Welcome to HyperString!

HyperString is a comprehensive and diverse library with well over 200 routines designed to exploit the full potential of the versatile new 32-bit long dynamic string type.   Full source code is available.

You must read the license agreement before using this product.

Math

function EnCodeBCD(**const** Source:AnsiString):AnsiString;
function DeCodeBCD(**const** Source:AnsiString):AnsiString;
function AddUSI(**const** X,Y:Integer):Integer;
function SubUSI(**const** X,Y:Integer):Integer;
function MulUSI(**const** X,Y:Integer):Integer;
function DivUSI(**const** X,Y:Integer):Integer;
function ModUSI(**const** X,Y:Integer):Integer;
function CmpUSI(**const** X,Y:Integer):Integer;
function USIToStr(**const** X:Integer):AnsiString;
function StrToUSI(**const** Source:AnsiString):Integer;

Binary Coded Decimal (BCD) conversion and unsigned integer (USI) math.

Note: The fact that the unsigned *Cardinal* type can only hold 31-bits leaves Delphi programmers without any true, native 32-bit unsigned integer type. A standard *Integer* can *hold* a full 32-bits; however, the value is always treated as signed. The above functions are designed to fill this gap by providing for *unsigned* manipulation of standard integer types. For the sake of clarity, a "u" naming prefix is recommended for integers being used as unsigned containers.

To initialize an integer to an unsigned value greater than 2^31, the constant must be provided in either hex or string format (with the help of StrToUSI). For example, the compiler will balk at uI := 4,294,967,295 but will accept uI := $FFFFFFFF or uI := StrToUSI('4,294,967,295').

Contents

API

function GetUser: AnsiString;
function GetNetUser: AnsiString;
function GetComputer: AnsiString;
function GetDrives: AnsiString;
function RemoteDrive(**const** Drv:Char):Boolean;
function GetDisk(**const** Drv:Char; **var** CSize,Available,Total:DWord):Boolean;
function GetVolume(**const** Drv:Char; **var** Name,FSys:AnsiString; **var** S:DWord):Boolean;
function GetWinDir: AnsiString;
function GetSysDir: AnsiString;
function GetTmpDir: AnsiString;
function GetTmpFile(**const** Path,Prefix: AnsiString): AnsiString;
function GetDOSName(**const** LongName: AnsiString): AnsiString;
function GetWindows: AnsiString;
function GetClasses: AnsiString;
function GetWinClass(**const** Title: AnsiString): AnsiString;
function GetCPU:AnsiString;
function GetDefaultPrn:AnsiString;
function IsWinNT:Boolean;
procedure GetMemStatus(**var** RAMTotal,RAMUsed,PGTotal,PGUsed:Integer);
procedure GetComList(Strings: TStrings);
function GetKeyValues(**const** Root:HKey;Key,Values:AnsiString): AnsiString;
function SetTaskBar(**const** Visible:Bool): Boolean;
procedure NoTaskBtn;
procedure KillOLE;
function GetProcID(**const** hWnd:THandle): THandle;
function KillProc(**const** ClassName:AnsiString): Boolean;
function DOSExec(**const** CmdLine:AnsiString; **const** DisplayMode:Integer): Boolean;
function WaitExec(**const** CmdLine:AnsiString; **const** DisplayMode:Integer): Integer;
procedure DebugConsole;
procedure DebugMsg;
function ShellFileOp(**const** S,D:AnsiString; **const** FileOp,Flgs:Integer):Boolean;
function FormatDisk(Drive:Word):Boolean;
procedure TrayInsert;
procedure TrayClose(**var** Action:TCloseAction);
procedure TrayDelete;
function SetAppPriority(**const** Priority:DWord):Boolean;
function GetFileDate(**const** FileName: AnsiString): AnsiString;
function MapNetDrive: Integer;

Contents

Convert

function IntToChr(**const** X: Integer): AnsiString;
function ChrToInt(**const** Src: AnsiString): Integer;
function WordToChr(**const** X: Word): AnsiString;
function ChrToWord(**const** Src: AnsiString): Word;
function SngToChr(**const** X: Single): AnsiString;
function ChrToSng(**const** Src: AnsiString): Single;
function DblToChr(**var** X: Double): AnsiString;
function ChrToDbl(**const** Src: AnsiString): Double;
function CurToChr(**var** X: Currency): AnsiString;
function ChrToCur(**const** Src: AnsiString): Currency;
function BinToInt(**const** Src: AnsiString): Integer;
function IntToBin(**const** X: Integer): AnsiString;
function HexToInt(**const** Src: AnsiString): Integer;
function NumToWord(**const** Src: AnsiString; Dollars: Boolean): AnsiString;
function OrdSuffix(**const** X: Integer): AnsiString;

Note: See Base64 for alternate numeric conversion routines which may be better suited for database use.

Contents

Search

```pascal
function  ScanF(const Src, Search: AnsiString; Start: Integer): Integer;
function  ScanR(const Src, Search: AnsiString; Start: Integer): Integer;
function  ScanC(const Src: AnsiString; X: Char; Start: Integer): Integer;
function  ScanB(const Src: AnsiString; Start: Integer): Integer;
function  ScanL(const Src: AnsiString; Start: Integer): Integer;
function  ScanU(const Src: AnsiString; X: Char; Start: Integer): Integer;
function  ScanCC(const Src: AnsiString; X: Char; Cnt: Integer): Integer;
function  ScanNC(const Src: AnsiString; X: Char): Integer;
function  ScanNB(const Src: AnsiString; X: Char): Integer;
function  ScanT(const Src, Table: AnsiString; Start: Integer): Integer;
function  ScanRT(const Src, Table: AnsiString; Start: Integer): Integer;
function  ScanNT(const Src, Table: AnsiString; Start: Integer): Integer;
function  ScanRNT(const Src, Table: AnsiString; Start: Integer): Integer;
function  ScanP(const Src, Search: AnsiString; var Start: Integer): Integer;
function  ScanW(const Src, Search: AnsiString; var Start: Integer): Integer;
function  ScanQ(const Src, Search: AnsiString; Start: Integer): Integer;
function  ScanQC(const Src, Search: AnsiString; Start: Integer): Integer;
function  ScanZ(const Src, Search: AnsiString; Defects: Integer; var Start: Integer): Integer;
```

Contents

Pad/Trim/Slice

function LTrim(**const** Src: AnsiString; X: Char): AnsiString;
function RTrim(**const** Src: AnsiString; X: Char): AnsiString;
function CTrim(**const** Src: AnsiString; X: Char): AnsiString;
function LStr(**const** Src: AnsiString; Cnt: Integer): AnsiString;
function RStr(**const** Src: AnsiString; Cnt: Integer): AnsiString;
function CStr(**const** Src: AnsiString; Index, Cnt: Integer): AnsiString;
procedure LPad(**var** Src: AnsiString; **const** X: Char; Cnt: Integer);
procedure RPad(**var** Src: AnsiString; **const** X: Char; Cnt: Integer);
procedure CPad(**var** Src: AnsiString; **const** X: Char; Cnt: Integer);
procedure LText(**var** Src: AnsiString);
procedure RText(**var** Src: AnsiString);
procedure CText(**var** Src: AnsiString);

Contents

Tokens

```pascal
function   Parse(const Src, Table: AnsiString; var Index: Integer): AnsiString;
function   ParseWord(const Src, Table: AnsiString; var Index: Integer): AnsiString;
function   ParseTag(const Src, Start, Stop: AnsiString; var Index: Integer): AnsiString;
function   Fetch(const Src, Table: AnsiString; Num: Integer; DelFlg: Bool): AnsiString;
function   GetDelimiter: Char;
function   SetDelimiter(Delimit: Char): Boolean;
function   InsertToken(var Src:AnsiString; const Token: AnsiString; Index: Integer): Boolean;
function   DeleteToken(var Src: AnsiString; var Index: Integer): Boolean;
function   ReplaceToken(var Src, Token: AnsiString; Index: Integer): Boolean;
function   GetToken(const Src: AnsiString; Index: Integer): AnsiString;
function   PrevToken(const Src: AnsiString; var Index: Integer): Boolean;
function   NextToken(const Src: AnsiString; var Index: Integer): Boolean;
function   GetTokenNum(const Src: AnsiString; Index: Integer): Integer;
function   GetTokenPos(const Src: AnsiString; Num: Integer): Integer;
function   GetTokenCnt(const Src: AnsiString): Integer;
```

Definitions:
Token - A properly delimited sub-string (may include null string).
Word - A non-null token.

Contents

Match

function Similar(**const** S1, S2: AnsiString): Integer;
function Soundex(**const** Src: AnsiString): Integer;
function MetaPhone(**const** Name: AnsiString): Integer;

Contents

Count

```
function   CountF(const Src: AnsiString; X: Char;Start: Integer): Integer;
function   CountR(const Src: AnsiString; X: Char; Start: Integer): Integer;
function   CountT(const Src, Table: AnsiString): Integer;
function   CountW(const Src, Table: AnsiString): Integer;
```

Contents

Compression

procedure IniRLE;
function RLE(**const** Bfr: AnsiString; L:Word): AnsiString;
function RLD(**const** Bfr: AnsiString; L:Word): AnsiString;
procedure IniSQZ;
function SQZ(**const** Bfr: AnsiString; L:Word): AnsiString;
function UnSQZ(**const** Bfr: AnsiString; L:Word): AnsiString;
function BPE(**const** Bfr: AnsiString; L:Word): AnsiString;
function BPD(**const** Bfr: AnsiString; L:Word): AnsiString;

Note: Though not exactly state-of-the-art, these routines can provide effective data compression for a wide variety of applications. Simple, easy to use and extremely compact; all of the above routines combined represent less than 5K of code.

Contents

Communicate

function ListComm: AnsiString;
function OpenComm(**const** Mode: AnsiString): THandle;
function ReadComm(**const** pHnd: THandle, **var** Bfr: AnsiString): Integer;
function WriteComm(**const** pHnd: THandle, **const** Bfr: AnsiString): Integer;
function CloseComm(**const** pHnd: Thandle): Boolean;

Note: These functions provide rudimentary support for establishing a communication session under Win32. Once a session has been established, additional setup and configuration (using API functions) will most likely be required for any serious work. See WIN32.HLP for more.

Contents

Test

function IsNum(**const** Src: AnsiString): Boolean;
function IsHex(**const** Src: AnsiString): Boolean;
function IsFloat(**const** Src: AnsiString): Boolean;
function IsAlpha(**const** Src: AnsiString): Boolean;
function IsAlphaNum(**const** Src: AnsiString): Boolean;
function IsMask(**const** Src, Mask: AnsiString; Index: Integer): Boolean;
function IsNull(**const** Src: AnsiString): Boolean;
function IsDateTime(**const** Src: AnsiString): Boolean;

Contents

Edit

function MakeNum(**var** Src: AnsiString): Integer;
function MakeFloat(**var** Src: AnsiString): Integer;
function MakeFixed(**var** Src: AnsiString; **const** Count: Byte): Integer;
function MakeAlpha(**var** Src: AnsiString): Integer;
function MakeAlphaNum(**var** Src: AnsiString): Integer;
function DupChr(**const** X: Char; Cnt: Integer): AnsiString;
procedure UCase(**var** Source: AnsiString; **const** Index, Count: Integer);
procedure LCase(**var** Source: AnsiString; **const** Index, Count: Integer);
procedure ProperCase(**var** Src: AnsiString);
procedure MoveStr(**const** S: AnsiString; XS: Integer; **var** D: AnsiString; **const** XD, Cnt: Integer);
procedure FillStr(**var** Src: AnsiString; **const** Index: Integer; X: Char);
procedure FillCnt(**var** Src: AnsiString; **const** Index,Cnt: Integer; X: Char);
function Compact(**var** Src: AnsiString): Integer;
function DeleteC(**var** Src: AnsiString; **const** X: Char): Integer;
function DeleteD(**var** Src: AnsiString; **const** X: Char): Integer;
function DeleteT(**var** Src: AnsiString; **const** Table: AnsiString): Integer;
function DeleteNT(**var** Src: AnsiString; **const** Table: AnsiString): Integer;
procedure ReplaceC(**var** Src: AnsiString; **const** X, Y: Char);
procedure ReplaceT(**var** Src: AnsiString; **const** Table: AnsiString; X: Char);
procedure ReplaceS(**var** Src: AnsiString; **const** Target, Replace: AnsiString);
procedure OverWrite(**var** Src: AnsiString; **const** Replace: AnsiString; Index: Integer);
procedure Translate(**var** Src: AnsiString; **const** Table, Replace: AnsiString);
procedure RevStr(**var** Src: AnsiString);
procedure IncStr(**var** Src: AnsiString);
function TruncPath(**var** Src: AnsiString; **const** Count:Integer): Boolean;

Contents

Arrays

procedure StrSort(**var** A: array of AnsiString; **const** Cnt: Integer);
function StrSrch(**var** A: array of AnsiString; **const** Target: AnsiString; Cnt: Integer): Integer;
function StrDelete(**var** A: array of Ansistring; **const** Target, Cnt: Integer): Boolean;
function StrInsert(**var** A: array of Ansistring; **const** Target, Cnt: Integer): Boolean;
procedure StrSwap(**var** S1, S2: AnsiString);
procedure Dim(**var** P; **const** Size:Integer; Initialize:Boolean);
function Capacity(**var** P):Integer;
type TIntegerArray
type TWordArray
type TSingleArray
type TDoubleArray
type TCurrencyArray

Contents

Hash/Encrypt

function   Hash(**const** Src: AnsiString): Integer;
procedure EnCipher(**var** Src: AnsiString);
procedure DeCipher(**var** Src: AnsiString);
procedure Crypt(**var** Src, Key: AnsiString);

Contents

Integer Date/Time

function TDT2IDT(**const** TDT: TDateTime): IDateTime;
function IDT2TDT(**const** IDT: IDateTime): TDateTime;
function StrToITime(**const** Source: AnsiString): IDateTime;
function StrToIDate(**const** Source: AnsiString): IDateTime;
function StrToIDateTime(**const** Source: AnsiString): IDateTime;
function IDateToStr(**const** IDT: IDateTime): AnsiString;
function ITimeToStr(**const** IDT: IDateTime): AnsiString;
function IDateTimeToStr(**const** IDT: IDateTime): AnsiString;
function EncodeITime(**const** D,H,M,S: Word): IDateTime;
procedure DecodeITime(**const** IDT: IDateTime; **var** D,H,M,S: Word);
function EncodeIDate(**const** Y,M,D: Word): IDateTime;
procedure DecodeIDate(**const** IDT: IDateTime; **var** Y,M,D: Word);
function RoundITime(**const** IDT: IDateTime;Mns: Word): IDateTime;
function WeekNum(**const** TDT: TDateTime): Word;
function ISOWeekNum(**const** TDT: TDateTime): Word;

Note: These routines are similar to the built-in *TDateTime* routines but use a more efficient Integer type; *IDateTime*, which is half the size of *TDateTime* . More importantly; integer data is more easily ported across operating and development environments. Efficiency and portability is achieved at the expense of range, accuracy and clarity (the units are more obscure, see below). However, the ensuing losses are insignificant for many applications. *IDateTime* can store any date/time value from 1900 through 2079 with an accuracy of +/- 2 seconds.

The internal *IDateTime* unit of time is a ‘tik’ with 65536 (2^16) tiks in a day.

Contents

Checksum/CRC

function CRC16(**const** IniCRC: Word; Src: AnsiString): Word;
function CRC32(**const** IniCRC: Integer; Src: AnsiString): Integer;
function ChkSum(**const** Src: AnsiString): Word;
procedure MakeSumZero(**var** Src: AnsiString);
function CreditSum(**const** Src: AnsiString): Integer;
function ISBNSum(**const** Src: AnsiString): Integer;

Contents

Base64

function EnCodeInt(**const** X: Integer): AnsiString;
function DeCodeInt(**const** Src: AnsiString): Integer;
function EnCodeWord(**const** X: Word): AnsiString;
function DeCodeWord(**const** Src: AnsiString): Word;
function EnCodeSng(**const** X: Single): AnsiString;
function DeCodeSng(**const** Src: AnsiString): Single;
function EnCodeDbl(**var** X: Double): AnsiString;
function DeCodeDbl(**const** Src: AnsiString): Double;
function EnCodeCur(**var** X: Currency): AnsiString;
function DeCodeCur(**const** Src: AnsiString): Currency;
function EnCodeStr(**var** Src: AnsiString): AnsiString;
function DeCodeStr(**var** Src: AnsiString): AnsiString;

Note: Encoding using the character subset specified by Internet proposal RFC 1521 (the MIME standard). The resultant data is safe for Internet, database, mainframe and registry use. EnCodeStr complies strictly with the RFC by adding 'fill' to produce a resultant whose length is a factor of 4. The specialized numeric routines produce efficient, fixed-length codes without 'fill'.

Useful for enhancing database formats with limited native support for numeric data. Virtually any type data can be encoded and stored in a registry string key. Provides a mild form of encryption.

Contents

+Miscellaneous

function UnSignedCompare(**const** X, Y: Integer): Boolean;
function LoBit(**const** X: Integer): Integer;
function HiBit(**const** X: Integer): Integer;
function RotL(**const** X, Cnt: Integer): Integer;
function RotR(**const** X, Cnt: Integer): Integer;
function TestBit(**const** X, Cnt: Integer): Boolean;
procedure IntSwap(**var** I1, I2: Integer);
procedure ISortA(**var** A: array of integer; **const** Cnt: Integer);
procedure ISortD(**var** A: array of integer; **const** Cnt: Integer);
function IntSrch(**var** A: array of integer; **const** Target, Cnt: Integer): Integer;
function UniqueApp(**const** Title: AnsiString): Boolean;
function CalcStr(**var** Source: AnsiString): Double;
function RndToFlt(**const** X:Extended):Extended;
function RndToInt(**const** X:Extended):Integer;
function IPower(**const** X,Y:Integer):Integer;
function IPower2(**const** Y:Integer):Integer;
procedure SpeakerBeep;
function iMin(**const** A,B: Integer): Integer;
function iMax(**const** A,B: Integer): Integer;
function iMid(**const** A,B,C: Integer): Integer;
function GetKeyToggle(**const** Key: Integer): Boolean;
procedure FlashSplash(BitMap:TGraphic; **const** Title: AnsiString);
procedure KillSplash;

Contents

function TDT2IDT (**const** TDT: TDateTime): IDateTime;

Converts TDateTime units to IDateTime format.

Group

function IDT2TDT (**const** IDT: IDateTime): TDateTime;

Converts IDateTime units to TDateTime format.

Group

function StrToITime (**const** Source: AnsiString): IDateTime;

Functional equivalent of StrToTime for IDateTime units.

Group

function StrToIDate (**const** Source: AnsiString): IDateTime;

Functional equivalent of StrToDate for IDateTime units.

Group

function StrToIDateTime (**const** Source: AnsiString): IDateTime;

Functional equivalent of StrToDateTime for IDateTime units.

Group

function IDateToStr (**const** IDT: IDateTime): AnsiString;

Functional equivalent of DateToStr for IDateTime units.

Group

function ITimeToStr (**const** IDT: IDateTime): AnsiString;

Functional equivalent of TimeToStr for IDateTime units.

Group

function IDateTimeToStr (**const** IDT: IDateTime): AnsiString;

Functional equivalent of DateTimeToStr for IDateTime units.

Group

function EncodeITime (**const** D,H,M,S: Word): IDateTime;

Similar to EncodeTime but with some important differences..

- Encoding of milli-seconds is not supported.

- A number of days (D) can be specified as part of the time increment to be encoded.

The reason for these differences is the more obscure units used by IDateTime. For example, an increment of 1 day, 6 hours in TDateTime units is simply 1.25. The same increment in IDateTime units is 81920. This is more a problem of interpretation rather than function. Aside from the different units, simple date/time arithmetic works much the same with IDateTime as with TDateTime.

Group

procedure DecodeITime (**const** IDT: IDateTime; **var** D,H,M,S: Word);

Similar to DecodeTime but with some important differences.   See EncodeITime for further discussion.

Group

function EncodeIDate (**const** Y,M,D: Word): IDateTime;

Functional equivalent of EncodeDate for IDateTime units.

Group

procedure DecodeIDate (**const** IDT: IDateTime; **var** Y,M,D: Word);

Functional equivalent of DecodeDate for IDateTime units.

Group

function RoundITime (**const** IDT:IDateTime; Mns:Word): IDateTime;

Native format rounding of IDateTime values to the nearest 1,5,6,15 or 30 minutes as specified by Mns. Factors for these common increments are hard coded. Any other increment is undefined and the original value will be returned.

Note: This routine eliminates the need to decode and re-encode merely for the sake of rounding. Remember that IDateTime values (and this function) are only accurate to within +/- 2 seconds. In other words, rounded values may still differ by up to 2 seconds from the correct HH:MM value upon decoding.

Group

function WeekNum (**const** TDT:TDateTime):Word;

Provides a work week index (0-52, Monday is first day of work week) for a given date per US calendar.   Week 0 is the week containing the first Monday of the year.

Note: Occasionally, there are 53 weeks per year, 1996 for example.

Group

function ISOWeekNum (**const** TDT:TDateTime):Word;

Provides a week-of-the-year index (0-52) for a given date per ISO 8601.   Week 0 is the week containing January 4, Monday is first day of week.

Note: Occasionally, there are 53 weeks per year, 1998 for example.

Group

function Compact(**var** Source:AnsiString):Integer;

Compact a string by moving embedded spaces and control char. to the right where they can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function DeleteC(**var** Source:AnsiString;**const** X:Char):Integer;

Convert specified char. into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function DeleteD(**var** Source:AnsiString;**const** X:Char):Integer;

Convert trailing duplicates of specified char. into right justified spaces which can be deleted if necessary using RTrim or SetLength. Only duplicates are affected, the first character in a run of duplicates is left in place.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function DeleteT(**var** Source:AnsiString;**const** Table:Ansistring):Integer;

Convert any Table chars into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function DeleteNT(**var** Source:AnsiString;**const** Table:Ansistring):Integer;

Convert any non-Table chars into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Example: One application might be to filter keystroke errors from user input.

```
Source:='$123X4.56   ';
I:=DeleteNT(Source,'$+-.0123456789');
```

On return, I=8, Source='$1234.56   '.   To remove trailing spaces, SetLength(Source,I);

Group

function IsFloat(**const** Source:AnsiString):Boolean;

Determine if a string contains characters,0-9,space,E,+.-.   This function only verifies the char. sub-set, see *IsMask* for format checking.

function IsNum(**const** Source:AnsiString):Boolean;

Determine if a string contains only digit characters, 0-9 and space.   This function only verifies the char. sub-set, see *IsMask* for format checking.

Group

function IsHex(**const** Source:AnsiString):Boolean;

Determine if a string contains only hexadecimal digit characters, 0-9, A-F and space.   This function only verifies the char. sub-set, see *IsMask* for format checking.

Group

function IsNull(**const** Source:AnsiString):Boolean;

Determine if a string contains only characters, 0-32 and 255.   This function only verifies the char. sub-set, see *IsMask* for format checking.

Group

function IsDateTime(**const** Source:AnsiString):Boolean;

Determine if a string contains only char. 0-9,space,-,DateSeperator,TimeSeparator. This function only verifies the char. sub-set, see *IsMask* for format checking.

Group

function IsAlpha(**const** Source:AnsiString):Boolean;

Determine if a string contains only alpha characters and spaces. See *IsMask* for format checking.

Group

function IsAlphaNum(**const** Source:AnsiString):Boolean;

Determine if a string contains only alpha characters, digits and spaces. See *IsMask* for format checking.

Group

function MakeAlphaNum(**var** Source:AnsiString):Integer;

Convert any non-alphanumeric char. into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function MakeAlpha(**var** Source:AnsiString):Integer;

Convert non-alpha char. into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function MakeNum(**var** Source:AnsiString):Integer;

Convert non-numeric char. into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function MakeFloat(**var** Source:AnsiString):Integer;

Convert chars other than 0..9,E,+,-, decimal into right justified spaces which can be deleted if necessary using RTrim or SetLength.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function MakeFixed(**var** Source:AnsiString; **const** Count:Integer):Integer;

Convert chars other than 0..9,+,-, decimal into right justified spaces, add or removes digits as necessary to provide Count decimal places.   Extend length if needed.

Returns: Valid char. count; length less any chars. moved and converted to spaces.

Group

function ChrToInt(**const** Source:AnsiString):Integer;

Convert any 4 char. string into an integer. See *IntToChr* for discussion.

Group

function ChrToWord(**const** Source:AnsiString):Word;

Convert any 2 Char string into a word. See *IntToChr* for discussion.

Group

function IntToChr(**const** X:Integer):AnsiString;

Convert any integer into a 4 byte MSB (Most Significant Byte first) string representation.

To preclude possible confusion, we’ll address some typical questions.

1) Why do this?

In order to turn the lowly string into a complex, variable length data container. For example, with this routine you can easily tag a string with one or more numeric integer values (a database record number perhaps) without using additional data structures. Simply convert the integer and append to the end of the string. To retrieve the integer value, apply the complimentary ChrToInt function to the last 4 chars. of the string. Much more elaborate structures are possible.

Admittedly, this is a hack but it works<g>.

2) Why MSB (also known as 'big-endian')?

Strings are normally compared and sorted on a left to right, MSB basis. Using this basis allows converted integers to be compared and sorted properly *as strings*. It should be pointed out that string comparisons are *unsigned*.

Note: This and related routines output characters in the range 0..255. Embedded nulls and control char. are fully supported with AnsiStrings; however, “binary” strings are not suitable for use in API calls or registry storage. The Base64 routines provide a similar but less efficient alternative that is safe for almost any use.

Group

function WordToChr(**const** X:Word):AnsiString;

Convert any word into a 2 Char string.   See *IntToChr* for discussion.

Group

function SngToChr(**const** X:Single):AnsiString;

Convert any single into a 4 byte MSB string representation. See *IntToChr* for discussion.   This conversion is 'internally exact'. In other words, converting from single to string and back again yields the original 'internal' representation exactly.   However; as is always the case with floating point, the internal representation only approximates the real value.

Group

function ChrToSng(**const** Source:AnsiString):Single;

Convert any 4 char. string into a single floating point value. See *IntToChr* for discussion.

Group

function DblToChr(**var** X:Double):AnsiString;

Convert any Double (or compatible type such as TDateTime) into an 8 byte MSB string representation.

See *IntToChr* for further discussion.

Group

function ChrToDbl(**const** Source:AnsiString):Double;

Convert any 8 char. string into a Double floating point value. See *IntToChr* for discussion. As is typical with floating point, the representation is only approximate.

Group

function CurToChr(**var** X:Currency):AnsiString;

Convert a Currency type into an 8 byte MSB string representation. See *IntToChr* for discussion.

Group

function ChrToCur(**const** Source:AnsiString):Currency;

Convert any 8 char. string into a Currency value. See *IntToChr* for discussion.

Group

function IntToBin(**const** X:Integer):AnsiString;

Convert any integer into a 32 byte right justified 1/0 string.   Use LTrim to remove leading zeros if desired.

Group

function BinToInt(**const** Source:AnsiString):Integer;

Convert a right justified 1/0 binary string into an integer.

Group

function HexToInt(**const** Source:AnsiString):Integer;

Convert a hexadecimal string into an integer. Prefixes and invalid characters (other than 0-9,A-F) are ignored.

Note: Similar results can be achieved by prefixing Source with ‘$’ and applying StrToInt; however, this function is generally more efficient since it eliminates the need to create an intermediate string.

Group

function EnCodeInt(**const** X:Integer):AnsiString;

Encode any integer as a truncated 6 char base64 string that is Internet, mainframe, database and Registry safe.

Group

function DeCodeInt(**const** Source:AnsiString):Integer;

Decode a 6 Char integer string created with EnCodeInt and return the original integer value.

Group

function EnCodeWord(**const** X:Word):AnsiString;

Encode any Word as a truncated 3 char base64 string that is Internet, mainframe, database and Registry safe.

Group

function DeCodeWord(**const** Source:AnsiString):Word;

Decode a 3 char Integer string created with EnCodeInt and return the numeric value.

Group

function EnCodeSng(**const** X:Single):AnsiString;

Encode any Single floating point value as a 6 char base64 string that is Internet, mainframe, database and Registry safe.   This conversion is 'internally exact'.

Group

function DeCodeSng(**const** Source:AnsiString):Single;

Decode a 6 char Integer string created with EnCodeSng and return the original floating point single value.

Group

```
function EnCodeDbl(var X:Double):AnsiString;
```

Encode any Double (or compatible type such as TDateTime) as an 11 char base64 string that is Internet, mainframe, database and Registry safe.   This conversion is 'internally exact'.

Group

function DeCodeDbl(**const** Source:AnsiString):Double;

Decode an 11 char Double string created with EnCodeDbl and return the original numeric floating point value.

Group

function EnCodeCur(**var** X:Currency):AnsiString;

Encode any Currency type as an 11 char base64 string that is Internet, mainframe, database and Registry safe.
This conversion is 'internally exact'.

Group

function DeCodeCur(**const** Source:AnsiString):Currency;

Decode an 11 char Currency string created with EnCodeCur and return the original Currency type value.

Group

function EnCodeStr(**var** Source:AnsiString):AnsiString;

Encode a string using base64 encoding compatible with Internet protocol RFC 1521 (the MIME standard).   Data encoded in this manner is safe for Internet, mainframe, database and Registry storage and use. By definition, the resultant string length is always a factor of 4 and at least 1/3 greater than the original.

Group

function DeCodeStr(**var** Source:AnsiString):AnsiString;

Decode a base64 string created with EnCodeStr and return the original. The length of the encoded Source string should be a multiple of 4, any fractional excess will be ignored.

Group

function Soundex(**const** Source:AnsiString):Integer;

Encode a string as a 4 byte integer value using the Soundex table originally provided by your friendly US Census Bureau:

Note: This function returns an Integer for faster and more efficient comparisons. If you prefer the more traditional string representation, simply apply *IntToChr* to the resultant integer.

Group

procedure EnCipher(**var** Source:AnsiString);

Fast, 7-bit ASCII (char. 32..127) encryption designed for database use. Control and high order (8 bit) characters are passed through unchanged.

Uses a hybrid method...random table substitution with bit-mangled output. No passwords to worry with, the algorithm and built-in table are the password.

The output from this procedure appears quite random and provides good, convenient security against the casual snoop armed with a file viewer.   However, keep in mind that the complementary *DeCipher* algorithm is available in the marketplace and anyone with access to it will be able to easily decode your data.

Note: When displaying enciphered strings, remember that some characters within the output range are interpreted by VCL components; for example, '&'.

Group

procedure DeCipher(**var** Source:AnsiString);

Decrypts a string previously encrypted with EnCipher.

Group

procedure Crypt(**var** Source,Key:Ansistring);

Encrypt AND decrypt strings using a reversible, ‘pseudo-key’ technique somewhat similar to S-Coder (DDJ, Jan. 1990). To decrypt, simply re-apply the procedure using the same password Key.

This is a much improved implementation of the popular, reversible XOR technique.   The provided Key is *not* used to directly encrypt the Source string.   Instead, Key is used to seed a linear feedback ‘pseudo-key’ generator.   The generated ‘pseudo-key’, which typically has a repeat cycle 256 times longer than Key itself, is actually used to encrypt the Source. The end result is a fast, simple and much more secure algorithm.   A similar level of security with simple XOR would require the use of a random, non-sense password hundreds of bytes long.   Here are some suggestions for further enhancing security.

- Use a longer Key.
- Double or triple encrypt the string using different keys. To decrypt, re-apply the keys in reverse order.
- EnCipher the string before using Crypt.   To de-crypt, simply re-apply Crypt then DeCipher.
- Use a combination of the above.

Note: Output characters are in the range, 0..255.   Key is case sensitive.

Group

function Hash(**const** Source:Ansistring):Integer;

Generate an integer hash key for the input string.

Returns: 32 bit +/- integer hash key, zero if null string

Note: This is a highly efficient, verified, general purpose hashing algorithm based upon the published research of Peter J. Weinberger of AT&T Bell Labs and others. This implementation has been used for years in UNIX object files.

CRC and Checksum routines can also be used effectively for hashing.

Group

function ScanF(**const** Source,Search:AnsiString;Start:Integer):Integer;

Forward scan from specified Start position looking for Search key. Search may contain any number of '?' wildcards to match any character. For case insensitive scan, specify Start as negative.

Returns: Position where/if found; otherwise, 0

Group

function ScanR(**const** Source,Search:AnsiString;Start:Integer):Integer;

Reverse scan from specified Start position (1 = First Char, 0 = String End) looking for Search key.   Search may contain '?' wildcards to match any character. For case insensitive scan, specify Start as negative.

Returns: Position where/if found; otherwise, 0

Group

function ScanC(**const** Source:AnsiString;X:Char;Start:Integer):Integer;

Forward scan from Start looking for next matching char. (X). This and the complementary ScanB (backwards scan) are optimized for case sensitive single character searching.

Returns: Position where/if found; otherwise, 0

Group

function ScanCC(**const** Source:AnsiString;X:Char;Count:Integer):Integer;

Forward scan from beginning of string looking for the Count instance of char. X.

Returns: Position where/if found; otherwise, 0

Group

function ScanB(**const** Source:AnsiString;X:Char;Start:Integer):Integer;

Backward/reverse scan from Start location (1 = First Char., 0 = String End) looking for single character, X.

Returns: Position where/if found; otherwise, 0

Group

function ScanL(**const** Source:AnsiString;Start:Integer):Integer;

Forward scan from Start location looking for next lowercase (ASCII 97..122) character.

Returns: Position where/if found; otherwise, 0

Group

function ScanU(**const** Source:AnsiString;Start:Integer):Integer;

Forward scan from Start location looking for next uppercase (ASCII 65..90) character.

Returns: Position where/if found; otherwise, 0

Group

function ScanNC(**const** Source:AnsiString;X:Char):Integer;

Forward scan looking for first NON-matching character.

Returns: Position where/if found; otherwise, 0

Group

function ScanNB(**const** Source:AnsiString;X:Char):Integer;

Backward/Reverse scan looking for first NON-matching character.

Returns: Position where/if found; otherwise, 0

Group

function ScanT(**const** Source,Table:AnsiString;Start:Integer):Integer;

Forward scan from Start looking for any char. in Table. For case insensitive scan, specify Start as negative.

Returns: Position where/if found; otherwise, 0

Group

function ScanRT(**const** Source,Table:AnsiString;Start:Integer):Integer;

Reverse scan from Start looking for any char. in Table. For case insensitive scan, specify Start as negative.

Returns: Position where/if found; otherwise, 0

Group

function ScanNT(**const** Source,Table:AnsiString;Start:Integer):Integer;

Forward scan from Start looking for first char. NOT in table. For case insensitive scan, specify Start as negative.

Returns: Position where/if found; otherwise, 0

Group

function ScanRNT(**const** Source,Table:AnsiString;Start:Integer):Integer;

Reverse scan from Start looking for first char. NOT in table. For case insensitive scan, specify Start as negative.

Returns: Position where/if found; otherwise, 0

Group

function ScanP(**const** Source,Search:AnsiString;**var** Start:Integer):Integer;

Exhaustive forward scan from Start looking for the longest partial match of Search. Search may contain '?' wildcards to match any character.

Returns: Match length. Start = Match position.

Note: To continue a search, Start must be manually incremented beyond the last returned match position. If a perfect match is found, resultant = Length(Search). If no partial match, resultant = 0 and Start = 0.

Group

function ScanW(**const** Source,Search:AnsiString;**var** Start:Integer):Integer;

Forward scan from Start looking for a match of Search pattern string containing wildcards:

```
   '*'   = match any string (including null string)
   '?'   = match any single character
   '#'   = match any numeric character (0..9)
   '@' = match any alpha character (A..Z, a..z)
 else = match given character
```

For case insensitive, specify Start as negative.

Returns: Matching length, Start = match location. If no match, Result = 0 and Start = 0;

Note: To continue a search, Start must be manually incremented beyond the last returned match position.

The function returns as soon as a substring satisfying the pattern has been found. When using wildcards, the results must be examined carefully. For example, given a Search pattern of 'abc*'; a match of the first 3 characters is all that is required. Therefore; if a match is found, the returned match length will *never* be greater than 3.

Likewise, with a search string of '*abc' ; the first character at Start will *always* match. In this case, match length may vary from 3 to the length of Source.

Group

function ScanQ(**const** Source,Search:Ansistring;Start:Integer):Integer;

"Quick" forward scan using the primary Boyer-Moore heuristic. Search key length is limited to 256 characters or less and may contain '?' wildcards to match any character. For case insensitive scan, specify Start as negative.

This algorithm is often dramatically faster than a brute force sequential search; however, there are cases where it may actually be slower.

1) Very short Search key string (less than 3 chars).
2) Relatively short Source string (less than 256 chars).
3) A match is located very near the given Start position.   Typically, there is no way to know this in advance.
4) Source contains an inordinately large number of a sub-string matching the rightmost part of Search key. For example, if Search ends with '...ing' (as in 'String') and Source contains many, many instances of the 'ing' sequence. In this case, the algorithm may waste significant time investigating false leads.

This function is probably at it's best when working with relatively long strings and medium sized keys.

Group

function ScanQC(**const** Source,Search:Ansistring; Start:Integer):Integer;

Continue a "Quick" forward scan begun by an initial call to ScanQ. This routine avoids time consuming initialization by assuming that Source and Search have not changed since the last call to ScanQ. The programmer is responsible for insuring that this is the case.

Example: The code snippet below counts all occurances of a search string. Note how Start is incremented beyond the previous instance.

```
Count := 0;
Start := ScanQ(Source, Search, 1);
while Start > 0 do begin
    Inc(Count);
    Start := ScanQC(Source, Search, Start + Length(Search));
end;
```

Group

function ScanZ(**const** Source,Search:Ansistring; Defects:Integer; **var** Start:Integer):Integer;

Forward scan from Start looking for an approximate, “fuzzy” match of Search. *Defects* is the max. number of character “defects” allowed in a matching sub-string. Typically, 0<Defects<Length(Search). A “defect” is defined as one of the following:

- Extra/missing character (isolated single characters only)
- Character mismatch
- Adjacent characters swapped

If Defects=0, a perfect match is required. If Defects >= Length(Search), any string will match. Use negative start for case insensitive scan (case difference *NOT* considered a defect).

Returns: Length of matching sub-string, Start = sub-string location. Resultant = 0 and Start is undefined if no match. Match length may differ from Length(Search) since extra/missing characters are allowable defects. To continue a search, manually adjust Start beyond the returned match.

Example:

Given:

Source := ’Where is Hillerd ?’;
Search := ’Dillard’;
Defects := 2;
Start := 5;

ScanZ(Source,Search,Defects,Start) returns a match length of 7 at Start = 10. In this case, ‘Hillerd’ and ‘Dillard’ are a match with defects at the 1st and 5th positions.

Notes:

The algorithm correctly diagnoses most minor spelling defects but will not identify words distorted by gross errors such as multiple adjacent characters missing.

The function always returns the first match found. An overly large Defect value often produces nothing more than a pre-mature or unwanted match.

“Fuzzy” algorithms can sometimes yield logically correct results that are somewhat unexpected. For example, ‘Hillerd’ is also a match for ‘billed’, ‘killed’ or ‘willed’ with two defects (mismatched 1st and extra/missing 6th).

Group

```
procedure ProperCase(var Source:AnsiString);
```

Upper case the first alpha character in each word, lower case all other characters. Any char. less than ASCII 48 (0) is considered a word delimiter.

Group

function CountT(**const** Source,Table:AnsiString):Integer;

Count all instances of Table characters within Source.

Group

function CountW(**const** Source,Table:AnsiString):Integer;

Count all words (non-null tokens) in Source delimited by any char in Table.   Use GetTokenCnt if null tokens are to be included.

Group

procedure Translate(**var** Source:AnsiString;**const** Table,Replace:AnsiString);

Replace all chars. found in Table with the corresponding character from Replace table.   By definition, the 2 tables must be the same size.

Group

procedure ReplaceT(**var** Source:AnsiString;**const** Table:AnsiString;X:Char);

Search and replace all chars. found in Table with a given replacement character.

Group

procedure ReplaceS(**var** Source:AnsiString;**const** Target,Replace:Ansistring);

Replaces all instances of Target sub-string with Replace sub-string.

Example: ReplaceS(Source,#9,DupChr(#32,8)); replaces all tabs with 8 spaces.

Group

function LStr(**const** Source:Ansistring;Count:Integer):Ansistring;

Conveniently retrieve Count chars from the left of Source.

For VB converts, similar to LEFT$().

Group

function RStr(**const** Source: Ansistring; Count: integer): Ansistring;

Conveniently retrieve Count chars from the right of Source.

For VB converts, similar to RIGHT$().

Group

function CStr(**const** Source:Ansistring;Index,Count:Integer):Ansistring;

Retrieve Count chars from any location inside Source.   Yes, we know it's essentially the same as Delphi's Copy but we just had to satisfy some strange sense of symmetry and completeness by rounding out the trio (LStr, RStr and CStr).   Anyway, it's cheaper than psychotherapy.

For VB converts, similar to MID$() function.

Group

function DupChr(**const** X:Char;Count:Integer):AnsiString;

Manufacture a string of length Count by duplicating char. X.

Group

procedure LPad(**var** Source: Ansistring;**const** X:Char;Count:Integer);

Append characters (X) to left of Source as required to increase length to Count.

Group

```
procedure RPad(var Source: Ansistring;const X:Char;Count:Integer);
```

Append characters (X) to right of Source as required to increase length to Count.

Group

procedure CPad(**var** Source: Ansistring; **const** X:Char;Count:Integer);

Append characters (X) to left and right of Source as required to increase length to Count while keeping existing text centered.

Group

procedure LText(**var** Source: AnsiString);

Left justify text within Source by moving leading spaces and control char. to end.   Length   is not changed.

Group

procedure RText(**var** Source: AnsiString);

Right justify text within Source by moving trailing spaces and control char. to front.   Length is not changed.

Group

procedure CText(**var** Source: AnsiString);

Center text within Source by moving leading and trailing spaces and control char. as necessary. Length is not changed.

Group

procedure FillStr(**var** Source:AnsiString;**const** Index:Integer;X:Char);

Fill Source starting at Index location using Char X. Includes range checking to prevent memory corruption.

Group

```
procedure FillCnt(var Source:AnsiString;const Index,Cnt:Integer;X:Char);
```

Fill Source starting at Index location with Cnt characters of X. Includes range checking to prevent memory corruption.

Group

procedure OverWrite(**var** Source:AnsiString; **const** Replace:AnsiString;Index:Integer);

Overwrite Source text at Index location with Replace text. A companion to the native Insert and Delete functions. Built-in range checking prevents memory corruption.

Group

procedure MoveStr(**const** S:AnsiString;XS:Integer;**var** D:AnsiString;**const** XD,Cnt:Integer);

Generic string move utility. Overwrite destination string D starting at location XD using Cnt characters taken from source string S at location XS. Full range checking is included to prevent memory corruption.

Example: MoveStr(Source,SIndex,Dest,DIndex,Count)

Group

function Parse(**const** Source,Table:AnsiString;**var** Index:Integer):AnsiString;

Sequential, left to right token parsing using a table of delimiter characters. Intended for applications where there is limited control over the delimiters. Index is a pointer (initialize to '1' for first token) updated by the function to point to next token.   To retrieve the next token, simply call the function again using the prior returned Index value.

Note: If returned Index > Length or Index < 1, no additional tokens are available.

Group

function ParseWord(**const** Source,Table:AnsiString;**var** Index:Integer):AnsiString;

Similar to Parse but null tokens are ignored. Intended for parsing “freeform” text.

Index is a pointer (initialize to '1' for first word) updated by the function to point to next word.   To retrieve the next word, simply call the function again using the prior returned Index value.

Note: If Length(Resultant) = 0, no additional words are available.

Example: Parse all words from freeform text (S) into a ListBox.

```
I := 1;
T := #9#10#13#32#44#46#59;   //common delimiters in freeform text
repeat
    W := ParseWord(S,T,I);
    if Length(W)>0 then
        ListBox1.Items.Add(W)
    else break;
until True=False;
```

Group

```
function ParseTag(const Source, Start, Stop:AnsiString;var Index:Integer):AnsiString;
```

Sequential, left to right parsing of tokens delimited by start/stop “tags” such as those commonly found in HTML and XML strings. Index is a pointer (initialize to '1' for first token) updated by the function to point to next token. To retrieve the next token, simply call the function again using the prior returned Index value.

Note: If returned Index > Length or Index < 1, no additional tokens are available.

Example: Parse all “anchor” tags in an HTML document.

- Allocate a string (S) to hold the entire document
- Allocate an index variable (I) and initialize to 1
- Open the document file and read into string
- Repeatedly call ParseTag(S, ’<A’ , ’/A>’, I) until no more tokens

Some HTML tokens (such as “paragraph”) may span multiple lines. This function returns the entire token, including line breaks. Use DeleteT with Table = line break (usually CRLF) to remove breaks from tokens if desired.

Group

function Fetch(**const** Source,Table:AnsiString; Num:Integer; DelFlg:Bool):AnsiString;

Retrieve tokens by number (first = 1) using a table of delimiter characters. Intended for applications where there is limited control over the delimiters.   If DelFlg is True, the returned token includes the terminating delimiter.

Note: If the specified token is not found, a null string is returned.   This routine was added by request as a convenience feature.   Locating a token by number requires a sequential search from the start of the string. GetToken is a much more efficient alternative if the approximate position of the desired token is known as a result of a scan or other means.

Group

function SetDelimiter(Delimit:Char):Boolean;

Set the *HyperString* delimiter character to be used by tokens and other functions. The default delimiter is a comma (ASCII 44). Returns False if delimiter is a null (zero); otherwise, True.

Group

function GetDelimiter:Char;

Return the current *HyperString* delimiter character being used by tokens and other functions.

Supports writing well behaved record handlers by allowing routines to save and restore (via SetDelimiter) the delimiter setting.

Group

```
function GetToken(const Source:Ansistring;Index:Integer):Ansistring;
```

Retrieves the token from Source associated with the given string Index position.

Tokens are referenced by string Index position. A valid Index is any string position between delimiters (string start and end are also delimiters). For example, with the string below, any Index position from 9 to 17 would retrieve the 'Wednesday' token.

```
1…      9…
Tuesday,Wednesday,Thursday
```

This method of token referencing is highly efficient and allows ordinary scan/search functions to be easily used with tokenized strings as well. After scanning for a desired sub-string, the corresponding token can be quickly and easily retrieved using the sub-string's index position.

Note: Index is indeterminate if Source[Index] = Delimiter. Use Index = 1 for first token, Index = Length(Source) for Last. Invalid or indeterminate Index returns a null string.

Group

function InsertToken(**var** Source:AnsiString; **const** Token:Ansistring;Index:Integer):Boolean;

Insert token into Source at the token position referenced by Index; shifting existing tokens as necessary.   Use zero Index to append a new token. Returns False if Index is invalid.

Tokens are referenced by string Index position.   A valid Index is any character position between delimiters (string start and end are considered delimiters).   For example, with the string below, Index positions from 9 to 17 all reference the 'Wednesday' token.

```
1…      9…
Tuesday,Wednesday,Thursday
```

This method of token referencing is highly efficient and allows oridnary scan/search functions to be used with tokenized strings as well.   After scanning for a desired sub-string, the corresponding token can be retrieved, deleted, replaced or a new token inserted using the sub-string’s index position.

Group

function DeleteToken(**var** Source:Ansistring; **var** Index:Integer):Boolean;

Delete token from Source at referenced Index position; shifting tokens as necessary to fill the voided position. Returns False if Index is invalid. If Source[Index] = Delimiter, the delimiter is deleted. Index points to the next token if successful (resultant = True). See GetToken for more.

Group

function ReplaceToken(**var** Source,Token:Ansistring;Index:Integer):Boolean;

Replace the token in Source at the given Index position.   Returns False if Index is invalid. See GetToken for more.

Group

function PrevToken(**const** Source:Ansistring;**var** Index:Integer):Boolean;

Move string Index pointer to preceding token in Source.   Returns False and Index is undefined if no token precedes current.   See GetToken for more.

Group

function NextToken(**const** Source:Ansistring;**var** Index:Integer):Boolean;

Move string Index pointer to following token in Source.   Returns False and Index is undefined if no following token is found. See GetToken for more.

Group

function GetTokenNum(**const** Source:Ansistring;Index:Integer):Integer;

Translate a string Index position into a token number (First = 1). Returns zero if Index is invalid or indeterminate.

Tokens are normally referenced by string index position (see GetToken) so this function should rarely be used.

Group

function GetTokenPos(**const** Source:Ansistring;Num:Integer):Integer;

Translate a token number (First = 1) into a string index position.

Returns: First valid string index for token Num;   zero if token is not found.

This function complements GetTokenNum.   Tokens are normally referenced by string index for greater efficiency.
See GetToken for more.

Group

function GetTokenCnt(**const** Source:Ansistring):Integer;

Count the total number of tokens in Source based upon the delimiters present in the string. Null tokens will be included in the total.   CountW is an alternative which ignores nulls and uses multiple delimiters from a table.

Returns: Total number of delimiters + 1

Group

function ChkSum(**const** Source:AnsiString):Word;

Fletcher's Checksum, IEEE Transactions on Communications, Jan. 1982. Error detection nearly as good as 16-bit CRC but much "cheaper" to calc, also has some special properties.

```
Max. error rates:
-------------------------------------
16-bit CRC          = 0.001526%
16-bit Fletcher = 0.001538%
```

Group

procedure MakeSumZero(**var** Source:AnsiString);

Appends 2 chars (range 0..255) to Source in order to force the string to "sum to zero" with the complementary Fletcher checksum routine. Make strings self-checking!

Note: The appended chars. may be null; therefore, the resultant string should not be cast as null terminated.

Group

function CreditSum(**const** Source:AnsiString):Word;

Shifted Mod 10 checksum of the type used for credit card encoding.   Result = 0 if Source is a *potentially* valid credit card number string. Result = -1 if Source is null.   Non-numeric ASCII characters (outside the range [‘0’..’9’]) are ignored.

Note: This checksum is primarily used to help detect data entry errors associated with 16 digit “major” credit card numbers.   Further checks are needed to verify that a given string is in fact a valid card number.   “Minor” issuers (department stores, auto clubs, etc.) may have their own encoding rules.   For example, an auto club (“gas card”) may use 14 digits and a checksum of 3.

“Major” US credit cards have 16 digit numbers with the first digit indicating card type as follows:

3 - American Express, Carte Blanche, Diners Club
4 - Visa
5 - Master Card, Choice
6 - Discover Card

Group

```
function ISBNSum(const Source:AnsiString):Boolean;
```

International Standard Book Number (ISBN) checksum.   Returns True if the given string is a potentially valid ISBN alpha-numeric book identifier.

Group

```
function CRC32(const IniCRC:Integer;Source:AnsiString):Integer;
```

Standard, table based CRC32 calculation. Initial call MUST use IniCRC:=-1 (or $FFFFFFFF). To add subsequent strings to the calcs, use IniCrc:= Prior CRC32 resultant. Final resultant must be inverted using NOT operator to conform to specs.   Equivalent Pascal implementation might be:

```
for I:=1 to Length(Source) do
   CRC:=((CRC SHR 8) AND $FFFFFF) XOR CRCTable[(CRC XOR Source[I]) AND $FF];
```

The code below calculates a combined CRC32 value (X) for strings S1 and S2.

```
var
   X :Integer;
begin
   X := -1;
   X := CRC32(X,S1);
   X := CRC32(X,S2);
   X := NOT X;
end;
```

Group

function CRC16(**const** IniCRC:Word;Source:AnsiString):Word;

Standard, table based CRC16 calculation. Initial string MUST use IniCRC:=-1 (or $FFFF). To add subsequent strings to the calcs, use IniCrc:= Prior CRC16 resultant. Final resultant must be inverted using NOT operator to conform to specs.   Equivalent Pascal implementation might be:

```
for I:=1 to Length(Source) do
   CRC:=((CRC SHR 8) AND $FF) XOR CRCTable[(CRC XOR Source[I]) AND $FF];
```

The code below calculates a combined CRC16 value (X) for strings S1 and S2.

```
var
   X : Word;
begin
   X := -1;
   X := CRC16(X,S1);
   X := CRC16(X,S2);
   X := NOT X;
end;
```

Group

function CountF(**const** Source: Ansistring;X:Char;Start:Integer): Integer;

Count instances of char X. from Start location Forward.

Group

function CountR(**const** Source: Ansistring;X:Char;Start:Integer): Integer;

Count instances of char X. in Reverse, from Start location backward.

Group

function LTrim(**const** Source:AnsiString;X:Char):AnsiString;

Trim specified char. X from the front of the string and return shortened string.

Group

function RTrim(**const** Source:AnsiString;X:Char):AnsiString;

Trim specified char. X from the end of Source string and return shortened string.

Group

function CTrim(**const** Source:AnsiString;X:Char):AnsiString;

Trim specified char. X from both ends of Source string and return shortened string.

Group

procedure ReplaceC(**var** Source: Ansistring;**const** X,Y:Char);

Search and replace all instances of char. X with char Y. To remove a character entirely, see DeleteC.

Group

procedure RevStr(**var** Source:AnsiString);

Reverse the characters; first to last, in the Source string.

Group

procedure IncStr(**var** Source:AnsiString);

Increment the characters in an alphanumeric string. Only string positions containing alphanumeric characters (0-9, A-Z, a-z) are considered. Therefore, the string to be incremented must be properly initialized. Incrementation is case-sensitive, overflows are ignored.

Example: IncStr('1a-9Z-99') yields '1b-0A-00'.

Group

function TruncPath(**var** Source:AnsiString; **const** Count:Integer)Boolean;

Attempts to shorten a file path to Count characters by replacing text between backslashes with an ellipsis and dropping characters from the file name if necessary. Retains as much of the file name as possible. Returns True if file path was shortened to Count or fewer characters.

Example:

```
S := ’c:\this\is\an\example\filepath.doc’;
TruncPath(S,24);
```

returns True with S = ‘c:\this\…\filepath.doc’

Group

procedure ISortA(**var** A:array of integer;**const** Cnt:Integer);

Sort an integer array into ascending, UNSIGNED order using CombSort; a generalized and much improved implementation of BubbleSort (see Byte magazine, April 1991). This assembler implementation is extremely compact and reasonably fast. Cnt = Total number of elements to be sorted if array is partially filled, use -1 for full array).

Note: BubbleSort is routinely chastised for being a “simple but slow” sorting technique, too slow for much practical use.   With regard to the typical BubbleSort, this may be true but there is more.   From a certain perspective, BubbleSort is merely a flawed implementation of an otherwise respectable sorting algorithm.   A few simple modifications transforms BubbleSort into the more generalized CombSort which provides very respectable performance while retaining the highly attractive simplicity.

Group

procedure ISortD(**var** A:array of integer;**const** Cnt:Integer);

Sort an integer array into descending, UNSIGNED order using CombSort. This assembler implementation is extremely compact and reasonably fast. Cnt = Total number of elements to be sorted (supports partially filled arrays, use -1 for full array). See ISortA for additional comments.

Group

function IntSrch(**var** A:array of integer;**const** Target,Cnt:Integer):Integer;

UNSIGNED binary search of an integer array.   Array is assumed to be sorted in ascending order.   Cnt = number of elements to search (supports partially filled arrays, -1 = All elements).

Returns: Element offset of match if found; otherwise,-1

Group

procedure StrSort(**var** A:array of Ansistring; **const** Cnt:Integer);

Fast, "semi-sort" (uses first 2 char. only) of a string array into ascending order. This is "good enough" in many cases but obviously of little value if most or all strings share the same first 2 characters.   A semi-sorted array can usually be searched much faster (see StrSrch) than a non-sorted one. The number of elements to be sorted must be provided in Cnt (-1 = All).   Any blank elements are up front after sorting.

Group

function StrSrch(**var** A:array of Ansistring;**const** Target:Ansistring; Cnt:Integer):Integer;

Binary search of string array for Target string.   Array is assumed to be in "semi-sorted" order as provided by StrSort.
Cnt = Number of elements to search for a partially filled array (-1 = All).

Returns: Offset of matching element if found; otherwise,-1

Group

function GetUser: Ansistring;

Returns the ID for the current system user. Returns null string if function fails.

<u>Group</u>

function GetNetUser: Ansistring;

Returns the network ID for the current system user. Returns null string if function fails.

Note: If a null string is returned, call GetLastError for an extended error code.   For example, if GetLastError = ERROR_NO_NETWORK, no network is currently available.   See GetLastError in WIN32.HLP for details.

Group

function GetComputer: Ansistring;

Returns the current workstation name. Returns null string if function fails.

Group

function GetDrives: Ansistring;

Returns a string containing all valid drive letters.   Removable drives are lower case, all others are upper case.

Group

function RemoteDrive(**const** Drv:Char**)**: Boolean;

Returns True if the specified drive letter can be identified as a networked drive.

Group

function GetDisk(**const** Drv:Char; **var** CSize,Available,Total:DWord):Boolean;

Returns disk size statistics --- cluster size, available and total number of clusters.

Group

function GetVolume(**const** Drv:Char; **var** Name,FSys:AnsiString; var S:DWord):Boolean;

Returns volume name, file system and serial number for a given drive.

Group

```
function GetWinDir: Ansistring;
```

Returns the Windows directory.

Group

function GetSysDir: Ansistring;

Returns the Windows\System directory.

Group

function GetTmpDir: Ansistring;

Returns the preferred directory for temporary files.

Group

function GetTmpFile(**const** Path,Prefix:AnsiString): Ansistring;

Returns a temporary filename. Use Path :='.' for current directory, Path:=GetTmpDir for Windows temp directory. Prefix is up to 3 char. used as the start of the file name. Returns null string on error.

Group

function GetDOSName(**const** LongName:Ansistring): Ansistring;

Returns the short, DOS equivalent for a long file name.

Group

function GetWindows: Ansistring;

Returns a tokenized string listing the titles of all currently active windows.   Use SetDelimiter first to specify the active token delimiter to be used.

Group

function GetClasses: Ansistring;

Returns a tokenized string containing the class names of all currently active windows.   Use SetDelimiter to specify the delimiter.

Example:

```
procedure ListAllClasses;
//Retrieve all class names and parse them into a listbox
var
   S,T,X:AnsiString;
   I:Integer;
begin
   SetDelimiter(#44); //use a comma delimiter
   S:=GetClasses;
   I:=1;
   repeat
      X:=Parse(S,',',I);
      if Length(X)>0 then ListBox1.Items.Add(X);
   until I=0;
end;
```

Group

function GetWinClass(**const** Title:Ansistring): Ansistring;

Returns the class name for a given Window title. Class names are fixed whereas windows titles may be changed at will.

Group

procedure GetMemStatus(**var** RAMTotal, RAMUsed, PGTotal, PGUsed:Integer);

Returns current status of Windows memory.

RAMTotal = Total physical memory available to Windows
RAMUSed = Percent of RAMTotal currently in use
PGTotal = Total swap file available to Windows
PGUsed = Percent of PGTotal currently in use

Group

procedure GetComList(Strings: TStrings);

Interrogates the Registry to obtain a list of all available COM ports and load them directly into a TStrings list.   If a modem is attached to the port, the ‘Model’ string parameter is included following the port name.

Example: GetComList(ListBox1.Items);   //loads list of available ports into ListBox1

Group

function GetCPU:AnsiString;

Returns identifying string for installed CPU type as follows:

80386	= Intel 386
80486	= Intel 486
80586	= Intel Pentium
4000	= Mips
21064	= DEC Alpha

Group

function GetDefaultPrn:AnsiString;

Returns info on the current system default printer as a tokenized string (comma delimited) of the form: PrinterName, DriverName, Port.   Returns null string on error.

Group

function IsWinNT:Boolean;

Returns true if OS is WIndowsNT, false otherwise (Windows95).

Group

function GetKeyValues(**const** Root:HKey;Key,Values:AnsiString):AnsiString;

Reads multiple, enumerated values from a given registry key and returns the data as a tokenized string.   If a given value is not found, a '?' is returned as a placeholder.

Note: The incoming Values string must be delimited using commas.   The resultant string is delimited using the current internal delimiter setting.   See SetDelimiter for more details.

Example: This example reads 5 values from the registry in order to set the caption of Label1 to show the Windows version, the version number, the current registered owner, the registered organization and the Windows product ID.

```
Label1.Caption:=GetKeyValues(HKEY_LOCAL_MACHINE, 'SOFTWARE\Microsoft\Windows\CurrentVersion',
                             'Version,VersionNumber,RegisteredOwner,RegisteredOrganization,ProductId');
```

Group

function SetTaskBar(**const** Visible:Bool):Boolean;

Enables/Disables the Windows taskbar based upon the provided Visible parameter.   Not a very 'Windows friendly' function but one that is necessary if you want your app to have the entire screen available.   Make sure the taskbar is enabled before your app ends.

Group

procedure NoTaskBtn;

Prevents the display of a taskbar button for the application.

Note: This routine is typically used in Project Source just after Application.Initialize.

Group

procedure KillOLE;

Unloads (actually de-references) OLE automation DLLs thus reducing the memory requirements of simple applications by roughly 1 meg.

WARNING: DO NOT use this routine unless you are confident that your app does not use OLE or variant data types. The data access components (TTable and TDataSource) have the potential to make use of variant data types.

Group

function GetProcID(**const** hWnd:THandle):THandle;

Returns a process handle given any Window handle created by the process.   The provided handle can be used with GetExitCodeProcess and other API functions to monitor and control the external process.

Note: To obtain the current process handle, use the GetCurrentProcess API function.

Example: The following function goes to great lengths to determine if NotePad is running.

```
function NotePadAlive: Boolean;
var
  hWnd:THandle;
  Status:DWord;
begin
   Result:=False;
   hWnd:=FindWindow('NOTEPAD',nil);
   if IsWindow(hWnd) then
      if GetExitCodeProcess(GetProcID(hWnd),Status) then
         Result := Status=STILL_ACTIVE;
end;
```

Group

function KillProc(**const** ClassName:AnsiString):Boolean;

Terminates the first process with the given window class name.

Example: KillProc('NOTEPAD') shuts down Windows Notepad if it is running.

Note: This is a rather rude, unconditional termination of an external process and should be used carefully. DLLs being used by the process may not be properly de-referenced.

Group

function DOSExec(**const** CmdLine:AnsiString; **const** DisplayMode:Integer):Boolean;

Execute a DOS app and automatically close the window on termination.   Path is optional but CmdLine must include the executable's extension.   DisplayMode is usually either sw_ShowNormal or sw_Hide.   Returns True if execution succeeds.

Note: Use *WaitExec* if a return code is required.

Group

function WaitExec(**const** CmdLine:AnsiString; **const** DisplayMode:Integer):Integer;

Execute an app, wait for it to terminate and then return the exit code.   DisplayMode is usually either sw_ShowNormal or sw_Hide.   Returns -1 if execution fails; otherwise, the return code is as provided by the executed app.

Group

procedure DebugConsole;

Implements a DOS-style console window for the display of debug messages using *DebugMsg()*. As a safety precaution, the Delphi IDE must be up and running; otherwise, this routine has no effect.

An internal flag tracks the current state and alternately creates and destroys the console. The initial call creates the console window, the next call destroys it.

Note: *To avoid a resource handle leak, the console must be destroyed before terminating the application.*

Group

procedure DebugMsg(**const** Msg:AnsiString);

Display a message in the debug console window created by *DebugConsole*.   If the console window is not active, this procedure has no effect.

Group

function ShellFileOp(**const** S,D:AnsiString; **const** FileOp,Flgs:Integer):Boolean;

Convenient interface to the myriad of options available for file operations using the Win95 shell. See *ShFileOperation* and *SHFileOpStruct* in WIN32.HLP for further discussion.

Note: *ShellAPI* must be added to the ‘uses’ clause.

(S)ource and (D)estination may contain multiple file paths and/or names separated by Delimiter.

FileOp can be FO_COPY, FO_DELETE, FO_MOVE or FO_RENAME.

Flgs can be any of the *fFlags* listed in *SHFileOpStruct*. Flags may be OR’ed together.

Returns True if operation succeeds and no user abort.

Example: Copy 3 files (File.001,File.002 and File.003) from the current directory to the \TEMP directory. Automatically rename the files if copies already exist.

```
SetDelimiter(#44); //use comma delimiter
ShellFileOp(‘File.001,File.002,File.003’, ’\Temp’,
                    FO_COPY, FOF_RENAMEONCOLLISION);
```

Group

function FormatDisk(Drive:Word):Boolean;

Convenient *modal* interface to disk formating dialog.   *Drive* is ASCII code for the disk drive, 65=A, 66=B, etc.

Returns True if *Drive* is valid and no user abort.

Group

procedure TrayInsert;

Simplified system tray interface. Places the application icon into the system tray with the application title used as the "hint" displayed whenever the mouse is held over the icon. Clicking the tray icon re-activates and re-displays the Main form.

Note: This procedure has no effect once an icon has been added to the tray.

Example: Standard usage places the following code in the Main form's OnClose event handler.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    TrayInsert;                //put icon into tray
    TrayClose(Action):     //alternative close
end;
```

It is often desirable to load the application directly into the tray with the Main form displayed only on request. The additional code below shows how to achieve this.

```
{Project source startup code}
begin
    Application.Initialize;
    Application.ShowMainForm := False;
    Application.CreateForm(TForm1, Form1);
    ShowWindow(Application.Handle, SW_HIDE);
    Application.Run;
end.
```

```
{Unit code}
procedure TForm1.FormCreate(Sender: TObject);
begin
    TrayInsert;   //add icon to tray on startup
end;
```

In order to manually terminate a tray application, some alternative means (button, menu selection, etc.) must be provided to remove the tray icon and actually end the program. The code below shows a tray application being terminated as a result of a button click.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    TrayDelete;                              //remove the tray icon
    Application.ProcessMessages;   //clear any pending events
    Application.Terminate;              //end the application, really
end;
```

Group

procedure TrayClose(**var** Action:TCloseAction);

Alternative close procedure for tray applications.   In effect, the Main form is hidden while the application remains active.   Has no effect unless an icon has been previously added to the system tray using TrayInsert.

Group

procedure TrayDelete;

Removes an icon previously inserted into the system tray using TrayInsert.

Group

```
function SetAppPriority(const Priority:DWord):Boolean;
```

Set the execution priority for the current application process.   Priority must be one of the following:

NORMAL_PRIORITY_CLASS
IDLE_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
REALTIME_PRIORITY_CLASS

See the Win32 docs for discussion.

Group

function GetFileDate(**const** FileName:AnsiString):AnsiString;

Returns file date and time string associated with a given filename.

Group

function MapNetDrive:Integer;

Invokes a dialog box for mapping network drives.   Returns NO_ERROR if a connection was established, returns -1 on user abort.   Any other value indicates either a network error or insufficient available memory.

Group

function UnSignedCompare(**const** X,Y:Integer):Boolean;

Does a full 32 bit unsigned integer compare for sorting and searching purposes. Returns True if X is greater than Y (exchange required for ascending order)

Group

function LoBit(**const** X:Integer):Integer;

Scans an integer for lowest non-zero bit.

Returns: bit#, (0-31) of lowest 1 bit, -1 if X = 0

Group

function HiBit(**const** X:Integer):Integer;

Scans an integer for highest non-zero bit.   See *LoBit* for return value.

Group

```
function RotL(const X,Cnt:Integer):Integer;
```

Rotate integer X left the number of bits specified in Cnt.

Group

function RotR(**const** X,Cnt:Integer):Integer;

Rotate integer X right the number of bits specified in Cnt.

Group

function TestBit(**const** X,Cnt:Integer):Boolean;

Tests the Cnt bit (0..31) of integer X. Returns True if bit is set (1); otherwise, False.

Group

function MetaPhone(**const** Name:AnsiString):Integer;

Phonetic spelling algorithm, similar to Soundex but more selective. Original algorithm by Lawrence Philips, 'Computer Language', Dec. 1990. There are a number of problems with this article --- discrepancies between code and text; some code is clearly missing. The implementation here more closely follows that of Gary Parker, 'C Gazette', June/July, 1991.

Returns: Integer representing phonetic spelling if Length(Name)>1; otherwise; 0. Integers are provided for faster comparison. If you prefer the more traditional string resultant, apply *IntToChr.*

Group

function NumToWord(**const** Source:AnsiString;Dollars:Bool):AnsiString;

Returns an English word translation of a numeric string.   'Dollars' is a logical flag (True/False) indicating that ‘Dollars’ and ‘Cents’ word units are to be included.

Group

```
function OrdSuffix(const X:Integerl):AnsiString;
```

Returns a two character English ordinal string suffix for an integer, ‘st’, ‘nd’, ‘rd’ or ‘th’ as in 1st, 2nd, 3rd, 4th, 12th, 542nd, etc..

Group

function Similar(**const** S1,S2:Ansistring):Integer;

Ratcliff/Obershelp pattern matching algorithm (DDJ, July 88). Returns a percentage ratio (0 - 100) representing the similarity of strings S1 & S2:

100 = Total match---identical
    0 = Total mismatch---nothing in common

Uppercase the two strings for case insensitivity.

Note: This routine uses an internal stack. To avoid potential overflow with very long strings, input strings are limited to 255 characters or less.

Group

procedure IntSwap(**var** I1,I2:Integer);

Quickly exchange the value of two Integers.

Group

function StrDelete(**var** A:array of Ansistring; **const** Target,Cnt:Integer):Boolean;

Delete Target element in a string array by doing a safe, "ring copy" of array elements downward (to lower index). If the array is zero based, Target = Element Index; otherwise, Target = Index - Base.

Cnt = Total active element count prior to the deletion; -1 = Full array.

Returns: True if successful and elements *following* Target are shifted downward to next lower index.

Note: The target element is not actually deleted but rather moved to the end of the array (to A[Cnt-1] if zero based). This last element can be deleted if desired; however, it may be more convenient and efficient timewise to simply leave it in place and adjust the active element count to exclude it.

Group

function StrInsert(**var** A:array of Ansistring; **const** Target,Cnt:Integer):Boolean;

Insert an element into a string array by doing a safe,"ring copy" of array elements upward (to next higher index) starting at Target element. If the array is zero based, Target = Element Index; otherwise, Target = Index - Base

Cnt = Total active element count prior to the insertion.   Must be less than the dimensioned size of the array (Dimensioned Size = High(A)-Low(A)+1).   In other words, the array must have space available to hold the inserted element.

Returns: True if Cnt is less than the dimensioned array size. Elements from Target to Cnt are shifted upward to next higher index.

Group

procedure StrSwap(**var** S1,S2:AnsiString);

Quickly exchange 2 strings.   Much, much faster than using assignments.

Group

procedure Dim(**var** P; **const** Size:Integer; Initialize:Boolean);

Allocate memory for a dynamic array.

P is a pointer to one of the pre-defined *HyperString* dynamic array types.   *Size* is the desired array size *in bytes*. *Initialize* indicates if *new* array elements are to be initialized to zero.   Existing elements retain their current value.

Upon return, *P* = address of the allocated memory block if sufficient memory was available; otherwise, *P* = nil.   Once initialized, *P* should not be manually altered.   Automatic de-referencing allows *P* to be used for array indexing without alteration or cumbersome pointer syntax.   See *dynamic array* above for examples.

Group

function Capacity(**const** P):Integer;

Determine the current memory allocation (bytes) for a dynamic array.

P is a pointer to one of the pre-defined *HyperString* dynamic array types previously initialized using *Dim*.

Group

Dynamic Arrays

HyperString offers a set of basic, pre-defined, single dimension dynamic array container types along with *Dim* and *Capacity* routines to simplify dynamic array use and management. By following the *HyperString* source code example, multi-dimension arrays can easily be created. Type names reflect the data contained; *TIntegerArray*, *TWordArray*, *TSingleArray*, *TDoubleArray* and *TCurrencyArray*.

Usage
With *HyperString;* fast, efficient dynamic arrays are as easy as 1-2-3.

1) Declare and initialize a pointer to the appropriate container type.
2) Dynamically allocate (and re-allocate as needed) an array size of up to 2 giga-bytes using *Dim.*
3) Access the array as needed using the pointer from step 1. With automatic de-referencing, the resulting syntax is virtually identical to that of a static array.

The currently allocated array size (bytes) is always readily available with the *Capacity* function. The following example illustrates the use of a dynamic integer array.

```
var
  iArray: ^TIntegerArray;                //pointer to dynamic integer array
begin
  iArray:=nil;                                    //initialize the pointer (very important);
  try
    Dim(iArray, 100*SizeOf(Integer),False);    //create 100 elements (0..99)
    if iArray<>nil then begin                              //size OK?
      iArray[50]:=123;                                          //set arbitrary value
      ShowMessage(IntToStr(iArray[50]));        //show that it worked
      Dim(iArray,1000*SizeOf(Integer),True); //re-size (with initialization)
      ShowMessage(IntToStr(iArray[999]));     //show initialization (should be 0)
      ShowMessage(IntToStr(iArray[50]));        //show persistence  (should be 123)
    end;
  finally
    Dim(iArray,0, False);                   //release array memory (very important)
  end;
end;
```

Indexing
As shown above, dynamic array elements are addressed in the same familiar manner as static arrays. In order to achieve maximum speed and efficiency, bounds checking is *NOT* provided. The user is solely responsible for proper array indexing.

CleanUp
The programmer is responsible for releasing all dynamic array memory (allocated by *Dim)* before the program ends. Failure to do so will produce a memory leak. For local arrays, a *try..finally* exception block can be used (see example above) to insure that memory gets released. For global arrays, a call to *Dim* should be placed inside Form.Close or some other event which is triggered in case of an abrupt shutdown. If an array has already been de-allocated, *Dim* has no effect.

Discussion
Array pointers ***must*** be initialized to **nil** before being used.

Empirical studies have shown that 2 is an ideal factor for efficient dynamic array management under Win32. For example; when expanding an array, the size should ideally be doubled. Likewise, an array should not be shrunk until it's size can be cut in half. This approach tends to minimize potential memory fragmentation and time spent on memory management.

With any dynamic array implementation, access speed must suffer due to the required runtime address manipulation. With the *HyperString* approach, access times are generally 2-3 times that of static arrays.

Group

Technical Support

We want to know about any problems you encounter when using this product. The best way to contact us is via e-mail at:

efd@mindspring.com

We use this library ourselves so in essence, your problems are our problems.   We can’t write your application for you; however, we will do our best to resolve any issues you bring to our attention.

Source

The full source code for *HyperString* is only $30US.   A great deal of the source is efficient, hand optimized and thoroughly commented BASM code. The Object Pascal compiler is certainly impressive but our tests show that the human mind armed with an assembler can still produce more optimal object code in many cases.

Currently, there are three ways to order the source code.

1) From CompuServe.

Type “GO SWREG” and register #14172.

Delivery is by e-mail to your CompuServe address.   This is the most efficient method for us so we absorb the transaction fees when you order from CIS.

2) From the World Wide Web.

With a major credit card, use the secure link from our web site at: http://www.mindspring.com/~efd

Delivery is by e-mail to your Internet address unless otherwise requested.   An additional $2 charge is added for credit card processing fees.   The generic order form describes this charge as ‘shipping & handling’.

3) By mail.

Send check or money order (US banks only) for $30US + 2$ shipping and handling to:

EFD Systems
304 Smokerise Circle
Marietta, GA 30067
USA

Delivery on 3.5” floppy disk is by a ground carrier of our choice.

function IsMask(**const** Source,Mask:AnsiString;Index:Integer):Boolean;

Validate Source from start to Index (-1 = Full) for conformance to a 'picture mask' (similar to Delphi's EditMask) composed from the following special character set.

A - Alphanumeric required (a..z,A..Z,0..9)
a - Alphanumeric permitted
C - Alphabetic required (a..z,A..Z)
c - Alphabetic permitted
0 - Numeric required (0..9)
9 - Numeric permitted
# - +/- permitted
? - Any required (#0..#255)
@ - Any permitted
| - Literal next, required
\ - Literal next, permitted

Example 1: IsMask(Source,’#09999\.999’,-1) fully validates a +/- numeric entry with 1 to 5 digits to the left of the decimal point and up to 3 trailing digits. Decimal point is allowed but not required.

Example 2: IsMask(Source,’00000\-9999’,-1) fully validates a US postal code. 5 digits are required, a dash and 4 additional digits are allowed but not required.

Example 3: IsMask(Source,’000|-00|-0000’,6) partially validates a US Employee ID number through the sixth char (the one just before the second dash). All digits and dashes are required. For this partial validation, the second dash and the last 4 digits may be missing.

Note: Extra trailing spaces are ignored, leading spaces are included in validation. Index provides support for partial, incremental validation. If Index <> -1, validation is only performed on the characters present. In other words, Source is allowed to be incomplete when compared to Mask. Index = -1 MUST be used to FULLY validate the entire Mask.

Group

procedure UCase(**var** Source:AnsiString;**const** Index,Count:Integer);

Upper case Count chars. in Source starting at Index.

Group

procedure LCase(**var** Source:AnsiString;**const** Index,Count:Integer);

Lower case Count chars. in Source starting at Index.

Group

function UniqueApp(**const** Title:AnsiString):Boolean;

Uses the Win32 GlobalAtom table to determine if an application is already loaded. Inserts a user-provided Title string into the GlobalAtom table as a signal that the app is loaded. Before insertion, the table is scanned (case-insensitive) for an existing instance of the title string. If found, the app must already be loaded OR a previous instance failed to cleanup the table. A dialog box alerts the user with the opportunity to either abort or continue loading.

Proper use requires an initial call at startup to modify the GlobalAtom table and the exact same call again at shutdown to clean things up. An internal flag tracks the current mode, either startup or shutdown. All your app must do is insure that the same Title string is provided to each call. Typically, the calls are placed inside the main form's OnCreate and OnClose events but Project Source can also be used (see examples below, use one approach or the other but not both). If the initial call fails (instance found, user aborts), the table is not altered and there is no need to clean up.

Returns: True if unique instance or user override; otherwise, False.

```
unit Example;   //Utilizing Main's OnCreate/OnClose events

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, ExtCtrls, HyperStr;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
const
  AppTitle='EFD Systems Test'; // Define our unique Title string

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  if Not UniqueApp(AppTitle) then Halt;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  UniqueApp(AppTitle);   //clean up the GlobalAtom table
end;

end.

//------------------------------------------------------
program Test;   //Utilizes Project Source.

uses
  HyperStr;    //Be sure to add this

{$R *.RES}

begin  //Note: This is a 'formless' app
  if Not UniqueApp('EFD Systems Test') then Exit;
  Application.Initialize;
  Application.Run;
  UniqueApp('EFD Systems Test');
end.
```

Group

function CalcStr(**var** Source:AnsiString):Double;

Provides mathematical string expression evaluation. Supports 4 basic math operators, powers, exponents, parenthesis's, Ln(), Exp() and Abs(). Other functions supported in source code version (MATH unit required). An exception is raised if the string cannot be fully evaluated for any reason.

Note: This function is a working demonstration of the power of *HyperString.* The entire source listing is less than 100 lines.

Group

function RndToFlt(**const** X:Extended):Extended;

Implements the popular 'biased' method of rounding whereby midpoint values are always rounded up to the next higher integer. This rounding method is widely used; hence, the demand for this function. Unfortunately, this method is based solely on convention rather than logic.

A value exactly mid-way between integers is as close to the lower integer as it is to the higher. Convention favors the higher value but there is no logical basis for favoring one over the other. The best (and perhaps the only) way to avoid bias in this case is to rely on random chance.

Computers are rather poor at random acts so Intel and the native Round() function do the next best thing by always favoring the nearest EVEN integer. For example; 1.5 rounds to 2, 2.5 also rounds 2 but 3.5 rounds to 4. This may initially seem ludicrous but statistically; over a range of randomly chosen midpoint values, it produces the same overall effect as random rounding. Midpoint values are rounded DOWN half the time and rounded UP the other half. Thus, the overall rounding errors tend to balance out and minimize any bias.

To illustrate, the table below computes the total of the first 4 midpoint fractions using both biased and unbiased rounding. As you can see, biased rounding (as provided by this function) quickly introduces a significant error into the overall resultant.

	Biased	Unbiased (even integer method)
	0.5 --> 1	0.5 --> 0
	1.5 --> 2	1.5 --> 2
	2.5 --> 3	2.5 --> 2
	3.5 --> 4	3.5 --> 4
Total	8.0 10	8.0 8

Group

function RndToInt(**const** X:Extended):Integer;

Implements the popular 'biased' method of rounding floating point values whereby fractions of 0.5 and greater are rounded up to the next higher integer.   Note that this function returns an INTEGER resultant.   See RndToFlt for discussion.

Group

function IPower(**const** X,Y:Integer):Integer;

Calculates integer powers (X^Y) without floating point Math unit.   Max. Y = 30.   Resultant = Zero on error or overflow.

Group

function IPower2(**const** Y:Integer):Integer;

Calculates integer powers of 2 (2^Y) without floating point Math unit. Max. Y = 30. Resultant = Zero on error or overflow.

Group

procedure SpeakerBeep;

Old fashioned DOS style beep using the system speaker in Win95.   Faster than the sound card, works even with sound driver muted.

Group

function iMax(**const** A,B:Integer):Integer;

Returns larger value, A or B.

Group

function iMin(**const** A,B:Integer):Integer;

Returns smaller value, A or B.

Group

function iMid(**const** A,B,C:Integer):Integer;

Returns mid-range value, A, B or C.   If any two are equal, the common value is returned.

Example: A:=5; B:=9; C:=10; iMid(A,B,C) returns 9;

Group

function GetKeyToggle(**const** Key:Integer):Boolean;

Returns True if the specified key is depressed or toggled on.   Any valid virtual key code may be specified.   The standard keyboard toggle switches are VK_INSERT,VK_NUMLOCK,VK_SCROLL and VK_CAPITAL.

Group

procedure FlashSplash(BitMap:TGraphic; **const** Title:AnsiString);

Dynamically create and display a modal splash form or ‘About’ box. Form shows a small user-specified graphic (64x64 max.) and form caption (Title) along with version info strings; ProductName, ProductVersion, LegalCopyright, CompanyName and LegalTrademarks (see Project | Options) and some Windows related info.

Example: Display a splash form with icon for 6 seconds.

```
FlashSplash(Application.Icon, ‘Welcome to’);
SleepEx(6000,False);
KillSplash;
```

Group

procedure KillSplash;

Destroy the splash form previously created by FlashSplash and free all associated memory.

Group

function ListComm:AnsiString;

Returns a tokenized string containing all available COM ports listed in the Registry.   Use SetDelimiter first to specify the token delimiter to be used.

Group

function OpenComm(**const** Mode:AnsiString):THandle;

Open a COM port for synchronous, non-overlapped read/write operation with default I/O buffer sizes, timeouts disabled and no flow control. Communication parameters are provided in a DOS-style MODE string. Returns the port handle if successful; otherwise, an exception is raised.

Example: Given a valid port identifier string (‘COM1:’, ‘COM2:’, etc.), the following function opens the port and checks for the presence of a modem.

```
function ModemThere(const Port :AnsiString):Boolean;
var
  Handle:THandle;
  Reply:AnsiString;
begin
  Result := False;
  try
    Handle := OpenComm(Port + ‘ BAUD=1200 PARITY=N DATA=8 STOP=1’);
    WriteComm(Handle,’ATZ’#13);
    SetLength(Reply,32);       //attempt to read 32 characters
    FillCnt(Reply,1,32,#0);    //initialize (just to be absolutely safe)
    Sleep(2000);               //give modem 2 seconds to respond
    if ReadComm(Handle, Reply) > 0 then Result  := Pos(‘OK’,Reply) > 0;
  finally
    CloseComm(Handle);
  end;
end;
```

TIP: The universal syntax for COM ports in Win32 is ‘\\.\COMXX:’ where XX is the port number. This syntax works for all ports but is only absolutely required for ports above COM9:.

Group

function ReadComm(**const** pHnd:THandle; **var** Bfr:AnsiString):Integer;

Attempt to read Length(Bfr) bytes from an open COM port.   Returns the number of bytes actually read.

Note: Any characters available in the receive buffer are returned in the event of a timeout.

Group

function WriteComm(**const** pHnd:THandle; **const** Bfr:AnsiString):Integer;

Attempt to write Length(Bfr) bytes to an open COM port.   Returns the number of bytes actually written.

Note: Flow control (if enabled) may cause this function to terminate with a count of the characters successfully transmitted.

Group

function CloseComm(**const** pHnd:THandle):Boolean;

Close an open COM port.   Returns True if successful.

Group

function EnCodeBCD(**const** Source:AnsiString):AnsiString;

Encodes an ASCII numeric string (0..9) into Intel packed Binary Coded Decimal format.

Group

function DeCodeBCD(**const** Source:AnsiString):AnsiString;

Decodes a string from Intel packed Binary Coded Decimal format into numeric ASCII (0..9).

Group

function AddUSI(**const** X,Y:Integer):Integer;

Unsigned 32-bit integer addition; X+Y.   An overflow exception is generated if the resultant exceeds 4,294,967,295.

Group

function SubUSI(**const** X,Y:Integer):Integer;

Unsigned 32-bit integer subtraction, X - Y. An overflow exception is generated if the resultant is negative (Y>X).

Group

function MulUSI(**const** X,Y:Integer):Integer;

Unsigned 32-bit integer multiplication. An overflow exception is generated if the resultant exceeds 4,294,967,295.

Group

function DivUSI(**const** X,Y:Integer):Integer;

Unsigned 32-bit integer division quotient, X Div Y.

Group

function ModUSI(**const** X,Y:Integer):Integer;

Unsigned 32-bit integer division remainder, X Mod Y.

Group

```
function CmpUSI(const X,Y:Integer):Integer;
```

Unsigned 32-bit integer comparison.   Returns zero if X=Y, positive if X>Y, negative if X<Y.

Group

function USIToStr(**const** X:Integer):AnsiString;

Convert an integer value to an unsigned numeric string.

Group

function StrToUSI(**const** Source:AnsiString):Integer;

Convert unsigned numeric string to an integer value. Generates an overflow exception if resultant exceeds 4,294,967,295 or the input is otherwise invalid.

Group

procedure IniRLE;

Initialize internal data structures in preparation for Run Length Encoding (RLE) or decoding of a new data stream.

Group

function RLE(**const** Bfr:AnsiString; L:Word):AnsiString;

Compress a string buffer containing repeated character sequences by applying a "safe" run length encoding (RLE) technique.   Effective buffer length may be specified in *L*. If *L*=0 or *L*>Length(Bfr) then Length(Bfr) is used.

This routine is "safe" in that it avoids adding control characters to the output.   High order ASCII characters (192..255) are used to represent repeat counts.

Note: Buffer length is intentionally limited to Word range to prevent overflow during de-compression.   Larger strings must be sub-divided for processing.

Input may include any binary 8-bit character; however, large numbers of widely dispersed higher order characters (192..255) can   result in output length actually exceeding input.

Group

function RLD(**const** Bfr:AnsiString; L:Word):AnsiString;

De-code a buffer string previously compressed using RLE.   An internal buffer will overflow and generate an access exception if Length(Result) exceeds Word range.   See RLE for further discusssion.

Be sure to re-initialize using IniRLE before start of de-compression.

Group

procedure InISQZ;

Initialize internal data structures in preparation for compression or de-compression of a new data stream.

Group

function SQZ(**const** Bfr:AnsiString; L:Word):AnsiString;

Compress a string buffer using an adaptive, variable-length encoding technique. Effective buffer length may be specified in *L*. If *L*=0 or *L*>Length(Bfr) then Length(Bfr) is used.

Note: Buffer length is intentionally limited to Word range to prevent internal overflow during de-compression.

This is a *sequential, stream oriented* compression method designed for use with 8-bit data. Large strings and files may be conveniently sub-divided for sequential processing. Initialization using IniSQZ is required prior to processing any string or file that is to be independently de-compressable.

Compression ratio may vary; a 30-40% reduction seems typical with English text. Both compression and de-compression are reasonably fast.

Example:

The code segment below sequentially reads a file of unknown size and produces a second, compressed output file. File open and close have been omitted.

```
IniSQZ;
SetLength(InBuffer,32767);
repeat
    BlockRead(InFile, InBuffer[1], Length(InBuffer), InSize);
    if InSize = 0 then break;
    OutBuffer := SQZ(InBuffer, InSize);
    BlockWrite(OutFile, OutBuffer[1], Length(OutBuffer));
until InSize < Length(InBuffer);
```

Group

function UnSQZ(**const** Bfr:AnsiString; L:Word):AnsiString;

De-compress a buffer string previously compressed using SQZ.

Note: An internal 64K buffer is used for de-compression. This buffer will overflow and generate an address exception if the expanded data exceeds 64K. To avoid any potential for overflow, Bfr should be 32K or less.

This is a *sequential, stream oriented* compression method. Large strings and files may be conveniently sub-divided for processing; however, initialization and data sequence must match that used for compression.

Example:

The code segment below sequentially reads a compressed file of unknown size and produces a second, de-compressed output file. File open and close have been omitted. See SQZ for a corresponding compression example.

```
IniSQZ;
SetLength(InBuffer,32767);
repeat
    BlockRead(InFile, InBuffer[1], Length(InBuffer), InSize);
    if InSize = 0 then break;
    OutBuffer := UnSQZ(InBuffer, InSize);
    BlockWrite(OutFile, OutBuffer[1], Length(OutBuffer));
until InSize < Length(InBuffer);
```

Group

function BPE(**const** Bfr:AnsiString; L:Word):AnsiString;

Compress a text string buffer using an exhaustive, multi-pass, block-oriented “byte pair encoding” technique. Effective buffer length may be specified in *L*. If *L*=0 or *L*>Length(Bfr) then Length(Bfr) is used.

Note: Buffer length is intentionally limited to Word range to prevent overflow during de-compression. Large files and strings must be sub-divided for processing. To avoid any potential for overflow, Bfr should be 32K or less.

This is a *block-oriented* compression method. Initialization is automatic at the start of each block. Output is a complete, stand alone compressed data block. De-compression with BPD requires the full, complete block as originally produced by this routine.

Designed for use with 7-bit ASCII text. Control and 8-bit characters are tolerated but the effective compression ratio may be reduced. Large numbers of 8-bit characters may result in output length actually exceeding the input.

Compression ratio may vary; a 50% reduction seems typical with English text. Compression is rather slow but de-compression is very fast.

Group

function BPD(**const** Bfr:AnsiString; L:Word):AnsiString;

De-code a string buffer previously compressed using BPE.

Note: This is a *block-oriented* compression method.   As such, only *whole, autonomous* data blocks as originally produced by BPE can be successfully de-compressed.   Incomplete, sub-divided blocks or other data will produce unpredictable results.

See BPE for further discusssion.

Group

License Agreement and Disclaimer

Long Dynamic Strings (AnsiStrings)

This is a concise background primer for those unfamiliar with the new 32-bit long dynamic string type known as AnsiString.

Structure

Under the hood, an AnsiString is fundamentally a pointer to a dynamic memory block. Implicit pointer de-referencing; provided courtesy of the compiler, tends to obscure this. In response to an AnsiString declaration, only a string header; which includes a pointer, is allocated. Memory storage for the actual string data (referenced by the pointer) is dynamically allocated on assignment. In contrast, the older Pascal style strings were always allocated a static 256 byte memory block. The first byte of the block held the effective length; therefore, usable string length was limited to 255 bytes.

Efficiency

The dynamic memory used by AnsiStrings is transparently managed (allocated, de/re-allocated as required) by the Delphi memory manager. As a result, an AnsiString can be assigned any length within the limits of available memory. A string that holds 2 bytes can be easily and effortlessly expanded to hold 2 million bytes. As you can see, this is much more powerful than the older string implementation. However, with power comes responsibility. Abuse of the memory manager is tempting and can adversely affect performance. Consider this example:

```
    Poor:           S2 := '';
        for I := 2 to Length(S1) do S2 := S2 + S1[I];

    Better:         SetLength(S2, Length(S1) - 1);
        for I := 2 to Length(S1) do S2[I-1] := S1[I];
```

The same result is achieved in each case but the second is potentially much more efficient. In the first case, string S2 is being built incrementally, byte by byte. As the string outgrows it's memory allocation, a whirlwind of behind the scenes activity is triggered. A new, larger memory block is allocated, existing string data is copied to the new block and finally the old block is freed. This is all handled transparently but it still takes time. Depending upon Length(S1), the re-allocation process may be triggered dozens of times from within the loop. In the second example, memory is explicitedly allocated once in advance.

Most complaints regarding AnsiString performance can be traced to memory manager abuse and overuse. As shown in the second code example, such abuse can be avoided; however, for those accustomed to the older static strings, some changes in coding style may be required. Aside from their more powerful, dynamic nature; AnsiStrings are simply a linear sequence of bytes (like all strings) and are every bit as fast and efficient as the older Pascal variety.

Compatibility

Windows is largely written in C. As a result, WinAPI functions expect C-style, null terminated strings. For compatibility, AnsiStrings are also null terminated. Outside the API functions, this terminating null is not normally accessible and thus *can not and does not* serve as the primary indicator of string length.

If a null doesn't do it, what does determine the length of an AnsiString? Instead of an embedded length indicator, effective string length is stored in the AnsiString header, alongside the dynamic memory block pointer. This requires a very small amount of storage overhead but the benefits are well worth it. AnsiStrings are much easier to use and more efficient than C-style strings and much more versatile than the older Pascal strings. String length is readily accessible using the Length() function whereas in C, a time consuming scan is required. Length can be set implicitedly by assignment or explicitedly using SetLength(). As was shown above, more efficient memory management can be achieved by explicitedly pre-setting length whenever possible.

Versatility

Within an AnsiString, a character is a character. No single character has any special significance over any other, not even a null. Therefore; an AnsiString is in effect a managed, dynamic, binary buffer. As such, AnsiStrings can be used for a whole range of applications previously beyond the reach of strings. For example, dynamically allocated buffers (GetMem(), GlobalAlloc(), etc.) with pointer addressing and mandatory cleanup (FreeMem(), GlobalFree(), etc.) can often be replaced by a safer, more convenient AnsiString buffer. To illustrate, HyperString features a rather unique implementation of dynamic numeric arrays using AnsiStrings as container structures.

Summary

AnsiStrings are a new, more powerful string type available for the first time with Delphi32.   With proper coding, these new strings are just as efficient and much more powerful and versatile than the older Pascal strings.   At the same time, they offer compatibility with the C strings used by the WinAPI.

With power, versatility and compatibility, why use anything but AnsiStrings?