```
// BigIni.pas v 5.00                                                    eh 2004-06-08
// Unit to read/write *.ini files even greater than 64 kB
// (c) Edy Hinzen 1996-2004 - Freeware
// See help file for details, disclaimer and contact information
//---------------------------------------------------------------------------

{*@summary Unit to read/write *.INI files even greater than 64 KByte.}
unit BigIni;

// Activate the following line, if you never access lines longer than 255 chars:
{ $ DEFINE UseShortStrings}

{$IFDEF UseShortStrings}
{$H-} {using short strings in old pascal style increases speed}
{$ENDIF}

{$IFDEF VER140}{$DEFINE DELPHI6_UP}{$ENDIF D6}
{$IFDEF VER150}{$DEFINE DELPHI6_UP}{$ENDIF D7}
{$IFDEF VER160}{$DEFINE DELPHI6_UP}{$ENDIF}

interface


uses classes,Windows,ShellAPI,SysUtils,Forms,Graphics;

const
  IniTextBufferSize  = $7000; {Note [1]: don't use more than $7FFF - it's an
integer}

  cIniCount          = 'Count';                 // count keyword
  cDefaultUsePrefixOnCount = False;             // see ReadNumberedList /
WriteNumberedList
type
  {*@summary Type of callback routines optionally used by
   TSectionList.EraseDuplicates
   *@desc Write TEraseSectionCallback event handler if you want
   programatically decide which of duplicate sections should be erased.<br>
   <br>
   <b>parameters</b><br>
   <i>sectionName</i> is the name of the section that was duplicate in the
   *.INI file.<br>
   <i>sl1</i> contains key/value pairs of the first section.<br>
   <i>sl2</i> contains key/value pairs of the second section.<br>
   <br>
   <b>Returns</b><br>
   <i>true</i> signals that the second section should be deleted (default if
   no callback is used).<br>
   <i>false</i> signals that the rist section should be deleted.}
  TEraseSectionCallback = function(const sectionName : string; const sl1,sl2 :
TStringList):Boolean of object;
```

```
{
TCommaSeparatedInfo is a Tool-Class to read and write multiple parameters/values
from a single, comma-separated string. These parameters are positional.

Please see descendant TCSIFont for some useful example.
}
  {*@summary Tool-Class wo read/write multiple values from and to
   comma-separated string.
   *@desc TCommaSeparatedInfo is a Tool-Class to read and write multiple
   parameters/values from a single, comma-separated string. These parameters
   are positional.<br>
   <br>
   Please see descendant <i>TCSIFont</i> in source code for some useful
   example.}
  TCommaSeparatedInfo = class
  private
    FValues        : TStringList;
  protected
    function       GetBoolean(Index: Integer): Boolean;
    function       GetColor(Index: Integer): TColor;
    function       GetElement(Index: Integer): string;
    function       GetInteger(Index: Integer): Integer;
    function       GetValue: string;
    procedure      SetBoolean(Index: Integer; const Value: Boolean);
    procedure      SetColor(Index: Integer; const Value: TColor);
    procedure      SetElement(Index: Integer; const Value: string);
    procedure      SetInteger(Index: Integer; const Value: Integer);
    procedure      SetValue(const Value: string);
  public
    constructor    Create;
    destructor     Destroy; override;
    {*@summary Returns value at index <i>index</i> as boolean.}
    property       AsBoolean[index:Integer]: Boolean read GetBoolean write
SetBoolean;
    {*@summary Returns value at index <i>index</i> as TColor.}
    property       AsColor[index:Integer]: TColor read GetColor write SetColor;
    {*@summary Returns value at index <i>index</i> as Integer.}
    property       AsInteger[index:Integer]: Integer read GetInteger write
SetInteger;
    {*@summary Returns value at index <i>index</i> as string.}
    property       Element[index:Integer]: string read GetElement write SetElement;
default;
    {*@summary Gets or sets comma-separated work string.}
    property       Value : string read GetValue write SetValue;
  end;

  {*@summary Helper class to translate <i>TFont</i> data to comma-separated
   string and vice versa.
   *! This class is used in <i>TBiggerIniFile.ReadFont()</i> and
```

```
 <i>TBiggerIniFile.WriteFont()</i>.}
TCSIFont = class(TCommaSeparatedInfo)
protected
  function       GetFontStyle: TFontStyles;
  procedure      SetFontStyle(const Value: TFontStyles);
public
  {*@summary Gets or sets element <i>Name</i> of <i>TFont</i> structure.}
  property       FontName  : string index 0 read GetElement write SetElement;
  {*@summary Gets or sets element <i>Size</i> of <i>TFont</i> structure.}
  property       FontSize  : Integer index 1 read GetInteger write SetInteger;
  {*@summary Gets or sets element <i>Style</i> of <i>TFont</i> structure.}
  property       FontStyle : TFontStyles read GetFontStyle write SetFontStyle;
  {*@summary Gets or sets element <i>Color</i> of <i>TFont</i> structure.}
  property       FontColor : TColor index 6 read GetColor write SetColor;
  {*@summary Gets or sets element <i>Charset</i> of <i>TFont</i> structure.}
  property       FontCharset : Integer index 7 read GetInteger write SetInteger;
  {*@summary Gets or sets element <i>Pitch</i> of <i>TFont</i> structure.}
  property       FontPitch : Integer index 8 read GetInteger write SetInteger;
end;

{*@summary Helper class to translate <i>TPoint</i> data to comma-separated
 string and vice versa.
 *! This class is used in <i>TBiggerIniFile.ReadPoint()</i> and
 <i>TBiggerIniFile.WritePoint()</i>.}
TCSIPoint = class(TCommaSeparatedInfo)
public
  {*@summary Gets or sets element <i>X</i> of <i>TPoint</i> structure.}
  property       X      : Integer index 0 read GetInteger write SetInteger;
  {*@summary Gets or sets element <i>Y</i> of <i>TPoint</i> structure.}
  property       Y      : Integer index 1 read GetInteger write SetInteger;
end;

{*@summary Helper class to translate <i>TRect</i> data to comma-separated
 string and vice versa.
 *! This class is used in <i>TBiggerIniFile.ReadRect()</i> and
 <i>TBiggerIniFile.WriteRect()</i>.}
TCSIRect = class(TCommaSeparatedInfo)
public
  {*@summary Gets or sets element <i>Left</i> of <i>TRect</i> structure.}
  property       Left   : Integer index 0 read GetInteger write SetInteger;
  {*@summary Gets or sets element <i>Top</i> of <i>TRect</i> structure.}
  property       Top    : Integer index 1 read GetInteger write SetInteger;
  {*@summary Gets or sets element <i>Right</i> of <i>TRect</i> structure.}
  property       Right  : Integer index 2 read GetInteger write SetInteger;
  {*@summary Gets or sets element <i>Bottom</i> of <i>TRect</i> structure.}
  property       Bottom : Integer index 3 read GetInteger write SetInteger;
end;

{*@summary Tool-Class for TBigIniFile.
 *@desc This class is a descendant of <i>TStringList</i> with "enhanced"
```

```
   <i>IndexOf()</i> function (and others).}
  TSectionList = class(TStringList)
  private
    FPrevIndex          : Integer;
  public
    {*@summary Creates new instance.}
    constructor Create;
    {*@summary Erase duplicate entries.
     *@desc Returns <i>true</i> if changes were made by erasing.<br>
     <br>
     <b>Parameters</b><br>
     <br>
     <i>callBackProc</i><br>
     You can set this parameter to a self-defined erase section callback
     function. If set to <i>NIL</i>, no callback takes place.}
    function    EraseDuplicates(callBackProc:TEraseSectionCallback = nil) : Boolean;
    {*@summary Access to property <i>SectionItems</i>.}
    function    GetSectionItems(Index: Integer): TStringList;
    {*@summary Returns the position of a string in the list.
     *@desc This is a version optimized for this unit since it is likely that
     the same section name will be needed several times.}
    function    IndexOf(const S: string): Integer; override;
{$IFDEF DELPHI6_UP} // must be overridden in D6 ff
    function    IndexOfName(const Name: string): Integer; override;
{$ELSE DELPHI6_UP}
    {*@summary Determines the position of the first string with key
     <i>Name</i>.
     *@desc If no key is found, -1 will be returned.}
    function    IndexOfName(const Name: string): Integer; virtual;
{$ENDIF DELPHI6_UP}
    {*@summary Used internally to retrieve <i>TStringList</i> object stored
     in <i>Objects[Index]</i>.}
    property    SectionItems[index: Integer]: TStringList read GetSectionItems;
  end;

  {*@summary TBigIniFile is a TIniFile replacement. It's main goal is to read
   / write *.INI files greater than 64 KByte.
   *@desc The <i>TBigIniFile</i> object is designed to work like
   <i>TIniFile</i> from the Borland unit <i>IniFiles</i>.<br>
   <br>
   The following procedures/functions/properties were added:<ul><br>
   function  <i>ReadAnsiString</i> - reads AnsiString types<br>
   procedure <i>AppendFromFile</i> - appends from other *.ini<br>
   procedure <i>FlushFile</i> - write data to disk<br>
   procedure <i>WriteAnsiString</i> - writes AnsiString types<br>
   property  <i>SectionNames</i><br>
   </ul>}
  TBigIniFile = class(TObject)
  protected
    FHasChanged           : Boolean;
```

```
  FTextBufferSize       : Integer;
private
  FEraseSectionCallback : TEraseSectionCallback;
  FFileName             : string;
  FPrevSectionIndex     : Integer;
  FFlagClearOnReadSectionValues,   {set true if clearing wanted}
  FFlagDropCommentLines,           {set false to keep lines starting with ';'}
  FFlagFilterOutInvalid,           {set false to keep lines without '='      }
  FFlagDropWhiteSpace,             {set false to keep white space around '='}
  FFlagDropApostrophes,            {set false to keep apostrophes around key }
  FFlagTrimRight        : Boolean; {set false to keep white space at end of line}
  FSectionList          : TSectionList;

  function    FindItemIndex(const aSection, aKey :string; CreateNew:Boolean;
                            var FoundStringList:TStringList):Integer;
  procedure   SetFileName(const aName : string);
  procedure   ClearSectionList;
public
  {*@summary Creates new instance.
   *@desc Parameter <i>FileName</i> determines the file name of the *.INI
   file to be used.<br>
   <br>
   Use empty file name if you want to hold data just in Memory or set some
   parameters before loading from the file.}
  constructor Create(const FileName: string);
  {*@summary Removes object from memory.
   *@desc Calls <i>FlushFile</>. That means that data will be saved if they
   have changed.
   *! Calls <i>FlushFile()</i>. That means that data will be saved if they
   have changed.}
  destructor  Destroy; override;

  {*@summary Appends data from File.
   *@desc Appends entries from given file to existing data. Duplicate
   sections will be ignored.
   *! Is internally also used to load data from file.}
  procedure   AppendFromFile(const aName : string); virtual;
  {*@summary Erases all data from the INI file.
   *! Property <i>HasChanged</i> is not affected by this operation.}
  procedure   Clear; virtual;
  {*@summary Deletes a key and it's associated value.}
  procedure   DeleteKey(const aSection, aKey: string); virtual;
  {*@summary Erases entire section.}
  procedure   EraseSection(const aSection: string); virtual;
  {*@summary Copies all 'lines' to TStrings object.
   *@desc In practice, the following line<br>
   <i>  myIniFile.getStrings(myStringList);</i><br>
   mostly produces the same result like<br>
   <i>  myIniFile.flushFile();<br>
   myStringList.loadFromFile(myIniFile.FileName);</i><br>
```

```
     <br>
     I say 'mostly', because <i>GetStrings</i> also works with virtual files
     (empty property <i>Filename</i>) or those that have property
     <i>HasChanged</i> set to <i>false</i>.}
    procedure   GetStrings(List: TStrings); virtual;
    {*@summary Populates TBigIniFile object with data from TStrings object.
     *@desc All sections, keys and values will be set similar to the
     execution of the following lines:<br>
     <br>
     <i>aStrings.SaveToFile('someTemporary.ini');<br>
     Filename = 'someTemporary.ini';</i>}
    procedure   SetStrings(const aStrings: TStrings); virtual;
    {*@summary Saves data to disk.
     *@desc Saves data to disk if<ul><br>
     Property <i>FileName</i> is not an empty string ('').<br>
     Property <i>HasChanged</i> is set to <i>true</i>.<br>
     </ul>}
    procedure   FlushFile; virtual;
    {*@summary Reads AnsiString value.}
    function    ReadAnsiString(const aSection, aKey, aDefault: string): AnsiString;
virtual;
    {*@summary Reads binary data into stream.
     *@desc <b>Parameters</b><br>
     <i>aSection, aKey</i><br>
     Section and key parameters as used in <i>ReadString()</i>.<br>
     <br>
     <i>aStream</i><br>
     <i>TStream</i>-object to write binary data into. Bytes will be written
     to the stream starting at current stream position.<br>
     <br>
     <b>Returns</b><br>
     Number of bytes written to <i>aStream</i>.}
    function    ReadBinaryStream(const aSection, aKey: string; aStream: TStream):
Integer; virtual;
    {*@summary Read boolean value.}
    function    ReadBool(const aSection, aKey: string; aDefault: Boolean): Boolean;
virtual;
    {*@summary Read TDateTime value.
     *@desc The string stored in the *.INI file will be parsed with function
     <i>StrToDate</i>. Thus, only the date portion of <i>TDateTime</i> will
     be set.
     *! The string stored in the *.INI file will be parsed with function
     <i>StrToDate</i>. Thus, only the date portion of <i>TDateTime</i> will
     be returned.}
    function    ReadDate(const aSection, aKey: string; aDefault: TDateTime):
TDateTime; virtual;
    {*@summary Read TDateTime value.}
    function    ReadDateTime(const aSection, aKey: string; aDefault: TDateTime):
TDateTime; virtual;
    {*@summary Reads a float value.}
```

```
    function    ReadFloat(const aSection, aKey: string; aDefault: Double): Double;
virtual;
    {*@summary Reads an Integer value.}
    function    ReadInteger(const aSection, aKey: string; aDefault: Longint):
Longint; virtual;
    {*@summary Reads all key names from specified section into TStrings object.
     *! Object <i>aStrings</i> will not be cleared on operation.}
    procedure   ReadSection(const aSection: string; aStrings: TStrings); virtual;
    {*@summary Reads the names of all sections into TStrings object.
     *! Object <i>aStrings</i> will be cleared on operation.}
    procedure   ReadSections(aStrings: TStrings); virtual;
    {*@summary Reads all lines of specified section to TStrings object.}
    procedure   ReadSectionValues(const aSection: string; aStrings: TStrings);
virtual;
    {*@summary Retrieves a string value from an *.INI file.
     *@desc This function is base of most other Read* functions.<br>
     <br>
     <b>Parameters</b><br>
     <br>
     <i>aSection</i><br>
     Section where the specified key/value is to be read from.<br>
     Section names are stored within brackets ([]) within the '.INI file.<br>
     <br>
     <i>aKey</i><br>
     Key under which the searched value is to be found.<br>
     <br>
     <i>aDefault</i><br>
     Is the value that will be assigned to the result string, if key
     <i>aKey</i> is not set in specified section.}
    function    ReadString(const aSection, aKey, aDefault: string): string; virtual;
    {*@summary Reads time value to TDateTime object.}
    function    ReadTime(const aSection, aKey: string; aDefault: TDateTime):
TDateTime; virtual;
    {*@summary Indicates whether a section exists.
     *@desc Returns true, if a section with the name specified in
     <i>aSection</i> exists.}
    function    SectionExists(const aSection: string): Boolean; virtual;
    {*@summary Flushes buffered data to disk.
     *@desc This function is equivalent to <i>FlushFile()</i>.}
    procedure   UpdateFile; virtual;
    {*@summary Retrieves whether a key exists.
     *@desc Returns true, if a key with name <i>aKey</i> exists in section
     with the name specified in <i>aSection</i>.}
    function    ValueExists(const aSection, aKey: string): Boolean; virtual;
    {*@summary Writes AnsiString value.}
    procedure   WriteAnsiString(const aSection, aKey, aValue: AnsiString); virtual;
    {*@summary Writes boolean value.}
    procedure   WriteBool(const aSection, aKey: string; aValue: Boolean); virtual;
    {*@summary Writes strean data.
     *@desc Writes binary data from buffer to *.INI file. Data will be
```

```
     written in their hexadecimal string representation.<br>
     <br>
     <b>Parameters</b><br>
     <i>aSection, aKey</i><br>
     Section and key parameters as used in <i>WriteString()</i>.<br>
     <br>
     <i>aStream</i><br>
     <i>TStream</i>-object to fetch data from. Bytes will be fetched starting
     at current stream position.}
    procedure   WriteBinaryStream(const aSection, aKey: string; aStream: TStream);
virtual;
    {*@summary Writes TDateTime value using the <i>DateToStr</i> function.
     *! Only the date portion will be stored (ignoring time information).}
    procedure   WriteDate(const aSection, aKey: string; aValue: TDateTime); virtual;
    {*@summary Writes TDateTime value.}
    procedure   WriteDateTime(const aSection, aKey: string; aValue: TDateTime);
virtual;
    {*@summary Writes a float value.}
    procedure   WriteFloat(const aSection, aKey: string; aValue: Double); virtual;
    {*@summary Writes an integer value.}
    procedure   WriteInteger(const aSection, aKey: string; aValue: Longint);
virtual;
    {*@summary Writes a <i>string</i>value.
     *@desc This procedure is base of most other write*() procedures.<br>
     <br>
     <b>Parameters</b><br>
     <br>
     <i>aSection</i><br>
     Section where the specified key/value is to be written to.<br>
     Section names are stored within brackets ([]) within the '.INI file.<br>
     <br>
     <i>aKey</i><br>
     Key under which the searched value is to be stored.<br>
     <br>
     <i>aValue</i><br>
     Is the value that will be stored.<br>
     <br>
     <b>Example</b><br>
     <i>WriteString('settings','firstname','Edy')</i> produces the following
     lines in the *.INI file:<br>
     <br>
     <i>[settings]<br>
     firstname=Edy</i>}
    procedure   WriteString(const aSection, aKey, aValue: string); virtual;
    {*@summary Writes the time portion of a TDateTime value.}
    procedure   WriteTime(const aSection, aKey: string; aValue: TDateTime); virtual;

    {*@summary Pointer to TEraseSectionCallback event handler.}
    property    EraseSectionCallback: TEraseSectionCallback read
FEraseSectionCallback write FEraseSectionCallback;
```

```
    {*@summary Determines if clearing is wanted for <i>aStrings</i> parameter
     of method <i>ReadSectionValues()</i>.
     *@desc Set to <i>true</i> if clearing is wanted for <i>aStrings</i>
     parameter of method <i>ReadSectionValues()</i>.<br>
     <br>
     In prior versions of TIniFile the target-Strings were <i>not</i> cleared
     on ReadSectionValues calls.<br>
     That's why my procedure didn't either.<br>
     Meanwhile, Borland changed their mind and I added this flag for D5
     compatibility.<br>
     <br>
     Default = <i>false</i>}
    property    FlagClearOnReadSectionValues : Boolean read
FFlagClearOnReadSectionValues write FFlagClearOnReadSectionValues;
    {*@summary Determines, if apostrophes around keys will be kept.
     *@desc Set to false to keep apostrophes around key.<br>
     Otherwise, keys like in<br>
     <i>'truth'=42</i><br>
     will be changed to<br>
     <i>truth=42</i><br>
     <br>
     Default = <i>false</i>}
    property    FlagDropApostrophes  : Boolean read FFlagDropApostrophes write
FFlagDropApostrophes;
    {*@summary Determines if comment lines (lines starting with ';') will be
     kept.
     *@desc Set to <i>false</i> to keep lines starting with ';'.<br>
     <br>
     Default = <i>false</i>}
    property    FlagDropCommentLines : Boolean read FFlagDropCommentLines write
FFlagDropCommentLines;
    {*@summary Determines if white space (blanks) around '=' will be dropped.
     *@desc Set to <i>false</i> to keep white space around '='.<br>
     <br>
     Default = <i>false</i>}
    property    FlagDropWhiteSpace   : Boolean read FFlagDropWhiteSpace write
FFlagDropWhiteSpace;
    {*@summary Determines, if lines without '=' were kept.
     *@desc Set to <i>false</i> to keep lines without '='.<br>
     Otherwise, all non-comment lines without an equal sign '=' will be
     dropped.<br>
     <br>
     Default = <i>false</i>}
    property    FlagFilterOutInvalid : Boolean read FFlagFilterOutInvalid write
FFlagFilterOutInvalid;
    {*@summary Determines if white space at end of line is to be kept.
     *@desc Set to <i>false</i> to keep white space at end of line.<br>
     Otherwise, it will be removed.<br>
     <br>
     Default = <i>false</i>}
```

```
    property    FlagTrimRight        : Boolean read FFlagTrimRight write
FFlagTrimRight;
    {*@summary Contains the name of the *.INI file from/to which to read/write.
     *@desc Set this property to an empty string ('') if you<ul><br>
     Want to hold the data just in memory (without saving them to disk
     later).<br>
     Want to set some other flags/properties before accessing the file.<br>
     </ul><br>
     <br>
     Changeing the filename will<ul><br>
     call <i>flushfile()</i> before setting the new property,<br>
     read the data from the given file (if it exists).<br>
     </ul>}
    property    FileName: string read FFileName write SetFileName;
  end;

  {*@summary The TBiggerIniFile class is a TBigIniFile descendant with some
   additional functions that came in handy at my projects.<br>
   Click on the links 'Properties' and 'Methods' above for a list.}
  TBiggerIniFile = class(TBigIniFile)
  private
    procedure   SetTextBufferSize(const Value: Integer);
  public
    {*@summary Erase a numbered list.
     *@desc Reads a set of string values into variable aStrings.<br>
     <br>
     <b>Parameters</b><br>
     <i>aSection</i><br>
     Name of section that contained the data.<br>
     <br>
     <i>aPrefix</i><br>
     Prefix for the used key names. For details, please see
     <i>WriteNumberedList</i>.<br>
     <br>
     <i>IndexStart</i><br>
     Value of first used index. For details, please see
     <i>WriteNumberedList</i>.<br>
     <br>
     <i>usePrefixOnCount</i><br>
     Flag: Is the count-variable prefixed, too?  For details, please see
     <i>WriteNumberedList</i>.}
    procedure   EraseNumberedList(const aSection: string;
                                  aPrefix: string = '';
                                  IndexStart: Integer = 1;
                                  usePrefixOnCount: Boolean =
cDefaultUsePrefixOnCount); virtual;
    {*@summary Starts editor to show the file.
     *@desc The appropriate program for the given file extension (mostly
     .ini) is evaluated from the user's registry.}
    procedure   LaunchInEditor;
```

```
    {*@summary Reads binary data into buffer.
     *@desc <b>Parameters</b><br>
     <i>aSection, aKey</i><br>
     Section and key parameters as used in <i>ReadString()</i>.<br>
     <br>
     <i>Buffer</i><br>
     Buffer to write binary data into. You must allocate memory to this
     variable before usage.<br>
     <br>
     <i>BufSize</i><br>
     Memory allocated to <i>Buffer</i> in bytes.<br>
     <br>
     <b>Returns</b><br>
     Number of bytes written to <i>Buffer</i>.}
    function    ReadBinaryData(const aSection, aKey: string; var Buffer; BufSize:
Integer): Integer; virtual;
    {*@summary Reads a TColor value.
     *@desc Reads a TColor value stored as hex-string by using WriteColor().}
    function    ReadColor(const aSection,
                          aKey: string;
                          aDefault: TColor): TColor; virtual;
    {*@summary Read a font's properties.
     *@desc Reads a TFont value stored by using WriteFont().}
    function    ReadFont(const aSection, aKey: string; aDefault: TFont): TFont;
virtual;
    {*@summary Reads a numbered list into TStrings.
     *@desc Reads a set of string values into variable aStrings.<br>
     <br>
     <b>Parameters</b><br>
     <i>aSection</i><br>
     Name of section to contain the data.<br>
     <br>
     <i>aStrings</i><br>
     TStrings object to recveive the data.<br>
     <br>
     <i>aDefault</i><br>
     Default value for those entries which are not in the section.<br>
     <br>
     <i>aPrefix</i><br>
     Prefix for the used key names. For details, please see
     <i>WriteNumberedList</i>.<br>
     <br>
     <i>IndexStart</i><br>
     Value of first used index. For details, please see
     <i>WriteNumberedList</i>.<br>
     <br>
     <i>usePrefixOnCount</i><br>
     Flag: Is the count-variable prefixed, too?  For details, please see
     <i>WriteNumberedList</i>.}
    procedure   ReadNumberedList(const aSection: string;
```

```
                                aStrings: TStrings;
                                aDefault: string;
                                aPrefix: string = '';
                                IndexStart: Integer = 1;
                                usePrefixOnCount: Boolean =
cDefaultUsePrefixOnCount); virtual;
    {*@summary Reads TPoint value.}
    function    ReadPoint(const aSection, aKey: string; aDefault: TPoint): TPoint;
    {*@summary Reads TRect value.}
    function    ReadRect(const aSection, aKey: string; aDefault: TRect): TRect;
    {*@summary Renames a key.
     *@desc Renames key with name given in <i>OldKey</i> to that one in
     <i>NewKey</i>.
     *! If key doesn't exist, no action takes place.}
    procedure   RenameKey(const aSection, OldKey, NewKey: string); virtual;
    {*@summary Renames a section.
     *@desc Renames section with name given in <i>OldSection</i> to that one
     in <i>NewSection</i>.
     *! If section doesn't exist, no action takes place.}
    procedure   RenameSection(const OldSection, NewSection : string); virtual;
    {*@summary Returns the number of sections.}
    function    SectionCount() : Integer; virtual;
    {*@summary Writes binary data.
     *@desc Writes binary data from buffer to *.INI file.}
    procedure   WriteBinaryData(const aSection, aKey: string; var Buffer; BufSize:
Integer); virtual;
    {*@summary Writes non-default boolean value.
     *@desc Writes boolean value if it's different from given
     <i>aDefault</i>.<br>
     Otherwise the key will be removed from the section.}
    procedure   WriteBoolDef(const aSection, aKey: string; aValue: Boolean; const
aDefault: Boolean); virtual;
    {*@summary Write TColor value.
     *@desc Writes TColor value as hex-string in the form <i>00bbggrr</i>
     (meaning bb for blue, gg for green, rr for red value).}
    procedure   WriteColor(const aSection,
                           aKey: string;
                           aValue: TColor); virtual;
    {*@summary Write a font's properties.}
    procedure   WriteFont(const aSection, aKey: string; aFont: TFont); virtual;
    {*@summary Writes TStrings object to *.INI file.
     *@desc <b>Parameters</b><br>
     <i>aSection</i><br>
     Name of section to contain the data.<br>
     <br>
     <i>aStrings</i><br>
     TStrings object with the data.<br>
     <br>
     <i>aPrefix</i><br>
     Prefix for the used key names. See details below.<br>
```


IndexStart
Value of first used index. See details below.

usePrefixOnCount
Flag: Shall the count-variable be prefixed, too? See details below.

Parameter and flag usage
By default, the data is stored in key/value pairs where the keys are simple numbers starting at 1</>.
The count of lines is stored in a key named Count.
Assumed, our source tempStringList contains three string values: 'Computing', 'Music' and 'Shopping'.

WriteNumberedList('Hobbies (1)',tempStringList); produces the following lines in the *.INI file:
[Hobbies (1)]
Count=3
1=Computing
2=Music
3=Shopping

To give the keys a prefix, use parameter aPrefix:
WriteNumberedList('Hobbies (2)',tempStringList,'Hobby_'); produces:
[Hobbies (2)]
Count=3
Hobby_1=Computing
Hobby_2=Music
Hobby_3=Shopping

Some programs don't count starting by 1 but perhaps at 0. Use param IndexStart for this purpose:
WriteNumberedList('Hobbies (3)',tempStringList,'Hobby_',0); procudes:
[Hobbies (3)]
Count=3
Hobby_0=Computing
Hobby_1=Music
Hobby_2=Shopping

To write several lists to one section, you may change the key for Count by using usePrefixOnCount:
WriteNumberedList('various',tempStringList,'Hobby_',1,true); produces:
[various]
Hobby_Count=3
Hobby_1=Computing
Hobby_2=Music
Hobby_3=Shopping}

```
    procedure   WriteNumberedList(const aSection: string;
                                  aStrings: TStrings;
                                  aPrefix: string = '';
                                  IndexStart: Integer = 1;
                                  usePrefixOnCount: Boolean =
cDefaultUsePrefixOnCount); virtual;
    {*@summary Writes TPoint value.}
    procedure   WritePoint(const aSection, aKey: string; aPoint: TPoint);
    {*@summary Writes TRect value.}
    procedure   WriteRect(const aSection, aKey: string; aRect: TRect);
    {*@summary Writes a complete section.
     *@desc All lines contained in aStrings will be written to given section.
     Lines that were in that section before, will be discarded.
     *! This technique can be used to write comment (and other) lines into a
     section.}
    procedure   WriteSectionValues(const aSection: string; const aStrings:
TStrings); virtual;

    {*@summary Indicates if values and / or keys have been changed. Set
     <i>HasChanged</i> to force or prevent data beeing saved to disk.
     *@desc <i>HasChanged</i> will be set<br>
     <ul>if a section is added, renamed or deleted<br>
     if a key is added, renamed or deleted<br>
     whenever  a value is changed<br>
     </ul><br>
     <br>
     <i>HasChanged</i> will be reset if<br>
     <ul>A new/different filename is given.</ul><br>
     <br>
     If <i>HasChanged</i> is not set, the file will not be (re-)written to
     disk in <i>FlushFile</i> or <i>Close</i> operations.}
    property    HasChanged : Boolean read FHasChanged write FHasChanged;
    {*@summary Use <i>TextBufferSize</i> to determine the buffer size for
     disk operations.
     *@desc Property <i>TextBufferSize</i> is used to set the size for
     <i>ReadLn</i> / <i>WriteLn</i> operations.<br>
     Set <i>TextBufferSize</i> to <i>0</i> switch buffering off.}
    property    TextBufferSize : Integer read FTextBufferSize write
setTextBufferSize;
  end;

  {*@summary The TAppIniFile class is a TBiggerIniFile descendant who's
   filename is derived from the application's exename.
   *@desc It's constructor <i>create()</i> has no parameters. The filename is
   the application's exename with with extension '.ini' (instead of '.exe').}
  TAppIniFile = class(TBiggerIniFile)
  public
    constructor Create;
    class function DefaultFileName : string; virtual;
  end;
```

```
  {*@summary TLibIniFile class is very similar to its super class
   TAppIniFile. But if the module is a library (e.g. DLL) the library name is
   used.}
  TLibIniFile = class(TAppIniFile)
  public
    class function DefaultFileName : string; override;
  end;

  {*@summary Gets the full path of the current module.
   *@desc Similar to <i>Application.ExeName</i>, the name of the current
   module is returned.<br>
   <br>
   If flag <i>getLibraryName</i> is set (and the module is a library) then
   the library name is returned. Otherwise the application's name is
   returned.<br>
   <br>
   The result is in proper mixed case using the same notation as the original
   path and file name (the similar function <i>Application.ExeName</i>
   returns all in uppercase chars).}
  function ModuleName(getLibraryName:Boolean) : string;

//---------------------------------------------------------------------------
implementation
//---------------------------------------------------------------------------

//.....................................................................
// classless functions/procedures
//.....................................................................

// . . . . . .
function ModuleName(getLibraryName:Boolean) : string;
var
  Buffer        : array[0..260] of Char;
  theHandle     : THandle;
  thePath       : string;
  theSearchRec  : TSearchRec;
begin
  if getLibraryName then theHandle := HInstance
                    else theHandle := 0;
  SetString(Result, Buffer, GetModuleFileName(theHandle, Buffer, SizeOf(Buffer)));
  { GetModuleFileName returns a result in uppercase letters only }
  { The following FindFirst construct returns the mixed case name }
  thePath  := ExtractFilePath(Result);
  if FindFirst(Result,faAnyFile,theSearchRec) = 0 then
  begin
    Result := thePath + theSearchRec.Name;
  end;
  FindClose(theSearchRec);
end;
```

```
//.....................................................................
{ TCommaSeparatedInfo }
//.....................................................................

constructor TCommaSeparatedInfo.Create;
begin
  FValues := TStringList.Create;
end;

destructor TCommaSeparatedInfo.Destroy;
begin
  FValues.Free;
  inherited;
end;

function TCommaSeparatedInfo.GetBoolean(Index: Integer): Boolean;
begin
  // '1' stands for 'true', any other value for 'false'
  Result := (Element[Index] = '1');
end;

function TCommaSeparatedInfo.GetColor(Index: Integer): TColor;
begin
  Result := StrToIntDef('$'+Element[Index],-1);
  if Result = -1 then
  begin
    try
      Result := StringToColor(Element[Index]);
    except
      Result := clBlack;
    end;
  end;
end;

function TCommaSeparatedInfo.GetElement(Index: Integer): string;
const
  default = '';
begin
  if (Index > FValues.Count-1) then Result := default
                               else Result := FValues[Index];
end;

function TCommaSeparatedInfo.GetInteger(Index: Integer): Integer;
begin
  Result := StrToIntDef(Element[Index],-1);
end;

function TCommaSeparatedInfo.GetValue: string;
begin
```

```
  Result := FValues.CommaText;
end;

procedure TCommaSeparatedInfo.SetBoolean(Index: Integer;
  const Value: Boolean);
const
  BoolText: array[Boolean] of string[1] = ('', '1');
begin
  SetElement(Index, BoolText[Value]);
end;

procedure TCommaSeparatedInfo.SetElement(Index: Integer;
  const Value: string);
begin
  while (FValues.Count -1) < Index do FValues.Add('');
  FValues[Index] := Value;
end;

procedure TCommaSeparatedInfo.SetInteger(Index: Integer;
  const Value: Integer);
begin
  SetElement(Index, IntToStr(Value));
end;

procedure TCommaSeparatedInfo.SetValue(const Value: string);
begin
  FValues.CommaText := Value;
end;

procedure TCommaSeparatedInfo.SetColor(Index: Integer; const Value: TColor);
begin
  Element[Index] := ColorToString(Value);
end;

//.....................................................................
{ TCSIFont }
//.....................................................................

function TCSIFont.GetFontStyle: TFontStyles;
begin
  Result := [];
  if AsBoolean[2] then Result := Result + [fsBold];
  if AsBoolean[3] then Result := Result + [fsItalic];
  if AsBoolean[4] then Result := Result + [fsUnderline];
  if AsBoolean[5] then Result := Result + [fsStrikeOut];
end;

procedure TCSIFont.SetFontStyle(const Value: TFontStyles);
begin
  AsBoolean[2] := fsBold      IN Value;
```

```
  AsBoolean[3] := fsItalic    IN Value;
  AsBoolean[4] := fsUnderline IN Value;
  AsBoolean[5] := fsStrikeOut IN Value;
end;

//.....................................................................
{ TSectionList }
//.....................................................................

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
constructor TSectionList.Create;
begin
  inherited Create;
  FPrevIndex    := 0;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
function TSectionList.GetSectionItems(Index: Integer): TStringList;
begin
  Result := TStringList(Objects[Index]);
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
function TSectionList.EraseDuplicates(callBackProc:TEraseSectionCallback = nil) :
Boolean;
var
  slDuplicateTracking   : TStringList;
  idxToDelete,
  ixLow,
  ixHigh,
  ix                    : Integer;

  { swap two integer variables }
  procedure SwapInt(var a,b:Integer);
  var
    c : Integer;
  begin
    c := a;
    a := b;
    b := c;
  end;
begin
  Result := False; { no changes made yet }

  if Count > 1 then
  begin
    slDuplicateTracking := TStringList.Create;
    slDuplicateTracking.Assign(Self);
    { store current position in the objects field: }
    for ix := 0 to slDuplicateTracking.Count-1 do slDuplicateTracking.Objects[ix] :=
```

```
Pointer(ix);
    { sort the list to find out duplicates }
    slDuplicateTracking.Sort;
    ixLow := 0;
    for ix := 1 to slDuplicateTracking.Count-1 do
    begin
      if (AnsiCompareText(slDuplicateTracking.Strings[ixLow],
                          slDuplicateTracking.Strings[ix]) <> 0) then
      begin
        ixLow := ix;
      end else
      begin
        ixHigh := ix;
        { find the previous entry (with lower integer number) }
        if Integer(slDuplicateTracking.Objects[ixLow]) >
           Integer(slDuplicateTracking.Objects[ixHigh]) then SwapInt(ixHigh,ixLow);

        if Assigned(callBackProc) then
        begin
          { ask callback/user wether to delete the higher (=true)
            or the lower one (=false)}
          if NOT callBackProc(slDuplicateTracking.Strings[ix],

SectionItems[Integer(slDuplicateTracking.Objects[ixLow])],

SectionItems[Integer(slDuplicateTracking.Objects[ixHigh])])
              then SwapInt(ixHigh,ixLow);
        end;
        idxToDelete := Integer(slDuplicateTracking.Objects[ixHigh]);

        { free associated object and mark it as unassigned }
        SectionItems[idxToDelete].Free;
        Objects[idxToDelete] := nil;
        Result := True; { list had been changed }
      end {if};
    end {for};

    ix := 0;
    while ix < Count do
    begin
      if Objects[ix] = nil then Delete(ix)
                           else Inc(ix);
    end;
    slDuplicateTracking.Free;
  end {if};

end;

// . . . . . .
function TSectionList.IndexOf(const S: string): Integer;
```

```
var
  ix,
  LastIX        : Integer;
  { This routine doesn't search from the first item each time,
    but from the last successful item. It is likely that the
    next item is to be found downward. }
begin
  Result := -1;
  if Count = 0 then Exit;

  LastIX := FPrevIndex;
  { Search from last successful point to the end: }
  for ix := LastIX to Count-1 do
  begin
    if (AnsiCompareText(Get(ix), S) = 0) then begin
      Result     := ix;
      FPrevIndex := ix;
      Exit;
    end;
  end;
  { Not found yet? Search from beginning to last successful point: }
  for ix := 0 to LastIX-1 do
  begin
    if (AnsiCompareText(Get(ix), S) = 0) then begin
      Result     := ix;
      FPrevIndex := ix;
      Exit;
    end;
  end;
end;

// . . . . . .

function TSectionList.IndexOfName(const Name: string): Integer;
var
  P: Integer;
  s1,
  s2 : AnsiString;
begin
  s2 := Name;
  for Result := 0 to GetCount - 1 do
  begin
    s1 := Get(Result);
    P := AnsiPos('=', s1);
    SetLength(s1,P-1);
    if (P <> 0) AND (
    CompareString(LOCALE_USER_DEFAULT, NORM_IGNORECASE,
      PChar(s1), -1,
      PChar(s2), -1)
      = 2) then Exit;
```

```
  end;
  Result := -1;
end;

//.....................................................................
{ TBigIniFile }
//.....................................................................

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
constructor TBigIniFile.Create(const FileName: string);
begin
  FSectionList    := TSectionList.Create;
  FTextBufferSize := IniTextBufferSize;   { you may set to zero to switch off }
  FFlagDropCommentLines := False;         { change this aDefaults if needed }
  FFlagFilterOutInvalid := False;
  FFlagDropWhiteSpace   := False;
  FFlagDropApostrophes  := False;
  FFlagTrimRight        := False;
  FFlagClearOnReadSectionValues := False;
  FFileName             := '';
  FPrevSectionIndex     := 0;
  FEraseSectionCallback := nil;
  SetFileName(FileName);
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
destructor TBigIniFile.Destroy;
begin
  FlushFile;
  ClearSectionList;
  FSectionList.Free;
  inherited Destroy;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
// clean up
procedure TBigIniFile.ClearSectionList;
var
  ixSections      : Integer;
begin
  with FSectionList do
  begin
    for ixSections := 0 to Count -1 do
    begin
      SectionItems[ixSections].Free;
    end;
    Clear;
    FPrevIndex := 0;
  end;
end;
```

```
// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBigIniFile.Clear;
begin
  ClearSectionList;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBigIniFile.AppendFromFile(const aName : string);
var
  CurrStringList  : TStringList;
  CurrSectionName : string;
  lpTextBuffer    : Pointer;
  Source          : TextFile;
  OneLine         : string;
  LL              : Integer;
  LastPos,
  EqPos           : Integer;
  nospace         : Boolean;
begin
  CurrStringList  := nil;
  lpTextBuffer    := nil; {only to avoid compiler warnings}
  FPrevSectionIndex := 0;
  if FileExists(aName) then
  begin
    Assign      (Source,aName);
    if FTextBufferSize > 0 then
    begin
      GetMem(lpTextBuffer,FTextBufferSize);
      SetTextBuf(Source,lpTextBuffer^,FTextBufferSize);
    end;
    Reset       (Source);
    while NOT Eof(Source) do
    begin
      ReadLn(Source,OneLine);
      if OneLine = #$1A {EOF} then OneLine := '';
      { drop lines with leading ';' : }
      if FFlagDropCommentLines then if (OneLine <> '') then if (OneLine[1] = ';')
then OneLine := '';
      { drop lines without '=' }
      if OneLine <> '' then begin
        LL := Length(OneLine);
        if (OneLine[1] = '[') AND (OneLine[LL] = ']') then
        begin
          CurrSectionName := Copy(OneLine,2,LL-2);
          CurrStringList  := TStringList.Create;
          FSectionList.AddObject(CurrSectionName,CurrStringList);
        end
        else begin
          if FFlagDropWhiteSpace then
```

```
          begin
            nospace := False;
            repeat
              { delete white space left to equal sign }
              EqPos := AnsiPos('=', OneLine);
              if EqPos > 1 then begin
                if OneLine[EqPos - 1] IN [' ', #9] then
                  Delete(OneLine, EqPos - 1, 1)
                else
                  nospace := True;
              end
              else
                nospace := True;
            until nospace;
            nospace := False;
            EqPos := AnsiPos('=', OneLine);
            if EqPos > 1 then begin
              repeat
                { delete white space right to equal sign }
                if EqPos < Length(OneLine) then begin
                  if OneLine[EqPos + 1] IN [' ', #9] then
                    Delete(OneLine, EqPos + 1, 1)
                  else
                    nospace := True;
                end
                else
                  nospace := True;
              until nospace;
            end;
          end; {FFlagDropWhiteSpace}
          if FFlagDropApostrophes then
          begin
            EqPos := AnsiPos('=', OneLine);
            if EqPos > 1 then begin
              LL := Length(OneLine);
              { filter out the apostrophes }
              if EqPos < LL - 1 then begin
                if (OneLine[EqPos + 1] = OneLine[LL]) AND (OneLine[LL] IN ['"',
#39]) then
                begin
                  Delete(OneLine, LL, 1);
                  Delete(OneLine, EqPos + 1, 1);
                end;
              end;
            end;
          end; {FFlagDropApostrophes}
          if FFlagTrimRight then
          begin
            LastPos := Length(OneLine);
            while ((LastPos > 0) AND (OneLine[LastPos] < #33)) do Dec(LastPos);
```

```
            OneLine := Copy(OneLine,1,LastPos);
          end; {FFlagTrimRight}
          if (NOT FFlagFilterOutInvalid) OR (AnsiPos('=', OneLine) > 0) then
          begin
            if Assigned(CurrStringList) then CurrStringList.Add(OneLine);
          end;
        end;
      end;
    end;

    if FSectionList.EraseDuplicates(FEraseSectionCallback) then FHasChanged := True;

    Close(Source);
    if FTextBufferSize > 0 then
    begin
      FreeMem(lpTextBuffer,FTextBufferSize);
    end;
  end;
end;

// . . . . . .
procedure TBigIniFile.SetFileName(const aName : string);
begin
  FlushFile;
  ClearSectionList;
  FFileName       := aName;
  if aName <> '' then AppendFromFile(aName);
  FHasChanged     := False;
end;

// . . . . . .
// find item in specified section
// depending on CreateNew-flag, the section is created, if not existing
function TBigIniFile.FindItemIndex(const aSection, aKey :string; CreateNew:Boolean;
          var FoundStringList:TStringList):Integer;
var
  SectionIndex    : Integer;
  LastIX          : Integer;
begin
  SectionIndex := -1;

  if FSectionList.Count > 0 then
  begin
    LastIX := FPrevSectionIndex -1;
    if LastIX < 0 then LastIX := FSectionList.Count -1;
    while (AnsiCompareText(aSection,FSectionList[FPrevSectionIndex]) <> 0)
      AND (FPrevSectionIndex <> LastIX) do
    begin
      Inc(FPrevSectionIndex);
      if FPrevSectionIndex = FSectionList.Count then FPrevSectionIndex := 0;
```

```
    end;
    if AnsiCompareText(aSection,FSectionList[FPrevSectionIndex]) = 0 then
    begin
      SectionIndex := FPrevSectionIndex;
    end;
  end;

  if SectionIndex = -1 then
  begin
    if CreateNew then begin
      FoundStringList   := TStringList.Create;
      FPrevSectionIndex := FSectionList.AddObject(aSection,FoundStringList);
    end
    else begin
      FoundStringList := nil;
    end;
    Result := -1;
  end
  else begin
    FoundStringList := FSectionList.SectionItems[SectionIndex];
    Result := FoundStringList.IndexOfName(aKey);
  end;
end;

// . . . . . .
function TBigIniFile.ReadString(const aSection, aKey, aDefault: string): string;
var
  ItemIndex       : Integer;
  CurrStringList  : TStringList;
begin
  ItemIndex := FindItemIndex(aSection,aKey,False,CurrStringList);
  if ItemIndex = -1 then
  begin
    Result := aDefault
  end
  else begin
    Result := Copy(CurrStringList[ItemIndex], Length(aKey) + 2, MaxInt);
  end;
end;

// . . . . . .
function TBigIniFile.ReadAnsiString(const aSection, aKey, aDefault: string):
AnsiString;
var
  ItemIndex       : Integer;
  CurrStringList  : TStringList;
begin
  ItemIndex := FindItemIndex(aSection,aKey,False,CurrStringList);
  if ItemIndex = -1 then
  begin
```

```
    Result := aDefault
  end
  else begin
    Result := CurrStringList.Values[aKey];
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteString(const aSection, aKey, aValue: string);
var
  ItemIndex       : Integer;
  CurrStringList  : TStringList;
  newLine         : string;
begin
  if aKey = '' then
  begin
    {behaviour of WritePrivateProfileString: if all parameters are null strings,
     the file is flushed to disk. Otherwise, if key name is a null string,
     the entire Section is to be deleted}
    if (aSection = '') AND (aValue = '') then FlushFile
                                         else EraseSection(aSection);
  end
  else begin
    newLine := aKey+'='+aValue;
    ItemIndex := FindItemIndex(aSection,aKey,True,CurrStringList);
    if ItemIndex = -1 then
    begin
      CurrStringList.Add(newLine);
      FHasChanged   := True;
    end
    else begin
      if (CurrStringList[ItemIndex] <> newLine) then
      begin
        FHasChanged := True;
        CurrStringList[ItemIndex] := newLine;
      end;
    end;
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteAnsiString(const aSection, aKey, aValue: AnsiString);
var
  ItemIndex       : Integer;
  CurrStringList  : TStringList;
  newLine         : AnsiString;
begin
  if aKey = '' then
  begin
    {behaviour of WritePrivateProfileString: if all parameters are null strings,
```

```
     the file is flushed to disk. Otherwise, if key name is a null string,
     the entire Section is to be deleted}
    if (aSection = '') AND (aValue = '') then FlushFile
                                         else EraseSection(aSection);
  end
  else begin
    newLine := aKey+'='+aValue;
    ItemIndex := FindItemIndex(aSection,aKey,True,CurrStringList);
    if ItemIndex = -1 then begin
      CurrStringList.Add(newLine);
      FHasChanged   := True;
    end
    else begin
      if (CurrStringList[ItemIndex] <> newLine) then
      begin
        FHasChanged := True;
        CurrStringList[ItemIndex] := newLine;
      end;
    end;
  end;
end;

// . . . . . .
function TBigIniFile.ReadInteger(const aSection, aKey: string;
  aDefault: Longint): Longint;
var
  IStr: string;
begin
  IStr := ReadString(aSection, aKey, '');
  if CompareText(Copy(IStr, 1, 2), '0x') = 0 then
    IStr := '$' + Copy(IStr, 3, 255);
  Result := StrToIntDef(IStr, aDefault);
end;

// . . . . . .
procedure TBigIniFile.WriteInteger(const aSection, aKey: string; aValue: Longint);
begin
  WriteString(aSection, aKey, IntToStr(aValue));
end;

// . . . . . .
function TBigIniFile.ReadBool(const aSection, aKey: string;
  aDefault: Boolean): Boolean;
begin
  Result := ReadInteger(aSection, aKey, Ord(aDefault)) <> 0;
end;

// . . . . . .
procedure TBigIniFile.WriteBool(const aSection, aKey: string; aValue: Boolean);
const
```

```
  BoolText: array[Boolean] of string[1] = ('0', '1');
begin
  WriteString(aSection, aKey, BoolText[aValue]);
end;

// . . . . . .
function TBigIniFile.ReadDate(const aSection, aKey: string; aDefault: TDateTime):
TDateTime;
var
  DateStr: string;
begin
  DateStr := ReadString(aSection, aKey, '');
  Result  := aDefault;
  if DateStr <> '' then
  try
    Result := StrToDate(DateStr);
  except
    on EConvertError do
    else raise;
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteDate(const aSection, aKey: string; aValue: TDateTime);
begin
  WriteString(aSection, aKey, DateToStr(aValue));
end;

// . . . . . .
function TBigIniFile.ReadDateTime(const aSection, aKey: string; aDefault:
TDateTime): TDateTime;
var
  DateStr: string;
begin
  DateStr := ReadString(aSection, aKey, '');
  Result := aDefault;
  if DateStr <> '' then
  try
    Result := StrToDateTime(DateStr);
  except
    on EConvertError do
    else raise;
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteDateTime(const aSection, aKey: string; aValue:
TDateTime);
begin
  WriteString(aSection, aKey, DateTimeToStr(aValue));
```

```
end;

// . . . . . .
function TBigIniFile.ReadFloat(const aSection, aKey: string; aDefault: Double):
Double;
var
  FloatStr: string;
begin
  FloatStr := ReadString(aSection, aKey, '');
  Result := aDefault;
  if FloatStr <> '' then
  try
    Result := StrToFloat(FloatStr);
  except
    on EConvertError do
    else raise;
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteFloat(const aSection, aKey: string; aValue: Double);
begin
  WriteString(aSection, aKey, FloatToStr(aValue));
end;

// . . . . . .
function TBigIniFile.ReadTime(const aSection, aKey: string; aDefault: TDateTime):
TDateTime;
var
  TimeStr: string;
begin
  TimeStr := ReadString(aSection, aKey, '');
  Result := aDefault;
  if TimeStr <> '' then
  try
    Result := StrToTime(TimeStr);
  except
    on EConvertError do
    else raise;
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteTime(const aSection, aKey: string; aValue: TDateTime);
begin
  WriteString(aSection, aKey, TimeToStr(aValue));
end;

// . . . . . .
function TBigIniFile.ReadBinaryStream(const aSection, aKey: string; aStream:
```

```
TStream): Integer;
var
  ix            : Integer;
  hexDump       : AnsiString;
  oneByte       : Byte;
begin
  hexDump := ReadAnsiString(aSection,aKey,'');
  Result  := Length(hexDump) DIV 2;
  for ix := 0 to Result -1 do
  begin
    OneByte := StrToIntDef('$' + Copy(hexDump,1 + ix*2,2) ,0);
    aStream.Write(oneByte,1);
  end;
end;

// . . . . . .
procedure TBigIniFile.WriteBinaryStream(const aSection, aKey: string; aStream:
TStream);
var
  ix            : Integer;
  bufPtr        : PAnsiChar;
  hexDump       : AnsiString;
  bufSize       : Integer;
begin
  bufSize := 512;
  getMem(bufPtr,bufSize);
  hexDump := '';
  bufSize := aStream.Read(bufPtr^,bufSize);
  while bufSize > 0 do begin
    for ix := 0 to BufSize-1 do
    begin
      hexDump := hexDump + AnsiString(IntToHex(Byte(bufPtr[ix]),2));
    end;
    bufSize := aStream.Read(bufPtr^,bufSize);
  end;
  WriteAnsiString(aSection,aKey,hexDump);
  freeMem(bufPtr);
end;

// . . . . . .
procedure TBigIniFile.ReadSection(const aSection: string; aStrings: TStrings);
var
  SectionIndex    : Integer;
  CurrStringList  : TStringList;
  ix              : Integer;
begin
  SectionIndex := FSectionList.IndexOf(aSection);
  if SectionIndex <> -1 then
  begin
    aStrings.BeginUpdate;
```

```
    CurrStringList := FSectionList.SectionItems[SectionIndex];
    for ix := 0 to CurrStringList.Count - 1 do
    begin
      if CurrStringList.Names[ix] = '' then Continue;
      if FFlagDropCommentLines AND (CurrStringList.Names[ix][1] = ';') then
Continue;
      aStrings.Add(CurrStringList.Names[ix]);
    end;
    aStrings.EndUpdate;
  end;
end;

// . . . . . .
procedure TBigIniFile.ReadSections(aStrings: TStrings);
begin
  aStrings.Assign(FSectionList);
end;

// . . . . . .
procedure TBigIniFile.ReadSectionValues(const aSection: string; aStrings: TStrings);
var
  SectionIndex    : Integer;
begin
  SectionIndex := FSectionList.IndexOf(aSection);
  if SectionIndex <> -1 then
  begin
    aStrings.BeginUpdate;
    {In prior versions of TIniFile the target-Strings were _not_ cleared
    That's why my procedure didn't either. Meanwhile, Borland changed their
    mind and I added the following line for D5 compatibility.
    Use FFlagClearOnReadSectionValues if needed}
    if FFlagClearOnReadSectionValues then aStrings.Clear; // new since 3.09,3.10
    aStrings.AddStrings(FSectionList.SectionItems[SectionIndex]);
    aStrings.EndUpdate;
  end;
end;

// . . . . . .
procedure TBigIniFile.GetStrings(List: TStrings);
var
  ixSections      : Integer;
  CurrStringList  : TStringList;
begin
  List.BeginUpdate;
  with FSectionList do
  begin
    for ixSections := 0 to Count -1 do
    begin
      CurrStringList := SectionItems[ixSections];
      if CurrStringList.Count > 0 then
```

```
      begin
        List.Add('['+Strings[ixSections]+']');
        List.AddStrings(CurrStringList);
        List.Add('');
      end;
    end;
  end;
  List.EndUpdate;
end;

// . . . . . .
procedure TBigIniFile.SetStrings(const aStrings: TStrings);
var
  CurrStringList  : TStringList;
  CurrSectionName : string;
  OneLine         : string;
  LL              : Integer;
  LastPos,
  EqPos           : Integer;
  nospace         : Boolean;
  ix              : Integer;
begin
  CurrStringList  := nil;
  FPrevSectionIndex := 0;
  for ix := 0 to aStrings.Count -1 do
  begin
    OneLine := aStrings.Strings[ix];
    { drop lines with leading ';' : }
    if FFlagDropCommentLines then if (OneLine <> '') then if (OneLine[1] = ';') then
OneLine := '';
    { drop lines without '=' }
    if OneLine <> '' then begin
      LL := Length(OneLine);
      if (OneLine[1] = '[') AND (OneLine[LL] = ']') then
      begin
        CurrSectionName := Copy(OneLine,2,LL-2);
        CurrStringList  := TStringList.Create;
        FSectionList.AddObject(CurrSectionName,CurrStringList);
      end
      else begin
        if FFlagDropWhiteSpace then
        begin
          nospace := False;
          repeat
            { delete white space left to equal sign }
            EqPos := AnsiPos('=', OneLine);
            if EqPos > 1 then begin
              if OneLine[EqPos - 1] IN [' ', #9] then
                Delete(OneLine, EqPos - 1, 1)
              else
```

```
              nospace := True;
            end
            else
              nospace := True;
          until nospace;
          nospace := False;
          EqPos := AnsiPos('=', OneLine);
          if EqPos > 1 then begin
            repeat
              { delete white space right to equal sign }
              if EqPos < Length(OneLine) then begin
                if OneLine[EqPos + 1] IN [' ', #9] then
                  Delete(OneLine, EqPos + 1, 1)
                else
                  nospace := True;
              end
              else
                nospace := True;
            until nospace;
          end;
        end; {FFlagDropWhiteSpace}
        if FFlagDropApostrophes then
        begin
          EqPos := AnsiPos('=', OneLine);
          if EqPos > 1 then begin
            LL := Length(OneLine);
            { filter out the apostrophes }
            if EqPos < LL - 1 then begin
              if (OneLine[EqPos + 1] = OneLine[LL]) AND (OneLine[LL] IN ['"', #39])
then
              begin
                Delete(OneLine, LL, 1);
                Delete(OneLine, EqPos + 1, 1);
              end;
            end;
          end;
        end; {FFlagDropApostrophes}
        if FFlagTrimRight then
        begin
          LastPos := Length(OneLine);
          while ((LastPos > 0) AND (OneLine[LastPos] < #33)) do Dec(LastPos);
          OneLine := Copy(OneLine,1,LastPos);
        end; {FFlagTrimRight}
        if (NOT FFlagFilterOutInvalid) OR (AnsiPos('=', OneLine) > 0) then
        begin
          if Assigned(CurrStringList) then CurrStringList.Add(OneLine);
        end;
      end;
    end;
  end;
```

```
  if FSectionList.EraseDuplicates(FEraseSectionCallback) then FHasChanged := True;
end;

// . . . . . .
procedure TBigIniFile.FlushFile;
var
  CurrStringList  : TStringList;
  lpTextBuffer    : Pointer;
  Destin          : TextFile;
  ix,
  ixSections      : Integer;
begin
  lpTextBuffer    := nil; {only to avoid compiler warnings}
  if FHasChanged then
  begin
    if FFileName <> '' then
    begin
      Assign     (Destin,FFileName);
      if FTextBufferSize > 0 then
      begin
        GetMem(lpTextBuffer,FTextBufferSize);
        SetTextBuf (Destin,lpTextBuffer^,FTextBufferSize);
      end;
      Rewrite    (Destin);

      with FSectionList do
      begin
        for ixSections := 0 to Count -1 do
        begin
          CurrStringList := SectionItems[ixSections];
          if CurrStringList.Count > 0 then
          begin
            WriteLn(Destin,'[',Strings[ixSections],']');
            for ix := 0 to CurrStringList.Count -1 do
            begin
              WriteLn(Destin,CurrStringList[ix]);
            end;
            WriteLn(Destin);
          end;
        end;
      end;

      Close(Destin);
      if FTextBufferSize > 0 then
      begin
        FreeMem(lpTextBuffer,FTextBufferSize);
      end;
    end;
    FHasChanged   := False;
  end;
```

```
end;

// . . . . . .
procedure TBigIniFile.UpdateFile;
begin
  FlushFile;
end;

// . . . . . .
procedure TBigIniFile.EraseSection(const aSection: string);
var
  SectionIndex    : Integer;
begin
  SectionIndex := FSectionList.IndexOf(aSection);
  if SectionIndex <> -1 then
  begin
    FSectionList.SectionItems[SectionIndex].Free;
    FSectionList.Delete(SectionIndex);
    FSectionList.FPrevIndex  := 0;
    FHasChanged := True;
    if FPrevSectionIndex >= FSectionList.Count then FPrevSectionIndex := 0;
  end;
end;

// . . . . . .
procedure TBigIniFile.DeleteKey(const aSection, aKey: string);
var
  ItemIndex       : Integer;
  CurrStringList  : TStringList;
begin
  ItemIndex := FindItemIndex(aSection,aKey,True,CurrStringList);
  if ItemIndex > -1 then begin
    FHasChanged := True;
    CurrStringList.Delete(ItemIndex);
  end;
end;

// . . . . . .
function TBigIniFile.SectionExists(const aSection: string): Boolean;
begin
  Result := (FSectionList.IndexOf(aSection) > -1)
end;

// . . . . . .
function TBigIniFile.ValueExists(const aSection, aKey: string): Boolean;
var
  S: TStringList;
begin
  S := TStringList.Create;
  try
```

```
    ReadSection(aSection, S);
    Result := S.IndexOf(aKey) > -1;
  finally
    S.Free;
  end;
end;

//.....................................................................
{ class TBiggerIniFile                                                }
//.....................................................................

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBiggerIniFile.setTextBufferSize(const Value: Integer);
begin
  if Value > $7FFF then FTextBufferSize := $7FFF
                   else FTextBufferSize := Value;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBiggerIniFile.WriteSectionValues(const aSection: string; const aStrings:
TStrings);
var
  SectionIndex    : Integer;
  FoundStringList : TStringList;
  ix              : Integer;
begin
  SectionIndex := FSectionList.IndexOf(aSection);
  if SectionIndex = -1 then
  begin
    { create new section }
    FoundStringList := TStringList.Create;
    FSectionList.AddObject(aSection,FoundStringList);
    FoundStringList.AddStrings(aStrings);
    FHasChanged := True;
  end
  else begin
    { compare existing section }
    FoundStringList := FSectionList.SectionItems[SectionIndex];
    if FoundStringList.Count <> aStrings.Count then
    begin
      { if count differs, replace complete section }
      FoundStringList.Clear;
      FoundStringList.AddStrings(aStrings);
      FHasChanged := True;
    end
    else begin
      { compare line by line }
      for ix := 0 to FoundStringList.Count - 1 do
      begin
        if FoundStringList[ix] <> aStrings[ix] then
```

```
        begin
          FoundStringList[ix] := aStrings[ix];
          FHasChanged := True;
        end;
      end;
    end;
  end;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBiggerIniFile.ReadNumberedList(const aSection: string;
                                    aStrings: TStrings;
                                    aDefault: string;
                                    aPrefix: string = '';
                                    IndexStart: Integer = 1;
                                    usePrefixOnCount: Boolean =
cDefaultUsePrefixOnCount);
var
  maxEntries            : Integer;
  ix                    : Integer;
  countKey              : string;
begin
  if usePrefixOnCount then countKey := aPrefix + cIniCount
                      else countKey := cIniCount;
  maxEntries := ReadInteger(aSection,countKey,0);
  aStrings.BeginUpdate;
  for ix := 0 to maxEntries -1 do begin
    aStrings.Add(ReadString(aSection,aPrefix+IntToStr(ix+IndexStart),aDefault));
  end;
  aStrings.EndUpdate;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBiggerIniFile.WriteNumberedList(const aSection: string;
                                        aStrings: TStrings;
                                        aPrefix: string = '';
                                        IndexStart: Integer = 1;
                                        usePrefixOnCount: Boolean =
cDefaultUsePrefixOnCount);
var
  prevCount,
  ix                    : Integer;
  prevHasChanged        : Boolean;
  oldSectionValues,
  newSectionValues      : TStringList;
  countKey              : string;
begin
  oldSectionValues      := TStringList.Create;
  newSectionValues      := TStringList.Create;
```

```
  if usePrefixOnCount then countKey := aPrefix + cIniCount
                      else countKey := cIniCount;

  try
    { store previous entries }
    ReadSectionValues(aSection,oldSectionValues);

    prevCount := ReadInteger(aSection,countKey,0);
    WriteInteger(aSection,countKey,aStrings.Count);
    prevHasChanged := HasChanged;

    { remove all previous lines to get new ones together }
    for ix := 0 to prevCount-1 do begin
      DeleteKey(aSection,aPrefix+IntToStr(ix+IndexStart));
    end;
    for ix := 0 to aStrings.Count -1 do begin
      WriteString(aSection,aPrefix+IntToStr(ix+IndexStart),aStrings[ix]);
    end;

    { check if entries really had changed }
    if NOT prevHasChanged then
    begin
      { read new entries and compare with old }
      ReadSectionValues(aSection,newSectionValues);
      HasChanged := NOT newSectionValues.Equals(oldSectionValues);
    end;
  finally
    oldSectionValues.Free;
    newSectionValues.Free;
  end;
end;

// . . . . . .
procedure TBiggerIniFile.EraseNumberedList(const aSection: string;
                                           aPrefix: string;
                                           IndexStart: Integer;
                                           usePrefixOnCount: Boolean);
var
  prevCount,
  ix                    : Integer;
  countKey              : string;
begin
  if usePrefixOnCount then countKey := aPrefix + cIniCount
                      else countKey := cIniCount;

  try
    prevCount := ReadInteger(aSection,countKey,-1);
    if prevCount >= 0 then
    begin
      HasChanged := True;
```

```
      DeleteKey(aSection,countKey);
      { remove all previous lines }
      for ix := 0 to prevCount-1 do begin
        DeleteKey(aSection,aPrefix+IntToStr(ix+IndexStart));
      end;
    end;
  finally
  end;
end;

// . . . . . .
function TBiggerIniFile.ReadColor(const aSection, aKey: string; aDefault: TColor):
TColor;
begin
  ReadColor  := StrToInt('$'+ReadString(aSection,aKey,IntToHex(aDefault,8)));
end;

// . . . . . .
procedure TBiggerIniFile.WriteColor(const aSection, aKey: string; aValue: TColor);
begin
  WriteString(aSection,aKey,IntToHex(aValue,8));
end;

// . . . . . .
function TBiggerIniFile.ReadFont(const aSection, aKey: string; aDefault: TFont):
TFont;
begin
  with TCSIFont.Create do begin
    Value := ReadString(aSection,aKey,'');
    if (Value <> '') then
    begin
      aDefault.Name    := FontName;
      aDefault.Size    := FontSize;
      aDefault.Style   := FontStyle;
      aDefault.Color   := FontColor;
      aDefault.Charset := FontCharset;
      aDefault.Pitch   := TFontPitch(FontPitch);
    end;
    Free;
  end;
  Result := aDefault;
end;

// . . . . . .
procedure TBiggerIniFile.WriteFont(const aSection, aKey: string; aFont: TFont);
begin
  with TCSIFont.Create do begin
    FontName    := aFont.Name;
    FontSize    := aFont.Size;
    FontStyle   := aFont.Style;
```

```
    FontColor   := aFont.Color;
    FontCharset := aFont.Charset;
    FontPitch   := Ord(aFont.Pitch);
    WriteString(aSection,aKey,Value);
    Free;
  end;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
{ read a Point's properties                                                    }
// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
function TBiggerIniFile.ReadPoint(const aSection, aKey: string; aDefault: TPoint):
TPoint;
begin
  with TCSIPoint.Create do begin
    Value := ReadString(aSection,aKey,'');
    if (Value <> '') then
    begin
      aDefault.X    := X;
      aDefault.Y    := Y;
    end;
    Free;
  end;
  Result := aDefault;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
procedure TBiggerIniFile.WritePoint(const aSection, aKey: string; aPoint: TPoint);
begin
  with TCSIPoint.Create do begin
    X  := aPoint.X;
    Y  := aPoint.Y;
    WriteString(aSection,aKey,Value);
    Free;
  end;
end;

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
function TBiggerIniFile.ReadRect(const aSection, aKey: string; aDefault: TRect):
TRect;
begin
  with TCSIRect.Create do begin
    Value := ReadString(aSection,aKey,'');
    if (Value <> '') then
    begin
      aDefault.Left    := Left;
      aDefault.Top     := Top;
      aDefault.Right   := Right;
      aDefault.Bottom  := Bottom;
    end;
```

```
    Free;
  end;
  Result := aDefault;
end;

// . . . . . .
procedure TBiggerIniFile.WriteRect(const aSection, aKey: string; aRect: TRect);
begin
  with TCSIRect.Create do begin
    Left    := aRect.Left;
    Top     := aRect.Top;
    Right   := aRect.Right;
    Bottom  := aRect.Bottom;
    WriteString(aSection,aKey,Value);
    Free;
  end;
end;

// . . . . . .
function TBiggerIniFile.SectionCount: Integer;
begin
  result := FSectionList.Count;
end;

// . . . . . .
procedure TBiggerIniFile.RenameSection(const OldSection, NewSection : string);
var
  SectionIndex    : Integer;
begin
  if NewSection <> OldSection then
  begin
    SectionIndex := FSectionList.IndexOf(OldSection);
    if SectionIndex <> -1 then
    begin
      FSectionList[SectionIndex] := NewSection;
    end;
    FHasChanged := True;
  end;
end;

// . . . . . .
procedure TBiggerIniFile.RenameKey(const aSection, OldKey, NewKey : string);
var
  ItemIndex       : Integer;
  CurrStringList  : TStringList;
begin
  if NewKey <> OldKey then
  begin
    ItemIndex := FindItemIndex(aSection,OldKey,False,CurrStringList);
    if ItemIndex <> -1 then
```

```
    begin
      WriteString(aSection,NewKey,ReadString(aSection,OldKey,''));
      DeleteKey(aSection, OldKey);
    end;
  end;
end;

// . . . . . .
function TBiggerIniFile.ReadBinaryData(const aSection, aKey: string; var Buffer;
BufSize: Integer): Integer;
var
  ix            : Integer;
  bufPtr        : PAnsiChar;
  hexDump       : AnsiString;
begin
  hexDump := ReadAnsiString(aSection,aKey,'');
  Result  := Length(hexDump) DIV 2;
  if Result > BufSize then Result := BufSize;

  bufPtr  := Pointer(Buffer);
  for ix := 0 to Result -1 do
  begin
    Byte(bufPtr[ix]) := StrToIntDef('$' + Copy(hexDump,1 + ix*2,2) ,0);
  end;
end;

// . . . . . .
procedure TBiggerIniFile.WriteBinaryData(const aSection, aKey: string; var Buffer;
BufSize: Integer);
var
  ix            : Integer;
  bufPtr        : PAnsiChar;
  hexDump       : AnsiString;
begin
  hexDump := '';
  bufPtr  := Pointer(Buffer);
  for ix := 0 to BufSize-1 do
  begin
    hexDump := hexDump + AnsiString(IntToHex(Byte(bufPtr[ix]),2));
  end;
  WriteAnsiString(aSection,aKey,hexDump);
end;

// . . . . . .
procedure TBiggerIniFile.WriteBoolDef(const aSection, aKey: string; aValue: Boolean;
const aDefault: Boolean);
begin
  if (aValue <> aDefault) then WriteBool(aSection,aKey,aValue)
                          else DeleteKey(aSection,aKey);
end;
```

```
// . . . . . .
procedure TBiggerIniFile.LaunchInEditor;
begin
  FlushFile;
  if FileExists(FileName) then
  begin
    ShellExecute(0,
                 'open',
                 PChar(FileName),
                 nil,
                 nil,
                 sw_ShowNormal);
  end;
end;

//.....
{ class TAppIniFile                                                    }
//.....
constructor TAppIniFile.Create;
begin
  inherited Create(DefaultFileName);
end;

// . . . . . .
class function TAppIniFile.DefaultFileName: string;
begin
  Result := ChangeFileExt(ModuleName(False),'.ini');
end;

//.....
{ class TLibIniFile                                                    }
//.....

class function TLibIniFile.DefaultFileName: string;
begin
  Result := ChangeFileExt(ModuleName(True),'.ini');
end;

end.
```