USER MANUAL

Version 3.0

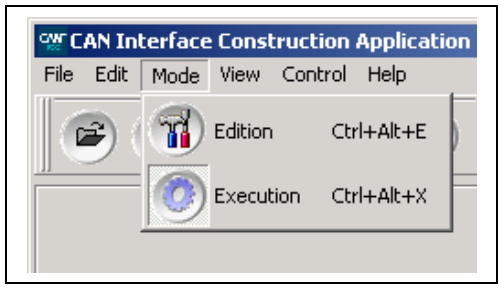
Table of Contents

1.INTRODUCTION

CANica is an engineering tool designed to assist the fast development of graphical applications based on communications buses like CAN and LIN.

1.1. CANica working modes

CANica has two different working modes easily switched either from shortcuts or from main menu.



 The Execution Mode

This is the normal mode in which you load a pre-defined CANica configuration and run it. From there, you can stop and resume the execution. The only additional view of this mode is the Development Trace.

Available actions:

-  Start the execution.
-  Stop the execution.
-  View the Development Trace.

 The Edition Mode

This is the mode used to develop new custom CANica configurations. From this mode, you have access to the Code editor window and to the Component Palette Toolbar. It's possible to be in Edition mode and run, which is commonly used during debugging process of the currently developed configuration.

Available actions:

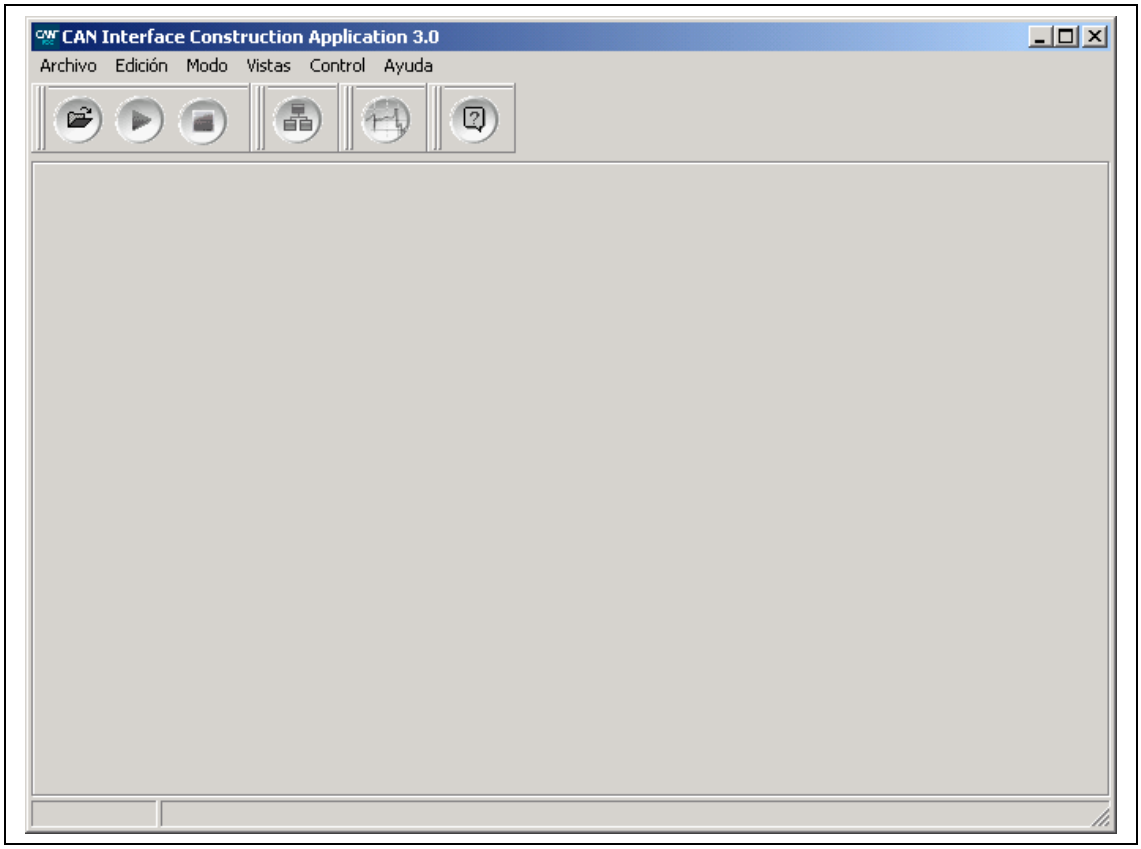
-  Start/Resume the execution.
-  Pause the execution.
-  Stop the execution.
-  Step by step execution (debugging).
-  View Development Trace.

-  View Components Palette.
-  View Code Editor.
-  View Component Inspect Window.

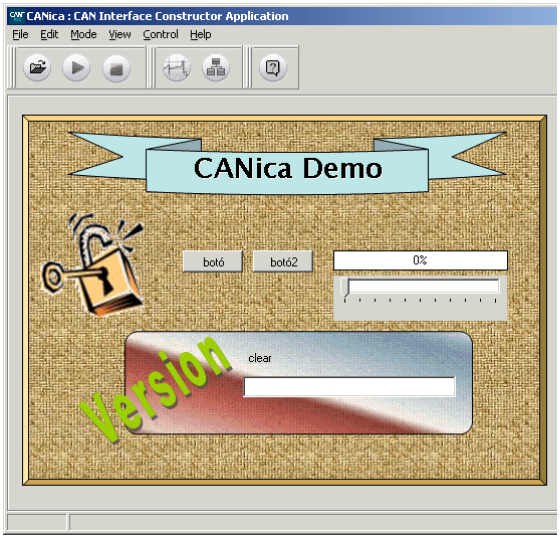
1.2. PROGRAM WINDOWS AND DIALOGS

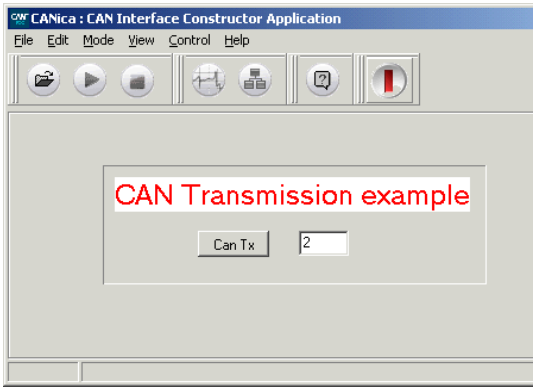
1.2.1. Main Window

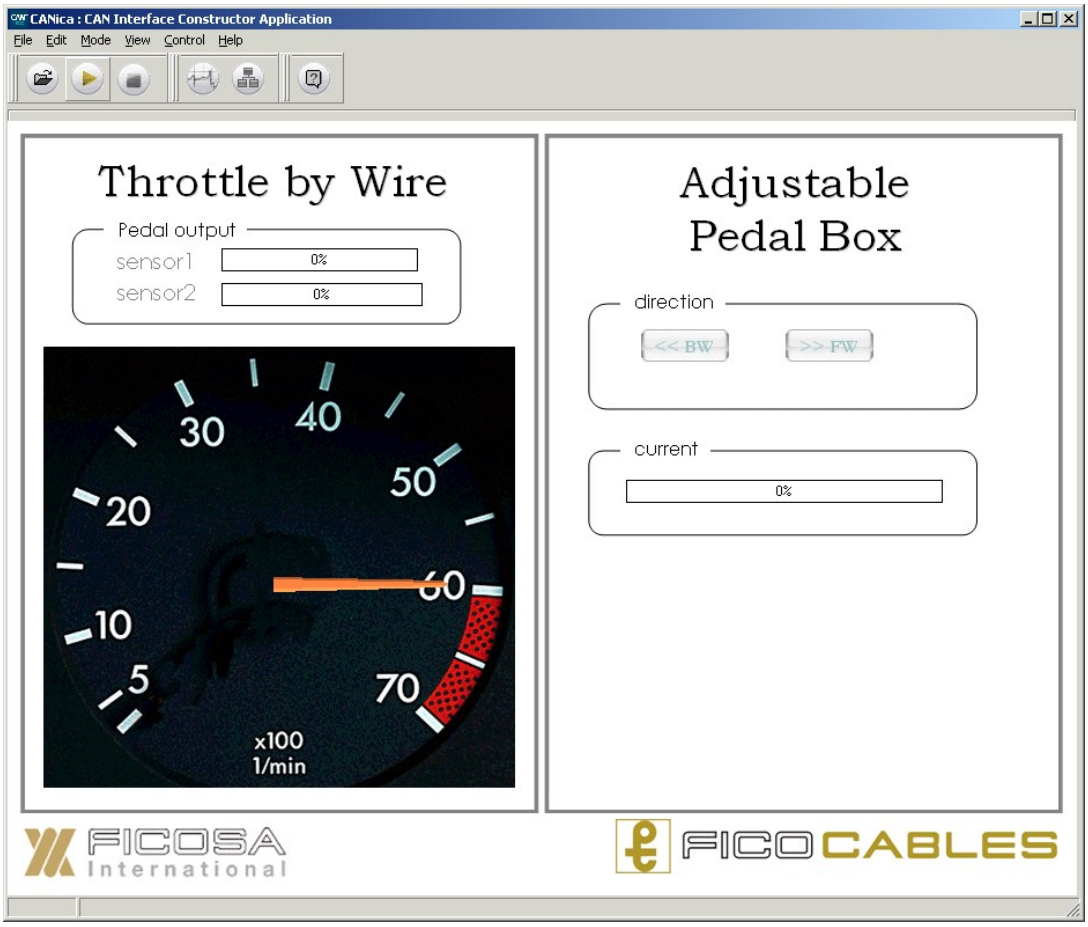
The main window is the working window of the application.



In the following images you can find some real applications made with CANica:

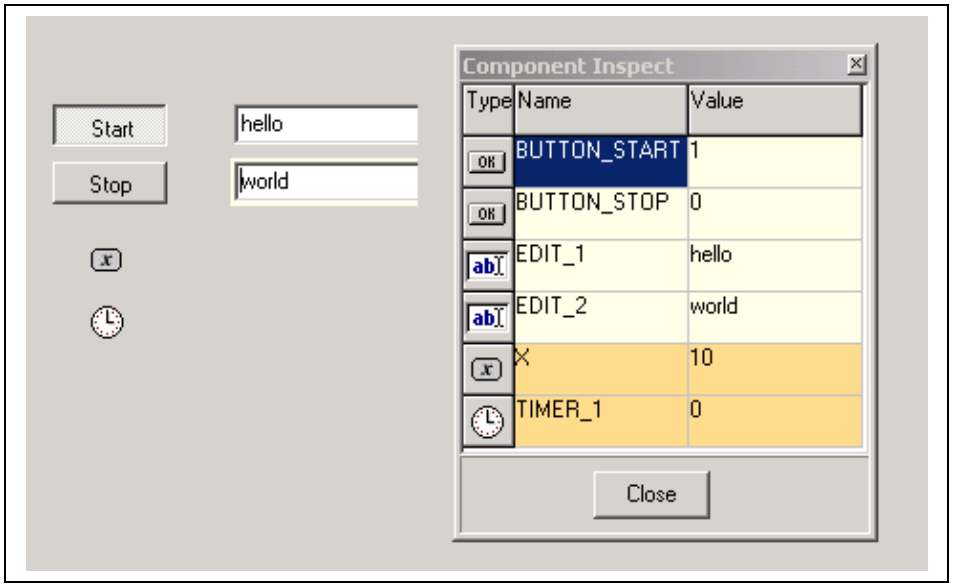






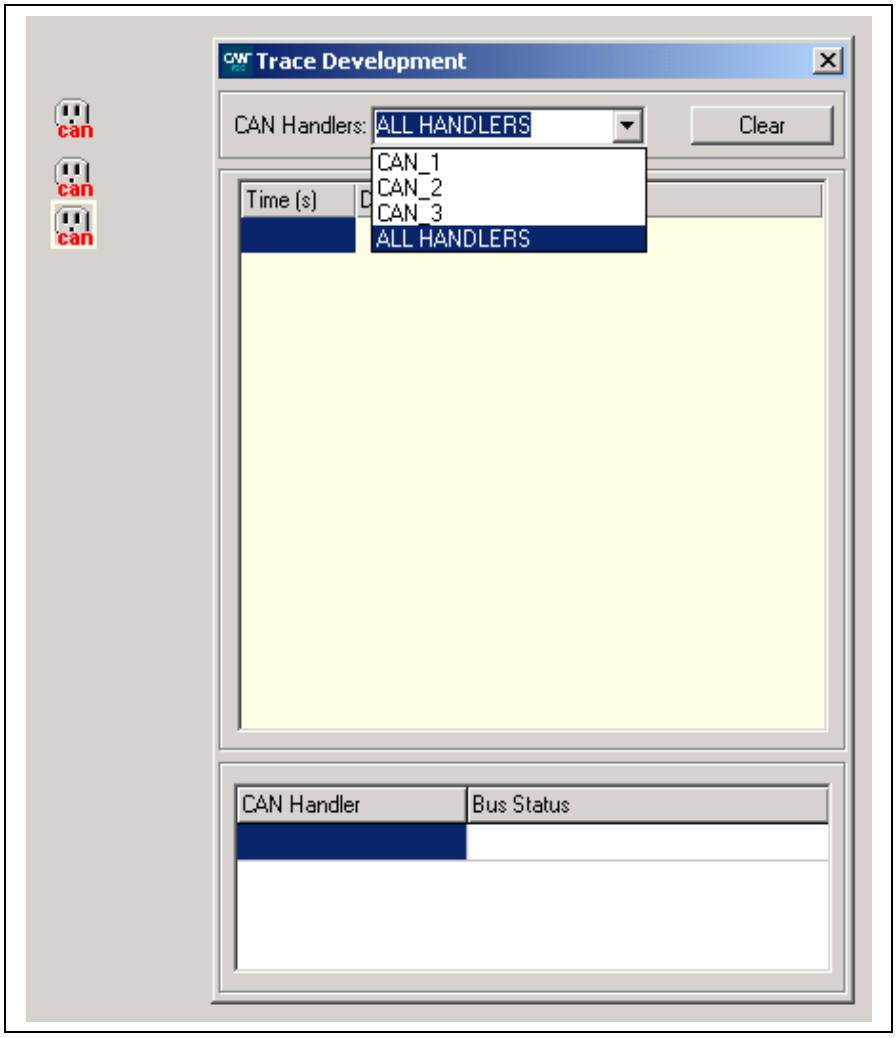
1.2.2. Inspect Window

The Inspect window is mostly used for debugging purposes. It helps the developer of CANica applications to inspect the value of the components defined, and see how they react to events and to program execution.
It's also possible to change the value of any component by double-clicking on it.



1.2.3. Development Trace

The Development Trace allows the user watch the bus CAN status every time he wants without the necessity of introduce his own Trace component in his application. This window operates as a Trace component but it can be modified only the associated handlers.

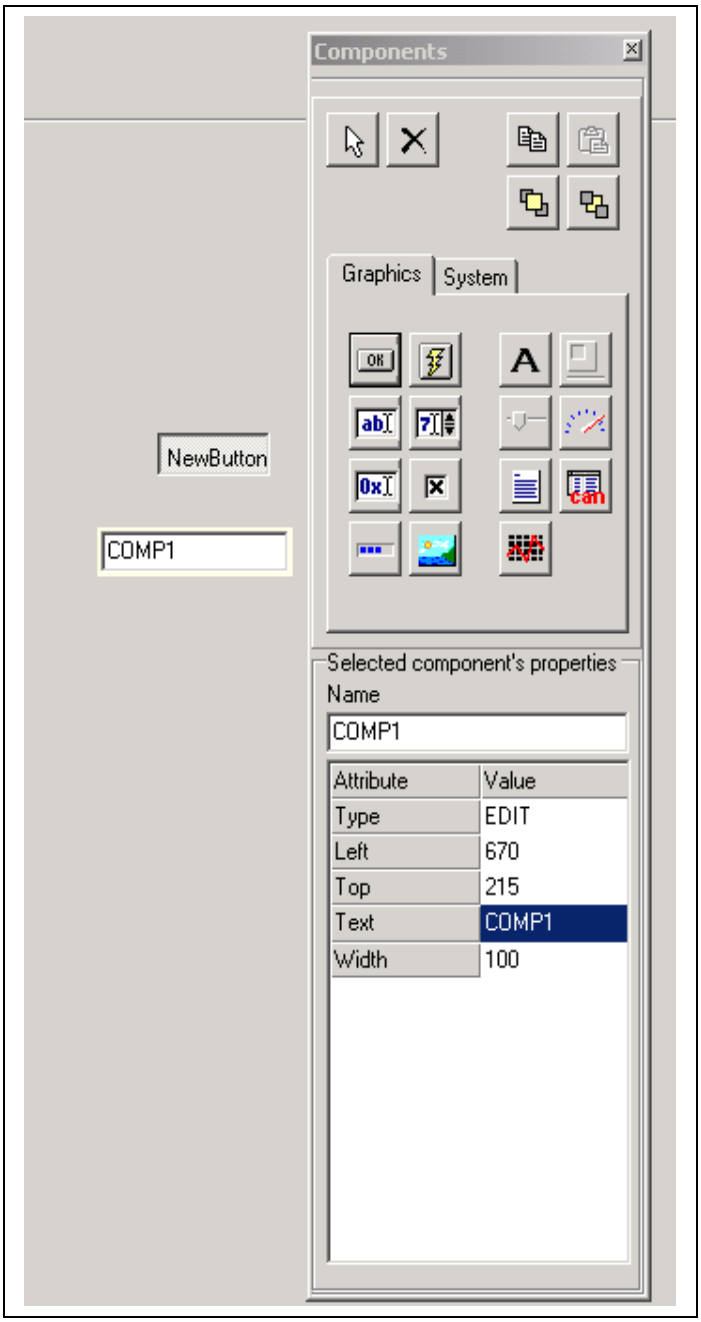


1.2.4. Component Palette

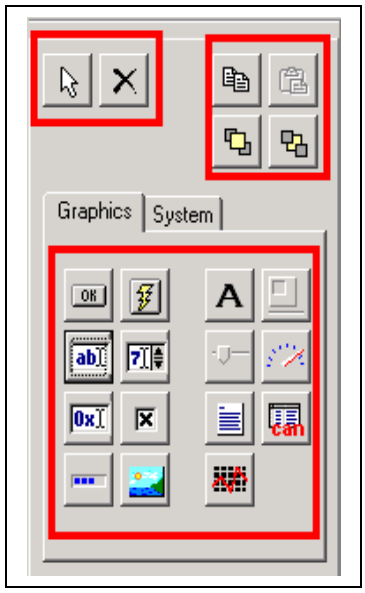
This palette appears when you enter the Edition Mode. It's based on a set of icons that represent the different types of component that CANica supports, organized in different Tabs.

At the bottom there is a list of attributes of the selected component in the main form (in the case that there is one) that you can manually change.

Some attributes are automatically updated, as for example the position, which changes anytime you move the component from its position in the screen.



Icons are arranged according to their category:



Selection and deleting tools:

 ***Selection*** button, used to select one component to change its attributes

 ***Delete*** button: the object you click on will be removed.

Copy/Paste and graphic positioning:

 ***Copy***: copies the selected component

 ***Paste***: pastes a previously copied component, creating a new one with the same attributes and type that the original one.

 ***Bring to front***

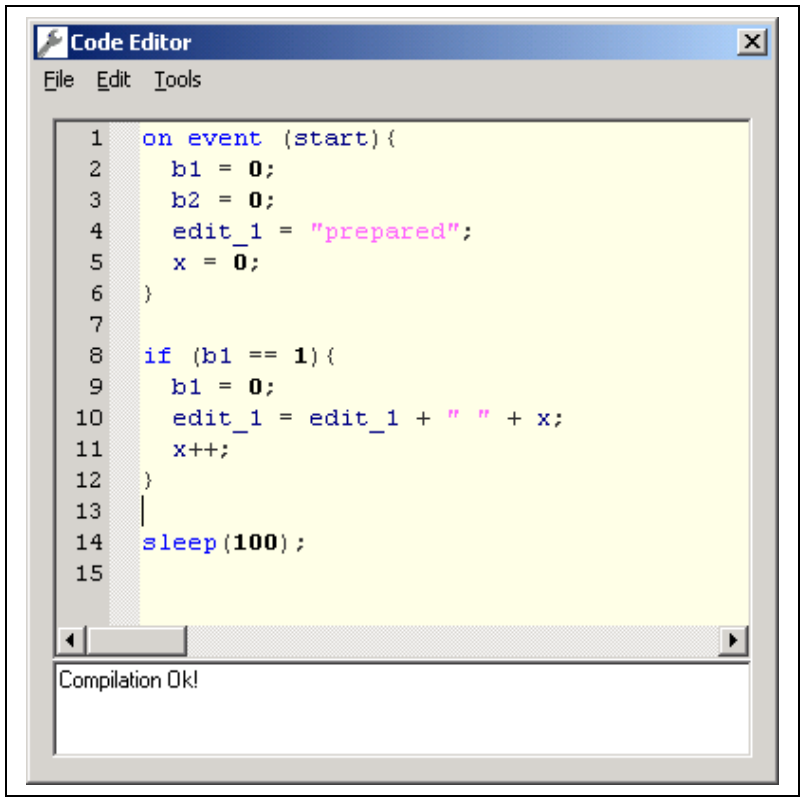
 ***send to back***.

Application components:

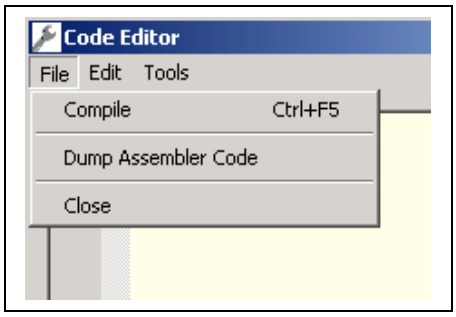
They allow to choose a kind of component to create new instances in the working area.

1.2.5. Code Window

The code window is the editor to type the code of the application.
It has been provided with syntax highlighting to simplify the detection of codification errors.
At the bottom of the window there is an output window where the compiler notifications are displayed.

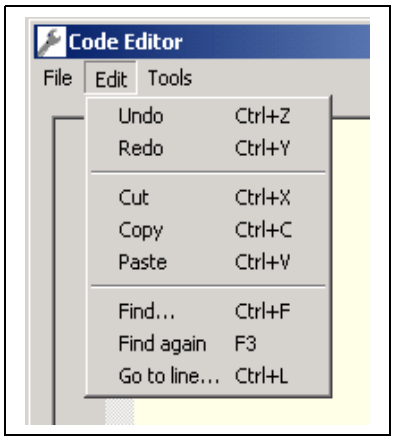


File Menu options:



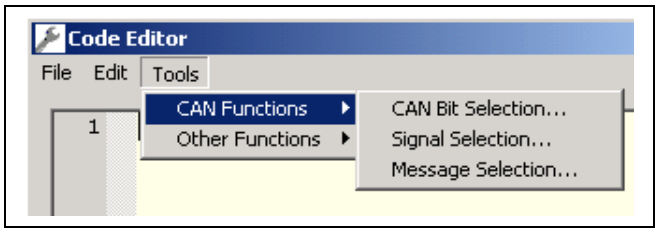
- **Grammatical Check:** Allows developers to check the grammatical compliance of developed code. It is useful to check developed code periodically to find easily syntax errors.
- **Dump Assembler Code:** Dump the internal assembly code generated by CANica from a user script. This file is the result of the internal compilation and it can be useful when a professional programmer need to debug an application.
- **Close:** Close the Code Window.

Edition Menu options:

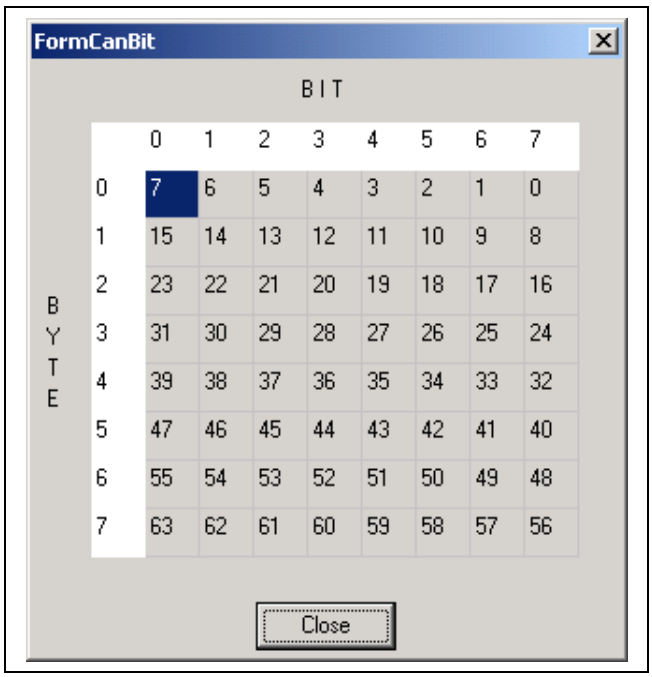


- Typical edition options.

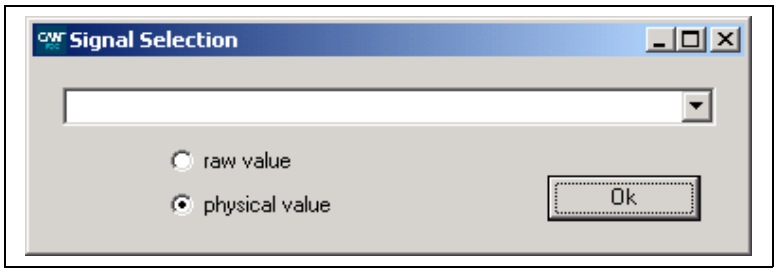
Tools Menu options (Assistants):



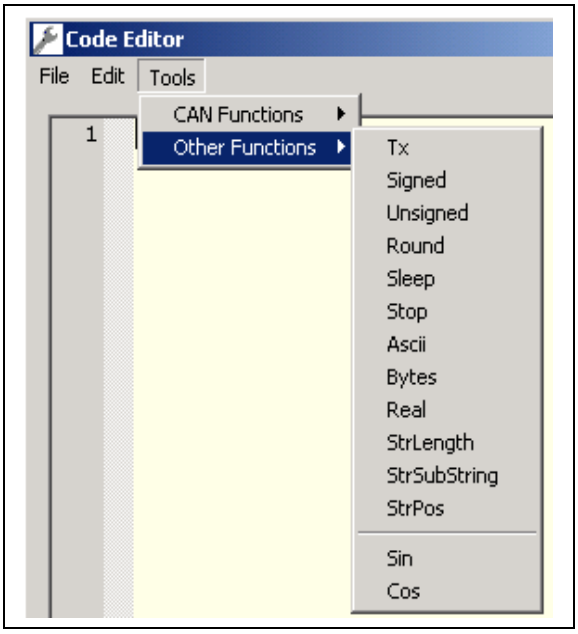
- **CAN Bit Selection:** Helps the user to select the frame bits graphically.



- **Network DB Explorer:** Is an assistant which lets navigate through the different frames and signals defined in a Network description data base, and assists users letting him copy the name of the selected element to the code editor in a fast way.



Tools Menu Options (Other Functions):



This menu offers several native functions to be used in application code by users.

1.3. Drivers and supported hardware

1.3.1. CAN Interfaces

Currently CANica supports the following hardware:

- KVASER CAN interfaces
- Vector Informatik CAN Interfaces
- IXXAT CAN Interfaces
- National Instruments CAN Interfaces

For additional interfaces please contact us.

1.3.2. LIN Interfaces

Currently CANica supports the following hardware:

- KVASER
- Vector Informatik LIN Interfaces

For additional interfaces please contact us.

1.4. Command Line Options

For normal handling, the use of command line parameters is not necessary, although for automating a few task the following available options could be useful:

Option	*Description*
-o FileName	Automatically opens the file FileName after starting the application.
-nodisclaimer	Human confirmation is not required to close the splash where a warning about the hazardous use of CANica connected to a communication bus without the required knowledge is shown. Hereby FicoTriad s.a, a company within FICOSA International expressly disclaims any liability for any bad usage of this option.

2.PROGRAMMING LANGUAJE

CANica has a scripting language that is syntactically similar to C/C++, with some simplifications and differences.

2.1. Constants

CANica allows three kind of constant values, integers, reals and character strings.

2.2. Variables

In CANica's programming language only components could be used. An instance of a component do not have any predefined type because CANica interprets dynamically it according to the context. This way of working simplifies the job of developing CANica based configurations because developers could assign integer, reals, and strings to components with no care.

2.3. Assignment statement

The basic statement in CANica is the assignment. It is the way of writing to and reading from components. Using assignments it's possible to change the value of a component, a part of a CAN message, send a string through the UART and other functions, depending on the component being written. In the same way, reading a component can have side-effects on it. As an example, if a UART component is read, it returns the pending buffer that arrived through it, and then it resets the buffer.

Syntax:

```
component = expression;
component.attribute = expression;
```

Examples:

```
Edit1 = 4;
Edit2 = "Hello World!";
Edit3 = 0xfe;
Edit3.width = x;
```

2.4. Expressions

Expressions are used to perform more that one mathematical operation within a single sentence. The result of an expression can be assigned to a variable, passed to a predefined function, etc.

Examples:

```
TrackBar1 = ( 4 + Edit1) / 2;
Aguja = 4+ int(16.5);
```

2.5. if...else statement

if..else statement is a conditional branch statement which evaluates just like in C or C++.

Syntax:

```
if (condition_expression) {
  instructions
}
```

```
if (condition_expression) {
  instructions
}
else {
  instructions
}
```

A condition_expression can be

- A boolean expression (e.g. `Edit1 >= 4, Check1 == 1`)
- A constant expression. It will evaluate true if it is an integer and it is not zero, and if it is a string and it is not null ”\0”. (e.g. 1, 0, “Hello”, “”)

Example:

```
if (var1 < 23) {
     var1++;
}
else {
     var1 = 0;
}
```

2.6. for statement

A for statement is a loop with an initial action, a remain condition and an end-of-loop action, exactly like C.

Syntax:

```
for (inicial_action; condition; instruction) {
     instructions
}
```

Example:

```
// this examples send the message with id 0x111
// 8 times. Each time with different length,
// according to the value of variable len.
for (len = 0; len < 8; len++) {
      h1.tx(can111h, len);
}
```

2.7. while statement

A while statement executes a loop while the condition evaluates true.

Syntax:

```
while (condition_expression) {
      instructions
}
```

Example:

```
while (var1 < 10) {
      var1 = edit1;
}
```

2.8. switch...case statement

A switch...case statement can be defined, with the following restrictions and characteristics:

- the default case is optional
- every case must have a brake statement, even the default.

Syntax:

```
switch (expresión) {
      case constante:
            instrucciones
            break;
      case constante:
            instrucciones
            break;
      default:
            instrucciones
            break;
}
```

Example:

```
switch (h1.byte(can200h,0) {
```

```
      case 0:
            edit1 = "P";
            break;
      case 1:
            edit1 = "N";
           break;
      default:
            edit1 = "?";
            break;
}
```

2.9. Functions

2.9.1. Native Functions

Native functions are a subset of functions internally implemented in CANica which could be used by developers. Each one is described in the following paragraphs.

2.9.1.1. Round

This functions returns the integer part of a number.

Syntax:

```
return_value = Round(x);
```

Example:

```
// edit1 will be 3
edit1 = Round(3.78);
```

2.9.1.2. Beep

Plays a simple beep.

Syntax:

```
Beep();
```

Example:

```
// play 'beep'
Beep();
```

2.9.1.3. Sleep

Sleeps a number of milliseconds the execution of CANica.

Syntax:

```
sleep(x);
```

Example:

```
// Application remains paused for 1 second
sleep(1000);
```

2.9.1.4. Ascii

This function returns the ASCII character that corresponds to the integer given.

Syntax:

```
return_value = Ascii(x);
```

Example:

```
// edit1 will be 'A'
edit1 = Ascii(65);
```

2.9.1.5. Signed

This functions returns the interpretation of an unsigned integer of given length as it was a signed integer.

Syntax:

```
return_value = Signed(x, nbits);
```

- **x:** integer
- **nbits:** length codification bits.

2.9.1.6. Unsigned

This function returns the interpretation of a signed integer of given length as it was an unsigned integer.

Syntax:

```
Unsigned(x, nbits);
```

- **x:** integer

- **nbits:** length codification bits.

2.9.1.7. Real

This function returns the conversion of a given 32 bit integer into a floating point according to the IEEE representation.

Syntax:

```
return_value = Real(x);
```

2.9.1.8. Bytes

This function separates a number of bytes from a buffer with the following format:
00 01 02 03 04 05 06 …where numbers are in hexadecimal and are separated by one space.
Use this function together with a full message assignment.

Syntax:

```
return_value = Bytes(buffer, ini, end);
```

- **buffer:** information buffer.
- **ini:** inicial byte.
- **end:** end byte.

Example:

```
//copies data received though Com2 to buff
buff = com2;
// copies the first two bytes of the buffer
// into p1
p1 = bytes (buff, 0, 2);
```

2.9.1.9. OpenDialog

This function returns the file name selected by users after opening a Select File Dialog.

Syntax:

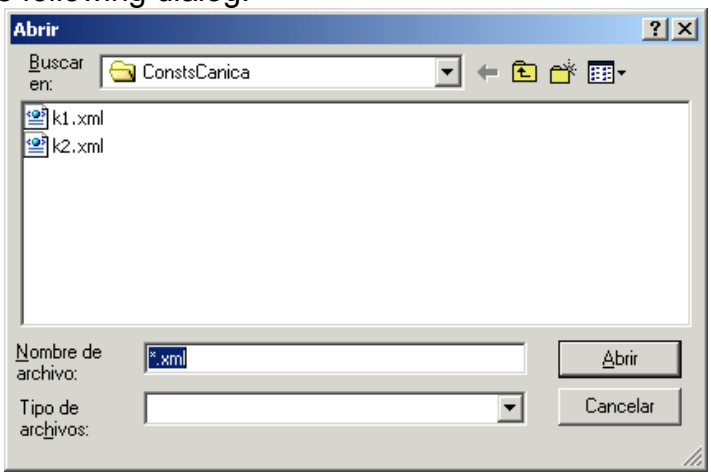
```
return_value = OpenDialog(filter);
```

- **filter:** indicates the extension of the consulting files.

Example:

```
F1 = OpenDialog("*.xml");
```

Would open the following dialog:



2.9.1.10. StrPos

This function returns the first position of a substring into a given string.

Syntax:

```
return_value = StrPos(str, substr);
```

- **str:** main string.
- **substr:** substring to search.

Example:

```
// edit1 wil be 4
edit1 = strops("my string", "str");
```

2.9.1.11. StrLength

This function returns the length of a string

Syntax:

```
return_value = Strlen(str);
```

Example:

```
// edit1 will be 9
edit1 = strlen("my string");
```

2.9.1.12. StrSubString

This function returns a new string that is a substring of the string given. The substring contains a number of characters beginning at an initial position.
Important note: the first position of a string is position 1.

Syntax:

```
return_value = StrSubString(str, ini, count);
```

- **str:** main string.
- **ini:** first position in main string to create the substring.
- **count:** number of characters of the new substring.

Example:

```
// edit1 will be "two"
edit1 = StrSubString("one two three", 5, 3);
```

2.9.1.13. Date

This function returns the current date as a string.

Syntax:

```
return_value = Date();
```

2.9.1.14. Time

This function returns the current time as a string.

Syntax:

```
return_value = Time();
```

2.9.1.15. Random

Returns a random number between 0 and 999999.

Syntax:

```
return_value = Random(x);
```

2.9.1.16. Math functions

2.9.1.16.1.sin

This function returns the sine of an angle passed in degrees.

Syntax:

```
return_value = sin(x);
```

2.9.1.16.2.cos

This function returns the cosine of an angle passed in degrees.

Syntax:

```
return_value = cos(x);
```

2.9.1.16.3.asin

Returns the arc sine of the input value.

Syntax:

```
return_value = asin(x);
```

2.9.1.16.4.acos

Returns the arc cosine of the input value.

Syntax:

```
return_value = acos(x);
```

2.9.1.16.5.tan

Calculates the tangent. Angles are specified in radians.

Syntax:

```
return_value = tan(x);
```

2.9.1.16.6.atan

Returns the arc tangent of the input value.

Syntax:

```
return_value = atan(x);
```

2.9.1.16.7.exp

Calculates the exponential e to the x.

Syntax:

```
return_value = exp(x);
```

2.9.1.16.8.log

Calculates the natural logarithm of x.

Syntax:

```
return_value = log(x);
```

2.9.1.16.9.pow10

Calculates 10 to the power of p.

Syntax:

```
return_value = pow10(x);
```

2.9.1.16.10.log10

Calculates the base ten logarithm of x.

Syntax:

```
return_value = log10(x);
```

2.9.1.16.11.pow

Calculates x to the power of y.

Syntax:

```
return_value = pow(x,y);
```

2.9.1.16.12.sqrt

Calculates the positive square root of the argument x.

Syntax:

```
return_value = sqrt(x);
```

2.9.2. User defined functions

CANica lets developers define their own user function. This kind of functions could either have parameter or do not have parameters, and could either return a result or do not return a result. In the case a result is required the developer must use the *return* statement at the end of the function with the value to be returned.
All user defined functions must be located at the end of the script code just bellow the on event statements.
If any function name is reused by two or more function within the same script only the first one will be executed, the rest will be ignored.
Native function must not be rewritten, if any user defined function uses the same name as a native function, the user defined one will be ignored.

Syntax:

```
/* -------- Main Program Section -------- */
/* ----------- Events Section ----------- */
/* ------- User Functions Section ------- */
function name(parameters)
{
    statements;
}
/* ----------------- End ---------------- */
```

```
/* -------- Main Program Section -------- */
/* ----------- Events Section ----------- */
/* ------- User Functions Section ------- */
function name(parameters)
{
    statements;
    return value;
}
/* ----------------- End ---------------- */
```

- **name:** Function mame.
- **parameters:** parameter list of the function. It can be none.
- **statements:** function body.
- **return value:** value is the value returned by the function.

Examples:

```
HelloWorld_1();
HelloWorld _2(edit1);
HelloWorld _3edit1, "Hello World!");

var = edit1;
HelloWorld _4(var);

edit1 = HelloWorld _5();

// the final behavior of these 5 functions is the
// same: edit1 = "Hello World!";
// ----------------------------------------
function HelloWorld_1()
{
     edit1 = "Hello World!";
}
// ----------------------------------------
function HelloWorld_2(var)
{
     var = "Hello World!";
}
// ----------------------------------------
function HelloWorld_3(var, string)
{
     var = string;
}
// ----------------------------------------
function HelloWorld_5()
{
     return "Hello World!";
}
```

```
edit1 = DateTime();
// ----------------------------------------
function DateTime()
{
     return date() + " , " + time();
}
```

Functions have the same scope as the main loop, so any defined component could be used inside functions. Functions could also be nested, but recursive call either direct or indirect are not allowed.

2.10. Comments

CANica allows comments as they are defined in C/C++.

```
// This is a comment
/* This is a comment */
```

3.COMPONENTS

Components are the elementary entities to be managed inside CANica. Components can be read and written. They have statically configurable attributes, and dynamically configurable attributes. They also have methods which could be executed, and could trigger events. All components have a statically configurable attribute called **name** used to reference them inside the script code, so it has to be unique inside the configuration.

All components are graphically shown in edition mode, so they could be placed inside the working area. Due to this, all components have two basic location attributes: Top and Left. The left-top corner of the working area is (0,0).

3.1. Attributes

Attributes of components could be either addressed for reading or writing using the following syntax.

Syntax:

```
Component.Attribute
```

Examples:

```
edit1.width = 200;
trackbar1.maxValue = Edit3;
button.left = edit1.left;
```

3.2. Methods

Methods of components are similar to functions. As functions, methods could either have parameters or do not have, and could either return a result or do not. The following section describes the right syntax.

Syntax:

```
Component.Method(p1)
```

Example:

```
// Write of byte 2 of a CAN frame
can1.byte(0x1000X, 2) = 200;
// Transmission of the frame
can1.tx(0x1000X, 8);
```

3.3. Events

As it has been mentioned above, components could notify asynchronously different events. Processing this event is possible using a on event statement.

All on event statement must be located at the end of the main cycle statements just before the user defined functions declaration section. The following paragraphs describes the right syntax.

Syntax:

```
/* -------- Main Program Section -------- */
/* ----------- Events Section ----------- */
// captura del cambio de estado de un botón
on event (event_name)
{
     instructions
}
/* ------- User Functions Section ------- */
/* ----------------- End ---------------- */
```

- **event_name:** Events usually are of the form '*ComponentName*', but could also be '*ComponentName.Event*' if the component could notify more than one event.

Example:

```
/* -------- Main Program Section -------- */
/* ----------- Events Section ----------- */
// captura del cambio de estado de un botón
on event (button_1)
{
    button.left++;
    button_1 = 0;
}
/* ------- User Functions Section ------- */
/* ----------------- End ---------------- */
```

3.4. Graphical Components

Graphical components are those which mange a graphic view located inside the working area during execution.

3.4.1. Button

Icon	
Description	This is a typical Windows toggle button: when clicked, it is

	pressed (down) and when clicked again, it is released (up).
Read	When read, a value of “1” means pressed, and a value of “0” means released**.**
Write	It can also be pressed or released by writing a “0” or a “1” on it.
Attributes	***State:*** The state of the button, a value of “1” means pressed, and a value of “0” means released. ***Width:*** The width of the button. ***Height:*** The height of the button. ***Text:*** The text displayed inside the button component. ***Group:*** The name of the group of buttons to which the button component belongs. Remember that in a group of button only one button could be pressed at the same time.
Methods	None
Events	***Name:*** State change.
Example	

3.4.2. ImageButton

Icon	
Description	This component is functionally identically to the previous described with the difference that it displays a user defined image on it.
Read	When read, a value of “1” means pressed, and a value of “0” means released**.**
Write	It can also be pressed or released by writing a “0” or a “1” on it.
Attributes	***State:*** The state of the button, a value of “1” means pressed, and a value of “0” means released. ***Image0:*** The name of the image file to be displayed on the button if it is released. ***Image1:*** The name of the image file to be displayed on the button if it is pressed.
Methods	None
Events	***Name:*** State change.
Example	

3.4.3. Edit

Icon	
Description	This is a simple windows Edit component.
Read	When read it returns the text contained in it.
Write	When set, it changes the text by an appropriate representation of the source (even if it’s a number).
Attributes	***Text:*** The text displayed inside the edit.

	Width: The width of the edit.
Methods	None
Events	***Name:*** Contents change.
Example	Hello

3.4.4. SpinEdit

Icon	
Description	This component shows only decimal numbers, and it provides spin buttons to increment or decrement its value.
Read	When read, it returns the number itself.
Write	When set, it represents the source.
Attributes	***Value:*** The value set. ***Width:*** The width of the spinedit.
Methods	None
Events	***Name:*** Contents change.
Example	0

3.4.5. HexEdit

Icon	
Description	This type of Edit component is reserved to display hexadecimal data.
Read	When read, it returns the number itself.
Write	When set, it represents the source using the hexadecimal representation.
Attributes	***Text:*** The value set. ***Width:*** The width of the hexedit.
Methods	None
Events	***Name:*** Contents change.
Example	0x4

3.4.6. Checkbox

Icon	
Description	A Checkbox component can be checked or unchecked by clicking on it.
Read	When read a value of “1” means checked, and a value of “0” is unchecked.
Write	When written it is checked if “1” is written and it is unchecked if “0” is written.
Attributes	***State:*** The state of the button, a value of “1” means pressed, and a value of “0” means released.
Methods	None

Events	***Name:*** Status changed.
Example	

3.4.7. Progress Bar

Icon	
Description	A progress bar represents from 0% to 100% by showing a blue bar. Its value can be modified inside the though the code, but the user can not interact with it.
Read	When read it returns its value.
Write	When set its value is changed.
Attributes	***Value:*** Components value. ***Width:*** The width of the component.
Methods	None
Events	None.
Example	

3.4.8. Image

Icon	
Description	An image component can display images on the application area by providing the file name of an existing image.
Read	When read, the name of the currently displayed image is retrieved.
Write	When written the source image is changed.
Attributes	***File:*** The initial image file to be displayed. ***Width:*** The width of the component. ***Height:*** The height of the component. ***Transparent***: Enables the image transparency.
Methods	None
Events	None.
Example	

3.4.9. Text

Icon	
Description	A text component simply shows a text on the window. The text to be displayed can be changed during execution time.
Read	When read, it returns the text it shows itself.
Write	When set, it changes the text it shows.
Attributes	***Text:*** The text displayed inside the text component.

	Size: The width of the text component. ***Color:*** Font color.
Methods	None
Events	None.
Example	This is a text

3.4.10. Bevel

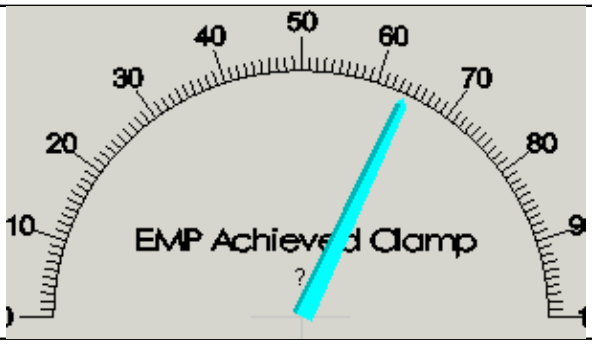
Icon	
Description	A bevel component is a rectangle that is used to visually group a number of components and to make the application more attractive. It has no implications on the behavior of the application.
Read	Not possible.
Write	Not possible.
Attributes	***Width:*** The width of the bevel component. ***Height:*** The height of the bevel component.
Methods	None
Events	None.
Example	

3.4.11. Trackbar

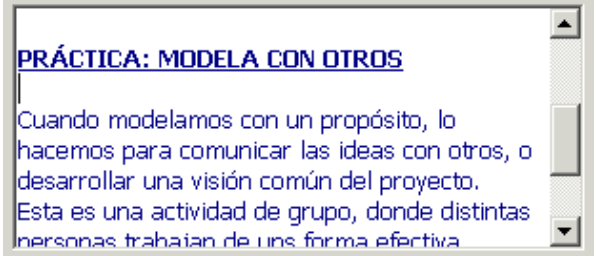
Icon	
Description	A trackbar component is used to select a value in the range of values between 0 and Max just by moving the bar that it contains.
Read	When read it gets its value.
Write	When written it sets its value.
Attributes	***Min:*** The minimum value admissible. ***Max:*** *Value*: The maximum value admissible. ***Width:*** The width of the trackbar component.
Methods	None
Events	***Name:*** Value changed.
Example	

3.4.12. Gauge

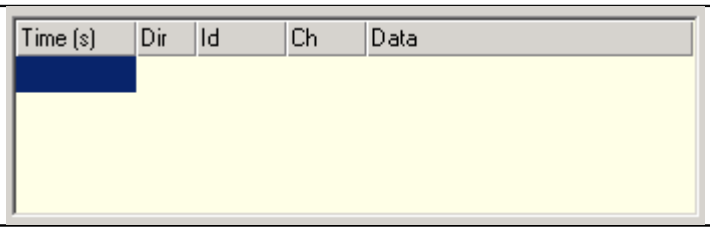
Icon	
Description	A Gauge component will show a needle that can rotate 360º.
Read	Reading it will return the current value. The conversion

	formula to degrees is grad = (val * factor) + offset
Write	Writing a value will change the rotation of the needle. The conversion formula to degrees is the showed in the previous section.
Attributes	***Value:*** Value of the component. ***Radius_1:*** Length of the needle. ***Radius_2:*** Portion not visible of the needle. This portion begins in the rotation point. ***Offset:*** Initial rotation of the needle. ***Factor:*** Conversion factor to degrees. ***Color:*** Color of the needle. ***MaxAlfa***: Max rotation angle.
Methods	None
Events	None.
Example	

3.4.13. Memo

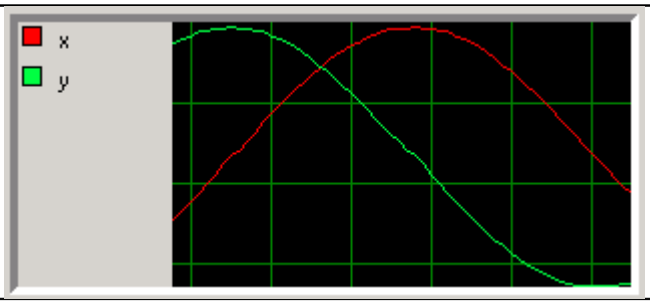
Icon	
Description	Similarly, a Memo component represents the content of a memo file. Accepted formats are: raw text, RTF. The Memo components are read only. Writing a text will try to open a new memo file and load it to the component.
Read	When read the file name which content is displayed is returned..
Write	When written a new file which content will be displayed is set.
Attributes	***Width:*** The width of the memo component. ***Height:*** The height of the memo component. ***File:*** The initial text file to be displayed. ***Color:*** The color of the component.
Methods	None
Events	None.
Example	

3.4.14. Trace

Icon	
Description	This component shows the traffic of network bus depending of the associated handlers.
Read	Not possible.
Write	Not possible.
Attributes	***Width:*** The width of the component. ***Height:*** The height of the component. ***Color:*** The color of the component. ***Handler:*** Handler associated. 'ALL HANDLERS' means that the trace will show the information of all the CanHandlers and LinHandlers in the application. ***TimeWidth:*** The width of Time column. ***DirWidth:*** The width of Direction column. ***IdWidth:*** The width of Id column. ***NameWidth:*** The width of Name column. ***ChWidth:*** The width of Channel column. ***ChecksumWidth:*** The width of the LIN checksum column.
Methods	None
Events	None.
Right Button	Popup Menu with option clear.
Example	

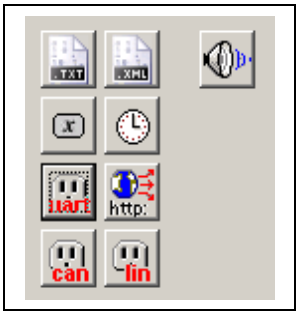
3.4.15. Graphic

Icon	
Description	This component shows a graphic with the selected signals varying in time. X axis represents the time and Y axis represents the signals value. This component offers a general vision about the signals behavior but not in a representative scale.
Read	Not possible.
Write	Not possible.
Attributes	***Width:*** The width of the component. ***Height:*** The height of the component. ***SignalWidth:*** The width of the signal inscription. ***Color:*** Background color.

	Grid: Indicate whether a grid is displayed or not. ***Xaxis:*** Indicates whether the Abscissae axis is displayed or not. ***YAxis:*** Indicates whether the Ordinates axis is displayed or not. ***Max***: Maximum value displayed in the graph. ***Min:*** Minimum value displayed in the graph. ***SampleTime::*** Sample frequency. ***Monitors:*** Open the form used to set up the signals monitored in the graph.
Methods	None
Events	None.
Example	

3.5. System components.

System components are those which does not have a graphical representation in the working area during execution.



3.5.1. Application

Icon	
Description	This component includes the working area attributes. It can't be created and it can't be destroyed.
Read	Not possible.
Write	Not possible.
Attributes	***Height:*** The height of the working area. ***Width:*** The width of the working area. ***Color:*** The color of the working area. ***FullScreen:*** Indicates if the application should be shown full sized. ***AutoStart:*** Indicates if the application will automatically start execution after load.
Methods	***Stop():*** Method to stop execution.
Events	***Start:*** Notification of execution starting. ***Stop:*** Notification of execution stopping.

3.5.2. Keyboard

Icon	
Description	This component manages the keyboard. Every CANica configuration has one and it could not be neither created nor deleted.
Read	When read the last key pressed is returned.
Write	No possible.
Attributes	None.
Methods	None.
Events	***key:*** Notification of a key pressed.

3.5.3. File

Icon	
Description	CANica supports the use of text files for both input and output. It must be selected a file name.
Read	When reading, it will return the next line.
Write	When writing to it, it will write a line of text showing the given value.
Attributes	***File:*** The name of the file to be associated to the File component.
Methods	***Seek(offset)***: Sets the pointer to the next readable byte to the byte located at offset from the beginning of the file.
Events	None.

3.5.4. XMLDoc

Icon	
Description	With CANica it's possible to read XML files and get information from it using XPath queries.
Read	When read, the result of the last XPath query is returned.
Write	When written, a new XPath query is sent to the XML component.
Attributes	***File:*** The name of the XML file to be associated to the XMLDoc component.
Methods	None
Events	None.

Example	Here's an example: suppose that we have some values for our components defined on a XML file. It has the following aspect:
	
	`<?xml version="1.0" encoding="iso-8859-1"?>`
	`<!DOCTYPE Constants SYSTEM "consts.dtd">`
	`<constant-list>`
	`  <const name="K1" value="34"/>`
	`  <const name="K2" value="0"/>`
	`  …`
	`</constant-list>`
	
	We may want to know the value of the constant called "K1". This is the XPath query that gets this information:
	
	`  /constant-list/const[@name='K1']/@value"`
	
	To read it we have to send the query to the XPath processor, and then get the result and assign it to the

	destination component, as in the example:
	
	`X1 = "/epb-constants/const[@nom='Kp']/@valor";`
	`Edit1 = X1;`

3.5.5. Uart

Icon	
Description	Serial communication with external devices could be achieved using the uart component. This component has two functioning modes, it could be used in binary mode or in string mode.
Read	***When reading in binary mode:*** it will return the next byte available in the receive buffer. ***When reading in string mode:*** it will return a string composed by all bytes available in the receive buffer. In any case if the receive buffer is empty a read instruction will return an empty string.
Write	***When writing to it in binary mode:*** the expected behavior varies depending on the data type written. ***Integer:*** One or more bytes are pushed ordered according the endianness specified, into the transmission buffer and they are transmitted as soon as possible. ***Float:*** The float is truncated to the nearest lower integer and it is sent as in the case described above. ***String:*** Zero is sent. ***When writing to the Uart in string mode:*** all data types are converted to a string if they are not, then the resulting string is pushed into the transmission buffer and all the characters are sent as soon as possible..
Attributes	***Port:*** The COM port managed by the Uart component. **Baudrate**: The communication baud rate configured either in the device connected to the port and the PC. ***Parity:*** The parity check used in the transmission of a byte. ***Bits:*** The number of data bits for every transmitted character, they are usually 7 or 8. ***Mode:*** The transmission mode.
Methods	None
Events	***Name:*** An event is thrown whether there are data available in the receive buffer.

3.5.6. HttpServer

Icon	
Description	CANica can work as an http server, responding to client's demands in order to remotely access to the information that is contained. As a http server you have to specify a port (numeric value e.g. 1024), and give it a name..
Read	A Read access to the component will return the last request made by a client.
Write	A Write access will send a response to all active connections, that is, to all clients that are currently connected.
Attributes	***Port:*** The server socket identifier through which clients will request information.
Methods	None
Events	***Name:*** New client request.

Example	As an example, you could program a timer at 1 second, that sends the state of a component through the http connection:
	`Http1 = Edit1;`
	You can test the application by opening a telnet session using the following command line:
	`C:\>telnet localhost 1024`
	With this simple example you will receive the value of the Edit1 component periodically.

3.5.7. Var

Icon	
Description	A var component is an internal variable with not a fixed type, and with no visual representation. Actual supported types are: String, Integer and Real. CANica is able to choose the appropriate type depending on what it's written on it. If you write "hello", it will return "hello", using String as the internal data type, If you write "0x04", it will return "4", interpreting that this is a hexadecimal value, and so on. See above chapter "type conversion" for more information. A Var has an initial value that can be changed during run-time.
Read	Reading it will return its value.
Write	Writing it will set its value.
Attributes	***Value:*** Value set.

Methods	None
Events	***Name:*** Notification of value change.

3.5.8. Timer

Icon	
Description	A timer is a simple counter from 0 to the selected time in milliseconds. Whenever it reaches the maximum, it comes back to 0 an restart counting.
Read	Reading it will return its value.
Write	Writing it will set its value.
Attributes	***Time:*** Period between notifications in ms. A value of zero disables notifications.
Methods	None
Events	***Name:*** Counter overflow (time achieved).

3.5.9. CANHandler

Icon	
Description	CANHandler component encapsulates the CAN bus communications from a configured channel. This component is required if the user wants to communicate with CAN bus.
Read	Reading it return the bus state.
Write	Not possible.
Attributes	***Channel:*** CAN channel to be opened by the component. One of the available channels must selected. If Auto is selected a selection dialog will appear before the first start after the configuration load. ***Bauds:*** Channel baud rate. ***CanDB***: File specifying the CAN data base, in format .dbc from vector, with the description of all frames and all signals which could be send through a Can Bus. ***FrameName***: Return the id of the frame named '*FrameName*' within the Can DB. Frames with standard id are in the range 0x000 to 0x7FF, and frames with extended id are in the range 0x00000000X al 0x1FFFFFFFX . ***SignalName***: Return or let modify the value of the signal named 'SignalName' within the Can DB. ***SignalName.Raw***: Return or let modify the value of the signal named 'SignalName' within the Can DB. ***SignalName.Phy***: Return or let modify the physical value of the signal named 'SignalName' within the Can DB. The physical value is defined as (raw_value * factor) + offset ***SignalName.NameCte***: Return the value of a constant named '*NameCte*' defined within the Can Db for the signal

	named '*NameSignal*'.
Methods	***Tx(id)***: Method to initiate the transmission of a frame with identifier *id.* The frame must be defined within the associated Can DB to allow finding it length. ***Tx(id, len)***: Method to initiate the transmission of the frame with identifier *id* and length *len bytes.* ***TxRemote(id)***: Method to initiate the transmission of a remote frame with identifier *id.* The frame must be defined within the associated Can DB to allow finding it length. ***TxRemote(id, len)***: Method to initiate the transmission of a remote frame with identifier *id* and length *len bytes.* ***Byte*(id, pos)**: Whether it is used on the left side of an assignment statement it lets modify the value of the byte *pos* of the frame with identifier *id otherwise it* is is used on the right side of an assignment statement it returns the value of of the byte *pos* of the frame with identifier *id*. ***Bits*(id, lsb, msb)**: Whether it is used on the left side of an assignment statement it lets modify the value of the signal codified between the bits *lsb* and *msb* of a frame with identifier *id*, otherwise it is used on the right side of an assignment statement it returns the value of the signal codified between the bits *lsb* and *msb* of a frame with identifier *id*. If the lsb (least significant bit) is lower than the msb (most significant bit) the signal codified is intended to be in intel format (little endian) otherwise it is intended to be in motorola format (big endian). ***ReStart***(): Restart of CAN communications.
Events	***Name***: Notification of a bus state change. ***Nombre.Id***: Notification of the reception of a new frame with identifier *id*. ***Nombre.NombreTrama***: Notification of the reception of a new frame named *'NombreTrama'* within the associated CanDB.

3.5.10. LINHandler

Icon	
Description	LINHandler component encapsulates the LIN bus communications from a configured channel. This component is required if the user wants to communicate with a LIN bus.
Read	Reading it return the bus state.
Write	Not possible.
Attributes	***Channel:*** CAN channel to be opened by the component. One of the available channels must selected. If Auto is selected a selection dialog will appear before the first start after the configuration load.

	Bauds: Channel baud rate. ***Master:*** Indicates whether the handler will act as a master or as a slave. ***Version:*** Indicates the version of the LIN protocol (LIN_VERSION_1_3 o LIN_VERSION_2_0). ***LinDB***: File specifying the LIN data base, in format .DBC (Vector format) or .LDF (LIN Description File), with the description of all frames and all signals which could be sent through the bus. After loading a .LDF file the values of the attributes Baudrate and Version will be overridden by the values defines in the loaded file. ***FrameName***: Return the id of the frame named '*FrameName*' within the LIN DB. Frames id are in the range 0x00 to 0x3F. ***SignalName***: Return or let modify the value of the signal named 'SignalName' within the LIN DB. ***SignalName.Raw***: Return or let modify the value of the signal named 'SignalName' within the LIN DB. ***SignalName.Phy***: Return or let modify the physical value of the signal named 'SignalName' within the LIN DB. The physical value is defined as (raw_value * factor) + offset ***SignalName.NameCte****: Return the value of a constant named 'NameCte' defined within the LIN Db for the signal named 'NameSignal'.*
Methods	***TxOn(id)***: Activation of the transmission of data bytes corresponding to frame with identification *id to be transmitted as soon as the master requests it. The definition of the frame with identification id in the configured LIN DB is mandatory to use this method with only the id parameter.* ***TxOn(id, dlc, chksum)***: Activation of the transmission of data bytes corresponding to frame with identification *id, data length dlc, and checksum type chksum to be transmitted as soon as the master requests it.* Possible values for the chksum parameter are: 0 : Classic checksum according LIN version 1.3 1 : Enhanced checksum according LIN version 2.0 ***TxOff(id):*** Deactivation of the transmission of data bytes corresponding to frame with identification *id when the master requests it.* ***Request(id)***: *In master mode requests the node owner of the frame with identification id the transmission of the data bytes associated with such frame.* *In slave mode the request is ignored.* *The definition of the frame with identification id in a LIN DB is mandatory to use this method with only the id parameter.* ***Request(id, chksum)***:*In master mode requests the node owner of the frame with identification id the transmission of the data bytes associated with such frame, expecting a checksum of type chksum.* *In slave mode the request is ignored.*

	Possible values for the chksum paramenter are: 0 : Classic checksum according LIN version 1.3 1 : Enhanced checksum according LIN version 2.0 ***NotifType(id):*** *Query about the type of the last receive notification for the frame with identification id.* *Possible values are:* *LIN_FRAME_OK: Correct frame.* *LIN_NO_ANSWER: No answered frame .* *LIN_CHKSUM_ERR: Frame with checksum error.* *LIN_ERR_FRAME: Frame with generic error.* *LIN_FRAME_UNKNOWN: Frame with unknown error.* ***Sleep( ):*** *Changes the node state from active mode to sleep mode.* ***WakeUp( )***: *Transmits a wake up notify frame to notify every node in the bus to change from sleep mode to active mode.* ***Byte*(id, pos)**: Whether it is used on the left side of an assignment statement it lets modify the value of the byte *pos* of the frame with identifier *id otherwise it* is is used on the right side of an assignment statement it returns the value of of the byte *pos* of the frame with identifier *id*. ***Bits*(id, lsb, msb)**: Whether it is used on the left side of an assignment statement it lets modify the value of the signal codified between the bits *lsb* and *msb* of a frame with identifier *id*, otherwise it is used on the right side of an assignment statement it returns the value of the signal codified between the bits *lsb* and *msb* of a frame with identifier *id*. If the lsb (least significant bit) is lower than the msb (most significant bit) the signal codified is intended to be in intel format (little endian) otherwise it is intended to be in motorola format (big endian).
Events	***Nombre.Id***: Notification of the reception of a new frame with identifier *id*. ***Nombre.FrameName***: Notification of the reception of a new frame named *'FrameName'* within the associated LinDB.

3.5.11. TraceLog

Icono	
Descripción	This component implements a log of thre different frames transmited through the differnt CAN or LIN busses.
Lectura	Not possible.
Escritura	Not possible.
Atributos	***File***: Name of the log file. ***Handler***: Handler associated. 'ALL HANDLERS' means that the trace will show the information of all the CanHandlers and LinHandlers in the application. ***AutoStart***: Flag indicative whether the logging has to be

	started with in conjunction with the CANica script start. ***AutoRename***: Flag indicative whether the log file name must be automatically renamed if there exits one with the same name or must be overwritten.
Métodos	***StartLog():*** Logging activation ***StopLog():*** Logging deactivation.
Eventos	None.

3.5.12. ISO15765FunHandler

Icon	
Description	This component implements a ISO15765 communication channel with functional addressing (Broadcast).
Read	Reading it returns the state of the bus. The list of possible values: IDLE: The channel is ready for transmission. BUSY: The channel is busy due an active transmission. TX_NOK: The channel is idle after an error occurred during the last transmission.
Write	Not possible.
Attributes	***CANHandler***: Name of the CAN handler used to create the Iso15765 channel with function addressing. ***IdReq***: CAN frame identification used to manage the transmission of the Iso15765 channel with functional addressing. ***P2Max***: Maximum Iso15765 channel response wait time after a request before a timeout error is issued. ***P2ExtMax***: Maximum Iso15765 channel response wait time after a request answered with an extra time required negative response before a timeout error is issued. ***P3Min***: Minimum time the Iso15765 channel remains busy after a request with no answer required is transmitted. ***S3Max***: Maximum allowed time without any request after which whether the AutoTP functionality is enables the channel will automatically send a TesterPreset with no answer required request to keep the diagnostic session active. ***AutoTP***: Flag indicative whether the channel should automatically send TesterPresent requests to keep a diagnostic session active.
Methods	***Tx(len, wait_resp)***: Transmission through the Iso15765 channel of a request of length len and answer required according the parameter wait_resp. Allowed values for the parameter wait_resp: 0: No response is expected. 1: Responses are expected. ***Byte*(pos)**: This method is only allowed as the left part of an assignment and it is used to change the value of the byte pos of the frame.

Events	***Name***: Channel status change.

3.5.13. ISO15765PhyHandler

Icon	
Description	This component implements a ISO15765 communication channel with physical addressing (Point to point).
Read	Reading it returns the state of the bus. The list of possible values: IDLE: The channel is ready for transmission. BUSY: The channel is busy due an active transmission. TX_NOK: The channel is idle after an error occurred during the last transmission. RX_OK: The channel is idle after the correct reception of an answer to a service request. RX_NOK: The channel is idle after an error occurred during the last reception. TIME_OUT: The channel is idle after a timeout error occurred due the expiration of the maximum response wait time to a service request.
Write	Not possible.
Attributes	***CANHandler***: Name of the CAN handler used to create the Iso15765 channel with physical addressing. ***ISO15765Fun:*** Name of the Iso15765 channel with functional addressing handler which will be answered when required using this Iso15765 channel with physical address. ***IdReq***: CAN frame identification used to manage the transmission of the Iso15765 channel with physical addressing. ***IdResp***: CAN frame identification used to manage the reception of the Iso15765 channel with physical addressing. ***P2Max***: Maximum Iso15765 channel response wait time after a request before a timeout error is issued. ***P2ExtMax***: Maximum Iso15765 channel response wait time after a request answered with an extra time required negative response before a timeout error is issued. ***P3Min***: Minimum time the Iso15765 channel remains busy after a request with no answer required is transmitted. ***S3Max***: Maximum allowed time without any request after which whether the AutoTP functionality is enables the channel will automatically send a TesterPreset with no answer required request to keep the diagnostic session active. ***AutoTP:*** Flag indicative whether the channel should automatically send TesterPresent requests to keep a diagnostic session active.
Methods	***Tx(len, wait_resp)***: Transmission through the Iso15765 channel of a request of length len and answer required according the parameter wait_resp.

	Allowed values for the parameter wait_resp: 0: No response is expected. 1: Responses are expected. ***Byte*(pos)**: This method is only allowed as the left part of an assignment and it is used to change the value of the byte pos of the frame. ***Len()***: length of the frame received in the last reception.
Events	***Name***: Channel status change.

3.5.14. Sound

Icono	
Descripción	This component allows playing a sound.
Lectura	Not possible.
Escritura	Determines the sound to be played at this moment.
Parámetros	None.
Métodos	None.
Eventos	None.

4.Using CANica

4.1. Simple programming

CANica is intended to be used with a minimum effort. As such, you can just type the code you need and it will be running continuously in an endless loop.

This method is simple, but it may not be very well-structured in complex applications

Example1:

```
if (b1==1) {
      beeper = 1;
      b1 = 0;
}
```

Example 2:

```
v3 = ((v3_ant_ant * 60) + (v3_ant * 10) + (v1 *
30)) / 100;
v3_ant_ant = v3_ant;
v3_ant = v3;

sleep(10);
```

4.2. Event-driven programming

When you need to respond to a lot of different events coming from the CAN bus, components or time, event-driven paradigm is very useful.
You can encapsulate all code in a CANica application in a series of “on event ()” statements, complemented with a number of functions.

This type of programming has several advantages:

- it’s easier to control the real-time behavior of the application
- it’s more readable and well-structured
- windows programmers are used to this kind of programming

Example 1:

Imagine you have to do something when the application starts, send a message in a periodic basis and each time you receive a certain frame you have to update some graphical elements. Additionally, when you press a button you have to send an eventual frame.

```
on event (start) {
      // start-up actions
}

on event (h1.can201h) {
      // process frame 201h
      process_201();
}

on event (h1.message_1) {
      // process message message_1
      process_can_message_1();
}

on event (timer1) {
      // fill-in frame 300h content
      fill_300();
      // send frame 300h
      h1.tx(can300h, 8);
}

on event (button_1) {
      // send eventual frame
      h1.tx(can203h,4);
}

function process_201()
{
      h1.byte(can201h,1) = edit1;
      // ...
}
```

```
function process_can_mesage_1()
{
      if (h1.signal1 == "position_1"){
            h1.signal1 = "p1_achieved";
            h1.signal2.phy = 0x12;
            h1.tx(h1.message_2);
      }
}
```

APPENDIX 1: Acknowledgments

CANica 3 has been developed using the following free software packages. These packages could be freely downloaded and information referent to their developers consulted from their own web pages.

Package	Source
Cintilla	http://www.scintilla.org
FreeImage	http://freeimage.sourceforge.net
Nsis	http://nsis.sourceforge.net

APPENDIX 2: End User License

USE LICENCE OF THE SOFTWARE CANIca

FICO- TRIAD , S.A. Y FICOSA INTERNATIONAL , S.A.
(Hereafter referred to as "FICOSA"), address Gran via Carles III, 98, 5 ,08028
Barcelona,
España.

IMPORTANT NOTICE

PLEASE READ THIS DOCUMENT BEFORE USING THE SOFTWARE: THIS
LICENCE OF THE SOFTWARE "CANIca" of FICOSA (HEREAFTER "LICENCE") IS
AN AGREEMENT SETTING THE RULES FOR THE USE OF THE SOFTWARE
"CANica" of FICOSA WHICH MAY BE FOUND IN THE PURCHASED SUPPORT,
INCLUDING INFORMATICS SOFTWARE AND RELATED DOCUMENTATION
(SOFTWARE). THE USE OF THE SOFTWARE WILL BE INTERPRETED AS A
UNEQUIVOCAL MATTER OF FACT OF YOU HAVING ACCEPTED THE TERMS
AND CONDITIONS OF THIS LICENCE. IF YOU HAVE ACCESS TO OUR
SOFTWARE THROUGH OUR WEBSITE OR THROUGH ANY OTHER
AUTHORIZED INTERNET SITE, THE FACT OF DOWNLOADING, INSTALLING,
COPYING OR USING THE SOFTWARE WILL BE INTERPRETED AS YOU HAVE
ACCEPTED THE LICENCE CONDITIONS. IF YOU DON'T ACCEPT THE
CONDITIONS OF THIS LICENCE DON'T USE OR DOWNLOAD THE SOFTWARE.

1.-General. Proprietary Rights of FICOSA.

FICOSA grants (does not sell) to User one unique non- exclusive and not transferable licence of use on the SOFTWARE CANica, the documentation and any other kind of information provided along with this Licence whether on disk, only lecture memory of a computer (ROM), any other support or in any other way ("SOFTWARE"), subject to the terms and conditions provided in this Licence. FICOSA reserves any and all other rights not granted expressly hereby to User.

This Licence does not imply the sale to User of the SOFTWARE, nor of the related documentation. All Intellectual and Industrial property rights on the SOFTWARE (including images, pictures, animation, video, music, text and other documents composing the SOFTWARE) belong exclusively to FICOSA. The SOFTWARE is protected by the Intellectual property laws and International treaties which protect software and computer programs. The infringement of the terms and of this Licence may be prosecuted by means of civil, administrative and/or criminal actions available in each jurisdiction.

2.- Evaluation Version

In the purchased SOFTWARE paquet or on the Web www.canica.biz, the User may find an Evaluation Version, which User may use, without charge, for the sole purposes of evaluation or trial without a commercial intent, during a period of 30 days after the first installation of the Evaluation Version. If User uses this software after the evaluation period, the Licence payment will be needed, according to the terms established in the Register Formulary. After receipt of the payment of the Licence, a registered User Password will be sent, being the software registered and User being authorised to use the software without time limitation subject to the terms and conditions of the present Licence. Otherwise, the User undertakes to destroy all copies and related materials of the Evaluation Version and not to use the same.

The use of the Evaluation Version after the evauation or trial period, without being registered, constitutes an infringement of the terms of the present Licence as well as Intellectual property laws and the International treaties.

User is hereby authorised, during the evaluation or trial period, to copy and distribute without requesting any charge the Evaluation Version, of any number of copies User may consider needed and by any means, without making any modification thereto, as well as of the related documentation of said Evaluation Version.

3. User obligations

A) The user expressly assumes the followings obligations:

a) To adopt all reasonable measures in order to protect the SOFTWARE against theft or non-authorized use according to the terms of this Contract.

b) To use and install the SOFTWARE only in one computer or equipment. The licence cannot be shared nor used at the same time by several computers, neither through an internal network or by any other mecanism or procedure enabling its use by more than on User at the same time.

B) The user undertakes not to realize any of the followings acts:

a) To modify, transform, dismantle, decompilate, separate or by any other means manipulate in reverse the SOFTWARE.
b) To copy o reproduce the SOFTWARE, except for one copy for security back up or archive purposes, as long as these contain all the information concerning the Intellectual property and Copyright or any other data concerning FICOSA's property as contained in the original.
c) To assign, sell, rent, lease, or by any other way distribute or enable the use of all or part, permanently or temporarily, of the SOFTWARE to any other person or legal entity.

4.- Expiration of de licence.

This LICENCE will be in force until its validity expires. The rights of use conferred by this Licence will expire automatically, without previous notice of FICOSA, in case User does not comply with any of the terms and conditions of this Licence. As soon as the Licence expires, the user must cease using the SOFTWARE and destroy any complete or partial copy of it.

5.- Limited Licence on the supports.

FICOSA guarantees to the best of its knowledge and belief that the support in which the SOFTWARE is recorded has no material and/or manufacture defects, in circumstances of normal use and during a time limit of (90) days from the date of acquisition of the Licence. The User has only the right to obtain a substitute of the SOFTWARE of FICOSA, which shall be returned to FICOSA or an authorised distributor along with the purchasing ticket. This guarantee is limited to ninety (90) days after the date of acquisition of the Licence. The limited guarantee indicated in this
section is the only one offered to you and it replaces any other, belonging to any other documents or paquet related to the SOFTWARE.

6.- Disclaimer of Warranties

User recognizes and accepts expressly that FICOSA cannot control under which conditions the User uses the SOFTWARE. Thus the User uses the SOFTWARE on his own risks, assuming totally the risk of the satisfactory quality, performance, accuracy and effort, except for the agreed in the limited guarantee over the supports above-indicated and as far as permitted by the applicable law.

The SOFTWARE is provided on an "AS IS" basis, with all its potential defects and without guarantees of any type. FICOSA excludes any warranties and/or conditions, expressed, implicit or legal, including but not limited to, warranties and /or implicit conditions of marketability, satisfactory quality, suitability for a specific purpose, accuracy, enjoyment and of no infringement of third parties rights, in relation with the SOFTWARE. FICOSA does not warrant, without prejudice to the use and accuracy of User of the SOFTWARE, that the SOFTWARE functions will satisfies its needs, that the SOFTWARE works without interruptions or errors nor that SOFTWARE defects will be corrected.

7- Limitation of Liability.

FICOSA has no control on the conditions under which the User uses the SOFTWARE, therefore FICOSA cannot guarantee and does not guarantee the performance or the results obtained o derived from the use of the SOFTWARE, neither will FICOSA be responsible, unless required by applicable law, for property and/or personal damages or for loss of profit or consequential damages, nor for general damages, direct or indirect, including but not limited to, damages for loss of benefit or data, the interruption of business activity or wahtever kind of damages and commercial loss, property or personal, caused by or related to the use or bad use of the SOFTWARE of FICOSA, whatever the cause of this damages is, independently of the civil liability theory (breach of contract, illicit act or other) and even if FICOSA had been informed about possibility of such damages.

8. Applicable law and independence of stipulations.

This Licence will be regulated and interpreted according to the Spanish laws. If for any reasons some or part of this Licence provisions are declared ineligible or ineffective by
a competent Court, the rest of the present Licence will continue to be in force and to

produce its effects.

9. Whole contract.

This licence constitutes the whole agreement between FICOSA and the User concerning the use of the SOFTWARE and it replaces all previous or current agreements related to its object. This licence may only be modified with the written agreement signed by FICOSA.