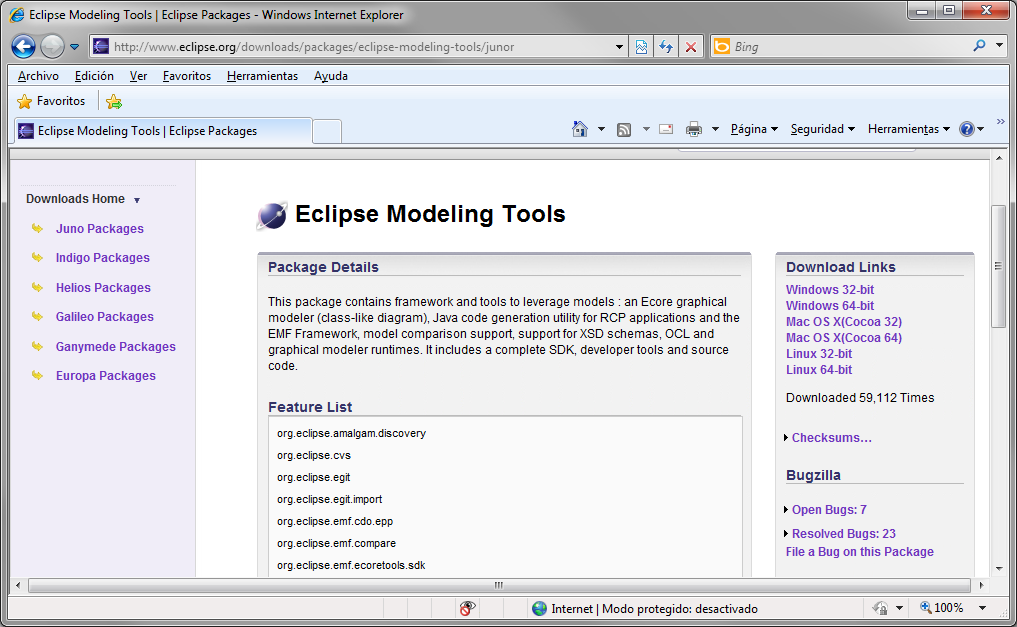
## Instalación.

BIT se entrega como un paquete de ficheros .jar, comprimidos en formato zip, que deben ser instalados manualmente en la carpeta "dropins" existente en el directorio donde tiene instalado Eclipse.





La instalación de Eclipse deberá disponer de los módulos Eclipse-EMF y Acceleo. Dichos módulos se encuentran disponibles desde el paquete *"Eclipse Modeling Tools"* que puede encontrar en la zona de descarga de la página oficial de <http://www.eclipse.org/downloads/>



También es recomendable que su instalación de Eclipse disponga del módulo Eclipse-CDT para poder visualizar y compilar el código fuente C/C++ que genere.



Eclipse CDT

Acceleo & Modeling

Junto con los plugins de Eclipse, se adjunta un proyecto C++ que dispone del código fuente necesario para compilar un simulador que "reconoce" su mensajería. Los detalles relativos a la compilación y uso del simulador se abordarán en las siguientes secciones.

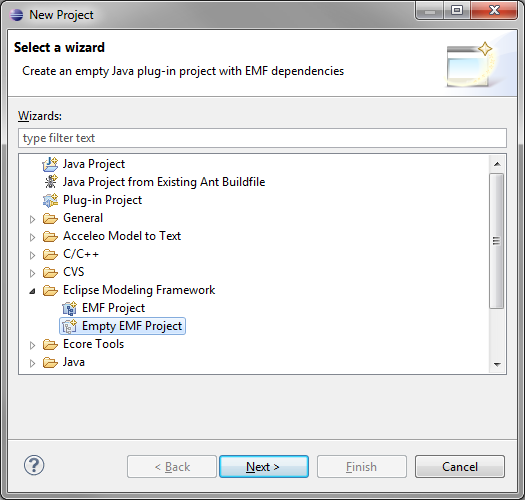
## El editor de mensajes.

El punto de inicio para empezar a trabajar con BIT es el editor de mensajes. El editor permite definir un *"ICD"* o mensajería en el que deberá ir añadiendo mensajes, dataItems y campos. Así mismo deberá ir rellenando uno por uno los atributos que afecten a cada uno de los elementos mencionados para definir su estructura y comportamiento.

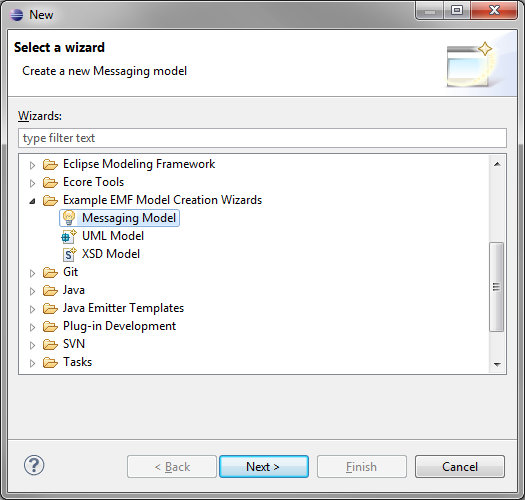
### Uso del editor de mensajes.

Vamos a definir una mensajería de ejemplo para abordar el uso del editor de mensajes.

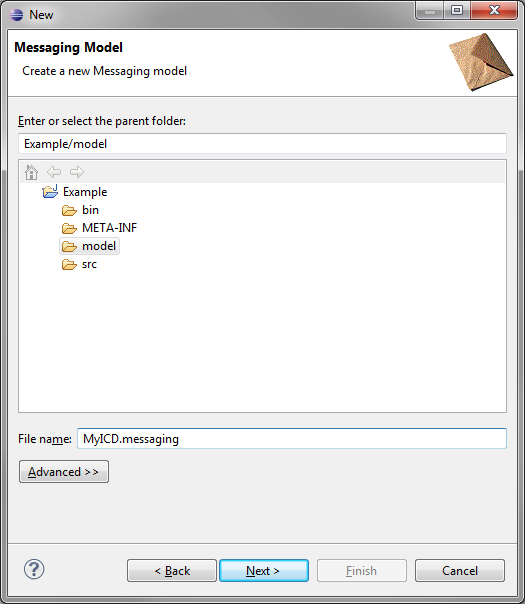
En primer lugar creamos un nuevo proyecto de modelado *"Empty EMF Project"*.



Y a continuación añadimos, dentro de la carpeta *"model"* del proyecto que acabamos de crear, un modelo del tipo *"Messaging Model"*.



Le asignamos nombre. Para nuestro ejemplo elegimos *"MyICD.messaging".* Después seleccionamos el *"Model Object"*, que **siempre deberá ser *"ICD"****,* y por último pulsamos *"Finish"*.

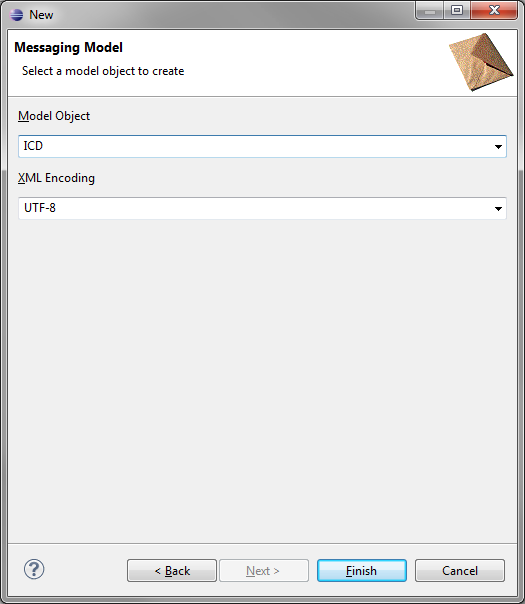
***Recuerde que el elemento base de toda mensajería definida con BIT es el ICD.***

***Del ICD cuelgan mensajes, que a su vez estarán compuestos de dataItems.***

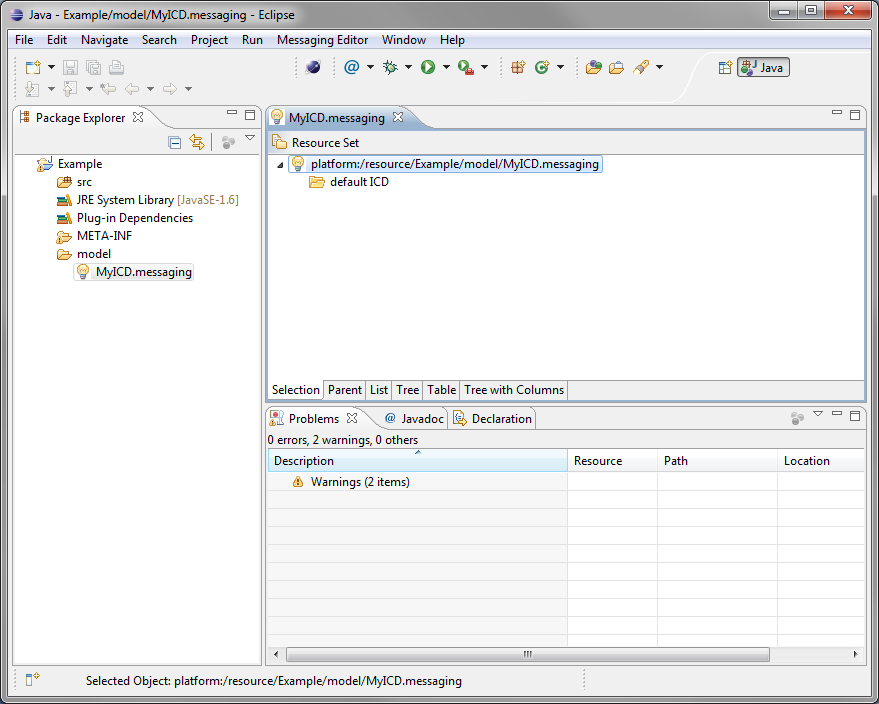
***Los dataItems actúan como estructuras de datos. Dichos datos serán los bitFields y los dataBlocks****.*

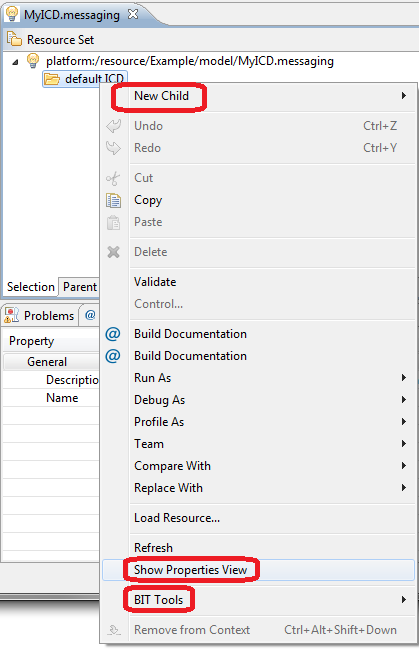
*Los bitFields son campos de bits con tamaños definibles entre 1 y 32 bits.*

*Los dataBlocks son bloques de bytes o búferes cuyo tamaño máximo deberá ser especificado.*



Llegados a este punto deberíamos tener algo parecido a esto:



En la pestaña *"MyICD.messaging"* se observa un editor de mensajes en el que hay un icono con forma de bombilla. Si pulsa en el triángulo que hay a la izquierda de la bombilla, se despliega un árbol con un único nodo llamado *"default ICD"* cuyo icono es el de una carpeta amarilla.

Al pulsar con el botón derecho sobre el icono de la carpeta, se despliega un menú en el que aparecen:

* Un menú *"New Child",* que sirve para ir añadiendo elementos a nuestro modelo. Dependiendo de qué elemento este seleccionado al mostrar el menú, nos permitirá añadir nuevos mensajes, dataItems y campos.
* Un menú *"BIT Tools"* que contiene los generadores de código que procesarán la mensajería que definamos.
* Una opción *"Show Properties View"* que despliega una vista donde podremos editar los atributos de cada uno de los elementos que insertemos en el modelo de mensajería.

Jugando con el menú *"New Child"* y los atributos que se despliegan en la vista *"Properties"*, que aparece al pulsar sobre *"Show Properties View*", podemos añadir mensajes a nuestro ICD.

Para nuestro ejemplo vamos a imaginar un escenario en el que un dispositivo controlador se comunica de forma inalámbrica con pequeños dispositivos de sensorización, distribuidos a los largo de un edificio, que recogen la temperatura, humedad, etc del lugar donde están situados.

Vamos crear un primer mensaje llamado *"Alive"*, que permitirá conocer al dispositivo controlador si los dispositivos de sensorización están disponibles. Dispondrá de un único dataItem llamado *"Header"*.

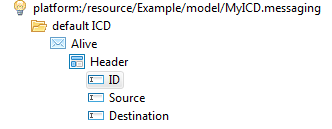
En el dataItem *"Header"* insertaremos tres campos:

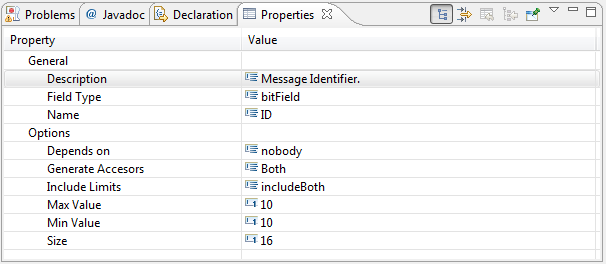
* Campo *"ID"* de 16 bits que identifica inequívocamente el tipo de mensaje. Tendrá un valor fijo de 10 en decimal.
* Campo *"Source"* de 8 bits que especifica quién envía el mensaje. Tomará valores entre 0 y 20, siendo 0 el dispositivo controlador.
* Campo *"Destination"* de 8 bits que especifica a quién va dirigido el mensaje. Tomará valores entre 0 y 20, siendo 0 el dispositivo controlador.



Header

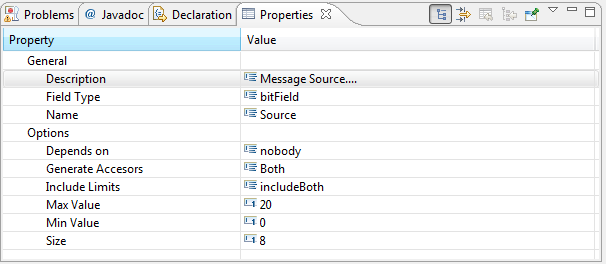
En el editor de mensajes, nuestro mensaje de *Alive* quedaría así:

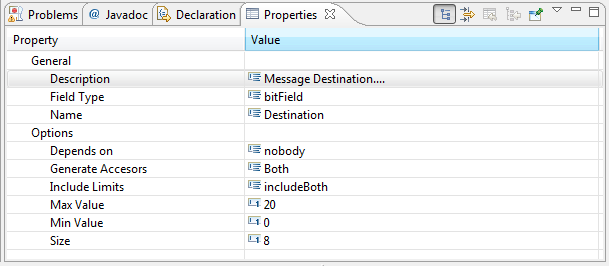


Como el atributo ID queremos que tome un valor constante e igual a 10, deberemos hacer que sus atributos *"Max Value"* y *"Min Value"* coincidan y sean iguales a 10.

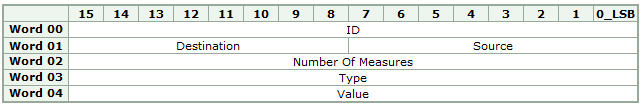
*Al rellenar los atributos de cada elemento, es aconsejable rellenar el atributo "Description", ya que luego figurará como parte de la documentación.*

Los atributos de los campos *"Source"* y *"Destination"* quedarían así:





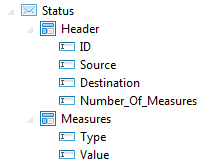
A continuación vamos a crear un nuevo mensaje llamado *"Status"*, con el que los dispositivos de sensorización reportarán las medidas realizadas. Cada dispositivo dispone de un número de sondas variable (10 como máximo). Para cada una de las sondas conectadas, se notificará tanto el tipo de medida realizada como el valor de la medida.

El mensaje debería quedar así:

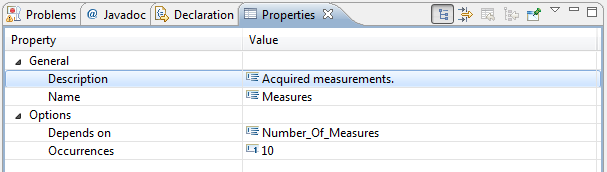
Header

Measures

El dataItem *"Measures"* contiene los campos *"Type"* y *"Value"* y además deberá ser repetitivo. Para ello asignaremos el atributo *"Occurrences"* a 10. Con esto obtenemos un array de estructuras con un máximo de 10 elementos.

Además se deberá repetir tantas veces como indique el campo *"Number\_Of\_Measures"*, por lo que el atributo *"Depends\_on"* deberá tomar el valor *"Number\_Of\_Measures"*.

Los atributos del dataItem *"Measures"* quedarían así:



Asegúrese de que el campo *"Number\_Of\_Measures"* toma valores comprendidos entre 0 y 10, o de lo contrario el número de medidas realizadas podría exceder del máximo especificado para el dataItem *"Measures"* y el mensaje no sería consistente.

Una inconsistencia en el diseño de la mensajería podría traducirse con un error tipo *"out of bounds"* en la librería C/C++ que luego genere. Por ese motivo sea cuidadoso a la hora de realizar el diseño.

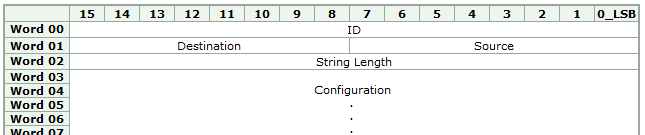
Como identificador de este mensaje utilice *ID = 20*. Procure incluir en cada mensaje de su diseño por lo menos un campo de valor constante para facilitar, a las librerías generadas, la identificación de mensajes en decodificación.

El campo *"Type"* especificará el tipo de sonda conectada, y podrá tomar valores comprendidos entre 0 y 99.

El campo *"Value"* tendrá una longitud de 16 bits y dispondrá de un rango de valores comprendido entre 0 y 65535. Dependiendo del tipo de sonda conectada, el valor de la medida tendrá un significado diferente.

Por último, crearemos otro mensaje *"SendConfig"* con el que cada dispositivo de sensorización notificará al controlador qué configuración está utilizando. La configuración vendrá dada en formato XML (texto plano) por lo que será necesario incluir un campo de tipo *"dataBlock"* llamado *"Configuration"* cuyo tamaño sea variable y dependiente de otro campo *"String\_Length"*. El campo *"Configuration"* tendrá un tamaño máximo de 1024 bytes.

El mensaje podría tener la siguiente forma:

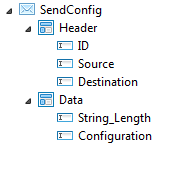


1024 bytes for XML file

Data

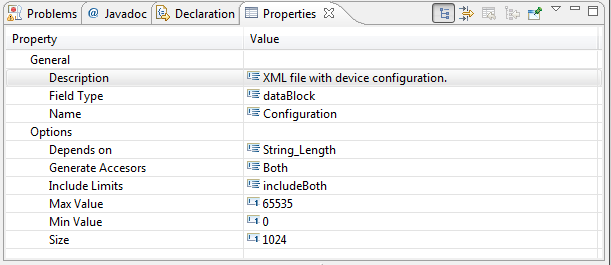
Header

Y su implementación con BIT debería verse del siguiente modo:

Para este mensaje el campo *"ID"* tomará el valor constante 30.

Además el campo *"String\_Length"* tomará valores comprendidos entre 0 y 1024. El rango deberá estar acorde con el tamaño máximo del campo *"Configuration"* para evitar inconsistencias.

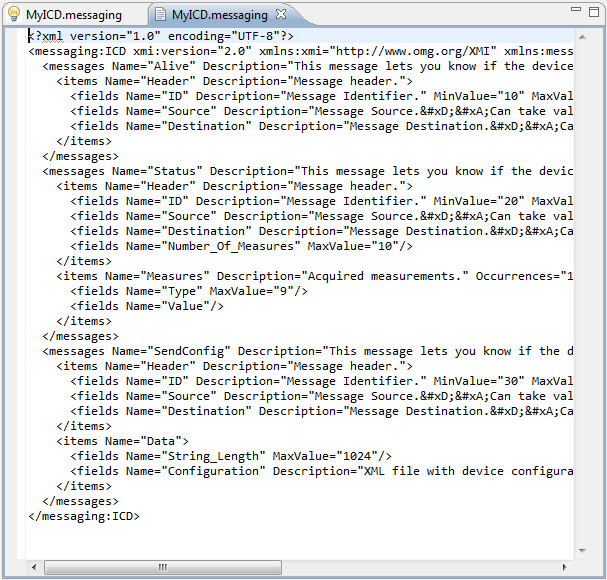
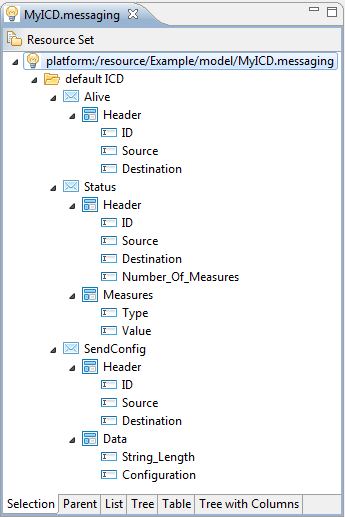
Los atributos del campo *"Configuration"* deberán ser los siguientes:

*Observe como el tamaño del dataBlock, expresado en bytes, viene especificado por el atributo "Size".*

La dependencia de tamaño del *dataBlock* viene determinada por el atributo *"Depends on"*, que en este caso deberá estar fijado al valor *"String\_Length"*.

Una vez diseñada la mensajería dispondremos de un fichero "MyICD.messaging" que contiene, en formato XML, la definición de todos nuestros mensajes.

En el editor de mensajes, presenta el siguiente aspecto:



### Otras consideraciones.

Entre los atributos disponibles a la hora de configurar dataFields, figuran *"Generate Accesors",* e *"Include Limits"*. Por lo general, lo valores asignados por defecto son válidos para una generación de código satisfactoria.

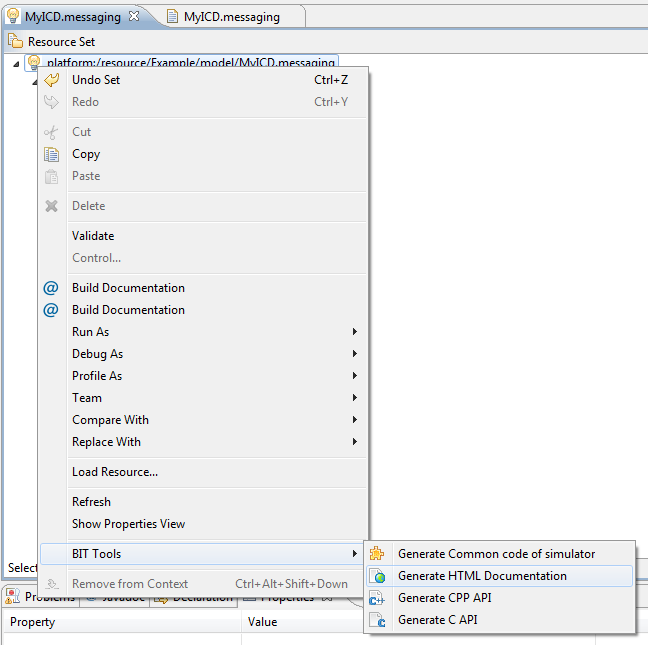
El atributo *"Include Limits"* indica si los valores *"Min Value"* y *"Max Value"* deben ser incluidos en el rango de valores válidos para el campo. Puede tomar los valores *"includeBoth"*, *"minLimitOnly"*, *"maxLimitOnly"*, y *"noIncludeLimits"*. El rango seleccionado tendrá efecto a la hora de generar el código fuente de las APIS que gestionan la mensajería, y solo afecta a los campos de tipo *"bitField"*.

El atributo *"Generate Accesors"* indica si a la hora de generar código se deben incluir solo Getters, solo Setters o Getters y Setters. Los campos de bits de valor constante solo disponen de Getters, independientemente de lo que configure en este atributo. A menos que tenga muy claro lo que significa este atributo, es recomendable asignar siempre el valor *"Both"*.

## Generación de código

A partir de nuestra mensajería podemos generar documentación HTML, una API en C y una API C++.

Para generar código, pulsamos con el botón derecho del ratón sobre el icono de la bombilla. Se despliega un menú en el que figura el submenú "BIT Tools".



El submenú "BIT Tools" dispone de herramientas que generan:

* Documentación en HTML
* API, en lenguaje C, de manejo de mensajería.
* API, en lenguaje C++, de manejo de mensajería.

Las APIs de manejo y gestión de mensajería disponen de las herramientas necesarias para manipular la mensajería:

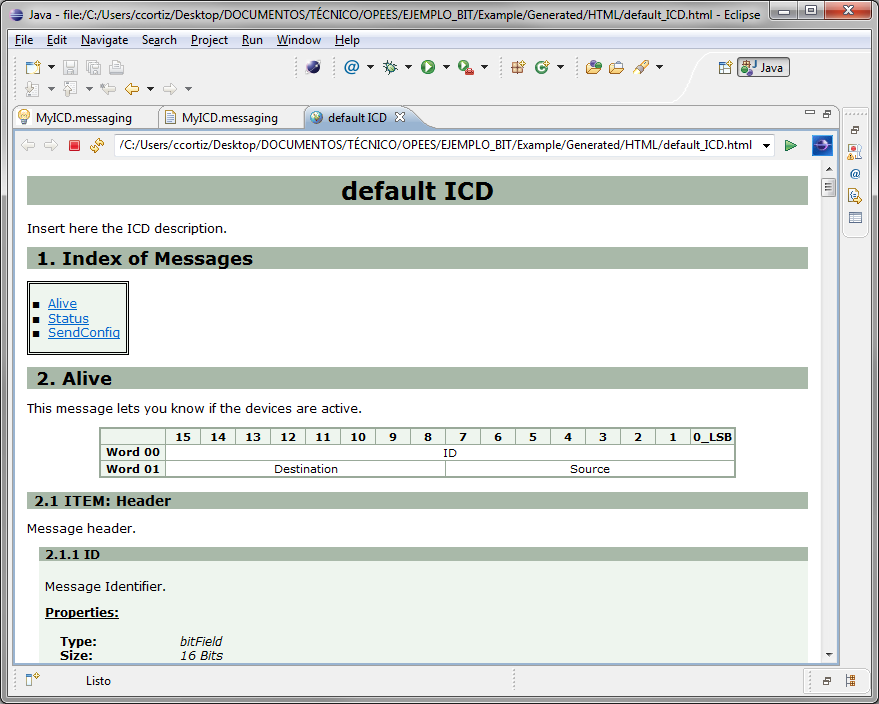
* Accesors (getters y setters) para los campos de los mensajes.
* Serializadores (codificadores de mensajes).
* Deserializadores (decodificadores de mensajes).
* Herramientas de búsqueda e identificación automática de mensajes.

Adicionalmente se incluye un generador que proporciona código fuente para la generación de un simulador que se adapta a la mensajería diseñada.

### El generador de documentos HTML

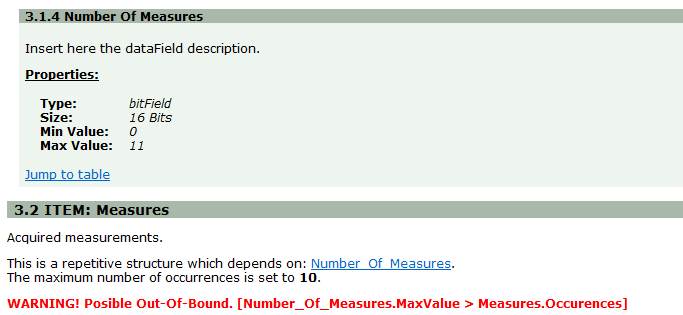
Al pulsar sobre la opción ***"Generate HTML Documentation"*** del menú *"BIT Tools"*, se activa el generador de código que genera documentación HTML.

En nuestro ejemplo, la documentación HTML generada consiste en un único fichero HTML, que cuelga de la carpeta "Generated/HTML" de nuestro proyecto y presenta el siguiente aspecto:



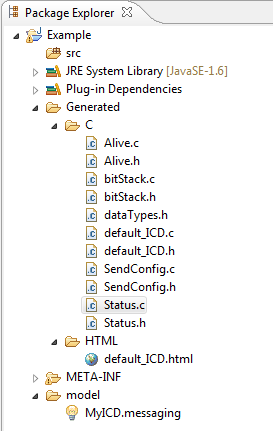
El documento generado, dispone de un índice mensajes y de la definición de todos y cada uno de los mensajes diseñados. Cada mensaje documentado incluye una tabla que representa la estructura del mensaje, y la descripción detallada de los dataItems y campos definidos en la mensajería.

Existen enlaces a las diversas zonas del documento para facilitar la navegación a través de él.

Además, el generador de HTML es capaz de detectar inconsistencias en la definición de los mensajes y le advierte sobre ello mediante avisos de color rojo.

### El generador de APIs C.

Al pulsar la opción "Generate C API" del menú *"BIT Tools"*, se activa el generador de código que proporciona la API, en lenguaje C, que permite manipular su mensajería.

La API generada consiste en un conjunto de ficheros .h y .c, que cuelgan de la carpeta "Generated/C" de nuestro proyecto.

Por cada uno de los mensajes definidos se genera un par de ficheros con el nombre del mensaje y extensiones .h y .c, que proporcionan las estructuras y funciones necesarias para manejar los datos de ese mensaje.

Se incluyen los ficheros *"bitStack.h"* y *"bitStack.c"*, que implementan las funciones de apilado de bits necesarias para el funcionamiento de la librería, y un fichero *"dataTypes.h"* que define los tipos utilizados por la librería.

En *"dataTypes.h"* además se define la macro *\_DEBUG\_* que , de estar disponible, proporciona mensajes de depuración a través de la salida estándar del sistema.

Junto con los ficheros mencionados, se adjuntan los ficheros, "default\_ICD.h" y "default\_ICD.c", que sirven como punto de entrada a la librería que se genera. El nombre de estos ficheros depende directamente del nombre que asigne a su ICD en el editor de mensajes.

El desarrollador que desee integrar la API generada dentro de su propio programa únicamente deberá incluir el fichero *"default\_ICD.h"*.

A partir de ahí tendrá todo lo necesario para acceder a todos los elementos de todos los mensajes, sin necesidad de entender todos los mecanismos subyacentes que funcionan por debajo de la librería.

#### Codificación de mensajes.

A continuación se presenta un ejemplo de uso de la librería generada en C, en el que se muestra cómo editar y codificar mensajes.

El mensaje que se desea codificar es el mensaje de *"Alive"* que definimos en nuestra mensajería de ejemplo.

El mensaje dispone de 3 campos, de los cuales el campo *"ID"* es de valor constante y por tanto no tiene *setter*. Los otros dos campos, *"Source"* y *"Destination"* los pondremos a 0 y a 1 respectivamente.

El código fuente que utilizamos es el siguiente:



Lo primero que observamos es que para utilizar la librería que hemos generado, basta con incluirla en nuestro programa.

A continuación deberemos obtener una instancia de la librería mediante el uso de la función ***new\_Default\_ICD(),*** que retorna un descriptor ***myICD*** que necesitaremos más adelante para poder utilizar la API. Tenga en cuenta que la función se llama "*new\_Default\_ICD()"* porque *"Default ICD"* es el nombre que se asigno al ICD. En caso de haber recibido un nombre distinto, la función se llamaría ***"new\_"*** seguido del nombre asignado.

A partir de ahí, se pueden empezar a utilizar los *getters* y *setters* de los mensajes definidos. El primer parámetro que se pasa a todas las funciones es el descriptor obtenido en la función *new\_Default\_ICD()*. **Gracias al uso de descriptores se pueden manejar simultáneamente varias instancias de la librería**.

Los *setters* siempre comienzan con el prefijo *"set\_"* seguido del nombre del campo que desea modificar. Así mismo, los *getters* comienzan con el prefijo *"get\_"* seguido del nombre del campo del que desea obtener valores.

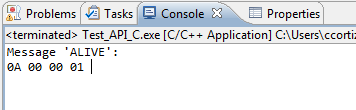
Los *getters* y los *setters* cuelgan directamente de los mensajes a los que pertenecen. Del mismo modo, los mensajes cuelgan directamente de **la instancia genérica *"Default\_ICD"***, que está predefinida en la librería generada. *"Default\_ICD"* coincide con el nombre que se designó para el ICD en el editor de mensajes.

Después de aplicar valores a cada uno de los campos del mensaje *Alive*, generamos el buffer del mensaje con la función ***serialize()***.

Así mismo podemos obtener el tamaño del buffer generado mediante el uso de la función ***get\_MsgSize()***.

El buffer generado garantiza que cada campo está situado y alineado correctamente atendiendo al diseño que se realizó desde el editor de mensajes.

Al ejecutar el programa obtenemos lo siguiente:



El valor 0x0A, 0x00 coincide con el ID del mensaje, cuyo valor era fijo e igual a 10 (0xA en hexadecimal).

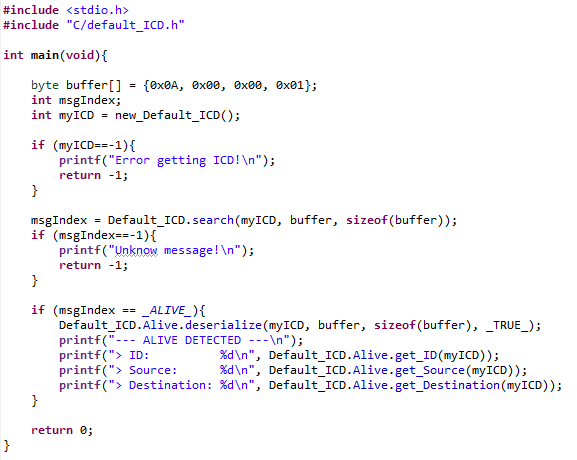
Los valores 0x00 y 0x01 coinciden respectivamente con los campos "Source" y "Destination" de 8 bits, que de acuerdo con el código del ejemplo están a 0 y a 1.

#### Decodificación de mensajes.

A continuación se presenta un ejemplo de uso de la librería generada en C, en el que se muestra cómo decodificar mensajes.

Vamos a decodificar un buffer de prueba, cuyo contenido representa un mensaje tipo *"Alive"* con los campos "Source" y "Destination" asignado a los valores 0 y 1 respectivamente. El contenido del buffer coincide con el resultado obtenido en el ejemplo anterior de codificación de mensajes.

El código fuente que utilizaremos es el siguiente:



Inicialmente obtendremos un descriptor para la API, y después utilizaremos la función **search()** para tratar de detectar automáticamente el tipo de mensaje contenido en el buffer.

La función *search()* recibe, como parámetros, el descriptor de su API, el buffer que desea analizar y la longitud de dicho buffer. Si a la función *search()* se le pasa un buffer que no es capaz de identificar, retorna el valor -1, en caso contrario devuelve un entero que representa el tipo de mensaje detectado.

Una vez identificado el tipo de mensaje, procederemos a su decodificación mediante el uso de la función ***deserialize()***.

La función *deserialize()* recibe, como parámetros, el descriptor de su API, el buffer que desea analizar, la longitud de dicho buffer y un cuarto parámetro que si toma el valor \_TRUE\_ decodifica el mensaje y asigna los valores decodificados a los campos de su mensaje. Si toma el valor \_FALSE\_, únicamente comprobará si el mensaje es decodificable, en cuyo caso devolverá el valor *\_DESERIALIZE\_OK\_.*

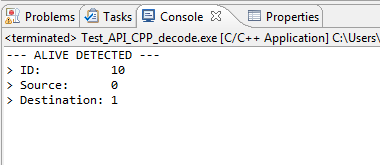
Si trata de pasar a la función *deserialize()* un buffer que no es capaz de decodificar, entonces retorna un valor distinto de *\_DESERIALIZE\_OK\_*.

Un mensaje se considera inválido en decodificación cuando:

* No tiene el tamaño esperado.
* Los campos definidos como constantes no tienen el valor que deberían.
* Los campos que se van obteniendo llevan valores que se salen del rango válido definido para cada campo.

Por último accederemos al contenido de los campos mediante el uso de sus **getters**.

El resultado de ejecutar el ejemplo es el siguiente:



#### Otras consideraciones.

Cuando maneja campos de tipo **dataBlock** (arrays de bytes), la forma de acceder a su valor es la siguiente:

* ***get\_dataBlock***(*descriptor*, *destBuffer*)
* ***set\_dataBlock***(*descriptor*, *srcBuffer*)

Observe que no es necesario especificar el tamaño de los búferes, ya que la librería sabe cuántos bytes debe copiar dependiendo del campo que este manipulando. En el caso del getter, es responsabilidad del desarrollador reservar la memoria suficiente para almacenar el buffer que la función retorna.

Cuando los campos a codificar/decodificar pertenecen a un dataItem repetitivo, el formato de los getter/setter varía ligeramente.

Si para un campo contenido en un dataItem no repetitivo la forma de acceder a su valor es:

* ***get\_Field***(*descriptor*)
* ***set\_Field***(*descriptor*, *value*)

En el caso de utilizar dataItems repetitivos las funciones quedan así:

* ***get\_Field***(*descriptor*, *index*)
* ***set\_Field***(*descriptor*, *index*, *value*)

Si además el campo es un dataBlock, las funciones de acceso quedan así:

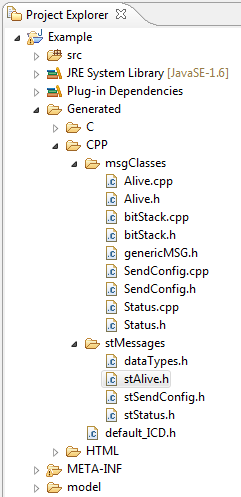
* ***get\_dataBlock***(*descriptor*, *index*, *destBuffer*)
* ***set\_dataBlock***(*descriptor*, *index*, *srcBuffer*)

Para los búferes y dataItems repetitivos, se considera que el primer elemento es aquel cuyo índice es igual a 0.

### El generador de APIs C++.

Al pulsar la opción "Generate CPP API" del menú *"BIT Tools"*, se activa el generador de código que proporciona la API en lenguaje C++ que permite manipular su mensajería.

La API generada consiste en un conjunto de ficheros .h y .cpp, organizados en las carpetas *msgClasses* y *stMessages*, que cuelgan de la carpeta *"Generated/CPP"* de nuestro proyecto.

El concepto que se aplica en esta librería es muy similar al visto para la librería C, con la diferencia de que en este caso la librería se apoya en la orientación a objetos para garantizar la encapsulación y ofrecer la posibilidad de generar múltiples instancias de la API.

La forma de utilizar la API C++ es muy parecida a la utilizada en la API C. Existen métodos *setter/getter, serialize(), getMsgSize(),* etc.

Para utilizarla, desde su programa deberá incluir el fichero "default\_ICD.h" y después de obtener una instancia de la API, podrá utilizar las características de codificación y decodificación que se le brindan.

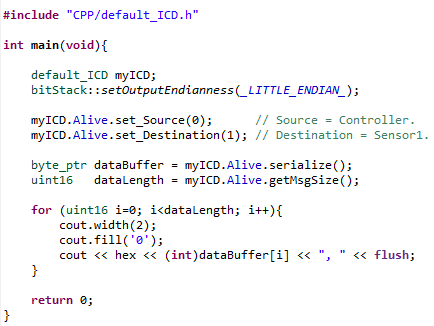
Tenga en cuenta que, del mismo modo que ocurría con la API C, el fichero "default\_ICD.h" recibe su denominación del nombre que le haya asignado al ICD en el editor de mensajes.

#### Codificación de mensajes.

A continuación se presenta un ejemplo de uso de la librería generada en C++, en el que se muestra cómo codificar mensajes.

El mensaje que se pretende codificar es el mensaje de "Alive", que dispone de los campos "ID", de valor constante e igual a 10, "Source" y "Destination". A estos últimos se les asignarán los valores 0 y 1 respectivamente.

El código empleado para realizar la codificación es el siguiente:



Lo primero que se hace es obtener una instancia de la clase *"default\_ICD".* El nombre de la clase *"default\_ICD"* coincide con el nombre asignado al ICD.

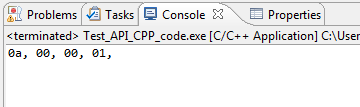
Sea cuidadoso al seleccionar el endianness apropiado para la codificación/decodificación de los mensajes. Para ello utilice el método estático *setOutputEndianness()* de la clase *bitStack*.

Observe como, después de crear la instancia *"myICD"* de la clase *"default\_ICD"*, ya no es necesario manejar descriptores como ocurría en el caso de la librería C.

Después, mediante los métodos *set\_Source()* y *set\_Destination()*, podemos asignar los valores deseados a los campos del mensaje.

Para finalizar, recurrimos a los métodos *serialize()* y *getMsgSize()* del mensaje *Alive* para codificar el mensaje en un búfer de bytes y obtener su tamaño.

El resultado de ejecutar el programa es el siguiente:

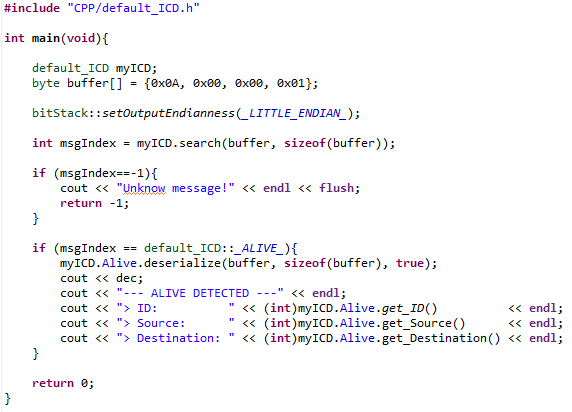


#### Decodificación de mensajes.

A continuación se presenta un ejemplo de uso de la librería generada en C++, en el que se muestra cómo decodificar mensajes.

Utilizando el buffer obtenido en el ejemplo anterior como base para este ejemplo, vamos a ver cómo utilizar las características de identificación automática y decodificación de mensajes.

El código empleado para realizar la decodificación es el siguiente:



Primero se obtiene la instancia *myICD* de la clase *default\_ICD* y especificamos el endianness que deseamos aplicar en la codificación/decodificación de nuestros mensajes.

A continuación utilizamos el método *search()* pasando como parámetros el buffer que deseamos analizar y su tamaño. En caso de que el mensaje sea reconocido por la librería, su identificador será retornado y guardado en la variable entera *msgIndex*. En caso contrario, el método retornará el valor -1.

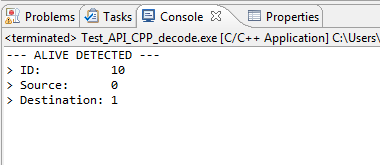
Una vez identificado el mensaje, recurrimos a decodificarlo y almacenar los valores del buffer dentro del mensaje. En este caso, el mensaje identificado es de tipo *\_ALIVE\_*.

Para decodificar el mensaje utilizamos el método *deserialize()*, pasándole como parámetros el buffer, su tamaño y un booleano que indica si deseamos decodificar el mensaje (true) o bien comprobar si el búfer que se pasa como parámetro es decodificable como mensaje tipo *Alive* (false).

Si *deserialize()* decodifica el buffer de forma satisfactoria, retorna *\_DESERIALIZE\_OK\_.*

Después de la decodificación, se puede acceder al contenido del mensaje decodificado mediante el uso de los getters apropiados.

El resultado de ejecutar el programa es el esperado:



#### Otras consideraciones.

Cuando maneja campos de tipo **dataBlock** (arrays de bytes), la forma de acceder a su valor es la siguiente:

* ***get\_dataBlock***(*destBuffer*)
* ***set\_dataBlock***( *srcBuffer*)

Cuando los campos a codificar/decodificar pertenecen a un dataItem repetitivo, el formato de los getter/setter varía ligeramente.

Si para un campo contenido en un dataItem no repetitivo la forma de acceder a su valor es:

* ***get\_Field*** ( )
* ***set\_Field*** ( *value* )

En el caso de utilizar dataItems repetitivos las funciones quedan así:

* ***get\_Field*** ( *index* )
* ***set\_Field*** ( *index*, *value* )

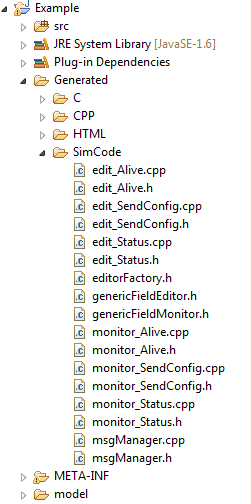
Si además el campo es un dataBlock, las funciones de acceso quedan así:

* ***get\_dataBlock***( *index*, *destBuffer* )
* ***set\_dataBlock***( *index*, *srcBuffer* )

Para los búferes y dataItems repetitivos, se considera que el primer elemento es aquel cuyo índice es igual a 0.

### El simulador de mensajes.

Junto con los generadores HTML/C/C++ se adjunta un generador de código que proporciona parte del código fuente de un simulador específico para la mensajería diseñada.

El simulador proporcionado permite editar el contenido de cada uno de los mensajes definidos y enviarlos a través de un canal de comunicación configurable (TCP, UDP, RS232).

También permite recibir tramas a través del canal de comunicación configurado e identificarlas como mensajes conocidos por nuestra mensajería.

Así mismo, permite monitorizar el envío y recepción de mensajes y guardar logs de la sesión.

El simulador, además de servirle como herramienta para el desarrollo y test de sus aplicaciones, sirve para demostrar la validez de la API C++ en la que se basa.

El código generado para el simulador cuelga de la carpeta *"Generated/SimCode"*.

El código fuente generado deberá ser sumado al código común del simulador, que se proporciona como parte de la distribución de BIT, y a la API C++ generada a partir de la misma mensajería.

De este modo, el código fuente del simulador se compone de:

* Código fuente común del simulador.
* Código fuente generado para el simulador.
* API C++ generada con BIT.

Generadores BIT

API C++

Código generado del simulador

Código fuente del simulador

Código común del simulador

Además, para poder compilar el proyecto deberá tener instalada la librería JUCE en su versión 1.52. La librería JUCE es GPL y gratuita para uso no comercial, aunque existen licencias comerciales en la web del autor:

<http://www.rawmaterialsoftware.com/juce.php>

Como compilador C/C++ se recomienda MinGW, que tiene licencia GPL y, además de ser gratuito para fines no comerciales, se integra perfectamente en Eclipse CDT.

<http://www.mingw.org/>

Código fuente del simulador

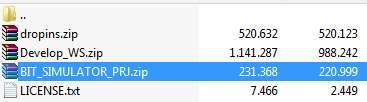
JUCE 1.52

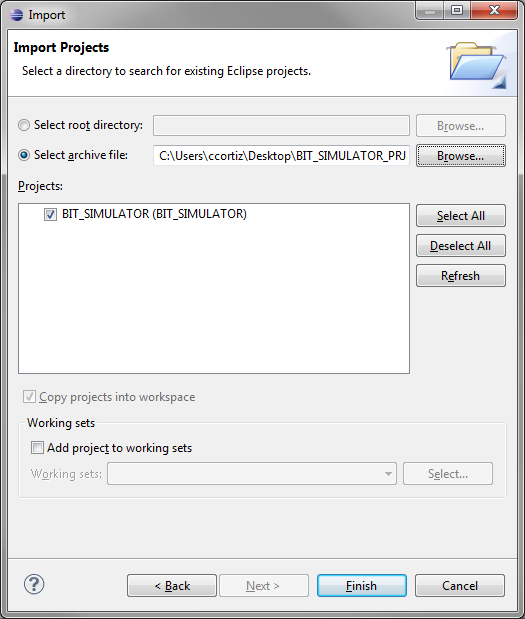
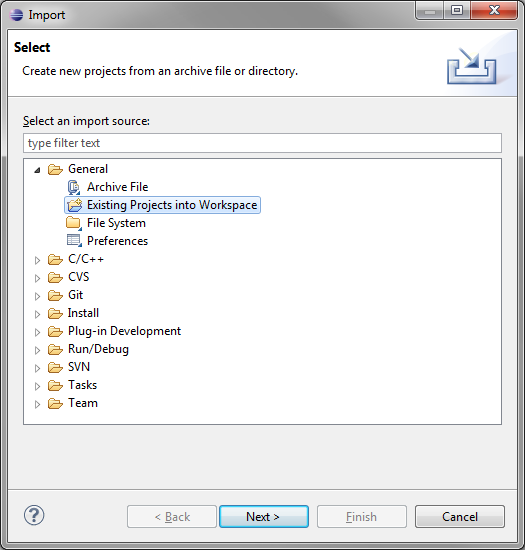
MinGW

Aplicación simuladora

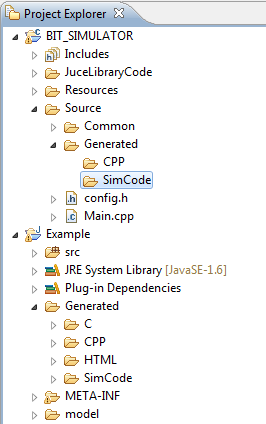
#### Compilación del simulador.

En el paquete de distribución de BIT se adjunta un proyecto de Eclipse llamado "*BIT\_SIMULATOR\_PRJ.zip"* en el que se encuentra el código común del simulador y la configuración necesaria para integrar JUCE.



Extraiga el fichero a una carpeta temporal de su elección y, desde Eclipse, importe el proyecto en su WorkSpace.

Una vez importado, debería ver algo similar a esto:

Observe como en la carpeta Source del proyecto importado figuran dos carpetas *"Common"* y *"Generated"*.

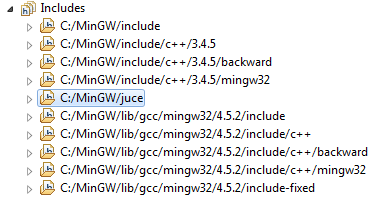
En *"Common"* se encuentra el código que es común a todos los simuladores que pueda generar con BIT.

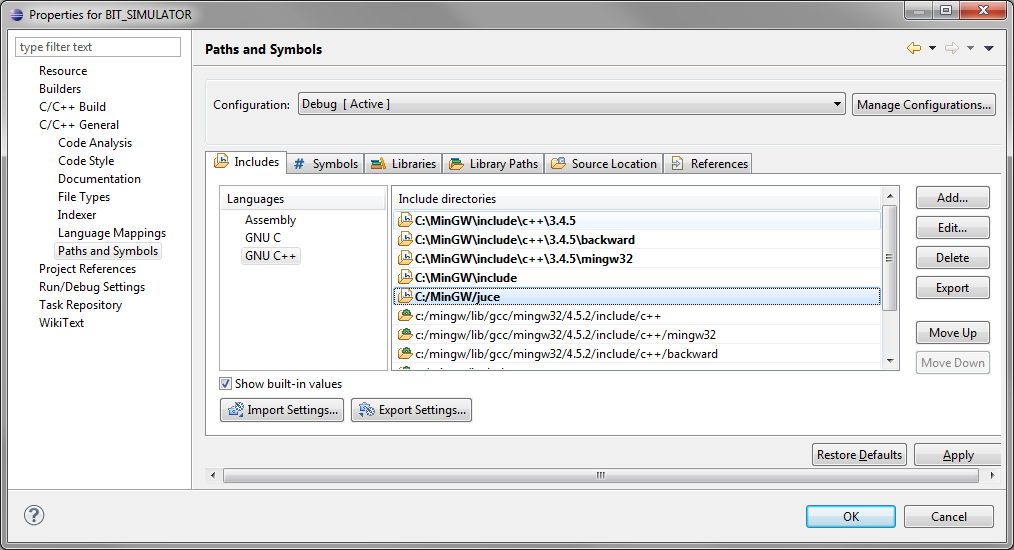
En *"Generated"* deberá incluir el código de la API C++ y del código *"SimCode"* generados a partir de su mensajería.

Copie el contenido de las carpetas *"CPP"* y *"SimCode"* desde el proyecto *"Example"* a la carpeta *"Generated"* del proyecto *"BIT\_SIMULATOR"*.

Verifique que la librería JUCE está debidamente instalada en la capeta *"C:\MinGW\juce"*.

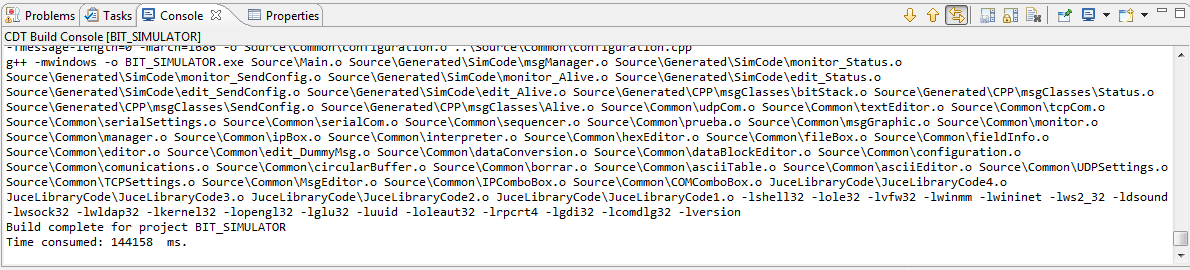
Si decide instalar JUCE en otra ruta diferente a la utilizada por defecto, deberá modificar los parámetros de configuración del proyecto para que encuentre la librería en la ruta especificada.



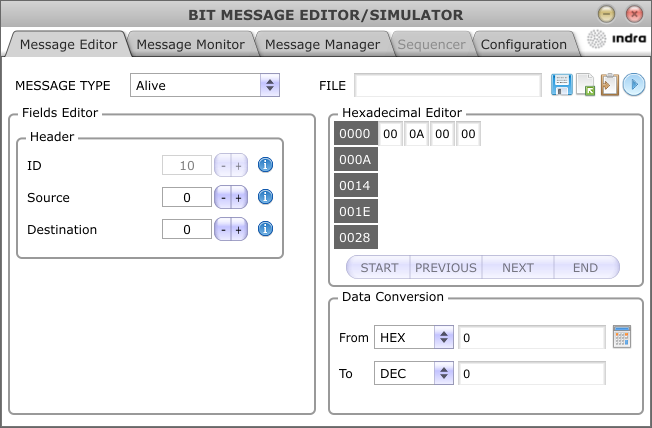


Puede configurar algunos parámetros en el fichero *"config.h"*, como el nombre que figurará en la ventana del simulador y el tamaño asignado a los diversos búferes del programa.

A continuación compile el proyecto.



Al cabo de unos minutos tendrá disponible su simulador:



#### Uso del simulador.

#### 

El simulador se distribuye en 4 pestañas, en cada una de las cuales se realizan tareas distintas:

* **Message Editor:** Permite editar el contenido de cada mensaje.
* **Message Monitor:** Permite enviar, recibir y monitorizar mensajes.
* **Message Manager:** Permite crear paquetes de mensajes.
* **Configuration:** Permite configurar el canal de comunicación empleado.

##### Message Editor.

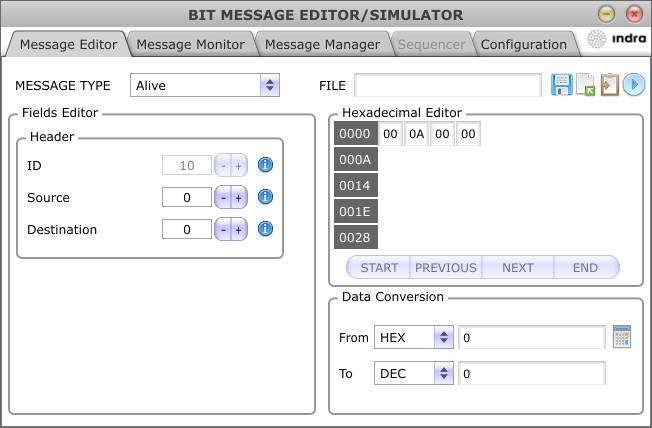
Permite configurar los campos de cada mensaje de forma individual. Una vez que tenga configurado un mensaje con los valores que desee, tiene la opción de guardarlo en disco o bien transmitirlo por el canal de comunicación configurado en la pestaña *"Configuration"*.

En la parte superior izquierda puede observa un comboBox *"Message type"*, que al desplegarse presenta una lista con todos los mensajes que ha diseñado. Además aparece un mensaje *"DUMMY\_MESSAGE"* que le permitirá editar un buffer de bytes de tamaño y contenido arbitrario.

Al seleccionar cada uno de los mensajes del comboBox, se observa como el formulario contenido dentro del grupo *"Fields Editor"* se actualiza para presentar cada uno de los dataItems y campos que conforman el mensaje. También se actualiza el *"Hexadecimal Editor"* para presentar el contenido, ya codificado, del mensaje que seleccione.

Tanto desde el *"Field Editor"* como desde el *"Hexadecimal Editor"* se puede modificar el contenido del mensaje. El programa utilizará la información disponible de cada campo para verificar que los valores introducidos estén dentro de rango.

Una vez configurado sus mensajes con los valores de prueba que desee, puede optar por guardar el mensaje en un fichero de su elección, copiar el buffer de datos del mensaje en el portapapeles (en formato C), o bien enviarlo por uno de los canales disponibles (TCP, UDP, RS232).



Send

Copy

Save

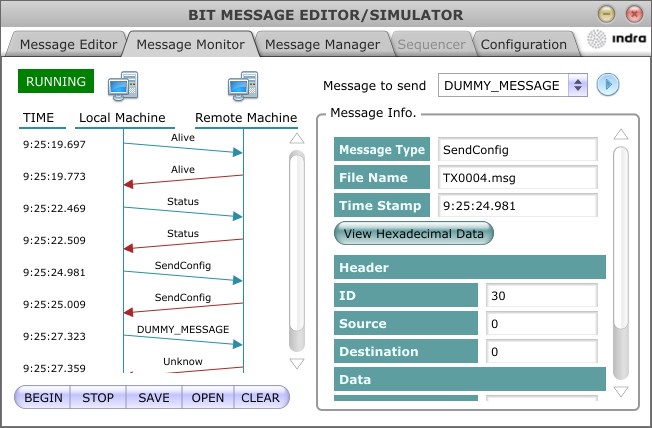
Load

Además dispone de un sencillo conversor de datos de hexadecimal a decimal y viceversa.

##### Message Monitor

La siguiente pestaña del simulador es el ***"Message Monitor"*** que permite registrar el trafico entrante y saliente del canal de comunicaciones que tenga configurado.

Además le da la opción de enviar mensajes desde el comboBox *"Message to send"* situado en la parte superior derecha. Los datos contenidos en los mensajes que envíe desde ahí, serán los que previamente haya configurado desde el *"Message Editor"*.



Save log

Clean log

Open log

Disable login

Enable login

Tanto en transmisión como en recepción, el monitor de mensajes trata de identificar el tipo de mensaje registrado. En caso de que la identificación sea satisfactoria, proporciona información de ese mensaje desglosada en campos.

También le da la oportunidad de ver el contenido del mensaje en formato hexadecimal/ASCII y de guardar el log completo en disco.

Para que el monitor este operativo deberá activarlo con el botón "BEGIN". Para detener el registro de mensajes pulse "STOP". En la parte superior izquierda de la ventana aparece un cuadro de texto de color indicándole el estado actual del monitor.

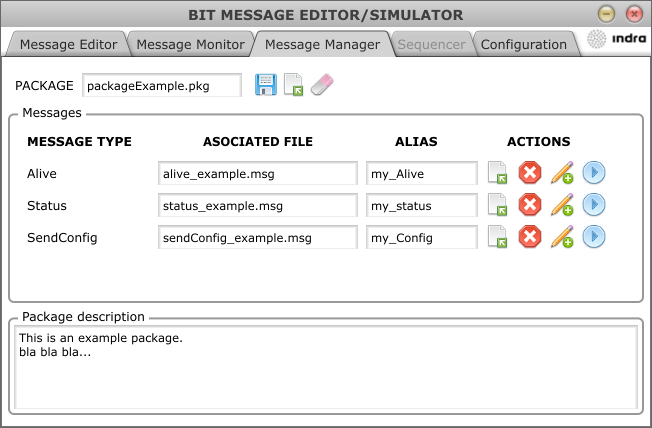
##### Message Manager.

La tercera pestaña es el ***"Message Manager"***, que permite definir un conjunto de mensajes predefinidos y guardarlo como un paquete/escenario de simulación.

Clean

Load

Save



Delete

Send

Edit description

Open File

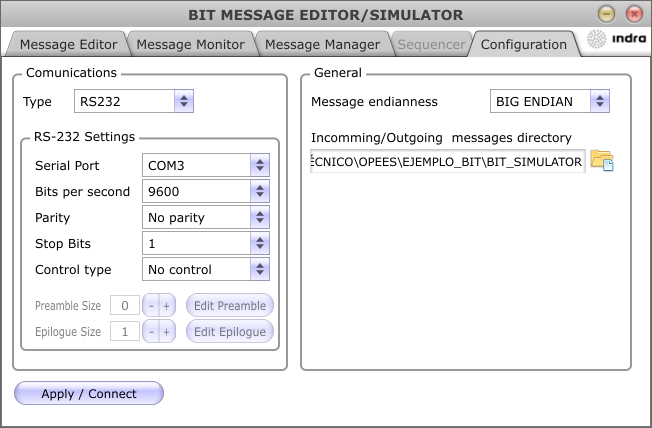
Los mensajes deben estar previamente guardados en ficheros. A cada uno de ellos se les puede asignar un alias, que será empleado en el *"Message Monitor"* para identificar los mensajes salientes.

También se les puede asignar una descripción que indique qué datos lleva el mensaje o para qué sirve.

El paquete completo también puede acompañarse de un texto que describa el contenido y objetivo del paquete.

##### Configuration.

La última pestaña del simulador es ***"Configuration"***, donde puedes configurar los parámetros de comunicación, el endianness utilizado para codificar la mensajería y el directorio donde guardar los mensajes de log recibidos en la pestaña *"Message Monitor"*.



Actualmente, el simulador soporta comunicaciones UDP, TCP modo cliente, TCP modo host y RS232.

Puede configurar los parámetros más comunes de cada una de los canales de comunicación disponibles.