Contents

[Introduction 2](#_Toc338671094)

[What is BIT? 2](#_Toc338671095)

[Where to apply BIT? 2](#_Toc338671096)

[Why do I need BIT? 2](#_Toc338671097)

[Installation 4](#_Toc338671098)

[Message editor 6](#_Toc338671099)

[Using the message editor 6](#_Toc338671100)

[Other considerations 13](#_Toc338671101)

[Code Generation 14](#_Toc338671102)

[HTML Document generator 15](#_Toc338671103)

[C API generator 16](#_Toc338671104)

[C++ API generator 21](#_Toc338671105)

[Message simulator 25](#_Toc338671106)

## Introduction

### What is BIT?

Binary Interface Tools (BIT) is a set of tools focused on the binary messaging development and test.

BIT uses the binary messages as a set of fixed size fields of bits that are piled in a specific way and fulfill some requirements.

BIT consists of a visual editor that allows defining the structure of each message and a set of code generators that produce C/C++ libraries to manage messages.

BIT also includes the ability to generate HTML documentation from the messaging, and to produce the source code of a simulator that, once compiled, provides a useful tool for developers.

### Where to apply BIT?

BIT is useful in those cases where it is necessary to define a binary messaging between two or more devices.

The messaging must fulfill the following requirements:

* A messaging consists of separate messages. The data within a message cannot have dependencies with the data on a different message.
* A message consists of "dataltems" or structures that include one or more fields. A dataltem can be single or repetitive (array structure). When it is repetitive, its multiplicity can be fixed or dependent on another field of the message.
* A dataltem consists of fields. The number of fields defined for each dataltem is fixed and it is always in the same order. There are two types of fields: "dataFields" or bit fields and "datablocks" or byte buffers. The datablocks length can be fixed or dependent on other fields of the same message.
* Within a message it is not possible to use the same name for different dataltems or dataFields.

These requirements cover a very important part of the protocols commonly used in embedded developments. However, messaging based on ASCII texts and messages with optional fields (e.g. ASTERIX) are not covered.

### Why do I need BIT?

The definition, implementation, test and documentation of the messaging between two devices are usually a recurring task in any embedded software project development.

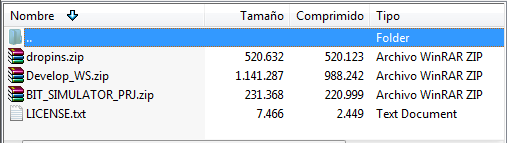
BIT pretends to provide an orderly procedure for these tasks, speeding up the development and automating the generation of the source code relative to messages management.

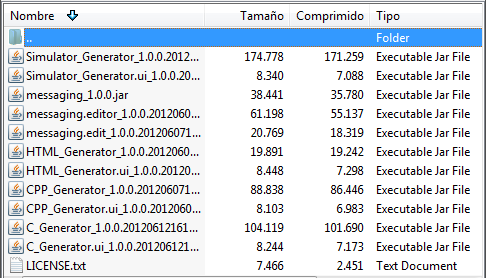
Automatic code generation saves time and minimizes coding bugs.

As a result, when the parties involved in the communication of messages use libraries generated from the same messaging, edited with BIT, many of the problems that may appear during the integration of the system can be avoided.

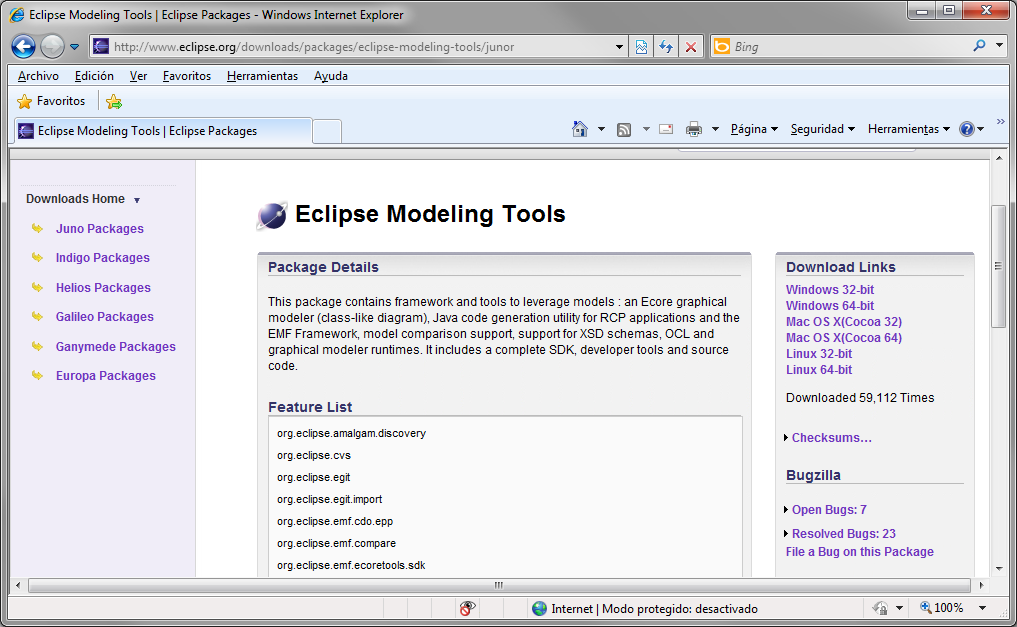
## Installation

BIT is delivered as several .jar files compressed in zip format. These must be manually installed in the "dropins" folder located in the directory where Eclipse is installed.





Eclipse installation should include Eclipse-EMF and Acceleo modules. These modules are available in the "*Eclipse Modeling Tools*" package that can be downloaded from the official web page <http://www.eclipse.org/downloads/>



It is also recommended that the Eclipse installation has the Eclipse-CDT module installed. Thus, visualize and compile the C/C++ source code will be enabled.



Eclipse CDT

Acceleo & Modeling

Besides the Eclipse plug-ins, a C++ project is attached with the source code necessary to compile a simulator that "recognizes" its messaging. The details about the compilation and use of the simulator are explained in following sections.

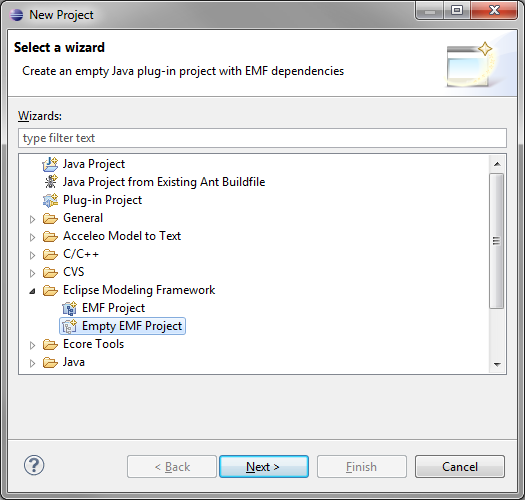
## Message editor

The first step is working with the message editor. The editor allows defining the *ICD[[1]](#footnote-1)* or the messaging where the messages, dataltems and fields will be added. In order to define the structure and behavior of each element, the user should fill the attributes that affect each one of the mentioned element.

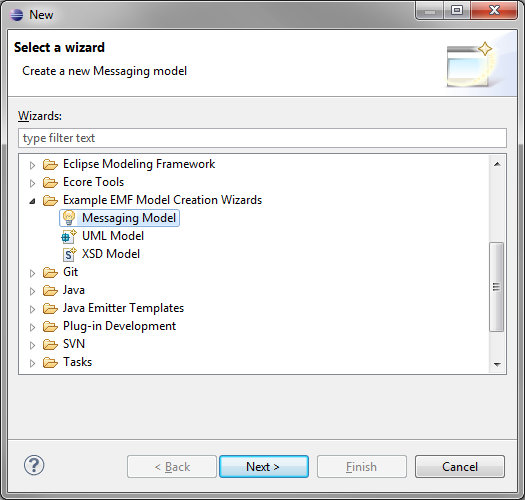
### Using the message editor

A messaging example is defined here to clarify the use of the message editor.

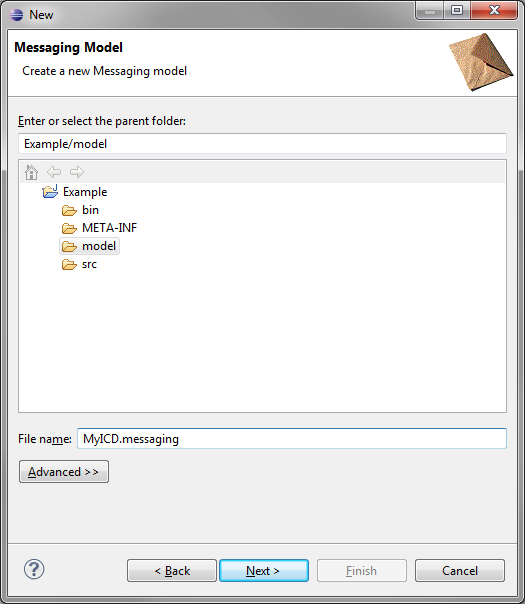
First of all we create a new project "*Empty EMF Project*".



Then we add a model type "*Messaging Model*", inside the "*model*" folder of the created project.



Assigning a name. In our example the chosen name is "*MyICD.messaging*". Then, select "*Model* *Object*" that always must correspond to "*ICD*" and press "*Finish*".

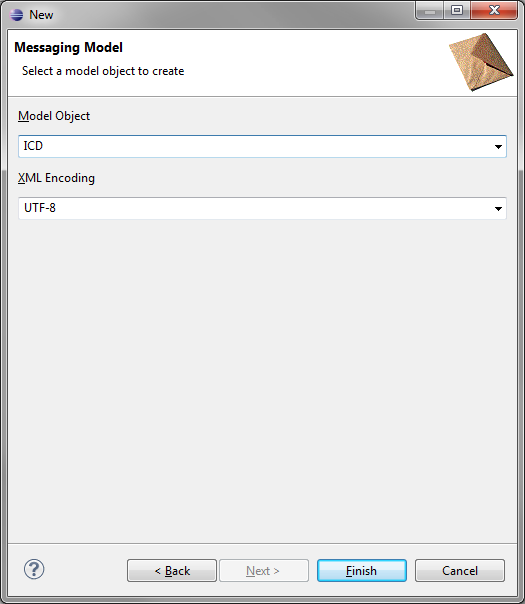
***Do not forget that the basic element of a messaging defined with BIT is the ICD.***

***ICD contains messages, which consist of dataltems.***

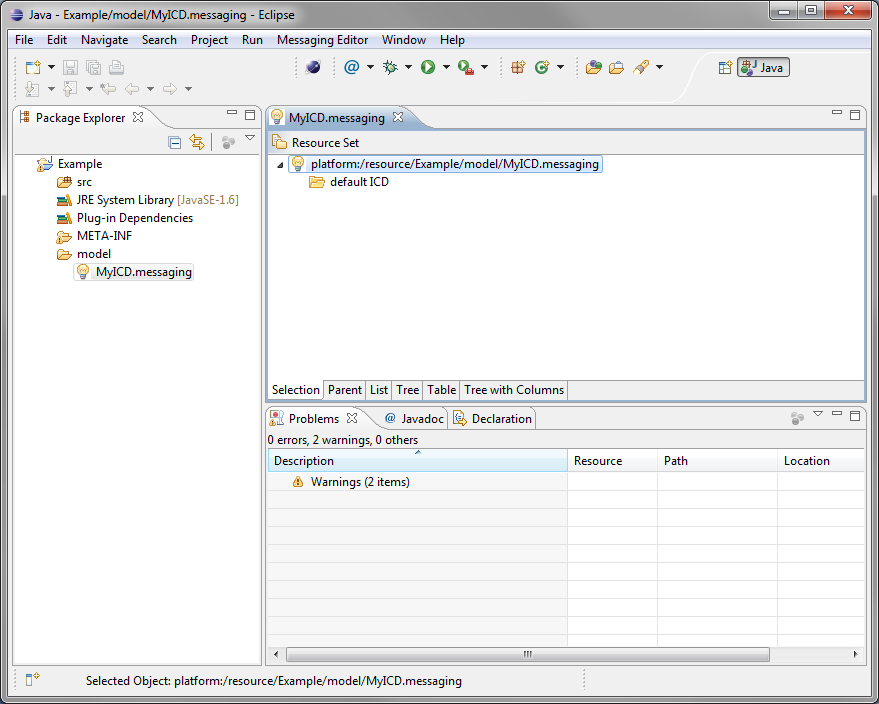
***Data Items act as data structure, that can be either bitFields or dataBlocks.***

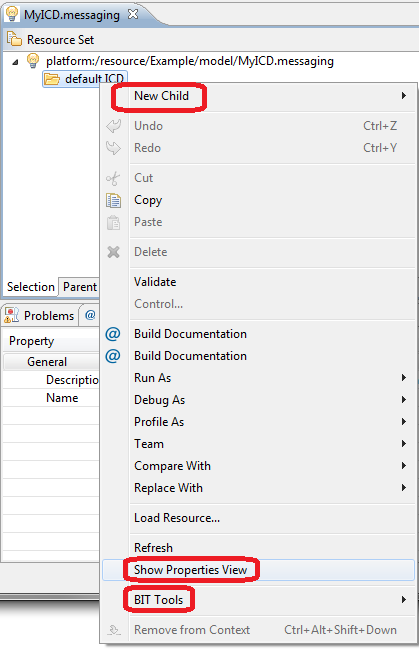
*BitFields are fields of bits with a definable size between 1 and 32 bits.*

*DataBlocks are blocks of bytes or buffers whose maximum size must be specified.*



At this point we should have something similar to what is shown on the next figure:



In the tab "*MyICD.messaging*" a message editor is shown with a bulb icon on the left. By clicking on the triangle at the left side of the icon, a tree with a node called "*default ICD*" is displayed. Its icon is a yellow folder.

Right-click over the folder and a menu is displayed with the following information:

* The "*New Child*" menu is useful to add items to our model. Depending on which item is selected, it is possible to add new messages, dataItems and fields.
* The "*BIT Tools*" menu contains the code generators that will process the messaging that we define.
* The "*Show Properties View*" option displays a view where the attributes of each one of the inserted items in the messaging model can be edited.

Messages can be added to our ICD from the "*New Child*" menu. The attributes are accessible from the view "*Properties*" of the option "*Show Properties View"*.

In our example, imagine a device controller that communicates wirelessly with small measurement devices distributed throughout a building. The controller collects the temperature, humidity, etc. of the room where the small devices are located.

Now we create a first message called "*Alive*" that will allow the device controller to identify the available measurement devices. This first message has a dataItem called "*Header*".

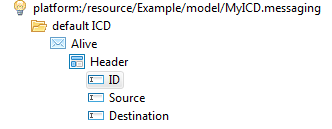
In the dataItem "*Header*" we insert three fields:

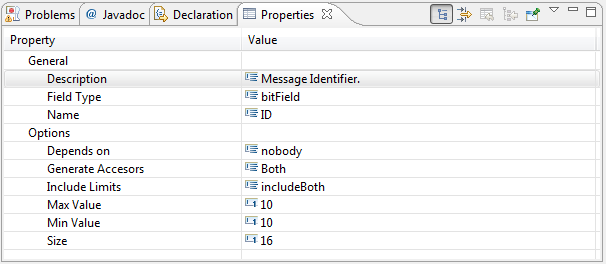
* ***ID*:** 16 bit field that uniquely identifies the message type. Set ID to fixed value 10DEC.
* ***Source*:** 8 bit field that specifies who delivers the message. It can take values between 0 and 20DEC, where 0 is the device controller.
* ***Destination*:** 8 bits field that specifies to whom the message is addressed. It can take values between 0 and 20DEC, where 0 is the device controller.



Header

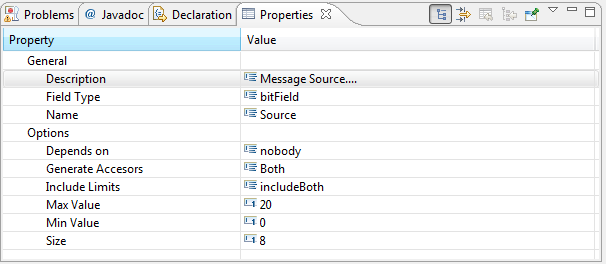
In the message editor our *Alive* message is shown as follows:

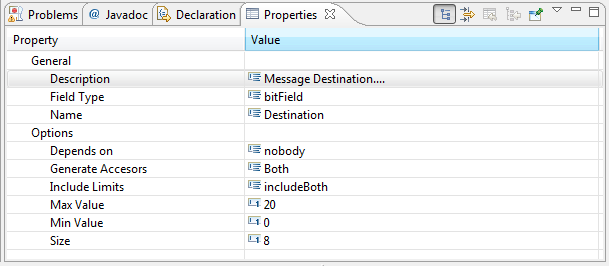


The *"ID"* attribute is expected to be fixed to 10DEC value. This means its attributes "*Max Value*" and "*Min Value*" must have the same value.

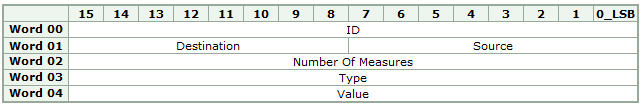
*When filling in the item's attributes, it is recommended to fill in the attribute "Description" because it will be part of the generated documentation.*

The attributes of the fields "*Source*" and "*Destination*" will be shown as:





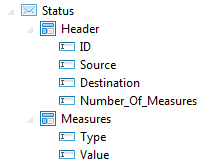
Now create a new message called "*Status*", it will be used by the sensing devices to report the measures. Each device has a variable number of probes (maximum 10). For each connected probe, the measurement type and its value will be notified.

The message should be like this:

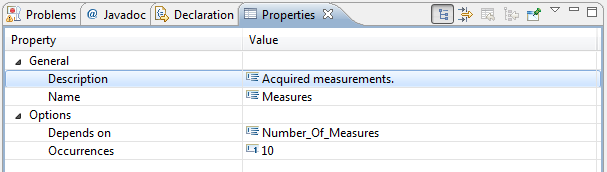
Header

Measures

The dataItem "*Measures*" contains two 16-bit fields "*Type*" and "*Value*", and should be repetitive. To achieve this, assign the value 10 to the attribute "*Occurrences*". This way an array of structures is created with a maximum of 10 items.

The dataItem "*Measures*" should be repeated as many times as indicated in the "*Number\_Of\_Measures*" field. Therefore its attribute "*Depends\_on*" must be set to "*Number\_Of\_Measures*".

The attributes of the dataItem "*Measures*" should be shown as follows:



Make sure the field "*Number\_Of\_Measures*" takes values between 0 and 10DEC, otherwise the number of measures could exceed the maximum specified value for the dataItem "*Measures*" and the message would not be consistent.

Messaging design inconsistencies could result in "*out of bounds*" errors, in the C/C++ generated library. For this reason be careful when designing the ICD.

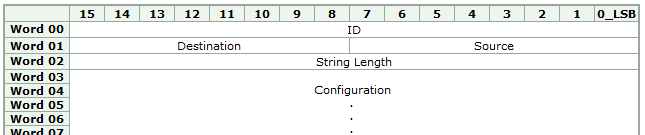
Set the identifier of this message to *ID*= 20DEC. Try to include, in each message of the design, at least a field with constant value in order to ease the generated libraries work during the decoding and identification of messages.

The "*Type*" field specifies the connected probe type, it takes values between 0 and 99DEC.

The "*Value*" field takes values between 0 and 65535DEC. Depending on the connected probe, the measure will have a different meaning.

Finally, create another message "*SendConfig*". It will be used by the sensing devices to report to the controller which configuration is using. The configuration is provided in XML (flat text files) format, so that it is necessary to include a "*dataBlock*" field called "*Configuration*", of variable size and dependent on the "*String\_Length*" field. The "*Configuration*" field will have a size of 1024 bytes.

The message could be as follows:

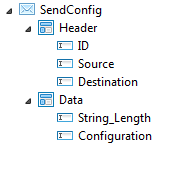


1024 bytes for XML file

Data

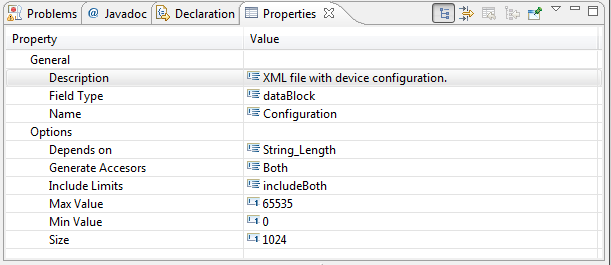
Header

And its BIT implementation should have the following appearance:

For this message the "*ID*" field will have the constant value of 30DEC.

The "*String\_Length*" field will have values between 0 and 1024DEC. In order to avoid inconsistencies, the range should be consistent with the maximum size of the field "*Configuration*".

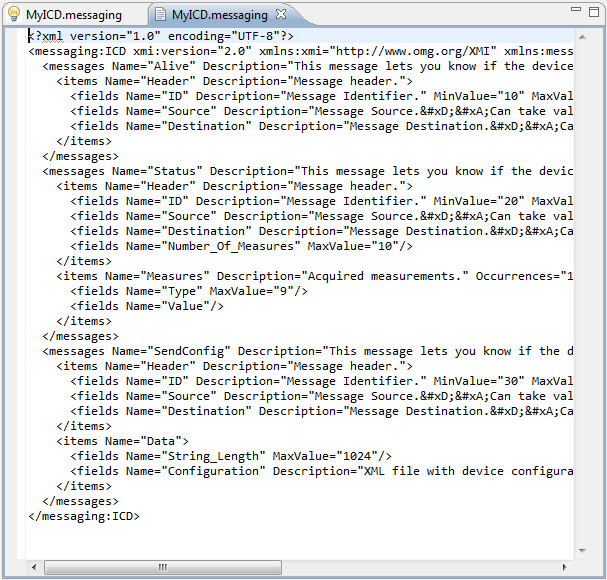
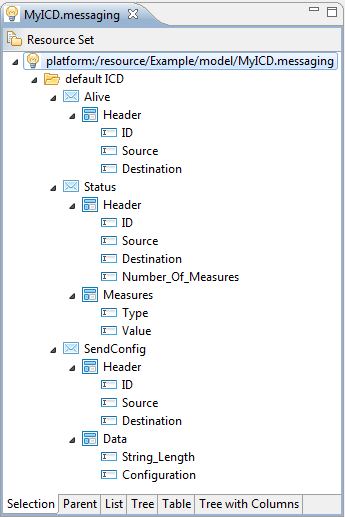
The attributes of the field "*Configuration*" must be the following:

*Observe how the size of the dataBlock, in bytes, is specified by the attribute "Size"*

The size dependency of the dataBlock field depends on the attribute "*Depends on*", in this case must be set to "*String\_Length".*

Once the messaging is designed, a file "MyICD.messaging" is available and will contain the definition of all our messages in XML format.

The message editor has the following appearance:



### Other considerations

The attributes "*Generate Accesors*" and "*Include Limits*" are available when configuring the dataField's attributes. In general, the default values are valid for a satisfactory code generation.

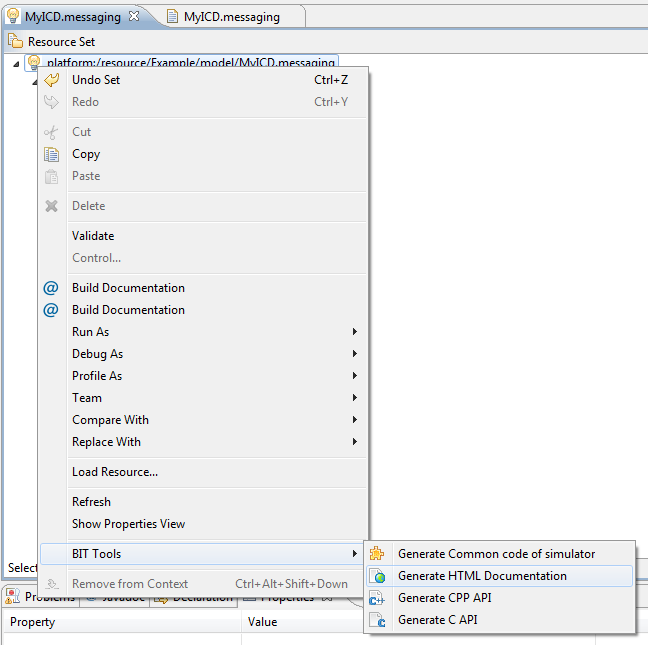
The attribute "*Include Limits*" indicates whether the values "*Min Value*" and "*Max Value*" should be included in the field range. It can be set to *"includeBoth"*, *"minLimitOnly"*, *"maxLimitOnly"*, or *"noIncludeLimits"*. The selected range will be considered during the APIs source code generation, and it will only affect to the "*bitField*" field type.

The attribute "Generate Accesors" indicates whether to generate code to include only Getters or Setters, or both of them. Bit fields with constant value only have getters, regardless of what it is set on this attribute. "Both" is a recommended value for this attribute.

## Code Generation

BIT gives to the user the possibility to generate HTML documentation, as well as a C/C++ API.

To generate code, right-click on the bulb icon. After that, the submenu "BIT Tools" is displayed.



The submenu "BIT Tools" provides tools to generate:

* HTML documentation.
* A C language API for messaging management.
* A C++ language API for messaging management.

The APIs that manage the messaging have the necessary tools to manipulate the messaging:

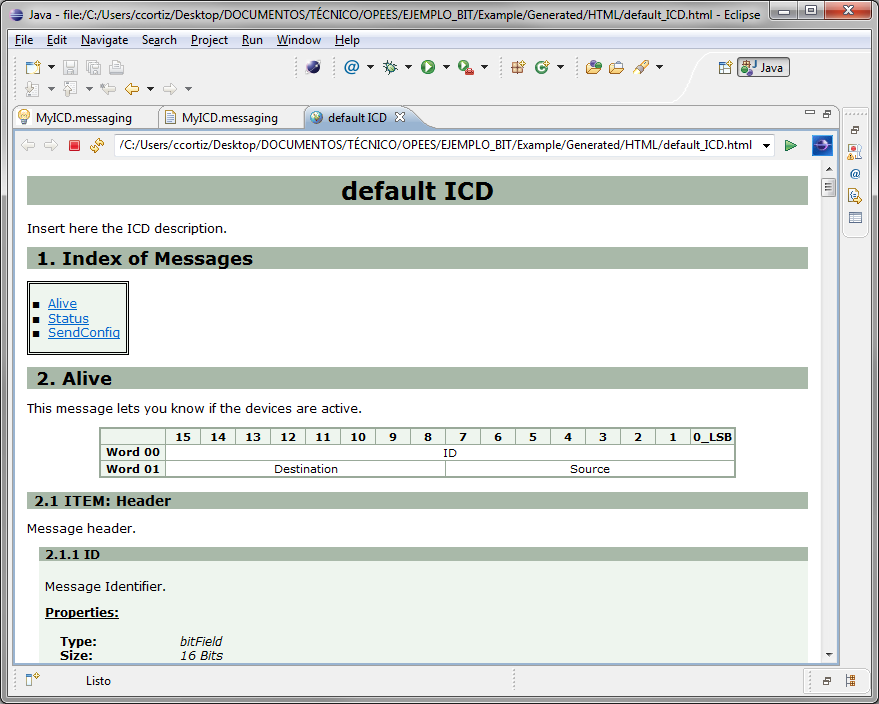
* Accesors (getters and setters) for the message fields.
* Serializers (message encoders).
* Deserializers (message decoders).
* Search tools and automatic identification of messages.

In addition, it is also included a generator that provides the source code for a simulator that is adapted to the designed messaging.

### HTML Document generator

By selecting "***Generate HTML Documentation***" in the "*BIT Tools*" menu, it is activated the code generator that generates HTML documentation.

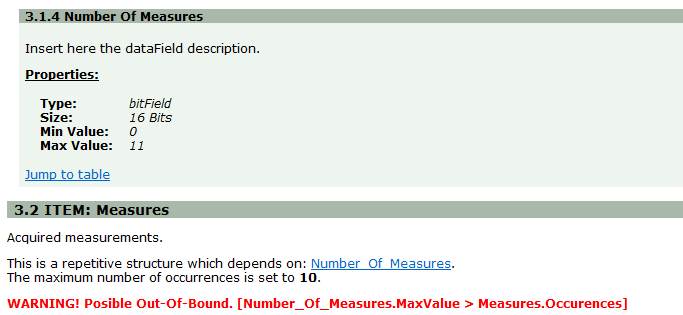
In our example, the generated HTML documentation consists of just one HTML file that is located in the folder "Generated/HTML" of our project and presents the following aspect:



The generated document has a messages index and the definition of all the previously designed messages. Each documented message includes a table that represents the structure of the message and the detailed description of the dataltems and defined fields in the messaging.

There are links to access the different parts of the document for easier navigation.

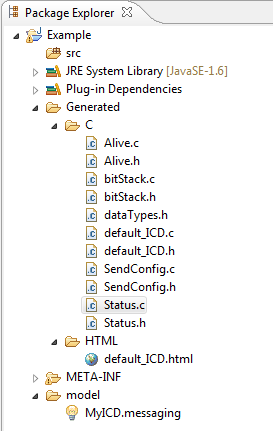
The HTML generator is able to detect inconsistencies in the definition of the messages, and warns about them by means of red colour warnings.



### C API generator

Selecting the "Generate C API" option from the "*BIT Tools*" menu activates the code generator. This provides the API in C language and allows the handling of the messaging .

The generated API consist of a group of files with .h and .c extension, and is located in the "Generated/C" folder of our project.

For each defined message a pair of files .h and .c with the name of the message is generated. This provides the necessary structures and functions to handle the data of that message.

The files "*bitStack.h*" and "*bitStack.c*" are included. These implement the function of bits piling needed to make the library work. The file "*dataTypes.h*" defines the types used for the libraries.

The file "*dataTypes.h*" defines the macro \_*DEBUG\_* that must be available. This provides debugging messages via the standard output of the system.

Along with the mentioned files, the files "default\_ICD.h" and "default\_ICD.c" are generated. They are the entry point to the library that is generated. The name of these files depends directly on the name of the ICD in the message editor.

The developer that wishes to call the generated API from his own code should only include the "*default\_ICD.h*" file.

From now on, the developer can access all the items of the messages even without the low-level knowledge of the working mechanisms of the library.

#### Messages Codifications

This chapter presents an example of use of the library generated in C. A clear explanation about how to edit and code the messages is shown.

The message we are about to code is the "*Alive*" message defined in our example of messaging.

This message has 3 different fields. The "*ID*" field has a constant value and therefore it does not have setter. The other two fields, "*Source*" and "*Destination*", will have the values 0 and 1 assigned respectively.

The source code used is:



The first thing to notice is that, in order to use the generated library, we only have to include “default\_ICD.h” in our program.

Then, we have to get an instance of the library by means of the function ***new\_Default\_ICD()*,** which returns a descriptor ***myICD*** that will be needed laterto use the API. Take into account that the function is called "*new\_Default\_ICD()"* because "*Default ICD*" is the name assigned to the ICD. In case of having a different name, the name of the function would be "***new\_***" and the assigned name.

From this point on, the *getters* and *setters* of the defined messages can be used. The first parameter delivered to all the functions is the descriptor obtained in the function *new\_Default\_ICD().* **By using descriptors several instances of the library can be simultaneously handled.**

The *setters* begin always with the prefix "*set\_*", followed by the name of the field to be modified. Also the *getters* begin with the prefix "*get\_*" and the name of the field to obtain the value.

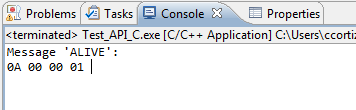
The *getters* and the *setters* are inside the messages to which they belong. The messages hang directly from the generic instance "***Default\_ICD***"that is predefined in the generated library. "*Default\_ICD*" matches up with the name designed to the ICD in the message editor.

After applying values to each of the fields of the *Alive* message, we generate the buffer of the message with the function ***serialize().***

The size of the generated buffer can be obtained by the function ***get\_MsgSize(),***

The generated buffer ensures that each field is located and correctly aligned as specified in the design made in the message editor.

When running the program, the following output is obtained:



The values 0x0A and 0x00 match up with the ID message, whose value is fixed and equal to 10 (0xA in hexadecimal)

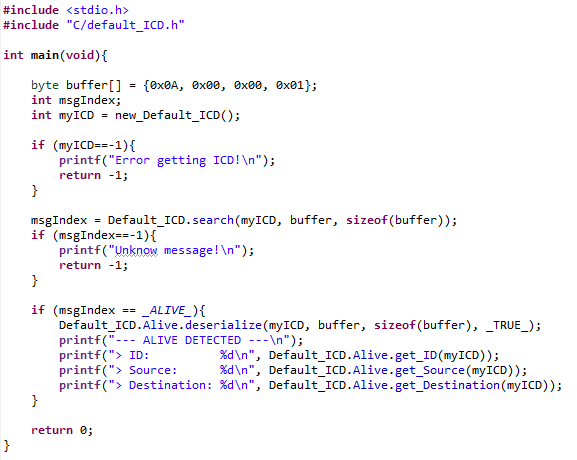
The values 0x00 and 0x01 match up with the fields "Source" and "Destination" of 8 bits that according with the code in the example, they are 0 and 1.

#### Decoding messages

#### Below is an example of use of the generated C library, which shows how to decode messages.

Now we are going to decode a test buffer, whose contents represent a message like "Alive" with fields "Source" and "Destination" assigned to the values ​​0 and 1 respectively. The buffer contents match the result obtained in the previous example of message encoding.

The source code used is:



Initially it will be obtained a descriptor for the API and then the function **search()** will be used to try to automatically detect the type of message contained in the buffer.

The function *search()* receives these parameters: the API descriptor, the buffer to be analyzed and its length. The function *search()* will return -1 in case of not identifying the message, otherwise the returned value will be an integer that represents the type of the detected message.

Once the type of message is identified, its decoding is carried out using the function ***deserialize().***

The function *deserialize()* receives these parameters: the API descriptor, the buffer to be analyzed, its length, and a fourth parameter whose values can be \_TRUE\_ or \_FALSE\_. When this parameter is set to \_TRUE\_ it decodes the message and assigns the decoded values to the fields of its message. When the parameter is \_FALSE\_, the function will only test if the message is decodable and in this case the value returned by the function is *\_DESERIALIZE\_OK\_.*

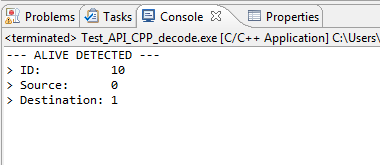
If the function *deserialize()* receivesan impossible to decode buffer as parameter, the returned value is different than *\_DESERIALIZE\_OK\_.*

When decoding a message, it is considered invalid if:

* It does not have the expected size.
* The fields defined as constant do not have the correct values.
* The fields that are being obtained have values out of range.

Finally the content of the fields is accessed using its **getters**.

The result of executing the example is:



#### Other considerations

The way to access the value for the fields of type **dataBlock** (arrays of bytes) is:

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

Observe that it is not necessary to specify the size of the buffers because the library knows how many bytes to copy depending on the fields. In case of using getters, it is the developer responsibility to reserve enough memory to store the buffer returned by the function.

When the fields to encode/decode belong to a repetitive dataItem, the format of the getter/setter varies slightly.

For a field inside a non-repetitive dataItem, the way to access its value is:

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

In case of using repetitive dataItems the function are as follows:

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

If the field is a dataBlock, the accessor functions are:

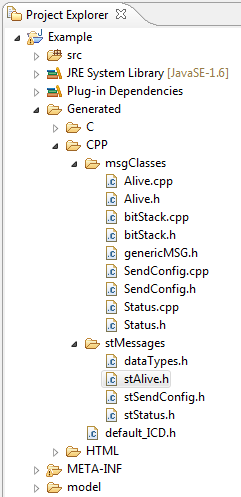
* ***get\_dataBlock***(*descriptor*, *index*, *destBuffer*)
* ***set\_dataBlock***(*descriptor*, *index*, *srcBuffer*)

For the buffers and repetitive dataItems, the first element is the one whose index is equal to 0.

### C++ API generator

By selecting the *"Generate CPP API"* option in the "*BIT Tools*" menu, the code generator that provides the C++ language API is activated. This allows handling its messaging.

The generated API consists of a set of files .h and .cpp, organized in the folders *msgClasses* and *stMessages* that are inside the folder "*Generated/CPP*" of our project.

The concept used in this library is very similar to the one used for the C library, with the difference that in this case the library is object-oriented, ensuring the encapsulation and allowing multiple API instances.

The way to use C++ API is very similar to the one used in C API. There are methods *setter/getter, serialize(), getMsgSize(),* etc.

To use it in a project, the user should include the file "default\_ICD.h" and after obtaining the API instance, the encoding and decoding features can be used.

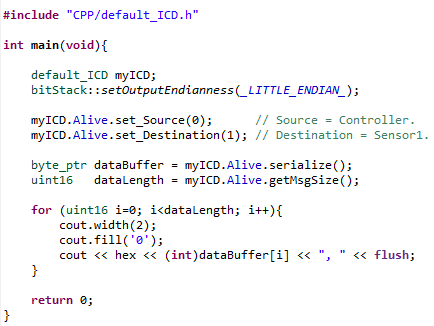
Take into account that like in the C API, the file name "default\_ICD.h" is taken from the name assigned to the ICD in the message editor.

#### Message encoding

Here it is presented an example of use of the C++ generated library where it is shown how to encode messages.

We are going to encode an "Alive" message that has the fields "ID" with constant value equal to 10, "Source" set to 0 and "Destination" set to 1.

The code used for encoding is as follows:



The first to do is to obtain an instance of the class "*default\_ICD*". The name of the class matches up with the name assigned to the ICD.

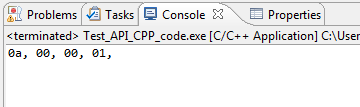
Be careful selecting the appropriate endianness for the encoding/decoding of the messages. To select the endianness, use the static method *setOutputEndianness()* of the class *bitStack.*

Observe how after creating the instance "*myICD*" of the class "*default\_ICD*" it is not necessary the handling of descriptors as it would be the case in C library.

Then we can assign the desired values to the message fields using the methods *set\_Source()* and *set\_Destination()*.

To finalize, use the methods *serialize()* and *getMsgSize()* of the message *Alive* to encode the message in a buffer of bytes and to obtain its size.

The result of running the program is:

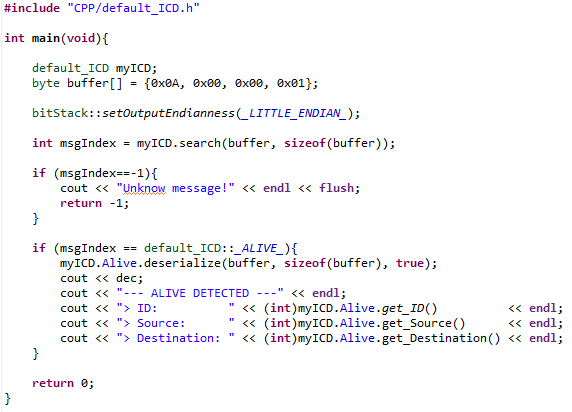


#### Messages decoding

Next it is presented an example of use of the C++ generated library, in which it is shown how to decode messages.

Using the buffer obtained in the previous example as basis of this example, it will be shown how to use the features of automatic identification and decoding of messages.

The code used to make the decoding is:



The first step is getting the *instance myICD* of the class *default\_ICD.* After that, we specify the desired endianness to apply in the encoding/decoding of our messages.

Next, we use the *search()* method giving as parameters the buffer to be analyzed and its size. In case the message is recognized by the library, its identifier will be returned and stored in the integer variable *msgIndex.* Otherwise, the method will return the value -1.

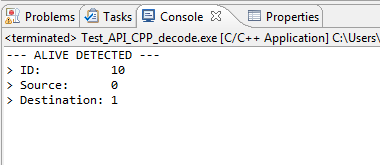
Once the message is identified, we decode it and store the values of the buffer inside the message. In this case, the identified message is type *\_ALIVE\_*.

To decode the message, use the method *deserialize()*, giving as parameters the buffer, its size and a Boolean that indicates whether to decode the message (true) or to test if the buffer used is decodable as message type *Alive* (false).

If *deserialize()* decodes the buffer successfully, *\_DESERIALIZE\_OK\_* is returned*.*

After the decoding, the content of the decoded message is accessible using the appropriate getters.

The result after running the program is the expected one:



#### Other considerations

When handling fields of type **dataBlock** (arrays of bytes), the way to access to the value is:

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

When the fields to encode/decode belong to a repetitive dataItem, the format of the getter/setter varies slightly.

For a field inside a non-repetitive dataItem the way to access to its value is:

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

In case of using repetitive dataItems the functions are:

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

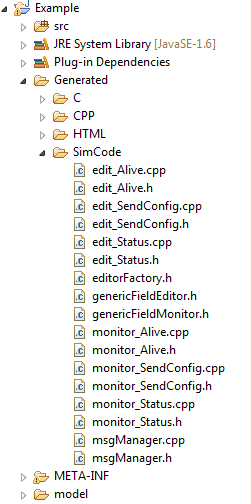
If the field is a dataBlock, the access functions are:

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

For the buffers and repetitive dataItems, the first element is the one whose index is equal to 0.

### Message simulator

Along with the HTML/C/C++ generators it is supplied a code generator that provides part of the source code of a specific simulator for the designed messaging.

The provided simulator allows to edit the content of every defined message and send it through a configurable communication channel (TCP, UDP, RS232).

It also allows to receive frames through the configurable communication channel and identify them as messages known by our messaging.

Likewise, it allows monitoring the sending and reception of messages and keeping the logs of the session.

The simulator is not only a tool for the development and test of the application but it is also a tool to confirm the validity of the C++ API it is based on.

The generated code for the simulator is located inside the folder *"Generated/SimCode".*

The generated source code must be added to the common code of the simulator, which is provided as part of the BIT distribution, and to the C++ API generated from the same messaging.

This way, the source code of the simulator consists of:

* The common source code of the simulator.
* The source code generated for the simulator.
* The generated C++ API with BIT.

BIT Generators

C++ API ++

Generated Code of the Simulator

Source code of the simulator

Common Code of the Simulator

Furthermore, in order to compile the project the 1.52 version of JUCE library must be installed. The JUCE library is GPL and free for non-commercial use, but there are commercial licenses in the library web site:

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

MinGW is recommended as C/C++ compiler. It has GPL license and not only has the advantage to be free for non-commercial use but it is perfectly integrated in Eclipse CDT.

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

Source Code of the Simulator

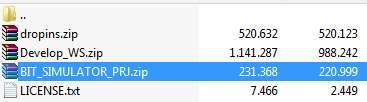
JUCE 1.52

MinGW

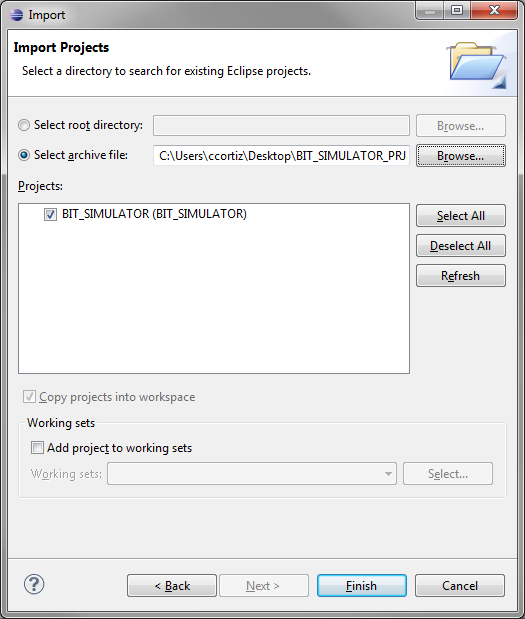
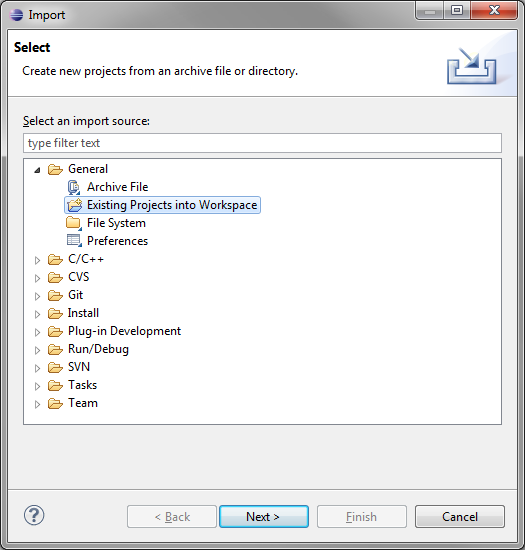
Application simuladora

#### Compilation of the simulator

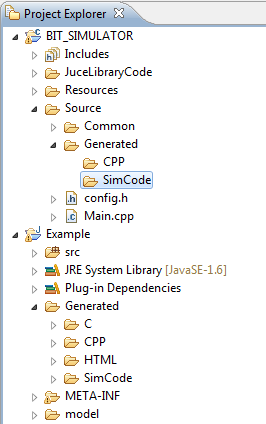
Within the BIT distribution package it is supplied an Eclipse project called "*BIT\_SIMULATOR\_PRJ.zip*” that contains the common source code of the simulator and the necessary project configuration options to integrate JUCE.



Extract the file into a temporary folder and, from Eclipse, import the project in your WorkSpace.



Once imported, something similar to this figure should be observed:

Observe two folders *"Common"* and *"Generated"* in the Source folder of the imported project.

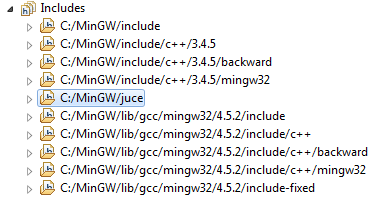
In the "*Common*" folder there is the common code to all the simulators that can be generated with BIT.

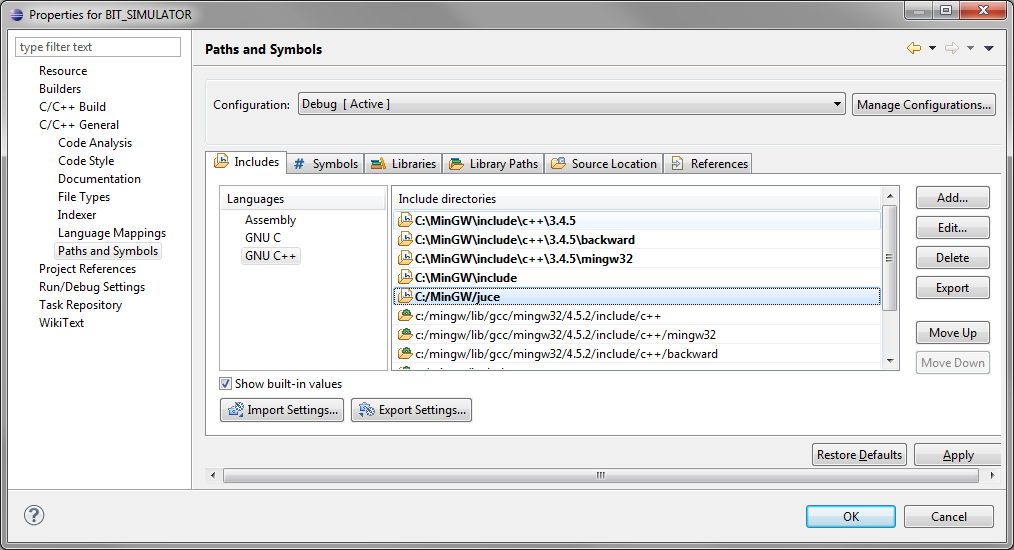
The C++ API and "*SimCode*" source code must be included in "*Generated*" from its messaging.

Copy the content of the folders *"CPP"* and *"SimCode"* from the project "*Example*" to the folder "*Generated*" of the project "*BIT\_SIMULATOR*".

Verify that the JUCE library is correctly installed in the folder *"C:\MinGW\juce"*.

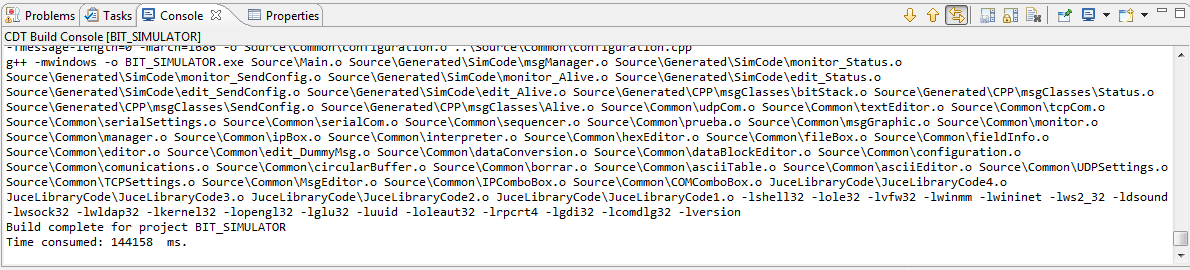
If you decide to install JUCE in a different path to the one used by default, the configuration parameters of the project must be modified to find the library in that specified path.



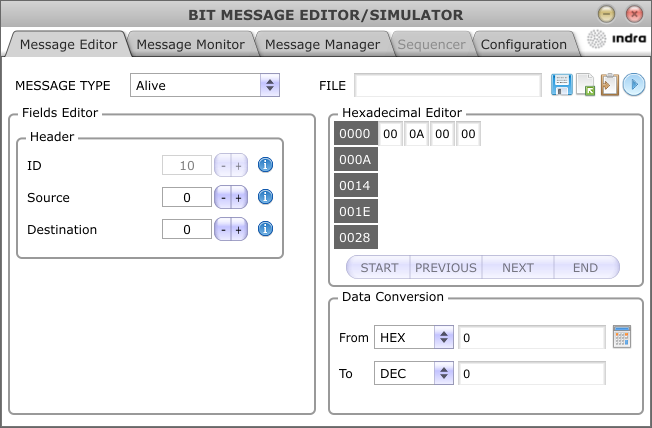


You can configure some parameters in the file "*config.h*", as the name shown in the window of the simulator and the size assigned to several buffers of the program.

Next compile the project.



After few minutes the simulator will be available:



#### Use of the simulator

The simulator is distributed in 4 tabs, each one perform different tasks:

* **Message Editor:** Allows editing the content of each message.
* **Message Monitor:** Allows sending, receiving and monitoring messages.
* **Message Manager:** Allows creating messages package.
* **Configuration:** Allows configuring the communication channel used.

##### Message Editor

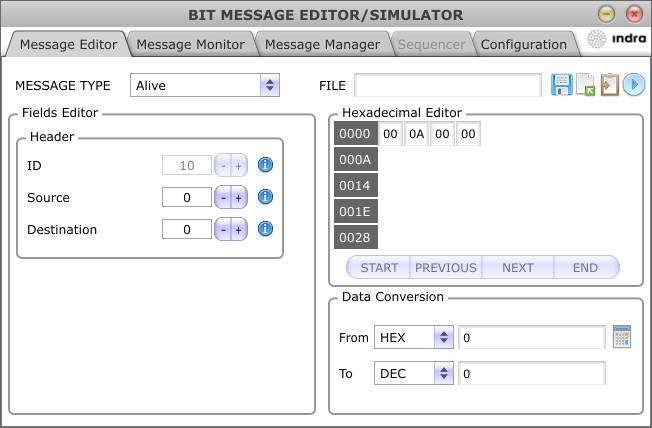
It allows configuring the fields of each message individually. Once a message is already configured with the desired values, you can save it in disk or send it using the configurable communication channel in the "*Configuration*" tab.

At the upper left side, a combo Box "*Message type*" is observed. This shows a list with all the designed messages. It also shows a message called "*DUMMY\_MESSAGE*" that allows editing a buffer of bytes with arbitrary size and content.

When selecting each one of the messages of the combo Box it is observed how the form included inside the "*Fields Editor*" group is updated to show all the dataItems and all the fields of the message. Also the "*Hexadecimal Editor*" is updated to show the content already encoded of the selected message.

From "*Field Editor*" and from "*Hexadecimal Editor*" the content of the message can be modified. The program will use the available information of each field to verify that the entered values are within the defined ranges.

Once the message is configured with the desired values, you can save it in a file of your choice, copy the data message buffer in the clipboard (in C format), or send it via one of the available channels (TCP, UDP, RS232).



Send

Copy

Save

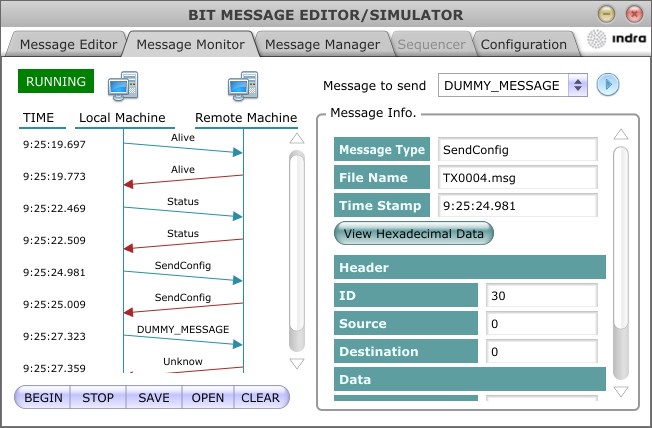
Load

A hexadecimal - decimal data converter is also available.

##### Message Monitor

The following tab of the simulator is "***Message Monitor***" that allows to register the incoming and outgoing traffic of the communication channel configured.

The option to send messages from the combo Box "*Message to send*" located on the upper right side is also available. The data contained in the messages sent from here will be the one previously configured in the "*Message Editor*".



Save log

Clean log

Open log

Disable login

Enable login

In transmission and reception, the messages monitor tries to identify the type of registered messages. In case of success, it provides information of that message broken down in fields.

It also has the option to see the content of the message in hexadecimal/ASCII format and save the complete log in disk.

Click the "BEGIN" button to have the monitor available. To stop the messages registration press "STOP". A color textbox appears at the upper left side of the window indicating the current state of the monitor.

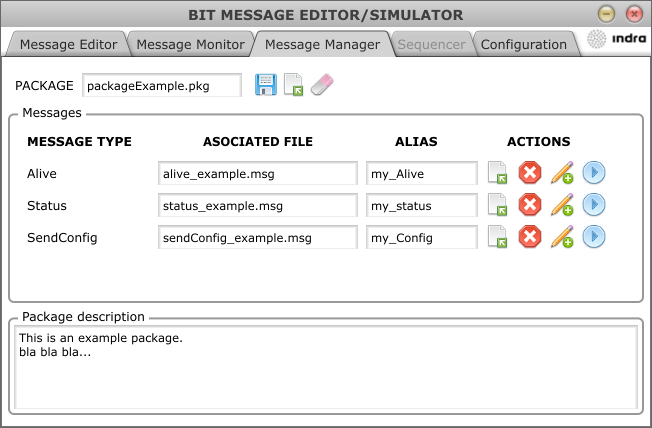
##### Message Manager

The third tab is the "***Message Manager***". This one allows defining a set of messages predefined and saving it as a simulation package/scenario.

Clean

Load

Save



Delete

Send

Edit description

Open File

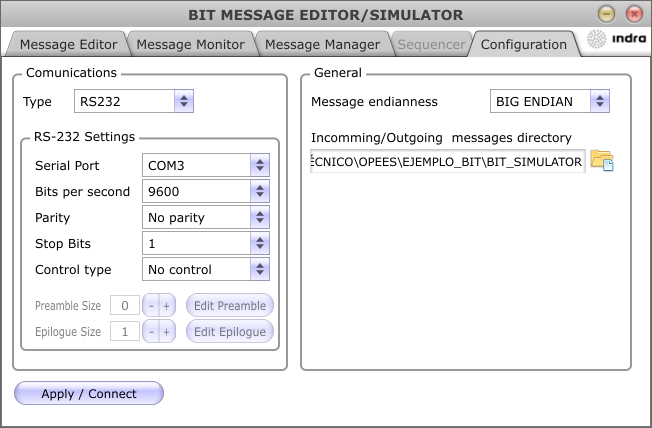
The messages must be previously saved in files. Each one can have an alias used in the "*Message Monitor*" to identify the outgoing messages.

Also a description can be written to indicate which data has the message or what it is for.

The complete package can also have a text to describe the content and scope of the package.

##### Configuration

The last tab of the simulator is "***Configuration***". This tab allows configuring the communication parameters, the endianness used to encode the messaging and the directory where to save the log messages received in the "*Message Monitor*" tab.



Currently, the simulator supports UDP, client mode TCP, host mode TCP and RS232 communications.

The most common parameters of each one of the available communication channels can be configured in this tab.

1. ICD, Interface Concept Document [↑](#footnote-ref-1)