Version 1.0

WARNING – SERIALIZED COPY

The software xConsole® is copyrighted and its usage is conditioned upon the acceptance by the user of the licence contract, which clearly states when and how the software can be used. **Under no circumstances the software should be distributed or transmitted to a third party in violation of the licence contract. Since every copy of the product is serialized, SZP is always able to determine the exact origin of an unauthorized copy thus individuating the original licensee, who will be prosecuted to the full extent for the violation of copyright and of the licence contract.**

Summary

Introduction

xConsole® is an Active-X control that allows to quickly develop console mode applications using common programming languages such as Visual Basic™, Delphi™, Visual C++™ and others.

Console applications created by using xConsole® are full 32 bit programs, which exploit all the features of the Windows™ operating system and of the languages which host the control.

xConsole® was written with the aim of creating a simple and flexible tool for easily handling string input, option selection, menus, etc.

Console applications are still useful, even when using modern graphical operating systems – sometimes text consoles are preferred for writing system utilities to be executed at the command line; moreover, text mode is the only way to go when applications are to be run by a Telnet Server (most RF portable terminals are loaded with a VT or Ansi Telnet client).

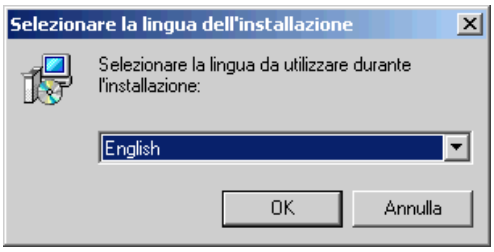
The advantages offered by xConsole® are many:

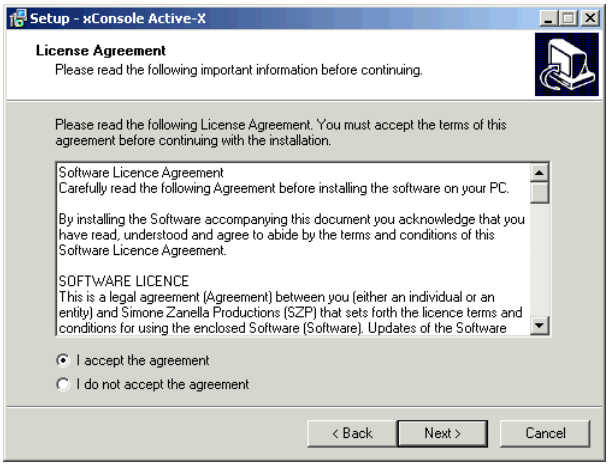
1. you can write console applications even when using languages which do not support this mode (e.g. Visual Basic™) or which have very basic functionalities (e.g. Deplhi™);
2. you can fully customize data input, thanks to a flexible event system;
3. sophisticated string input functions are available, with features ranging from masks to regular expressions (simple and extended), validation for date and time, integer and floating point numbers, password fields, scrolling, etc.
4. you get a full set of useful routines: single or multiple choice option lists, menus, multiline text fields, routines for drawing lines, boxes and shadows, functions to save and restore screen contents, to print messages (optionally accompanied by confirmation buttons), etc.

Even though Windows™ APIs allow to create text consoles, handling input and output, the highly optimized and fully tested functionalities offered by xConsole® saves you a lot of time and let you create faster, better working, more readable and stable applications.

Installation

To install the package, insert the cd-rom in the drive of your PC, which must be running a Windows™ operating system (95, 98, ME, NT, 2000, XP or better). Start Windows Explorer™, select the drive letter corresponding to the drive and run the program SETUP.EXE that you will find in the root directory.





Select your language and press Ok; read the licence, select “I agree with the terms of the licence agreement” and press Next. Continue pressing Next (in the following forms) and wait until the program is installed; at the end, press “Finish”.

The Active-X control will finally be registered and become available in your development environment.

Using xConsole® under Visual Basic™

xConsole® is an Active-X control; to use it, you must first configure your development environment.

In Visual Basic™, you can do it by adding the control to the list of components (under Project menu); at this point, the xConsole® icon will appear in the toolbox. You can now select the control and drop it on a form, just like any other Visual Basic™ control. Since the application you are going to develop will use a text console, there will be a single form, having the *Visible* attribute set to *False*.

You will write your procedures and functions inside a module, referring the control inserted in the form; text mode does not depend on events: it follows a linear flow, so you need a starting point to begin your program (using *Form_load* is not reccomended).

The first method you must invoke is InitConsole (*True*), which creates the console used by the following methods and resets to default all property values.

If the application you are developing is a command line utility, it could be useful to save screen size and contents (by using MaxCol, MaxRow and the method ScreenSave), to restore everything as it was upon exit.

You can set the colors according to your personal taste, or to the limitations of the remote portable terminal where the application will be run. A simple but effective test to determine if the program is being run inside Visual Basic™ IDE or is compiled are the following instructions:

```
Err.Clear
On Error Resume Next
Debug.Print 1 / 0
If Err.Number <> 0 Then
  ' Inside IDE
Else
  ' Compiled application
End If
```

You might need to Resize the screen size to match the maximum screen size of the remote terminals.

At this point you can set length, justification, frame and shadow type and start invoking the methods to draw user interface.
We first reccomend to write down the various screens of your application by using a text editor (possibly one which displays current cursor position: row and column), so that you have a reference when writing your program.

All the events for the control are fired inside the main form; you can get a list by double clicking on the xConsole® control.

Before shutting down your application, it is reccomended to invoke ShutDownConsole to destroy the text window (but ONLY inside the IDE).

When your application is compiled, it is necessary to change its type: Visual Basic™ only creates graphical applications. To correctly execute the compiled program you need to tell to the operating system that your program is a console application.

You can do it in two ways:
a) using EditBin, which is installed by Visual Studio™;
b) using the command line utility ConsoleMode, which is freely shipped with xConsole® Active-X.

This is how EditBin should be invoked:

```
editbin /SUBSYSTEM:CONSOLE program.exe
```

ConsoleMode is even simpler:

```
consolemode program.exe
```

In both cases, at the end your application will be ready to be run inside a console or a Telnet Server.

Check out the “CodQt” example for additional details and suggestions.

Using xConsole® under Visual C++™

xConsole® is an Active-X control; to use it, you must first configure your development environment.

Under Visual C++™, follow these steps:
1. create a new project and choose “MFC AppWizard (exe)”; fill in the project name and press OK;
2. at Step 1, choose "Dialog based" and press Next;
3. at Step 2, check only “ActiveX Controls” and “Automation” (optionally check “Windows Sockets” if required) and press Next;
4. at Step 4, choose if you want remarks and how MFC library will be linked and press Next;
5. press Finish to generate the support files.

At this point, open the dialog window and select:

Project > Add to Project > Components and Controls

From "Registered ActiveX Controls", choose XCONSOLE Control and press Insert; confirm with Ok.
Keep the class name (CXCONSOLE) and change header and implementation file names to:

```
XCONSOL1.h
XCONSOL1.cpp
```

Confirm with Ok; at the end, close the dialog.
On the toolbox the xConsole™ icon will appear; select it and dropt it on the dialog window.
Set the property *Visible* of the dialog window to *False* (invisible window).

From the menu, choose Edit > ClassWizard; click on the tab Member Variables; in the class field select the dialog class name; under Control_ID choose the one belonging to the xConsole™ control (default: IDC_XCONSOLECTRL1).
Press Add Variable and set variable name to `m_xc`; press Ok and close ClassWizard.

Open the source file (cpp) for the dialog window and look for the function OnInitDialog; inside its body, after the line:

```
// TODO: Add extra initialization here
```

insert the following text:

```
m_xc.InitConsole(TRUE);
```

Immediately after, invoke the function which represents the entry point of your program, using the variable `m_xc` which will be passed as a pointer to a CXCONSOLE object.

Upon return from this function (i.e. end of program), insert a call to PostMessage to close the dialog (which is useful only for hosting the xConsole® control):

```
[dialog name]::PostMessage(WM_CLOSE, 0, 0);
```

Obviously, `[dialog name]` should be replaced by the name of your dialog (in the example, CXcdemoDlg).

Do not forget to remove any compilation flags requesting double-byte character strings: xConsole® only supports single byte strings.

If the application you are developing is a command line utility, it could be useful to save screen size and contents (by using MaxCol, MaxRow and the method ScreenSave), to restore everything as it was upon exit.

You can set the colors according to your personal taste, or to the limitations of the remote portable terminal where the application will be run.
You might need to Resize the screen size to match the maximum screen size of the remote terminals.

At this point you can set length, justification, frame and shadow type and start invoking the methods to draw user interface.
We first reccomend to write down the various screens of your application by using a text editor (possibly one which displays current cursor position: row and column), so that you have a reference when writing your program.

To handle events fired by the control, it is necessary to write event sinks by following these steps:
1. on the View menu, click ClassWizard;
2. click the Message Maps tab;
3. in the Class name box, select the dialog box class that contains the ActiveX control;
4. in the Object IDs box, select the control ID of the embedded ActiveX control (e.g. IDX_XCONSOLECTRL1). The Messages box displays a list of events that can be fired by the embedded ActiveX control. Any member function shown in bold already has handler functions assigned to it;
5. select the message you want the application to handle; press “Add Function” to add a handler, or “Edit Code” to jump to the event handler code in the implementation (.CPP) file.

Before terminating your application, it is reccomended to invoke the method ShutDownConsole to destroy the console you created.

When your application is compiled, it is necessary to change its type: to correctly execute the compiled program you need to tell to the operating system that your program is a console application.

You can do it in two ways:
c) using EditBin, which is installed by Visual Studio™;
d) using the command line utility ConsoleMode, which is freely shipped with xConsole® Active-X.

This is how EditBin should be invoked:

```
editbin /SUBSYSTEM:CONSOLE program.exe
```

ConsoleMode is even simpler:

```
consolemode program.exe
```

In both cases, at the end your application will be ready to be run inside a console or a Telnet Server.

Check out the “xcdemo” example for additional details and suggestions.

xConsole® methods

Below you will find a short description of all the methods supported by the xConsole® control and the most relevant interactions between them (emphasized by a common prefix).

The methods are printed in **BLUE**, the properties in **RED**, the events in **GREEN**.

Constants are always expressed as mnemonic identifiers, whose values can be looked up in the module XCONSOLE.BAS and in the header file XCONSOLE.H.

You will find two syntaxes: the blue one refers to Visual Basic™, the gray one refers to Visual C++™; keep in mind the following type conversions:

Visual Basic™ type	Visual C++™ type
Boolean	BOOL
Integer	short or short * (when passed by reference)
Long	long or long * (when passed by reference)
String	LPCTSTR (parameter in methods) CString (property value) BSTR (value returned by a method)

Note:

1) All the methods having X and Y coordinates in their parameters (both implicit or explicit) adds to them the values of OffsetX and OffsetY, so you can quickly move your masks to any place on the screen without changing a single coordinate.

2) For performance reasons, the xConsole® control does minimum tests on the parameters with which its methods are invoked; take care not to specify coordinates outside the screen area.

Methods (alphabetical list)

AboutBox ()
void AboutBox()

Opens a graphical dialog window displaying information about control version and copyright. No value is returned. This is the only method which produces graphical output.

Alert (*ByVal Tag as Long*) *as Boolean*
BOOL Alert(long Tag);

Opens a box containing text and buttons; it returns True if the user selected a button, False if he pressed Esc. Only a button at a time is displayed on screen. The behaviour of the method is influenced by the following properties:

AlertText *as String*	Text to be printed inside the box
AlertButtons *as String*	Text to be displayed inside the buttons; the string must have the following format: “button 1[#button 2..]”, i.e. you must use the character “#” to separate one button label from the following

AlertCurrentButton *as Integer*	Number of the default button (if invalid, the first button becomes the default); this property is updated at the end of the selection, even if the user presses Esc
AlertBackColor *as Integer* AlertForeColor *as Integer*	Background and foreground colors used for printing message text
AlertButtonBackColor *as Integer* AlertButtonForeColor *as Integer*	Background and foreground colors used for printing buttons
AlertFrameForeColor *as Integer*	Background and foreground colors used for printing the frame (the buttons have AlertButtonForeColor as the frame color)
Frame *as Boolean* Frame3D *as Boolean* FrameChars *as String* ShadowMode *as Integer*	Frame status, type and characters; shadow type (none, right, left)

The parameter *Tag* determines how the method reacts to the introduction of data by the user; if zero, the default behaviour is the following:

cursor keys	Change the button displayed, allowing to select the answer
Enter, space	Accept the current selection
Esc	Exit without selection

If *Tag* is not zero, whenever a key is pressed the following event is fired:

AlertKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)

where:

KeyAscii *as Integer*	Key pressed by the user; can be modified to fake a different key was pressed; set to 0 to discard it
Action *as String*	Determine the action requested in response; can be updated with one of the values specified below
Tag *as Integer*	User chosen identification number for this Alert

The value of *Tag* can be used to discriminate which Alert is active, to adopt different behaviours according to the circumstances.

The possible values for *Action* are listed below:

ALERT_ACCEPT	Process the key as usual
ALERT_DISCARD	Ignore the key (same as setting *KeyAscii* to 0 and *Action* to ALERT_ACCEPT)
ALERT_SELECT	Select the button AlertCurrentButton and remove the box
ALERT_SELECTNR	Select the button AlertCurrentButton and return leaving the box on screen (no restore)
ALERT_ABORT	Abort and return removing the box
ALERT_ABORTNR	Abort and return leaving the box on screen (no restore)
ALERT_NEXT	Display next button
ALERT_PREVIOUS	Display previous button

ALERT_FIRST	Display first button
ALERT_LAST	Display last button

Attribute () *as Integer*
short Attribute();
AttributeXY (*ByVal X as Integer, ByVal Y as Integer*) *as Integer*
short AttributeXY(short X, short Y);

Returns the video attribute at current or specified coordinates. The attribute combines foreground and background colors; you can obtain the two separate colours by applying the method AttributeSplit to the result.

AttributeJoin (*ByVal ForegroundColor as Integer, ByVal BackgroundColor as Integer*) *as Integer*
short AttributeJoin(short ForegroundColor, short BackgroundColor);

Returns the video attribute corresponding to the combination of the specified foreground and background colours; it is the opposite of the method AttributeSplit.

AttributeSplit (*ByVal Color as Integer, ByRef ForegroundColor as Integer, ByRef BackgroundColor as Integer*)
void AttributeSplit(short Color, short* ForegroundColor, short* BackgroundColor);

Splits the video attribute *Color* into the corresponding foreground and background color; it is the opposite of the method AttributeJoin.

Box (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer*)
void Box(short Left, short Top, short Right, short Bottom);

Draws a frame from (*Left, Top*) to (*Right, Bottom*); frame appearance depends on the following properties:

Frame3D *as Boolean*	Draw the frame with a pseudo-3D appearance (two sides are darker than the others)
FrameChars *as String*	8-byte string which represents the characters to be used for drawing the frame (clockwise, starting from upper left). By default after InitConsole you can draw single line frames (FRAME_SINGLE) by using semi-graphical character. You can set the value to other constants (FRAME_DOUBLE, FRAME_SNGDOU, FRAME_DOUSNG, FRAME_DOTS) for different appearances.
FrameBackColor *as Integer* FrameForeColor *as Integer*	Background and foreground colours to be used for drawing the frame.
Pattern *as Integer*	ASCII code of the character used to fill the frame (default is 32 = space).
ShadowMode *as Integer*	Shadow type; possible values are: NO_SHADOW = no shadow SHADOW_LEFT = shadow to the left SHADOW_RIGHT = shadow to the right

ClearArea (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer, ByVal Color as Integer, ByVal Pattern as Integer*)
void ClearArea(short Left, short Top, short Right, short Bottom, short Color, short Pattern);

Clears the area from (*Left, Top*) to (*Right, Bottom*), using the attribute *Color* and the character corresponding to the ASCII code *Pattern*.

Cls ()
void Cls();

Clears the screen, using the current colours and fill pattern.

ColorizeArea (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer, ByVal ForeColor as Integer, ByVal BackColor as Integer*)
void ColorizeArea(short Left, short Top, short Right, short Bottom, short ForeColor, short BackColor);

Replaces the foreground and background colors in the delimited area with those specified.

GetMaxColRow ()
void GetMaxColRow();

Sets MaxCol and MaxRow to the number of columns and rows that the screen holds. Usually, it is not necessary to invoke this method, because both InitConsole and Resize automatically update these properties.

GetXY ()
void GetXY();

Sets X and Y to the current cursor position.

GotoXY (*ByVal X as Integer, ByVal Y as Integer*)
void GotoXY(short X, short Y);

Moves the cursor to the specified coordinates and update the properties X and Y; it is possible to achieve the same result by separately setting the two properties.

HPrint (*ByVal Text as String*) *as Integer*
short HPrint(LPCTSTR Text);
HPrintXY (*ByVal X as Integer, ByVal Y as Integer, ByVal Text as String*) *as Integer*
short HPrintXY(short X, short Y, LPCTSTR Text);

Prints *Text* at the current or specified coordinates, enhancing each character prefixed by the symbol "~"; returns the number of lines used (or –1 if justification is impossible). The appearance of the printed text depends on the following properties:

ForegroundColor *as Integer* BackgroundColor *as Integer*	Background and foreground colours used
Justification *as Integer*	Justification; can be set to any of these values: J_NOJUST = No justification J_LEFT = Align to the left J_CENTER = Center text J_RIGHT = Align to the right J_JUST = Full justification
JustificationLength *as Integer*	Justification length (should always be greater than or equal to the length of the text being justified)

Pattern *as Integer*	ASCII code of the character used to fill the string for justification

If *X* = 0, the method does nothing (emulation mode) but calculates and returns the number of lines needed for printing.

HiColor (*ByVal Color as Integer*) *as Integer*
short HiColor(short Color);

Returns the video attribute corresponding to *Color* enhanced. This is the transformation function invoked by HPrint and other functions.

InitConsole (*ByVal UseExisting as Boolean*)
void InitConsole(long UseExisting);

Initializes the control and sets all properties to their default values; it is necessary to invoke this method before any other and before accessing any control property. The parameter *UseExisting* lets you choose if you want to use the console associated with the process (True, default) or creating a new one (False).

InputString (*ByVal Tag as Integer*) *as Boolean*
BOOL InputString(long Tag);
InputStringXY (*ByVal X as Integer, ByVal Y as Integer, ByVal Tag as Integer*) *as Boolean*
BOOL InputStringXY(short X, short Y, long Tag);

String input, with optional validation; the second form allows to specify the starting point for input, the first one uses current cursor coordinates. Returns True if the user confirmed with Enter, False if he aborted with Esc. The behaviour of this method is influenced by the following properties:

InputDefault *as String*	Initial value for the string (it holds the edited value upon return); this property gets updated even if the user presses Esc
InputMaxLength *as Integer*	Maximum accepted length for the string
InputWindowLength *as Integer*	Editing window width (in columns)
InputStartPos *as Integer*	Position inside editing window; this property gets updated even if the user presses Esc
InputWindowOffset *as Integer*	Editing window offset (relative to the beginning of the string); this property gets updated even if the user presses Esc
InsertMode *as Boolean*	Insert mode status: when True, every character typed moves forward the following characters; when False, every character overwrites the existing one; this property gets updated even if the user presses Esc
InputBackColor *as Integer* InputForeColor *as Integer*	Foreground and background colours used
InputPicture *as String*	Input mask to automatically validate what is typed; when empty, no validation takes place. The table below illustrates the possible values.
DateType *as Integer* Epoch *as Integer*	Date and time formats, used for date/time validation.
SilentMode *as Boolean*	If True, no acoustic warning will be played when

	validation fails (the event SoundRequest will be fired anyway)

The characters in InputPicture have the following meaning:

!	convert all alphabetical characters to upper case
*	print "*" in place of any character typed (e.g. password request)
N	accept an integer number
F	accept an integer or decimal number
D	accept a date; the interpretation depends on the properties DateType and Epoch. The first one can take one of the following values: DATE_US = american format (month/day/year) DATE_EUROPE = european format (day/month/year) DATE_JAPAN = japanese format (year/month/day) Epoch determines how years should be interpreted in short dates (where only two digits are used to specify the year); in this case, if the last two digits of the year are below the last two digits in Epoch, the year is considered in the following century, otherwise in the same century; e.g. Epoch = 1970 Year in: 01/01/69 = 2069 Year in: 01/01/70 = 1970 Year in: 01/01/97 = 1997
H	accept a time in the format "hh:mm:ssx", where mm and ss are between 0 and 59, hh is between 1 and 12 (if x is "p" or "a") or between 0 and 23 (if x is a space or is missing); x must be "a", "A", "p", "P" or a space
R*s*	string is validated only if it satisfies *s* (extended regular expression)
T*s*	string is validated only if it satisfies *s* (case unsensitive extended regular expression)
P*s*	string is validated only if it satisfies *s* (regular expression)
O*s*	string is validated only if it satisfies *s* (case unsensitive regular expression)
M*s*	specify a mask for data input; *s* can include the following characters: X = any character N = digit 0-9 O = digit 0-7 H = digit 0-9 or A-H B = digit 0 or 1 A = alphabetical character U = alphabetical character or digit (0-9) other = literal character

For additional information on regular expressions please consult the chapter later in this manual.

The paramater *Tag* determines how the method react to the data being typed by the user; if zero, the default behaviour is the following:

left and right cursor keys	move the cursor inside editing buffer
Enter	confirm what was typed; if validation fails, a sound is played and the user remains in editing mode
Backspace/Canc	delete previous or currenct character
Esc	abort
Ins	switch between insert and overwrite mode (cursor shape and InsertMode value change)
Home	move the cursor to the beginning of the line
End	move the cursor to the end of the line
Other character between 32 and 255	accept the character into the string (if validation rules are satisfied)

If *Tag* is not zero, whenever a key is pressed the following event is fired:

InputKeyPress(*ByRef KeyAscii as Integer*, *ByRef Action as Integer*, *ByVal Tag as Long*)

where:

KeyAscii *as Integer*	Key pressed by the user; can be modified to fake a different key was pressed; set to 0 to discard it
Action *as String*	Determine the action requested in response; can be updated with one of the values specified below
Tag *as Integer*	User chosen identification number for this InputString

The value of *Tag* can be used to discriminate which InputString is active, to adopt different behaviours according to the circumstances.

The possible values for *Action* are listed below:

ALERT_ACCEPT	Process the key as usual
ALERT_DISCARD	Ignore the key (same as setting *KeyAscii* to 0 and *Action* to ALERT_ACCEPT)
ALERT_SELECT	Select the button AlertCurrentButton and remove the box
ALERT_SELECTNR	Select the button AlertCurrentButton and return leaving the box on screen (no restore)
ALERT_ABORT	Abort and return removing the box
ALERT_ABORTNR	Abort and return leaving the box on screen (no restore)
ALERT_NEXT	Display next button
ALERT_PREVIOUS	Display previous button
ALERT_FIRST	Display first button
ALERT_LAST	Display last button

INPUT_ACCEPT	Process the key as usual
INPUT_UPDATE	InputDefault modified; update editing buffer and continue
INPUT_UPDATEANDCONFIRM	InputDefault modified; update editing buffer and accept the new string
INPUT_ABORT	Abort and return

INPUT_CONFIRM	Confirm input
INPUT_DISCARD	Ignore the key
INPUT_LEFT	Move cursor to the left
INPUT_RIGHT	Move cursor to the right
INPUT_HOME	Move cursor to the beginning of the input field
INPUT_END	Move cursor to the end of the input field

KeyHit () *as Long*
long KeyHit();

Returns the code of the next key available in the keyboard buffer, or zero if the buffer is empty. This method returns immediately; the key is not removed from the keyboard buffer. KeyHit takes into account keys stuffed using KeyStuff. Use KeyInput or KeyInputTimed to read the key and remove it from the buffer.

KeyInput () *as Long*
long KeyInput();

Waits for a keypress and returns its code; takes into account keys stuffed using KeyStuff. This method stops program execution until a key becomes available; use KeyHit to determine if a key is available without removing it from keyboard buffer. Use KeyInputTimed if you need a timeout for input.

KeyInputTimed (*ByVal Seconds as Integer*) *as Long*
long KeyInputTimed(short Seconds);

Waits for a keypress (with timeout) and returns its code; takes into account keys stuffed using KeyStuff. This method stops program execution until a key becomes available or the timeout expires; use KeyHit to determine if a key is available without removing it from keyboard buffer. If *Seconds* is zero, this method is functionally the same as KeyInput. Use KeyHit to determine if a key is available without removing it from keyboard buffer.

KeyStuff (*ByVal KeyAscii as Long*)
void KeyStuff(long KeyAscii);

Stuffs the key corresponding to *KeyAscii* into keyboard buffer; all the methods in xConsole® take into account keys stuffed using this method, exactly as if the user had typed them using the keyboard.

LineFromTo (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer*)
void LineFromTo(short Left, short Top, short Right, short Bottom);

Draws a line from (*Left, Top*) to (*Right, Bottom*), using the first character of the property LineCharsHV if the line is horizontal or the second if it is vertical; if LineCharsHV is undefined or too short, the method uses character 2 and 4 of the property FrameChars. The colours used are FrameForeColor and FrameBackColor.
Only horizontal and vertical lines can be drawn using this method.

List (*ByVal Tag as Integer*) *as Boolean*
BOOL List(long Tag);
ListXY (*ByVal X as Integer, ByVal Y as Integer, ByVal Tag as Integer*) *as Boolean*
BOOL ListXY(short sX, short sY, long Tag);

Opens a box on screen filled with a list of options, using the current or specified coordinates; returns True if the user confirmed the selection, False if he aborted pressing Esc. The behaviour of the method is influenced by the following properties:

ListTitle *as String*	title for the list of options (printed on the top frame, centered); visible only if frame is enabled
ListOptions *as String*	text for options; the string must have the following format: "option 1[#option 2..]", i.e. use the character "#" to separate one option from the following
ListCurrentOption *as Integer*	option currently selected; this property is updated upon exiting, even if the user pressed Esc
ListRows *as Integer*	number of visible lines
ListColumns *as Integer*	visible width (if 0, the length of the longest option is used)
ListWindowOffset *as Integer*	offset for printing options (number of characters to skip at the beginning of every option); this property is updated at the end of the selection, even if the user pressed Esc
ListCurrentLine *as Integer*	line where the selected option is displayed (default: 0); updated at the end of the selection
ListMultiSelect *as Boolean*	(dis)allow multiple selections
ListSelection *as Integer*	ASCII code of the character used to show that an option is selected when multiple selections are allowed (default: 16)
ListMap *as String*	empty string (if ListMultiSelect = False), or string where each character is "0" or "1" according to the selection status of each option ("0" = unselected, "1" = selected); this property is updated at the end of the selection, even if the user pressed Esc
TitleBackColor *as Integer* TitleForeColor *as Integer*	background and foreground colours used to print the title
SelectedBackColor *as Integer* SelectedForeColor *as Integer*	background and foreground colours used to print the current option
UnselectedBackColor *as Integer* UnselectedForeColor *as Integer*	background and foreground colours used to print all the options (except the current one); background for the frame
ListFrameForeColor *as Integer*	foreground colour for the frame and for the thumb elevator
Frame *as Boolean* Frame3D *as Boolean* FrameChars *as String* ShadowMode *as Integer*	frame status, type and characters used to draw it; shadow type

The parameter *Tag* determines how the method react to the data being typed by the user; if zero, the default behaviour is the following:

cursor keys	change current option (up/down) or the horizontal offset (left/right)
Enter	select the current option (and return, if ListMultiSelect is False)
Space	select the current option and move to the next

	(only if ListMultiSelect is True)
Esc	abort
Ctrl+Enter	confirm selection (only if ListMultiSelect is True)
Tab	change selection status for all the options (select or deselect all the options)
Home	jump to the first option
End	jump to the last option

If *Tag* is not zero, whenever a key is pressed the following event is fired:

ListKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)

where:

KeyAscii *as Integer*	Key pressed by the user; can be modified to fake a different key was pressed; set to 0 to discard it
Action *as String*	Determine the action requested in response; can be updated with one of the values specified below
Tag *as Integer*	User chosen identification number for this List

The value of *Tag* can be used to discriminate which List is active, to adopt different behaviours according to the circumstances.

The possible values for *Action* are listed below:

LIST_ACCEPT	Process the key as usual
LIST_DISCARD	Ignore the key
LIST_SELECT	Select the option ListCurrentOption (and remove the box, if ListMultiSelect is False)
LIST_SELECTNR	Select the option ListCurrentOption (and leave the box on screen, if ListMultiSelect is False)
LIST_ENDSEL	Only when ListMultiSelect is True: confirm selection and remove box from screen
LIST_ENDSELNR	Only when ListMultiSelect is True: confirm selection and leave the box on screen)
LIST_ABORT	Abort and remove box from screen
LIST_ABORTNR	Abort and leave box on screen
LIST_ENHANCE	Jump to ListCurrentOption
LIST_NEXT	Move to next option
LIST_PREVIOUS	Move to previous option
LIST_FIRST	Jump to first option
LIST_LAST	Jump to last option
LIST_REDRAW	Redraw the list (ListWindowOffset was modified)

Menu (*ByVal Tag as Integer*) *as Boolean*
BOOL Menu(long Tag);
MenuXY (*ByVal X as Integer, ByVal Y as Integer, ByVal Tag as Integer*) *as Boolean*
BOOL MenuXY(short X, short Y, long Tag);

Opens a box on screen filled with a list of options, using the current or specified coordinates; returns True if the user confirmed the selection, False if he aborted pressing Esc. The behaviour of the method is influenced by the following properties:

MenuOptions *as String*	options in the menu; the string must have the following format: "option 1[|description 1]#option 2[|description 2].." i.e. use the character "#" to separate the options and the character "|" to separate the option from its description. If option*x* is empty, description*x* is the character that will be used to fill the separation line; use "|" alone to get an empty line
MenuUnselectable *as String*	string where each character is "0" (selectable option) or "1" (unselectable option); if an option has no matching character in MenuUnselectable then it is selectable
MenuCurrentOption *as Integer*	current option number; this property is updated at the end of the selection, even if the user pressed Esc
SelectedBackColor *as Integer* SelectedForeColor *as Integer*	foreground and background colours for current option
UnselectedBackColor *as Integer* UnselectedForeColor *as Integer*	background and foreground colours used to print all the options (except the current one); background for the frame
UnselectableBackColor *as Integer* UnselectableForeColor *as Integer*	background and foreground colours used to print unselectable options
MenuFrameForeColor *as Integer*	foreground colour for the frame
Frame *as Boolean* Frame3D *as Boolean* FrameChars *as String* ShadowMode *as Integer*	frame status, type and characters used to draw it; shadow type
ScoreboardBackColor *as Integer* ScoreboardForeColor *as Integer*	foreground and background colours used to print descriptions
ScoreboardStatus *as Boolean* ScoreboardX *as Integer* ScoreboardY *as Integer* ScoreboardJustification *as Integer* ScoreboardLength *as Integer*	determine if descriptions are printed, their position and the justification

Descriptions are a brief note that accompany every menu item; they are displayed whenever an option become the current option, using the properties Scoreboard[..].

The parameter *Tag* determines how the method react to the data being typed by the user; if zero, the default behaviour is the following:

cursor up/down	change current option
Enter, right cursor, space	select the current option
Esc	abort
Home, PagUp	Jump to the first option
End, PadDn	jump to the last option

Every character prefixed by "~" (ASCII 126) inside an option appears enhanced onscreen and becomes the key for direct selection of the item.

If *Tag* is not zero, whenever a key is pressed the following event is fired:

MenuKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)

where:

KeyAscii *as Integer*	Key pressed by the user; can be modified to fake a different key was pressed; set to 0 to discard it
Action *as String*	Determine the action requested in response; can be updated with one of the values specified below
Tag *as Integer*	User chosen identification number for this Menu

The value of *Tag* can be used to discriminate which Menu is active, to adopt different behaviours according to the circumstances.

The possible values for *Action* are listed below:

MENU_ACCEPT	process key as usual
MENU_DISCARD	ignore key
MENU_SELECT	select MenuCurrentOption, return True
MENU_SELECTNR	select MenuCurrentOption, return True and leave menu box on screen (no restore)
MENU_ABORT	return False and remove menu box from screen
MENU_ABORTNR	return False and leave menu box on screen
MENU_ENHANCE	jump to MenuCurrentOption
MENU_NEXT	move to the next option
MENU_PREVIOUS	move to the previous option
MENU_FIRST	jump to the first option
MENU_LAST	jump to the last option
MENU_ENABLE	process MenuUnselectable again, changing the "selectable" status of every menu item

OSD (*ByVal Text as String*) *as String*

CString OSD(LPCTSTR Text);

Opens a window in the center of the screen containing the specified text; returns a string which can be used by OSDRestore to restore the underlying video. The behaviour is influenced by the following properties:

AlertBackColor *as Integer* AlertForeColor *as Integer*	background and foreground colours used for printing the text of the message
AlertFrameForeColor *as Integer*	foreground colour for the frame
Frame *as Boolean* Frame3D *as Boolean* FrameChars *as String* ShadowMode *as Integer*	frame status, type and characters used to draw it; shadow type

Every character prefixed by "~" is printed using enhanced colours.

OSDRestore (*ByVal Screen as String*)
void OSDRestore(LPCTSTR OSDBuffer);

Restores the contents of the screen overwritten by a previous call to OSD; *Screen* must have been previously returned by a previous call to OSD.

Resize (*ByVal Width as Integer, ByVal Height as Integer*)
void Resize(short Width, short Height);

Changes the size of the console to those specified (if possible); if successfull, the properties MaxCol and MaxRow become equal to *Width* and *Height*.

ReverseArea (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer*)
void ReverseArea(short Left, short Top, short Right, short Bottom);

Reverses video colours in the area from (*Left*, *Top*) to (*Right*, *Bottom*).

SPrint (*ByVal Text as String*) *as Integer*
short SPrint(LPCTSTR Text);
SPrintXY (*ByVal X as Integer, ByVal Y as Integer, ByVal Text as String*) *as Integer*
short SPrintXY(short X, short Y, LPCTSTR Text);

Prints *Text* at the current or specified coordinates; returns the number of lines used (or –1 if justification is impossible). The appearance of the printed text depends on the following properties:

ForegroundColor *as Integer* BackgroundColor *as Integer*	Background and foreground colours used
Justification *as Integer*	Justification; can be set to any of these values: J_NOJUST = No justification J_LEFT = Align to the left J_CENTER = Center text J_RIGHT = Align to the right J_JUST = Full justification
JustificationLength *as Integer*	Justification length (should always be greater than or equal to the length of the text being justified)
Pattern *as Integer*	ASCII code of the character used to fill the string for justification

If *X* = 0, the method does nothing (emulation mode) but calculates and returns the number of lines needed for printing. See also HPrint.

ScreenClear (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer*)
void ScreenClear(short Left, short Top, short Right, short Bottom);

Clears the area from (*Left*, *Top*) to (*Right*, *Bottom*). The colours used are ForegroundColor and BackgroundColor, the character to fill the area is Pattern.

ScreenRestore (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer, ByVal Screen as String*)

void ScreenRestore(short Left, short Top, short Right, short Bottom, LPCTSTR ScreenBuffer);

Restores the video block *Screen* (obtained with a previous call to ScreenSave) at the specified coordinates. Destination area size must match the source (width and height); coordinates can be different.

ScreenSave (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer*) *as String*

CString ScreenSave(short Left, short Top, short Right, short Bottom);

Returns a string representing the video block (text and attributes) for the area going from (*Left, Top*) to (*Right, Bottom*); this area can later be restore by using ScreenRestore.

ScrollHorizontally (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer, ByVal Columns as Integer*)

void ScrollHorizontally(short Left, short Top, short Right, short Bottom, short Columns);

Scrolls horizontally the specified area; scrolls left if *Columns* is positive, scrolls right otherwise.

ScrollVertically (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer, ByVal Rows as Integer*)

void ScrollVertically(short Left, short Top, short Right, short Bottom, short Rows);

Scrolls vertically the specified area; scrolls up if *Columns* is positive, scrolls down otherwise.

SettingsRestore (*ByVal Settings as String*)

void SettingsRestore(LPCTSTR SavedSettings);

Restores all the properties of the control; *Settings* must have been returned by a previous call to SettingsSave.

SettingsSave () *as String*

CString SettingsSave();

Returns a string which stores all the current values for the properties of the control; in this way, it is possible to make any change (including recursive calls) as long as you restore the original values by using SettingsRestore between calls.

Shadow (*ByVal Left as Integer, ByVal Top as Integer, ByVal Right as Integer, ByVal Bottom as Integer*)

void Shadow(short Left, short Top, short Right, short Bottom);

Paints a shadow for the area from (*Left, Top*) to (*Right, Bottom*); the kind of shadow depends on the property ShadowMode, which can take one of the following values:

NO_SHADOW = no shadow
SHADOW_LEFT = shadow to the left
SHADOW_RIGHT = shadow to the right

ShutdownConsole ()

void ShutDownConsole();

Closes the handle and frees the console allocated by InitConsole; this should be the last method invoked before terminating your program. Usually, it is not necessary to make an explicit call to this method: when your program ends, all the handles belonging to the process (including the console) are automatically closed.

TextBox (*ByVal Tag as Integer*) *as Boolean*

BOOL TextBox(long Tag);

TextBoxXY (*ByVal X as Integer, ByVal Y as Integer, ByVal Tag as Integer*) *as Boolean*

BOOL TextBoxXY(short X, short Y, long Tag);

Allows to edit a rectangular text buffer, at the current or specified coordinates; returns True if the user confirmed the editing (Enter at the last line, or Ctrl+Enter anywhere), False if he pressed Esc. The behaviour is influenced by the following properties:

TextBoxDefault *as String*	initial buffer value (and resulting buffer upon return); this property is updated even when the user pressed Esc
TextBoxColumns *as Integer*	number of columns for the editing window
TextBoxRows *as Integer*	number of lines for the editing window
TextBoxStartPosition *as Integer*	cursor position in the rectangular buffer; this property is updated even when the user pressed Esc
InsertMode *as Boolean*	insert mode status: if True, every character typed moves forward the following characters; if False, it overwrites the current character; this property is updated even when the user pressed Esc
TextBoxBackColor *as Integer* TextBoxForeColor *as Integer*	foreground and background colours for the editing window
SilentMode *as Boolean*	if True, no acoustic warning will be played upon error (SoundRequest will be fired anyway)

The parameter *Tag* determines how the method react to the data being typed by the user; if zero, the default behaviour is the following:

Ctrl+Enter, Enter on the last line	Confirm editing
Esc	Abort editing
Tasti cursore	Navigate area
Home, End	Start/end of line
Ctrl+Home, Ctrl+End	Start/end of buffer
Ctrl+Left, Ctrl+Right	Previous/next word
Delete	Delete current character and move backward the rest of the text
Backspace	Delete previous characters and move backward the rest of the text
Enter	Insert spaces until the end of the line (move to the next line what follows) if insert mode is enabled; otherwise, insert a paragraph break at the end of the current line (if possible)
Ctrl+Y	Delete current line
Ins	Change cursor shape and insert mode status

Ctrl+B	Show/hide paragraph breaks
Ctrl+W	Move to the beginning of the next line the word to the left (wrap)
Ctrl+E	Move the cursor at the end of the word being edited
Ctrl+N	Clear the buffer and move cursor to the beginning
Ctrl+S	Save current buffer (checkpoint)
Ctrl+L	Restore buffer to the last checkpoint

Paragraph breaks are handled by introducing in the text the symbol chr(255), which is invisible in console mode. Upon exiting the method, remember to replace this symbol with a space before using the string, because in graphical mode this symbol is usually visible.

If *Tag* is not zero, whenever a key is pressed the following event is fired:

TextBoxKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)

where:

KeyAscii *as Integer*	Key pressed by the user; can be modified to fake a different key was pressed; set to 0 to discard it
Action *as String*	Determine the action requested in response; can be updated with one of the values specified below
Tag *as Integer*	User chosen identification number for this TextBox

The value of *Tag* can be used to discriminate which TextBox is active, to adopt different behaviours according to the circumstances.

The possible values for *Action* are listed below:

TEXTBOX_ACCEPT	Accept character into buffer
TEXTBOX_UPDATE	Buffer updated, continue editing
TEXTBOX_UPDATEANDCONFIRM	Buffer updated, confirm editing
TEXTBOX_ABORT	Abort
TEXTBOX_CONFIRM	Confirm input
TEXTBOX_DISCARD	Discard character
TEXTBOX_LEFT	Move cursor to the left
TEXTBOX_RIGHT	Move cursor to the right
TEXTBOX_HOME	Move cursor to the beginning
TEXTBOX_END	Move cursor to the end
TEXTBOX_DELLINE	Delete current line
TEXTBOX_BOL	Jump to the beginning of the line
TEXTBOX_EOL	Jump to the end of the line
TEXTBOX_PWORD	Next word
TEXTBOX_NWORD	Previous word
TEXTBOX_PARSIGN	Hide/Display paragraph breaks
TEXTBOX_WRAP	Move to the beginning of the next line the word to the left (wrap)
TEXTBOX_EOW	Move the cursor to the end of the word being edited
TEXTBOX_CLEAR	Clear buffer and move cursor at the beginning of

	the string
TEXTBOX_CHKRESTORE	Restore buffer to the last checkpoint
TEXTBOX_CHECKPOINT	Saves checkpoint

ThumbElevator (*ByVal Current as Long, ByVal Total as Long, ByVal Column as Integer, ByVal FirstRow as Integer, ByVal LastRow as Integer, ByVal ForeColor as Integer, ByVal BackColor as Integer, ByRef LastPosition as Integer*)

void ThumbElevator(long Current, long Total, short Column, short FirstRow, short LastRow, short ForeColor, short BackColor, short* LastPosition);

Draws a thumb elevator from (*Column, FirstRow*) to (*Column, LastRow*), using the colours *ForeColor* / *BackColor*. *Current* is the initial value (start from 0), *Total* is the maximum value; *LastPosition* keeps the cursor position and is updated upon return; if it is –1, the thumb elevator is completely redrawn (use it as the initial value).

Tone (*ByVal Frequency as Long, ByVal Duration as Long*)

void Tone(long Frequency, long Duration);

Plays a sound tone using the specified frequency (in Hertz) and duration (in milliseconds). If the property SilentMode is True, no sound is played and the following event is fired:

SoundRequest (*ByVal Frequency as Long, ByVal Duration as Long*)

In this way, the user has the opportunity of giving an alternate signal, or sending specific escape sequences to produce sound on the remote terminal.

xConsole® properties

Below you will find a short description of all the properties supported by the xConsole® control and the most relevant interactions between them (emphasized by a common prefix).

The methods are printed in **BLUE**, the properties in **RED**.

Constants are always expressed as mnemonic identifiers, whose values can be looked up in the module XCONSOLE.BAS and in the header file XCONSOLE.H.

The descriptions you will find refer to Visual Basic™; keep in mind the following type conversions:

Visual Basic™ type	Visual C++™ type
Boolean	BOOL
Integer	short or short * (when passed by reference)
Long	long or long * (when passed by reference)
String	LPCTSTR (parameter in methods) CString (property value) BSTR (value returned by a method)

Under Visual C++™ all the properties of the control are set by invoking functions whose name is the same of the property, prefixed by "Set"; these functions all have a single parameter, which is the value to be assigned to the property; e.g. MenuOptions is set with:

```
SetMenuOptions(options);
```

In the same way, when you need to get the value of a property you use a pseudo-function with the prefix "Get":

```
options = GetMenuOptions();
```

All methods having two variants (with and without "XY" suffix) are referred to with the simplest form. All properties are read/write.

Alphabetical list of the properties (see related methods for additional information):

AlertBackColor *as Integer*	Background color for Alert
AlertButtonBackColor *as Integer*	Background button color for Alert
AlertButtonForeColor *as Integer*	Foreground button color for Alert
AlertButtons *as String*	Button labels for Alert
AlertCurrentButton *as Integer*	Current button index for Alert
AlertForeColor *as Integer*	Foreground color for Alert
AlertFrameForeColor *as Integer*	Foreground frame color for Alert
AlertText *as String*	Alert message
BackgroundColor *as Integer*	Background color for SPrint, Hprint
ConsoleTitle *as String*	Window title (only visible when the program is not run full screen)
CursorType *as Integer*	Cursor type; possible values:

	CUR_OFF = hidden cursor CUR_BIG = block CUR_SMALL = underline Changing the property immediately set the cursor to the new type.
DateType *as Integer*	Date format, used for automatic date validation in InputString; possible values: DATE_US = month/day/year DATE_EUROPE = day/month/year DATE_JAPAN = year/month/day
Epoch *as Integer*	Epoch, used for automatic date validation in InputString. Epoch determines how years should be interpreted in short dates (where only two digits are used to specify the year); in this case, if the last two digits of the year are below the last two digits in Epoch, the year is considered in the following century, otherwise in the same century.
ForegroundColor *as Integer*	Foreground color for SPrint, Hprint
Frame *as Boolean*	Flag that determines if a frame will be added when using methods such as Alert, List, OSD, etc.
Frame3D *as Boolean*	Flag that determines if the frame will be drawn with a 3D effect (two sides are darker than the others)
FrameBackColor *as Integer*	Background color for Box
FrameChars *as String(8)*	8-byte string which represents the characters to be used for drawing the frame (clockwise, starting from upper left). By default after InitConsole you can draw single line frames (FRAME_SINGLE) by using semi-graphical character. You can set the value to other constants (FRAME_DOUBLE, FRAME_SNGDOU, FRAME_DOUSNG, FRAME_DOTS) for different appearances.
FrameForeColor *as Integer*	Foreground color for Box
FullScreen *as Boolean*	Determines if an application is running in a window (False) or full screen (True)
InputBackColor *as Integer*	Background color for InputString
InputCodePage *as Long*	CodePage used for input
InputDefault *as String*	Default/return value for InputString
InputForeColor *as Integer*	Foreground for InputString
InputMaxLength *as Integer*	Maximum string length for InputString
InputPicture *as String*	Validation format for InputString
InputStartPos *as Integer*	Initial cursor position for InputString
InputWindowLength *as Integer*	Width of the editing window for InputString
InputWindowOffset *as Integer*	Offset in editing window for InputString
InsertMode *as Boolean*	Insert/overwrite mode, used by InputString and TextBox

Justification *as Integer*	Text justification for SPrint, HPrint; possible values: J_NOJUST = No justification J_LEFT = Align to the left J_CENTER = Center text J_RIGHT = Align to the right J_JUST = Full justification
JustificationLength *as Integer*	Justification length for SPrint, Hprint
KeyLast *as Long*	Value of the last key read by KeyInput; read/write property
LineCharsHV *as String(2)*	2-character string that influences LineFromTo; the first character is used to draw horizontal lines, the second character is used for vertical lines
ListColumns *as Integer*	Number of columns filled by List options
ListCurrentLine *as Integer*	Current line for List
ListCurrentOption *as Integer*	Current option for List
ListFrameForeColor *as Integer*	Frame foreground color for List
ListMap *as String*	Selection map of the options for List
ListMultiSelect *as Boolean*	Allow multiple selections in List
ListOptions *as String*	Options for List
ListRows *as Integer*	Number of rows for the options in List
ListSelection *as Integer*	Selection character for List
ListTitle *as String*	Title for List box
ListWindowOffset *as Integer*	Offset for the options (number of characters to be skipped) in List
MaxCol *as Integer*	Screen columns
MaxRow *as Integer*	Screen rows
MenuCurrentOption *as Integer*	Current option for Menu
MenuFrameForeColor *as Integer*	Frame foreground color for Menu
MenuOptions *as String*	Option list (and description) for Menu
MenuUnselectable *as String*	Unselectable option map for Menu
OffsetX *as Integer*	Horizontal shift
OffsetY *as Integer*	Vertical shift
OutputCodePage *as Long*	CodePage used (useful only when FullScreen is True)
Pattern *as Integer*	Character used when clearing/filling an area
ScoreboardBackColor *as Integer*	Background color for descriptions (Menu)
ScoreboardForeColor *as Integer*	Foreground color for descriptions (Menu)
ScoreboardJustification *as Integer*	Justification for descriptions (Menu)
ScoreboardLength *as Integer*	Justification length for descriptions (Menu)
ScoreboardStatus *as Boolean*	Enable/disable printing descriptions (Menu)
ScoreboardX *as Integer*	Column for printing descriptions (Menu)
ScoreboardY *as Integer*	Row for printing descriptions (Menu)
SelectedBackColor *as Integer*	Background color for current option (List, Menu)
SelectedForeColor *as Integer*	Foreground color for current option (List, Menu)
ShadowMode *as Integer*	Shadow type; possible values: NO_SHADOW = no shadow SHADOW_LEFT = left shadow

	SHADOW_RIGHT = right shadow The shadow is added automatically when a Box is drawn (explicitly or implicitly)
SilentMode *as Boolean*	Flag: if false, Tone plays a sound; if true, Tone fires the event SoundRequest
TextBoxBackColor *as Integer*	Background color for TextBox
TextBoxColumns *as Integer*	Number of columns for TextBox
TextBoxDefault *as String*	Default/return text for TextBox
TextBoxForeColor *as Integer*	Foreground color for TextBox
TextBoxRows *as Integer*	Number of rows for TextBox
TextBoxStartPosition *as Integer*	Starting position in buffer for TextBox
ThumbElevatorChars *as String(4)*	4-character string that defines the characters to be used for drawing the thumb elevator: 1 – upper terminator 2 – lower terminator 3 – background 4 – elevator
TitleBackColor *as Integer*	Background title color for List
TitleForeColor *as Integer*	Foreground title color for List
UnselectableBackColor *as Integer*	Background color for unselectable options in Menu
UnselectableForeColor *as Integer*	Foreground color for unselectable options in Menu
UnselectedBackColor *as Integer*	Background color for unselected options in List, Menu
UnselectedForeColor *as Integer*	Foreground color for unselected options in List, Menu
X *as Integer*	Cursor column; read/write property (when the value is updated, the cursor is immediately repositioned to the new coordinate)
Y *as Integer*	Cursor row; read/write property (when the value is updated, the cursor is immediately repositioned to the new coordinate)

xConsole® events

Below you will find a short description of all the events fired by the xConsole® control.
The methods are printed in **BLUE**, the properties in **RED**, the events in **GREEN**.

Constants are always expressed as mnemonic identifiers, whose values can be looked up in the module XCONSOLE.BAS and in the header file XCONSOLE.H.

The descriptions you will find refer to Visual Basic™; keep in mind the following type conversions:

Visual Basic™ type	Visual C++™ type
Boolean	BOOL
Integer	short or short * (when passed by reference)
Long	long or long * (when passed by reference)
String	LPCTSTR (parameter in methods) CString (property value) BSTR (value returned by a method)

Under Visual C++ you need to add an event handler, as specified in the chapter that explains the usage of the control in this language.

Events fired when a key is pressed

Every key pressed fires the following event:

KeyPress(*ByRef KeyAscii as Integer*)

KeyAscii holds the key pressed by the user; this variable can be updated to simulate a different key, or set to 0 to discard the key.

The following events all have the same structure; they are invoked by the respective methods when *Tag* is not zero:

AlertKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)
InputStringKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)
ListKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)
MenuKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)
TextBoxKeyPress(*ByRef KeyAscii as Integer, ByRef Action as Integer, ByVal Tag as Long*)

where:

KeyAscii *as Integer*	Holds the key pressed by the user; can be updated to simulate a different key or set to 0 to discard the key
Action *as String*	Determines the action requested; can be updated with one of the values specified in the methods
Tag *as Integer*	Identification number of the method that fired the event

Other events

SoundRequest (*ByVal Frequency as Long, ByVal Duration as Long*)

This event is fired by Tone when SilentMode is True; this allows to substitute the default sound routine with a different one chosen by the user.

Regular expressions

xConsole® supports two kinds of regular expressions (for InputString): extended and simple.

Extended regular expressions

Extended regular expressions are more powerful, but also more complex to use; they consist in a string of characters where a few are interpreted literally, while others are control characters with a special meaning. This is a brief explanation about them:

a) '\' followed by a single character x means "match the character x";
b) '^' means "start of line"; '$' means "end of line";
c) '.' means "any character";
d) any character x, without a special meaning, means "match the character x";
e) a string enclosed between [square brackets] means "match any character in the string";
f) ASCII character ranges can be abbreviated as 'a-z0-9'. An isolated closing bracket (']') can appear only as the first character in the regular expression. A literal '-' can only appear where it can not be interpreted as a range indicator. If the first character is '^', then any character not matching the expression will be accepted;
g) a postfix '*' means "accept 0 or more repetitions";
h) a postfix '+' means "accept 1 or more repetitions";
i) a postfix '?' means "accept 0 or 1 repetitions";
j) two adjacent regular expressions (chained) means "match the first, then the second";
k) two regular expressions separated by '|' means "match the first or the second";
l) a regular expressions between parenthesis means "match what is inside the parenthesis".

The evaluation order for operators at the same level of parenthesis is (from highest to lowest priority):

[] *+? concatenation |

A few examples of extended regular expressions (used by {REXMATCH}, {REXIMATCH} and others):

"^a"	accept any string beginning with 'a'
"^apples"	accept any string beginning with 'apples'
"a$"	accept any string ending with 'a'
"oranges$"	accept any string ending with 'oranges'
"f..e"	accept any string of 4 letters beginning with 'f' and ending with 'e' (e.g. 'free', 'fare' but not 'force')
"[ab]"	accept any string containing 'a' or 'b'
"[^ab]"	accept any string not containing 'a' and 'b'
"^[0-3][0-9]/[0-1][0-9]/[0-9][0-9]$"	accept any date (like "30/12/97")
"su(m|n)"	accept any string containing the word 'sum' or 'sun' (not 'su')
"worl?d"	accept any string containing "word" or "world"
"^[0-9]*$"	accept an empty string or a number containing only the digits '0'-'9'
"^[a-zA-Z]+[a-zA-Z0-9]*$"	accept an identifier name (start with a letter, can contain only alphanumeric characters, is at least one character long)
"^(hello)|(goodbye)$"	accept only the two strings 'hello' and 'goodbye'

Simple regular expressions

Simple regular expressions are easier to use than extended regular expressions; they only include two special characters:

1. '*' replaces zero, one or more characters
2. '?' replaces a single character

A few examples of simple regular expressions:

"*su?*"	accept any string where the two characters 'su' are followed by a single character (e.g. 'sum', 'sun', etc.)
"c*"	accept any string starting with 'c'
"*a"	accept any string ending with 'a'
"???"	accept any string 3-character long
"*one*two*three*"	accept any string including the words 'one', 'two' and 'three' in this order

Software Licence Agreement

Carefully read the following Agreement before installing the software on your PC.

By installing the Software accompanying this document you acknowledge that you have read, understood and agree to abide by the terms and conditions of this Software Licence Agreement.

SOFTWARE LICENCE

This is a legal agreement (Agreement) between you (either an individual or an entity) and Simone Zanella Productions (SZP) that sets forth the licence terms and conditions for using the enclosed Software (Software). Updates of the Software shall also be subject to the terms and conditions of this Agreement. This Agreement is effective until terminated by destroying the Software and all of the diskettes and documentation provided in this package, together with all copies, tangible or intangible. In this Agreement, the term “use” means loading the Software into RAM, as well as installing it onto a hard disk or other storage device.

The Software is owned by SZP and is protected under Italian copyright laws as well as international treaty provisions. You must treat the software as you would any other copyrighted material.

The purchase price for the Software grants you a non-exclusive licence to use the Software with the following restrictions: the Software can be redistributed as part of a package developed by the purchasing company, but the new package cannot be a derivative of xConsole®.

You may make one copy of the software solely for archival purposes.

You may not rent, sell, lease, sub-licence, time-share or lend the Software to a third party or otherwise transfer this Licence without written permission from SZP. You may not decompile, disassemble, reverse-engineer or modify the Software.

It is strictly and expressly prohibited the redistribution of the Software as part of a package that can be considered generally competitive with the Software.

If you fail to comply with any of the terms and conditions of this Agreement, this Licence will be terminated and you will be required to immediately return to SZP, the Software, diskettes and documentation provided in this package, together with all back-up copies. The provisions of this Agreement which protect the proprietary rights of SZP will continue in force after termination.

LIMITED LIABILITY

The software and documentation are sold AS IS. You assume responsibility for the selection of the Software to achieve your intended results, and for the installation, use and results obtained from the Software. SZP makes no representations or warranties with regard to the Software and documentation, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose.

SZP shall not be liable for errors or omissions contained in software or manuals, any interruption of service, loss of business or anticipatory profits and/or for incidental or consequential damages in connection with the furnishing, performance or use of these materials.

LIMITED WARRANTY

For a period of twentyfour (24) months from date of purchase, SZP warrants to the original purchaser, that the disks on which the Software is recorded are free from defects in material and faulty workmanship when subject to normal use and service. If, during this twentyfour (24) month period, a defect should occur, the disk will be replaced free of charge after it is returned to SZP.

If a defect occurs after the expiration of this warranty period, certain charges may apply. SZP reserves the right to refuse repeated replacement requests.

This Limited Warranty gives you specific legal rights and you may also have other rights which vary from state to state. Some states do not allow the limitation or exclusion of implied warranties or of consequential damages, so the above limitations or exclusions may not apply to you.

You agree that this is the complete and exclusive statement of the Agreement between you and SZP which supercedes any proposal or prior agreement, oral or written, and any other communications between us regarding the subject matter of this Agreement. This Agreement shall be construed, interpreted and governed by the Italian laws and any controversy will be treated by the forum of Venice – Italy. If any provision of this Agreement is found unenforceable, it will not effect the validity of the balance of this Agreement, which shall remain valid and enforceable according to its terms.

Index